How To Touch a Running System

Reconfiguration of Stateful Components

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig-Maximilians-Universität München
zur Erlangung des Grades Doctor rerum naturalium (Dr. rer. nat.)

vorgelegt von

Moritz Hammer

eingereicht am 24.4.2009

Berichterstatter:	Prof. Dr. Martin Wirsing
	Prof. Dr. František Plášil
	Prof. Dr. Alexander Knapp
Tag des Rigorosums:	7.5.2009

Abstract

The increasing importance of distributed and decentralized software architectures entails more and more attention for adaptive software. Obtaining adaptiveness, however, is a difficult task as the software design needs to foresee and cope with a variety of situations. Using reconfiguration of components facilitates this task, as the adaptivity is conducted on an architecture level instead of directly in the code. This results in a *separation of concerns*; the appropriate reconfiguration can be devised on a coarse level, while the implementation of the components can remain largely unaware of reconfiguration scenarios.

We study reconfiguration in component frameworks based on formal theory. We first discuss programming with components, exemplified with the development of the CMC model checker. This highly efficient model checker is made of C++ components and serves as an example for component-based software development practice in general, and also provides insights into the principles of adaptivity. However, the component model focuses on high performance and is not geared towards using the structuring principle of components for controlled reconfiguration. We thus complement this highly optimized model by a message passing-based component model which takes reconfigurability to be its central principle.

Supporting reconfiguration in a framework is about alleviating the programmer from caring about the peculiarities as much as possible. We utilize the formal description of the component model to provide an algorithm for reconfiguration that retains as much flexibility as possible, while avoiding most problems that arise due to concurrency. This algorithm is embedded in a general four-stage adaptivity model inspired by physical control loops. The reconfiguration is devised to work with stateful components, retaining their data and unprocessed messages. Reconfiguration plans, which are provided with a formal semantics, form the input of the reconfiguration algorithm. We show that the algorithm achieves perceived atomicity of the reconfiguration process for an important class of plans, i.e., the whole process of reconfiguration is perceived as one atomic step, while minimizing the use of blocking of components. We illustrate the applicability of our approach to reconfiguration by providing several examples like fault-tolerance and automated resource control.

Zusammenfassung

Die wachsende Bedeutung von verteilter und dezentralisierter Software steigert das Interesse an adaptiver Software. Adaptivität is jedoch schwierig zu erreichen, da das Design der Software eine Vielzahl von unterschiedlichen Situationen vorhersehen und behandeln können muss. Rekonfiguration von Komponenten verspricht diese Aufgabe zu vereinfachen, da die Adaptivität auf der Architekturebene und nicht direkt auf der Codeebene stattfindet. Dadurch kann "*separation of concerns*" erreicht werden – die Rekonfiguration wird auf einem grobgranularen Level geplant und durchgeführt, während die feingranular geschriebenen Komponenten grösstenteils unbehelligt bleiben und nicht angepasst werden müssen.

Wir untersuchen Rekonfiguration in Komponentenframeworks, die auf einer formalen Theorie aufgebaut sind. Dazu diskutieren wir zuerst, wie mit Komponenten Software entwickelt werden kann; dies wird exemplarisch an der Entwicklung des CMC Model Checkers beschrieben. Dieser Model Checker ist extrem optimiert und in C++ geschrieben; er dient als Beispiel für den Entwicklungsprozess und zeigt erste Möglichkeiten für Adaptivität auf. Da das verwendete Komponentenmodell jedoch Priorität auf Performanz legt, kann Rekonfiguration nur bedingt Nutzen aus der strukturierenden Eigenschaft von komponentenorientierter Softwareentwicklung ziehen. Wir wenden uns daher einem Komponentenmodell zu, das durch exklusive Verwendung von "message passing" zur Kommunikation von Komponenten maximale Trennung der Komponenten erzielt und Rekonfigurierbarkeit zu einem zentralen Prinzip erhebt.

Rekonfiguration kann von einem Framework unterstützt werden, indem vor dem Programmierer möglichst viele Probleme des Rekonfigurationsprozesses verborgen werden. Auf der Basis einer formalen Beschreibung des Komponentenmodells entwickeln wir einen Algorithmus, der möglichst viel Flexibilität bezüglich des Systems und der darauf realisierten Rekonfiguration bewahrt, gleichzeitig jedoch Probleme mit nebenläufiger Komponentenausführung vermeidet. Diesen Algorithmus betten wir in ein vierstufiges Adaptivitätsmodell ein, das physikalischen Kontrollschleifen nachempfunden ist. Rekonfiguration betrachtet dabei zustandsbehaftete Komponenten, deren Daten und unbearbeitete Nachrichten bei der Rekonfiguration erhalten bleiben sollen. Als Eingabe für den Rekonfigurationsalgorithmus werden Rekonfigurationspläne mit einer klar definierten Semantik verwendet. Für eine wichtige Klasse dieser Pläne zeigen wir, dass die Ausführung des Algorithmus als ein atomarer Schritt wahrgenommen wird, während nur eine unbedingt notwendige Menge von Komponenten blockiert werden muss. Die Anwendbarkeit dieses Ansatzes wird mit einer Reihe von Beispielen, wie Fehlertoleranz und automatisierter Ressourcenkontrolle, illustriert.

Acknowledgments

I thank Prof. Dr. Martin Wirsing for supervising my thesis. I will always remember the scientific atmosphere as well as the warmth and friendliness that Prof. Wirsing established at the PST (Programming and Software Technique) chair. I thank Prof. Dr. Alexander Knapp, for countless discussions and suggestions, proofreading and education in the finer aspects of writing a scientific document. I also thank Prof. Dr. František Plášil for agreeing to be my second referee, and for the valuable suggestions that helped much to improve this thesis.

I have to thank everybody working at the PST chair, being the best colleagues one can wish for, but I want to point out the support and friendship I experienced from Toni Fasching, Andreas Schroeder and Philip Mayer. Also, Stephan Merz and Dirk Pattinson, who are no longer working here, contributed much to my scientific progress.

I started working on this thesis as a participant of the "Graduiertenkolleg Logik in der Informatik", funded by the German Research Society (DFG). I would like to thank Prof. Dr. Helmut Schwichtenberg for the opportunity to participate in this program, and for the generous support of the "Munich Model Checking Day", which sparked the beginning of my research that eventually led to the interest in reconfiguration.

Michael Weber was my co-author for a paper that, at first, appeared totally out of line and then became a major influence for this thesis. The research described in this paper would not have been possible without the generous sharing of the VMSSG virtual machine by Michael, and the paper would not have been half as good if it was not for his strive for clarity.

I also have to thank Gustaf Ganz, Sonja Hofer, Andi Pietsch, Norbert Romen and Sascha Burger for valuable discussions. Without their comments and suggestions, writing this thesis would have been much more difficult. I sincerely hope that I can repay their priceless support by helping them in their own endeavors in the future.

Reconfiguration needs to keep some parts of the system unchanged. For their invaluable help in this regard, I have to thank Katja and Manuel Ströh, David Born and Kerstin and Thomas Langenberg. And finally, I have to thank Eva Kneitz for her invaluable help with this thesis, which was far more than what I could have hoped for.

Contents

Contents vi

List of Figures viii

List of Tables xi

Chapter 1. Introduction 1
- 1.1. Adaptive Software 1
- 1.2. Frameworks for Adaptive Software 1
- 1.3. Components and Reconfiguration 3
- 1.4. Contributions and Scope 4
- 1.5. Notational Conventions 5
- 1.6. Structure of this Thesis 6

Chapter 2. Components and Reconfiguration 8
- 2.1. What are Components? 8
- 2.2. What are No Components? 11
- 2.3. Key Terms and Concepts 14
- 2.4. Why Use Components? 15
- 2.5. The Difficulty of Reconfiguration 23

Chapter 3. Related Work 27
- 3.1. Component Frameworks 29
- 3.2. Component Frameworks Supporting Reconfiguration 37
- 3.3. Formal Approaches to Adaptive Component Systems 56
- 3.4. Special Aspects and Applications of Reconfiguration 57
- 3.5. An Assessment 63

Chapter 4. Formal Foundation of Components 66
- 4.1. Components 66
- 4.2. Component Process Terms 70
- 4.3. Component Setups 71
- 4.4. Concepts for Describing Dynamic Behavior 73
- 4.5. Component Models 78
- 4.6. A "Programming Language" for Components 79

Chapter 5. A Case Study of a Synchronous Component Model: The CMC Model Checker 81
- 5.1. Evolution of the CMC Model Checker 83
- 5.2. The Component Framework of CMC 86
- 5.3. Data Flow of the CMC Model Checker 94
- 5.4. Benchmarking CMC 98
- 5.5. Benefits of Component-Based Software Engineering for CMC 100
- 5.6. Future Work 103
- 5.7. Conclusions 106

Chapter 6. A Component Model for Reconfiguration: JComp 109
6.1. Requirements 109
6.2. Design Principles 113
6.3. The Component Model 115
6.4. The Approach to Reconfiguration 116
6.5. Reconfiguration Plans 119
6.6. Rules for Reconfiguration 121
6.7. Reconfiguration Plan Implementation 125
6.8. State Transferal 135

Chapter 7. Implementing the JComp Model in a Component Framework 143
7.1. The Active Object Pattern 143
7.2. Implementation 145
7.3. Distributing JComp 153
7.4. Case Studies 158
7.5. Extensions of the JComp Component Model for Supporting Reconfiguration 161

Chapter 8. Applications of Reconfiguration 170
8.1. MAPE 172
8.2. Hot Code Updates 189
8.3. Handling External Changes 195
8.4. Dynamic Cross-Cutting Concerns 206
8.5. Reconfiguration for cmc 224
8.6. Discussion 227

Chapter 9. Component Correctness and Correct Reconfiguration 229
9.1. Concurrent Contracts 229
9.2. Patterns for Precondition-Preserving Concurrency 235
9.3. Verification of Specifications 239
9.4. Evaluating Correctness of Reconfiguration 247
9.5. Mutual Consistent States 249
9.6. Transaction-Preserving State Transferal 252

Chapter 10. Conclusion 257
10.1. Discussion 257
10.2. Future Work 265

Bibliography 268

Index 290

List of Figures

1.1 Example of attaining adaptivity through reconfiguration 4

2.1 Components of the Space Shuttle 9
2.2 Extend of views for a component setup and services 12
2.3 Strategy pattern 21
2.4 Ingredients of reconfiguration 25

3.1 Java BeanBuilder screen-shot 31
3.2 Patch bays, in real life and virtual 57
3.3 Complexity of related work 64

4.1 Static elements of a component 67
4.2 Provided interfaces and roles 68
4.3 "Reverse" Chain of Responsibility pattern 69
4.4 A component in a UML component diagram 70
4.5 Example run reduction 74

5.1 State Space Visualization for Peterson's Mutual Exclusion Protocol 81
5.2 First ideas of cmc 83
5.3 Pseudo-code for the cmc algorithm 84
5.4 Problems with the "Auto-Atomic Filter" improvement of cmc 86
5.5 Distribution of CPU time for cmc runs 88
5.6 Example runs comparing loops and self-invocations 92
5.7 Life-cycle of states in cmc 94
5.8 Specification of the simplified core algorithm 95
5.9 Distributing cmc 97
5.10 Effectiveness of caches with different parameters 100
5.11 Smooth degradation 101
5.12 Speedup obtained by various zLIB compression levels 101
5.13 Components of the cmc model checker 108

6.1 Steps of reconfiguration 118
6.2 Continuation of synchronous calls during reconfiguration 121
6.3 Sets of components for the coarse-grained rule 121
6.4 Message copying via δ 122
6.5 State machine of a component, with rules to reach new states 123

6.6 A reconfiguration scenario illustrating the problems with message retainment 126
6.7 Example for embedding of reconfiguration 131
6.8 A problem with non-injective shallow reconfiguration plans 134
6.9 Problems with an image intersecting $C \setminus R$ for ρ 134
6.10 Extended component state machine 138
6.11 Rules for indirect state transferal 138
6.12 Hybrid state transfer 141

7.1 Active Object pattern, as described in [LS96] 144
7.2 Active Object pattern, as realized in JComp 146
7.3 Message passing cost 147
7.4 Component setup for the producer/consumer example 150
7.5 Component graph with edge/node weights 156
7.6 Proxy component for network connections 157
7.7 Sample output of NewsCondensed 159
7.8 Components of NewsCondensed 160
7.9 Components of the web crawler 161
7.10 Component with self-invocation capability 162
7.11 UML class diagram for reconfiguration plan classes 167

8.1 Closed control loop 172
8.2 MAPE loop 174
8.3 Filter component 177
8.4 Adapter component 178
8.5 Progression of a cmc run 180
8.6 SPO graph transformation rule example 185
8.7 Graph transformation rule for filter removal 186
8.8 Hybrid reconfiguration – logging example 192
8.9 Example JavaScript code for state retainment 194
8.10 Impact of reconfiguration on an asynchronous system's message latency 195
8.11 Chain of Command pattern 196
8.12 Substitution of a multiport by a Chain of Responsibility 197
8.13 Organization of RSS reader chain 199
8.14 Reconfiguration of an RSS reader 202
8.15 Adding a placeholder for handling network failure 203
8.16 Hierarchical component abstractions 206
8.17 Three-stage reconfiguration of a network edge 208
8.18 Architectural reconfiguration scenarios 210
8.19 The "unbalanced producer/consumer" example 213
8.20 Hybrid automaton for the analyzer of the unbalanced producer/consumer example 214
8.21 Degrading replay accuracy 215

8.22 Fault tolerance components 217
8.23 Specification of the fault tolerance example 218
8.24 Time measurements of `java.util.TreeSet` 221
8.25 Cost of reconfiguring `java.util.TreeSet` 222
8.26 Comparing insertion and iteration efficiency 223
8.27 Speedup obtained by reconfiguring 223

9.1 Phases of message execution 230
9.2 Evaluation of a method specification 233
9.3 Bank example with quotas 236
9.4 Bank example with a proxy 237
9.5 State automaton example 240
9.6 Specification Büchi automaton example 241
9.7 A queue automaton and its abstraction 246
9.8 A queue automaton that cannot be finitely abstracted by regular model checking 246
9.9 Atomic and interleaved reconfiguration checking 251
9.10 Client/Hash table example 253

List of Tables

3.1 Overview of component frameworks supporting reconfiguration 41
3.2 State-considering frameworks 55

5.1 A comparison of I/O effort for different forms of state storage of the CMC model checker 85
5.2 Some statistics for smaller models checked in CMC 85
5.3 Rules for the CMC component model 90
5.4 Large models checked in CMC 99
5.5 Comparison of different reclaiming strategies 102

6.1 Configuration transition rules 115
6.2 Reconfiguration transition rules 124

7.1 Component network creation commands in JCOMP 149
7.2 Observable stages of message processing in JCOMP 152

CHAPTER 1

Introduction

Any improvement will be for the better.
— Jack McKay

1.1. Adaptive Software

Ubiquitous computing, a term coined in [**Wei91**], is becoming more and more a reality (although maybe not exactly a reality as it was originally envisioned, cf. [**BD06**]). While computers are involved in a variety of everyday scenarios, they become less and less visible. When purchasing a washing machine, advertisement no longer puts the fact that it is controlled by a small computer on top of the features list. Computers in cars are no longer even avoidable, since their benefit is so vast – in terms of improved comfort for the driver as well as cost-saving for the car manufacturer. As computers become more and more commonplace, the concept of computation also changes: Often, the role of a single computer is not to be a powerful tool that is used under close observation by a knowledgeable user, but to run unattended and unbeknownst to their owner for long times and preferably never require interaction with an expert user anymore.

This changes the way the software running on those computers needs to be written: Instead of requiring frequent action of the user to move away obstacles that hinder the successful execution, software now needs to cope with problems on its own. This even holds true for classical software run on regular personal computers – it is more and more expected that the software runs on a variety of systems with different hardware and software present; the days where a computer game could dictate which sound-card was to be installed are long gone. The integration of computers in everyday life requires the software to be more flexible.

If a software remains flexible while it is running, we consider it to be adaptive. Instead of (or additionally to) being able to operate in a variety of environments, adaptive software should hence be capable of responding to *changes* in the environment. Adapting to those changes comes in a variety of forms: Fault tolerance (i.e., to continue working even if some fault condition has been met, say by the loss of a required communication partner), self-tuning (i.e., taking advantage of changes in the environment to improve one's performance) or hot code updates (i.e., integrating improved software into the running system) are aspects of adaptivity.

But adaptivity is not limited to those direct responses to changes of the environment. Instead, it is a very broadly defined approach towards writing any software. In this thesis, we will investigate this software engineering approach rather than individual adaptive solutions. This leads to the definition of a *framework* that facilitates adaptiveness and allows for experimentation with such software.

1.2. Frameworks for Adaptive Software

Software usually interacts with its environment. Even for scientific computing, where calculations are running autonomously for a very long time, the environment

needs to be considered, e.g., when requesting memory from the operating system. Since the environment is usually prone to unexpected and sudden changes (e.g., by having the memory taken away by an independent process), most software can be considered to be adaptive.

But this is certainly not the kind of adaptiveness that is associated with the term "adaptivity" and the work that analyzes it (e.g., [**Lad00**]). Rather, adaptiveness describes the way the software is devised; instead of writing a monolithic program that checks whether the memory is available before it allocates some (and usually fails with an error message if the memory available is insufficient), special precautions are added to facilitate the process of adapting to a changed environment. Thus, the scientific computing applications might utilize a framework that takes care of giving the memory to those pieces of software that need it most, and possibly changes the structure of the software if this goal cannot be met. The main topic of this thesis is to provide such a framework, and study the problems emerging from its use.

Talking about goals immediately illustrates the close relationship to agent-style reasoning and planning. Agents are usually written in a framework that facilitates planning and reasoning [**KMW03, Lin01**]. Since planning is a difficult problem, the agent community's research interest is focused on its challenges. In this thesis, however, we will place more emphasis on the utilization of a plan (which we consider to be given by the user, most of the time), and rather research the problems related to actually writing adaptive software. The resulting framework might then be used to implement a software that enables planning and reasoning for agents.

Writing adaptive software is hard. Anyone who has implemented a complex data structure knows the reason: For example, when implementing a hash table, most of the code is required for the events encountered least often: Collision resolution and, possibly, table growth. It is quite easy to get a hash table implementation correct with respect to a single element insertion at the first try, but it is equally easy to screw up the code that reorganizes a hash table in an attempt to adapt to a larger-than-anticipated data volume by hash table growth (preferably, without keeping everything in memory twice). This illustrates the basic problem of adaptive software: The code that is hardest to write is executed least often. Furthermore, testing that only verifies the API (i.e., if an element added to the hash table indeed shows up during a later search) is prone to missing such problems, as the coverage is obviously insufficient – as the reorganization of the hash table is not experienced during the (black-box) tests. Things are made worse if nondeterminism is involved (possibly in the disguise of a concurrent system, or an open environment), as this makes the discovery of bugs non-reproducible.

One of the major requirements for a framework supporting adaptiveness is to ease the process of adaptation as much as possible, in order to relieve the software designer from coping with most of the problems induced by adaptivity. Also, the framework might provide valuable information to the designer if some known causes of errors are detected and reported. For example, adaptation in concurrent systems is very prone to deadlocks. The framework might help the user by avoiding adaptation to happen in a way that induces a deadlock, or at least hint at the risk of deadlocks, should the adaptation be attempted in a risky situation.

In this thesis, we describe adaptation as a process with a well-defined begin and end; this is a design decision that stems from the idea of maintaining a software system and changing its structure from time to time (and, as stated previously, only in rather exceptional cases). This idea of adaptivity dominates all design decisions, but it should be noted that, from a very broad point of view, adaptation does not necessarily have to be a distinct process.

A framework that helps a software developer in creating adaptive software needs to fix the way in which adaptivity is achieved in order to provide some guarantees and guidance. In the memory allocation example, adaptivity can only be supported if the treatment of the memory shortage is more or less handled by the framework. An abundance of possibilities how to achieve this has been proposed; e.g., by employing an exception handling mechanism like it is done in the JAVA programming language, or by supplying an event-handling architecture (as most GUI frameworks do). In this thesis, we will employ *components* and *reconfiguration*.

1.3. Components and Reconfiguration

The term "component" describes a very broad range of concepts, but all definitions have a common denominator: The explicit declaration of the communication partner requirements. Usually, this is done in the form of *interfaces*, i.e., sets of *methods* that can be invoked on the component or that the component itself needs to invoke (called the *provided* and *required* interfaces). Little more can be assumed to be subsumed by the term "component" (we will clarify the use of the term in the context of this thesis in Sect. 2.1), but this is already sufficient to see how reconfiguration will fit into component models: Since components make their communication capabilities and requirements explicit, we can substitute components with suitable capabilities. This is a coarse-grained approach, as we will replace entire components, but since the term "component" is so broad, this does not necessarily pose a restriction yet. Since an important aspect of components is their *configuration*, i.e., which component exists and how they are connected, we call the adaptivity of component systems *reconfiguration*.

Using reconfiguration as a means to realize adaptivity helps to maintain a *separation of concerns*, a term introduced by Dijkstra [**Dij82**]: Instead of putting the normal functionality and the adaptivity-related functions into the same code, the issues are addressed separately. The memory allocation problem, which could be implemented using a mundane case distinction within a component (i.e., if not enough memory is available, the program branches and takes appropriate action, say, by asking other data structures to reorganize themselves in order to free some memory), might be solved by adding a listener to out-of-memory errors: Such a listener (which is maintained and invoked by the framework) might then be able to devise a treatment for resolving the situation by conducting a reconfiguration that replaces some memory-wasting components with other (maybe not so efficient in terms of runtime, but more memory-preserving) components. The separation of concerns is given by the fact that in the latter solution, the implementer of the memory-requesting code does not have to worry about resolving the situation; given a powerful enough framework, the code only needs to signal the error to a listener and maybe wait for being notified that a solution has been found. How this solution is obtained becomes somebody else's concern, whereas in the case-distinction solution, the implementer needs to think of such a solution at the time the problem-prone code is written.

Of course, a clean distinction is hard and most of the time impossible. In the course of this thesis, however, we will frequently encounter separation of concerns again (usually manifested in a separation of roles), and see how we can benefit from such an approach.

An important aspect of this thesis is the explicit consideration of component *states*. The components we consider are stateful, i.e., the past communication influences the future communication. This makes adaptation harder: Not only has a suitable substitute to be found, but the state needs to be retained also. In

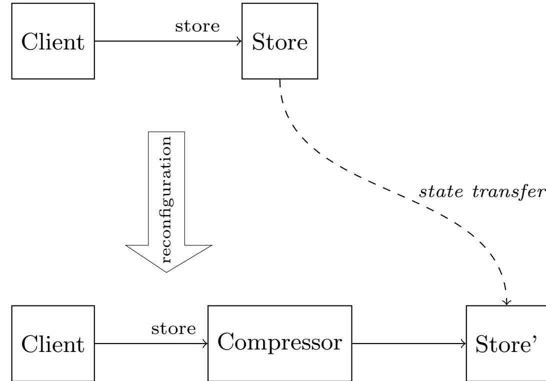


Figure 1.1: Example of attaining adaptivity through reconfiguration

the memory-allocation example, substituting a component by a component that requires less memory is an easy approach to resolve memory shortage, but the state of the substituted component needs to be retained; hence the state of the old component needs to be copied, which can become tricky if the memory is too scarce to hold the old and new component in memory at once. However, the problems we are interested in need to employ stateful components, and since we expect reconfiguration being hard to debug, we need to consider the transferal of state within the reconfiguration framework as well as the actual component substitution.

Fig. 1.1 illustrates how reconfiguration can be used to achieve adaptivity: Originally, a client component is connected to a store component, which it utilizes to have some data stored. The client effects storing by sending communication to the store component. The component is, however, not aware of the exact identity of the store component, it just knows about the existence of some connection to another component that provides data storing services.

Now, we might run into a situation where the store component cannot offer these services any longer because the memory is exhausted. In order to adaptively respond here, we want to compress the data that is stored, and all further data. This is implemented by the addition of a compressor component that pre-processes the data before storing it in a new store component. This component might be a new copy of the old store component, if this one is generic enough to store the compressed data as well, or a different implementation. During the reconfiguration, the data accumulated in the old store component needs to be transported to the new store component, and it needs to become compressed during this transfer.

The important property of such a reconfiguration is given by the fact that the client component can stay completely ignorant of the substitution of the store component. It may continue sending messages with data store requests, and does not need to care that these messages are now received by a compressor rather than the original store component.

1.4. Contributions and Scope

In this thesis, we investigate reconfiguration of component systems, which is an area of increasing interest (cf. Sect. 3.5). Many frameworks have been built to support and research reconfiguration, with varying scope and elaboration. In this thesis, we make the following contributions to the area of reconfiguration and its use in component-based software engineering:

(1) The definition of the JComp component model, which is a formally described component model with reconfiguration capabilities. The semantics of this component model are described by means of term rewriting on a small-step granularity, obtaining a component model that can be readily implemented in a regular programming language.
(2) Within JComp, a reconfiguration algorithm is defined which is minimally invasive as it stops only the components that are about to become removed. We prove that this algorithm still appears as an atomic step to an outside observer, i.e., it acts as if the entire system had been stopped for the duration of the reconfiguration.
(3) We discuss ways to transfer a component's state during reconfiguration, based on a distinction introduced by Vandewoude [**Van07**]. We propose a novel approach for state retainment that combines the benefits of the proposed approaches.
(4) We report on the implementation of the JComp model in a component framework, and report on a number of examples which exhibit various domains where reconfiguration can be applicable.
(5) We present a case study of a high-performance model checker which uses a novel algorithm for storing states in order to illustrate and justify the design decisions we made for JComp.
(6) We report on ways to specify the behavior of components based on a novel view on contracts in a concurrent environment. We extend this view to reconfiguration and discuss how the prevalent theoretical consideration of reconfiguration, namely the attaining of quiescent states [**KM90**], can be substituted by state transferal in our approach.

We report on practical experience with reconfiguration and component-based software engineering. The inclusion of both formal consideration and practical experience is important, as we believe that neither should be done without the other. Reconfiguration in an asynchronously communicating, highly concurrent system is difficult to maintain; it is important to have some guarantees that alleviate the process of designing a working reconfiguration. A clear semantics of the reconfiguration process is indispensable for this. On the other hand, practical experience is required to get a consistent idea on what reconfiguration has to provide (and, sometimes, what is superfluous – this might be even more important).

A formal definition that carries over to practical implementations, however, requires a severe limitation of features. While we feel that the reconfiguration-enabled framework detailed in this thesis is fairly powerful, it lacks many features that a production-grade framework needs. Some of these features are too complicated or inconvenient for formal description. Others have not been added because their utility only became apparent at a later point in time, or because they can be substituted by the means already present. Overall, the framework is highly experimental, and its sole purpose should be seen in exploring the power of reconfiguration. Still, we hope that it offers interesting insights into this field of software engineering and contributes to a general understanding of the utility of reconfiguration.

1.5. Notational Conventions

In the context of this thesis, we use some notational conventions. Most of them are introduced on-the-fly, but some are so fundamental that we will introduce them here:

For mathematical functions, we write $\rightarrow$ to indicate a function where no implicit property is assumed (e.g., whether it is partial, injective, etc.). For partial functions, we write $\rightharpoonup$ if the function being partial is critical to its understanding (e.g., for

an infinite set C of component identifiers, the function that assigns a finite subset of C a value from the state set $\mathcal{S}$ (and hence describes the active components with their associated state), we write $C \rightharpoonup \mathcal{S}$). For a partial function $f : A \rightharpoonup B$, we name the subset of A for which f is defined the *domain* of f and denote it by $\operatorname{dom}(f) = \{a \in A \mid \exists b \in B. f(a) = b\}$. B is called the *co-domain*, and the subset $\operatorname{ran}(f) = \{b \in B \mid \exists a \in A \,.\, f(a) = b\}$ the *range*.

We write $f[x \mapsto y]$ for the function

$$\lambda z. \begin{cases} f(z), & \text{if } z \neq x, \\ y, & \text{if } z = x. \end{cases}$$

Often, we will explicitly define functions with a (very) finite domain, by writing them as $\{v_1 \mapsto v_1', \ldots, v_n \mapsto v_n'\}$, describing a (partial) function

$$f(x) = \begin{cases} v_1', & \text{if } x = v_1, \\ \vdots \\ v_n', & \text{if } x = v_n. \end{cases}$$

For sequences, i.e., finite words over a given alphabet Σ, we write :: for concatenation (defined for all combinations of elements $e \in \Sigma$ and sequences $s \in \Sigma^*$) and ε for the empty sequence. We write $\langle a, b, c\rangle$ for the sequence consisting of the elements a, b and c, hence $\langle\rangle = \varepsilon$.

As for names, we will use SMALL CAPITALS to indicate that a name describes a software (product), a programming language, or an algorithm with an artificial name. So we will write about JAVA, CORBA and the CMC model checker, and about LTL and Bloom filters. We retain the capitalization chosen by the authors if convenient.

1.6. Structure of this Thesis

In the next chapter, we will refine the idea of using components and runtime reconfiguration to implement adaptivity a little more. We will discuss the various definitions of components, and how their utility is described in the literature. In Chapter 3, we discuss related work; we discuss the major component frameworks, but the major focus is placed on component frameworks capable of reconfiguration, as this is the central interest of this thesis.

We then build our own approach in a number of successive chapters: Chapter 4 presents the formal background that we use to describe our component model with, as well as tools to investigate the dynamic behavior of applications. We then present the CMC model checker with its dedicated component framework in Chapter 5. The CMC model checker has been an interesting influence, and although it does not directly contribute to the main approach of this thesis, we feel it offers a number of interesting insights, and many design decisions done later can be related to the specific properties of this application.

In Chapter 6, we introduce the JCOMP component model, which we use to investigate reconfiguration with. In this chapter, we describe how the component model was designed and how reconfiguration is integrated. This chapter extends the results published in [HK08]. We prove some valuable properties for the reconfiguration approach, which is based on an implementation-related, small-step semantics. We proceed to discuss how this approach can be implemented in JAVA, while retaining the properties shown, in Chapter 7.

In Chapter 8, we give a number of examples for reconfiguration with the JCOMP framework. We first introduce the so-called MAPE loop as a uniform framework

for realizing reconfiguration, and show how different applications of reconfiguration can be realized.

Chapter 9 discusses the specification of components, and how these specifications can be verified. This is extended to reconfiguration, discussing ways to ensure the validity of a reconfiguration execution. We conclude this thesis in Chapter 10.

CHAPTER 2

Components and Reconfiguration

*The modules we constructed were made to exhibit
a black-box characteristic, so that their
internal idiosyncrasies could be safely ignored.*

— Tom DeMarco and Timothy Lister, Peopleware

We have chosen components as the basis for our approach towards adaptivity, using the reconfiguration of component systems. It is quite challenging to find a suitable definition from the wealth of different views on components. We will try to give a general idea of components in Sect. 2.1, and refine our definition by pointing out differences to similar approaches in Sect. 2.2. We will then fix the relevant terms a little more in Sect. 2.3, and discuss why components are useful in our context in Sect. 2.4.

2.1. What are Components?

The idea of "software components" has been among the first proposals towards structuring software. The term was first used in the key-note talk by Douglas McIlroy at the famous NATO conference on software engineering in Garmisch [**McI69**]. McIlroy believed that using software components in a way similar to using hardware components in complex machines might help overcome the so-called "software crisis". While his example – a sinus-function component that can be sold in varying degrees of accuracy and efficiency – has certainly been too fine-grained, many commercially successful approaches have been built around the term of components, most notable the various frameworks by Microsoft such as COM [**Box98**] or ActiveX, and industry standards like Corba [**OMG06a**]. The idea of selling components instead of whole shrink-wrapped software products or programming services has ever since attracted programmers (e.g., [**Cox90**]); currently, it experiences a renaissance in the form of web services.

The idea of what a "component" actually describes is surprisingly consistent in industry and research, maybe because it capitalizes on the well-established understanding in the hardware industry. (Consider Fig. 2.1, which shows the major components NASA's Space Shuttle is comprised of. Most of the components are built by different contractors, thus making use of their special skills and experiences. The final assembly was conducted by Rockwell International [**Hep02**].) It is, however, quite vague with respect to the granularity addressed, and usually has to be seen in context. Some examples are:

- In the context of the Object Constraint Language (OCL) [**OMG06b**], a component is given by a set of classes grouped together, for example in a Java package. The term *component visibility* then refers to a visibility granted to all classes within the same package.
- In a cryptography context, a component refers to a replaceable algorithm like a cypher or a public key system [**LB03**].

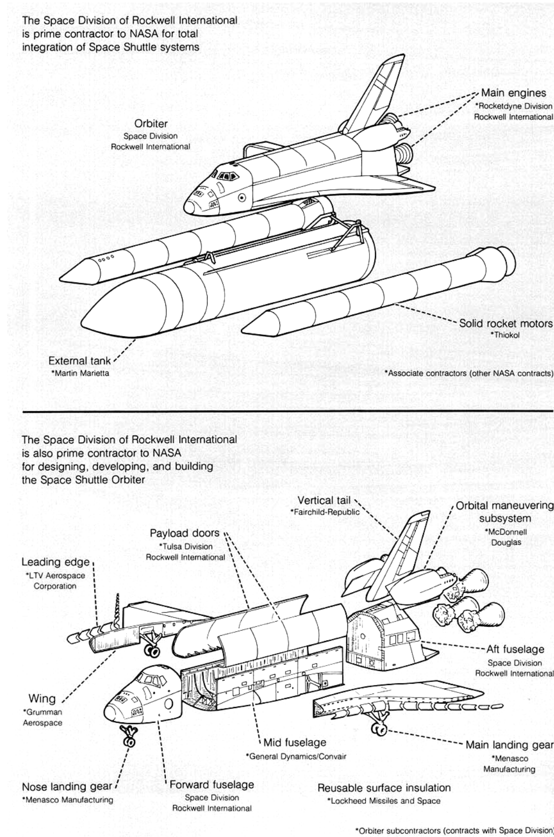


Figure 2.1: Components of the Space Shuttle, image courtesy of NASA

- In commercial component frameworks like Microsoft's ActiveX, components usually are complex libraries of code that provide a well-documented interface (but usually do not require one).
- Other frameworks, like Corba [**OMG06a**] use components in a sense similar to ours: Black boxes with explicitly declared provided and required interfaces.

The most established definition of a component in the context of component-based software engineering, i.e., a discipline of software engineering that perceives components to be the main ingredient of building software systems, is due to Clemens Szyperski. In [**Szy98**, pp. 30], he writes:

> *The characteristic properties of components are:*
> - *A component is a unit of independent deployment.*
> - *A component is a unit of third-party composition.*
> - *A component has no persistent state.*

The latter point then was changed to read "[A component] has no (externally) observable state" in [**SGM02**, pp. 36]. Basically, this is just a rephrasing: In both definitions, a component instance must not be distinguishable from another instance of the same type by means of communication. Szyperski notes exceptions, like attributes that do not influence functionality, or internal caches. Nevertheless, he concludes that any component only needs to be instantiated once, as having a copy does not make sense.

We differ from Szyperski in the statelessness of components; instead, we consider components to be inherently stateful. This is due to a different view on the granularity of components: Szyperski exhibits a rather technical view, where a "database" can be a component (although this example does not seem to work well with not having a persistent state), whereas we see components as parts of a system, which usually consists of nothing but components – hence requiring components to store state. However, Szyperski proceeds to provide a more generic notion of components [**Szy98**, pp. 34] that resulted from a discussion at a dedicated workshop [**Mue97**]:

> *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.*

This is a widely accepted definition (cf. [**Hop00**, **Völ03**, **GG03**, **GA04**], which are paper focused on the definition of components), but there is a broad range of differing refinements of this definition, cf. the discussion in [**BDH+98**]. Also, the self-containment property (*"can be deployed independently"*) is a little stronger than we envision it here – since we will place strong emphasis on required interfaces, and the satisfaction of the connection requirements at system start-up time.

One further aspect, namely the one of a component model, is given by George Heineman and William Councill [**CH01**]:

> *A* software component *is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.*
>
> *A* component model *defines specific interaction and composition standards. A* component model implementation *is the dedicated set of executable software elements required to support the execution of components that conform to the model.*

This accentuation on the definition of interaction and composition standards is unique to this definition; although no other definition directly disagrees. In this thesis we will see that making the component model an integral part of the component definition is very much desirable.

Another interesting definition – obtained by listing the key properties – is provided by Bertrand Meyer. In [**Mey00**], he lists seven points that constitute a component:

(1) *May be used by other software elements (clients).*
(2) *May be used by clients without the intervention of the component's developers.*
(3) *Includes a specification of all dependencies (hardware and software platform, versions, other components).*
(4) *Includes a precise specification of the functionalities it offers.*
(5) *Is usable on the sole basis of that specification.*
(6) *Is composable with other components.*
(7) *Can be integrated into a system quickly and smoothly.*

Compared to Szyperski's second definition, the explicit catering of a client and the inclusion of specification is noteworthy. The latter is quite understandable given Meyer's research background in contract-based software engineering [**Mey97**, **NM06**]. The consideration of a client addresses an interesting point: Are components to be used by clients, or just by other components? The *use* of

components is not described in Szyperski's definitions. Both definitions also (presumably deliberately) lack a specification of *granularity*: is there any limit to the functionality a single component might provide?

We will not exactly give a concise definition of our understanding of components here, but answer those two questions for this thesis: We will consider components to be used by other components only. If there is something as a client, it has to be packaged to become a component itself. This results in a different viewpoint on software development: In Meyer's definition, a component is used to provide some *service*, some functionality that a client needs. In our definition, components are assembled to collaborate on a common computation, and all required code will be provided in the form of components[1]. Also, we provide an understanding of granularity, albeit a very informal one: A (non-composite) component should group the functionality that cannot be broken down in such a way that some subfunctionality might be replaced by some other implementation profitably.

This is rather vague, but in the course of this thesis, the idea should become clear. An example often used is a hash table, i.e., a set implementation that calculates the hash value of an object and puts it into an appropriate bucket. Such a hash table could be used as a component – but it would violate the definition, as it will work with any hash function. It is *profitable* to externalize the hash function, since the proper choice of the actual implementation might depend on the data at hand (a cheap hash function might work fine for objects that are cheap to compare, but if collisions are expensive, a more elaborate hash function might pay off). So part of the functionality – the hash function – needs to be externalized to a second component. Of course, in other situations it might be judged to be unprofitable to externalize the hash function, and it might become integrated in the hash table component. Ultimately, such a decision needs to be made in the context of the application that the component is written for. Nevertheless, a hash table component that uses case distinction to choose from a number of hash functions (maybe at compile time with preprocessor commands) violates our definition of a component for sure.

2.2. What are No Components?

"Autistic computing"
— some merry people [2]

As the term "component" is a fairly general one, it might be useful to refine its meaning within this thesis a little more by pointing out the differences to other, well-established concepts of software engineering. This investigates the idea of components by finding out what is missing or not accurate enough in other software engineering technologies.

2.2.1. Objects. Object-orientation has been one of the major advances in software engineering [**Mey97**]. On a technical level, the concept is closely related to components, and the concepts can be expected to have inspired each other. (We will later elaborate on the use of object-oriented languages, i.e., C++ and JAVA, as host languages utilized to actually implement a component framework.) There are minor technical differences, like inheritance, which is usually not considered for components (although the CMC model checker employs inheritance concepts, albeit

[1]There will be some exceptions, mostly related to reconfiguration, where further code is required to steer it.

[2]Andreas Schroeder, Carolyn Talcott and me; trying to imagine what the opposite of service-oriented computing might be.

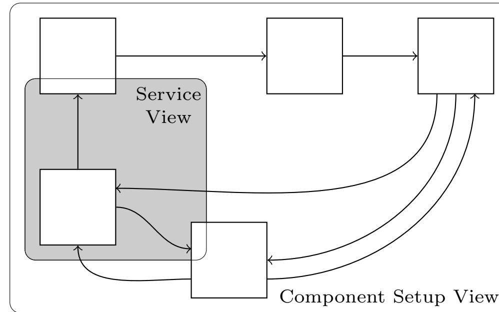


Figure 2.2: Extend of views for a component setup and services

just for avoidance of tedious redefinitions; see Sect. 5.2), or hierarchical components, which have no language level correspondence in object-oriented languages (but which can be easily provided on the implementation level by the Composite pattern [**GHJV95**]).

The key difference of object-orientation is the lack of a differentiation of "control structures" and "data structures", which we desire for component systems. In a JAVA-based web application, the central controller is defined in a class, as is the tiniest data-storing bean. It is perfectly feasible to pass a reference to an instance of that controller class to other objects, so it becomes a data object itself. This is where components are requested to distinguish more strictly: Either some code is concerned with processing data, managing calculations etc., in which case it is part of a component, or it defines data structures (possibly with some elaborated accessor functions), in which case it is a data object.

Of course, it takes little effort to just follow some simple rules and not pass a reference to the main controller object to other objects. Then, object orientation and component-based software quickly become identical, and in fact it takes little overhead to enrich an object-oriented language to become "component-based" – the programming language (we use C++ and JAVA in this thesis) then becomes a *host language* for the component framework. The important point is that the distinction of control-flow-managing components and data-encapsulating objects becomes more visible; and even more enforced – which helps to obtain some invariants that cannot be guaranteed in "fully-enabled" object-oriented languages.

However, components retain one of the most successful features of objects: Clear borders, and explicit communication. As Kent Beck writes in [**Bec00**, pp. 24]:

> *Message sending is a powerful way to cheaply build many opportunities for change. Each message becomes a potential point for future modification, a modification that can take place without touching the existing code.*

Components make communication even more explicit; this amplifies the benefits Beck mentions.

2.2.2. Services. Being one of the major research topics in software engineering, services have become a paradigm that offers both a new revenue model as well as interesting research opportunities.

As mentioned before, the difference between components and services is small; if services are considered as black-boxes that need to be connected, the term "component" is describing the entities involved in just about the sense we are using it in

this thesis. Actually, components offer functionality to other components, and this is often referred to as provided *services*. If a difference is to be made, it can be found here: Components also *require* functionality to be provided by other components, which is not discussed for services.

Hence, the view on services is shallow and one-way only, as illustrated in Fig. 2.2. This means that the user of a service – i.e., the programmer that writes an application that queries a service – should only be aware of a single layer of services. The internal processing of a query – which, of course, might involve further queries to other services – should be opaque to the caller. In any case, the call should not result in a call back to the caller – the call-graph should not contain circles.

For components as we understand them here, exactly the opposite is true: They are assembled by a central instance that understands the whole of the application. While services can be viewed like this as well (it is not strictly required to remain ignorant of the called service's behavior), services are not supposed to require application-wide restrictions of communication, like requiring circular data flow, i.e., the called components sends data back to the caller on a different route than the method return. While this situation seems artificial at first, we will encounter it in some rather straightforward implementations like the CMC model checker in Sect. 5.3.

Much of the research advances associated with services have impact on component-based software engineering as well; but often, the focus is slightly, yet articulately different. This especially holds true for *service discovery*, a process of detecting and choosing suitable services for a given purpose [**LMH07**]. As discussed before, this might also be used for detecting components that provide the services a given component requires; but these components might require further services on their own, some of which might – or even must – be provided by existing components, and others that need further components to be discovered and instantiated. Components require that an overall architecture is adhered to, which needs to be explicitly described [**GS93**]. As a result, the automated *configuration* of components is a good deal more complicated, and rarely attempted (cf. Sect. 8.1.4), whereas service discovery is a matured field of research [**ZBBF07**].

2.2.3. Agents. Among the concepts presented in this section, agents are undoubtedly least likely to be called components in our sense. It is still interesting to stress the difference, which we perceive to be the absence of autonomy of components.

Autonomy is one of the key concepts of software agents [**Fon93**]:

> *"A more autonomous agent can pursue agenda independently of its user. This requires aspects of periodic action, spontaneous execution, and initiative, in that the agent must be able to take preemptive or independent actions that will eventually benefit the user."*

The significant difference to components is given by words like *spontaneous* and *preemptive*, which require the agent to be capable of self-triggered actions; components on the other hand are required to respond to external messages sent over the interfaces. Everything else is subject to the black box metaphor – while we certainly do not want to prohibit components from spontaneously sending messages, we also cannot enforce it to be a key feature. Therefore, agents impose a more restricted definition than generic software components; but there is evidence that components as described here are a very good way to implement agents [**KMW03**].

For the purpose of this thesis, agents are interesting because they often require *mobility*, i.e., the capability to change the computer they use for execution at runtime. This requires *migration* [**IKKW01, LS05**], a special type of reconfiguration (cf. Sect. 8.4).

2.2.4. Modules. Interestingly, modules – i.e., groupings of functions, data types, classes etc., as provided by a programming language – are sometimes called components, e.g., in the OCL specification [**OMG06b**], where packages (in the sense of the JAVA programming language; such packages constitute what we mean by modules here) provide the scope for *component invariants* [**HBKW01**].

Of course, the altogether unspecified extend of a JAVA package (i.e., the lack of a criteria which classes should be put in one package) makes them more generic than the component definition used here. Furthermore, a module cannot be instantiated. It describes a static code library, an assembly of structures that serve some purpose together. While parts of a module can be instantiated – and might also be called components in our sense – an entire module does not form a stateful entity at runtime, which is exactly what we consider components to be.

2.3. Key Terms and Concepts

Let us briefly introduce the central terms in an informal way; the formal definition will be given in Chapter 4.

A *component* describes an entity that performs calculation, in the broadest sense of the word. Usually, a component is an instance of some *component type*. Depending on the actual component model, a component may have *ports* or *interfaces*; in any case, the component has some *connections* to other components over which communication is conducted. A component usually *provides* services to the component it is connected to, and also *requires* services to be provided by connected components. The components and their connections form a *component graph*, also referred to as a *component setup* or a *configuration*. If the connections bear some delicate semantics or have a state of their own, they are represented by *connectors*, which can be made first-class entities of a component model [**LEW05**]. A *component model* describes how components are to be executed. A component *framework* is a software that implements a component model.

Usually, components are considered to be *black boxes*, i.e., they are entities whose inner operations cannot (or rather do not have to) be inspected (at least at runtime), or as *gray boxes* [**BW97, dB00**], where some aspects of the inner structure (of the component type) are known, but not accessible at runtime (e.g., by means of inspection). Contrary, the inter-component communication is usually regarded as observable. In this thesis, we consider components to be *stateful*. Following the black box view, this state cannot be directly observed from the outside, but it influences the communication behavior of the component. Likewise, communication received by the component changes its state. Depending on the component model, state changes might also occur spontaneously.

Another ingredient is the *assembly*, which is some kind of startup code that instantiates and connects components and maybe manages runtime *reconfiguration* of the component graph. Developing components and assemblies is a discipline called *component-based software engineering*.

As an example, we might consider a component type that implements the hash table mentioned before[3]. This component type provides a service that can be invoked to add an element and to query it, and requires a service, to be provided by a connected component, that calculates the hash code for a given element. A component can be created that instantiates this component type, and every such instance needs to be connected to a hash function component in order to have its requirements satisfied (which might be the same component for all hash table instantiations, or a different one); this is done in the assembly code that builds the

[3] Stemming from the model checking experience, we use the word hash table for a set implementation; in JAVA this would be called a hash set.

component graph. The component model will then describe how the communication is conducted; for example, it might mandate that the query for existence of a certain element is to be done by using the host programming language's means of method invocation. At runtime, we will either consider the hash table component as a black box, knowing nothing about its internals, but being aware of its external description, connections and communication behavior, or as a gray box, in which case we might have the source code of the component available for inspection, but no means to read internal data at runtime.

The state of the component is comprised of the contents of the hash table, combined with structures that are defined by the component model – e.g., message queues. The term "stateful" usually refers to the existence of the former state part, which we refer to as the *data state*. The contents of the hash table influence the communication: The response to an element query will depend on whether the hash table already stores this element.

A word on the use of the term "component" itself: Usually, a proper distinction between "component" and "component type" does not take place (cf. the discussion in [**KBHR08**]). Instead, both are called "component" and the actual meaning can (hopefully) be derived from the context. Usually, there is not much difference: A component type is defined to become instantiated, and in many settings, it is instantiated only once. In this thesis, we will not consider component types in the formal parts – everything is embedded in the component definition. In the informal parts, we will be a bit more vague and talk about the "hash table component" that is connected to the "hash function component", although these names both describe component types which are instantiated once and connected in every component graph we are currently considering.

2.4. Why Use Components?

2.4.1. Software Reuse. McIlroy introduced components as a means to reuse implementations in the way that hardware is reused in product line assemblies [**McI69**]. Components are considered to be independently developed and marketed as suitable processors of sub-tasks of complex applications. These "COTS" (commercial-of-the-shelf) components are still perceived to be one of the major benefits of component technology (cf. [**Szy98**]). There is, however, a serious drawback to overcome: Most of the time, components need to provide a functionality more complex than originally discussed by McIlroy (who discussed sine function implementations). For example, components might provide functionality for image processing. Finding a component that does exactly what is required can be so difficult that programmers seem to favor rewriting the code, an observation that sparked the AMPHION project of NASA [**LPPU94**]:

> *However, even when a subroutine library is developed following the best conventional software engineering practices, users often have neither the time nor the inclination to fully familiarize themselves with it. The result is that most users lack the expertise to properly identify and assemble the routines appropriate to their problems. This represents an inherent knowledge barrier that lowers the utility of even the best-engineered software libraries: The effort to acquire the knowledge to effectively use a subroutine library is often perceived as being more than the effort to develop the code from scratch.*

AMPHION utilizes the SNARK theorem prover [**SWL+94**] to find and assemble appropriate components – requiring a specification of what is to be processed on a physical level. The few research that investigates the utility of code reuse tends

to draw a similar picture: While reuse seems to offer the possibility of adding functionality for free, in practice much of it is consumed by the requirement of understanding the specification and ironing out the minor (maybe undocumented) discrepancies that emerge [**Tra88**, **Tra94**, **Fav91**, **GAO95**]. This has led some people to claim that component reuse is not profitable, stating that a truly black-box, reusable component takes 3-5 times the development effort a "good enough component" needs [**Lam04**][4].

In this thesis, we will not discuss software reuse much. First of all, the research we report on was done on projects too small to make credible statements about the utility of components for software reuse. Instead, we understand component-based software engineering as a different approach towards writing software, which enhances the clarity of a software system (which, in the end, is a necessary prerequisite for reconfiguration). If we want to specify the behavior of components, this will be done to reason about the correctness of an application, and not to identify components that provide a given functionality – as would be required for reuse considerations.

The impact of this choice is greater than it might first appear: When specifying for correctness, the context is narrowed down significantly compared to a specification for functionality. If we want to specify an image processing component such that its utilization in a component application is verified for safety properties, we might restrict ourselves to specifying "sane communication behavior" if this suffices for the correctness properties (e.g., stating that the result of a method call to `findHumanFace` is either a coordinate within the boundaries of the picture or `null`). If we want to specify this image processing component for reuse, we would have to fully describe the algorithm it implements. Obviously, the difference can be huge.

2.4.2. Algorithm Substitution. McIlroy also provides another reason for using components. His example of a sine function also discusses that these functions might be provided in varying degrees of efficiency and accuracy. Obviously, this is done to further stress the metaphor of hardware components (where simple things like screws are available in immense ranges of quality). However, it also implies an interesting view on components: Rather than writing a single sine function and passing the required accuracy as a parameter, a number of sine functions with preset accuracy are provided as components. The user then takes the component with the appropriate accuracy and uses it as part of the component application.

We believe that this approach carries over to component-based development even though only one person is involved. Although we write all the sine functions ourselves, there is a conceptual difference between choosing a sine function component and passing an accuracy parameter: The time the decision is made. When passing a parameter, this time is not defined. The calling method might supply a fixed parameter, or the accuracy might be given by the user on the command line and passed through the entire application. With components, however, the choice of accuracy is invariably postponed until the components are assembled.

This effects a changed view on the application. The choice of a different sine function component produces a different application. We might choose a low-quality

[4]The author does not feel comfortable with this generalization. There are well-established components that see frequent reuse, in domains like cryptography or when being shipped with a programming language (note that [**Lam04**] lists these exceptions as well). But there are also some domains where components are sold with considerable profit, e.g., as plug-ins for media editing software. It is most likely true that software can never be assembled from pre-existing components alone, but that does not void the general utility of reusable components (cf. the experience with CMC, reported in Sect. 5.5).

function for development, and a high-quality, mind-numbingly slow one for the final production version. We might even select an especially bad performing one to check out how errors propagate. And, with reconfiguration, we might add a monitor that, from time to time, checks if the sine function currently in use is appropriate, and might reconfigure the application to use one with better accuracy if this is deemed beneficial.

All this is done at assembly time. We do not even have to know that we will utilize different sine functions when first writing the components. This is what we understand as the separation of concerns – when writing a single sine function component, we do not have to care whether it is going to be replaced by a better performing one – whereas, with an added accuracy parameter, we have to deal with this usage scenario right away.

2.4.3. Programming-in-the-large. Programming-in-the-large, a term coined by Frank DeRemer and Hans Kohn [DK75] describes a discipline of programming that is concerned with wiring components on a high level, as opposed to programming-in-the-small, which is about actually implementing algorithms. Programming-in-the-large emphasizes or even enforces modularity of programs, reaping many benefits like the possibility to do "proving-in-the-small" to establish the correctness of components with respect to their specification, and to proceed to do "proving-in-the-large" by combining components whose abstract behavior description is then known.

Writing software in modular fashion has always been encouraged (cf. [Par72]), but modularity can be achieved for a number of different aspects of software. For example, the source code of an application written in C can be distributed over multiple files, or an application can utilize a library that is linked after the compilation process or even just loaded at runtime. The software can also be written to run distributed on a number of computers. Programming-in-the-large describes a different modularity: That of dividing functionality into a series of sub-functionalities that are assembled on a different level.

This kind of programming encourages a different perspective on the programming task at hand. Rather than asking "what needs to be added to get the program running" the question becomes "what needs to be assembled" – an approach that is less iterative, but also with an articulately succinct goal. With the experience obtained throughout this thesis, this kind of consideration supports a different approach towards analyzing the software problem at hand, as it puts more emphasis on the flow of data, which provides a different mental image as traditional "operate on a given data structure" programming does.

For some problems, this might be very beneficial. It is most likely not by coincidence that the best-known programming languages that implement the programming-in-the-large paradigm are business process modelling languages like the "business process execution language" BPEL [AAA+06] or YAWL [AH03], a language based on workflow patterns [AHKB03]. Workflow patterns describe repetitive structures in business processes, which are descriptive for the aforementioned class of problems where programming-in-the-large excels. Problems that can be thought about as workflows – where multiple steps are executed in a sequential or parallel fashion, each doing some work on data that is transported along the workflow – seem especially suitable for component-based programming. Within this thesis, a number of examples will be given for problems that are best viewed from this perspective. In fact, almost all systems that do not require frequent user interaction (and this exclusion is only necessary because this is outside the scope

of this thesis – it may well be that GUI applications also work well with dataflow considerations) can be viewed as workflow systems, but sometimes, efficiency constraints will rule out this approach.

A slightly different view at the programming-in-the-large paradigm is given by the notion of "architectural programming" [**ACN02a, BHH+06**]. Here, the high-level assembly of components (which is still regarded as programming) is emphasized. By programming assemblies at a coarseness level that admits immediate understanding, the code becomes its own model, suitable for describing the application. This might remedy the problem of *architectural erosion* [**PW92**], which describes how a model – used to describe how a software is implemented – often is not updated when the software is extended and modified, leading to an ever-increasing gap between the source and the model that is supposed to describe it. *Architectural description languages* (ADLs) aim at supporting such programming, examples being WRIGHT [**All97**] and DARWIN [**MDK93**] (cf. Sect. 3.1.3). While this thesis lacks examples that are long-termed and large enough to truly illustrate the benefits of this programming approach (although the CMC model checker described in Chapter 5 draws near), it is encouraging to see the general idea of component-based software engineering supported by such a multitude of approaches.

2.4.4. Separation of Roles. Programming-in-the-large also promotes a software development process that involves at least two roles with distinct concerns. One of these roles is concerned with writing components, the other one with plugging them together "in-the-large". Such a differentiation is often encountered when coarse-level architectural programming languages are employed (e.g., DURRA [**Wei89, BDWW89**]). We will refer to the first role as the *component implementer* or *programmer* or *designer*, and to the second role as the *system designer*. Discerning these roles is an application of the separation of concerns paradigm to software engineering [**CBG+04**]: it is the concern of the component implementer to write components that offer some functionality and are optimized in one way or another, and the concern of the system designer to choose appropriate components and integrate them to a correct and efficient application. In the context of this thesis, the difference between the two roles is very important, since it is the primary motivation to do reconfiguration. We will thus mention this *separation of roles* frequently.

It is not always possible to distinguish between the roles accurately. In hierarchical component models (e.g., SOFA [**BHP06**] or JAVA/A [**Hac04**]), the system designer of a composite component acts as a component implementer in the next hierarchy level. Also, pretty often the roles tend to mix; e.g., if the system designer briefly takes the role of a component designer in order to write an adapter component that is required to link two slightly incompatible components. And obviously, roles can also be filled by teams of people.

The best separation of roles is obtained if the component designer is unaware of the system designer. In scenarios like the one envisioned by McIlroy [**McI69**], a programmer employed by a component vendor writes a whole series of components without targeting a specific application. On the contrary, a wide range of nonfunctional properties is covered in order to give the system designer more choices. Often, the separation of roles is made more explicit by the maintenance of a component repository [**Ye01**]. The concern of the component designer is to add components to this repository, while the system designer queries the repository and uses the components found.

Such a strong separation, however, is outside the scope of this thesis, as we are more interested in building software more or less from the scratch; separating the roles is done for obtaining a clean, robust software architecture rather than

to emphasize component reuse. We argue that using component-based software development is beneficial even if the software is written by a single person only – as most of the examples in this thesis are. The roles involved are thus not to be understood as distinct *persons*, but rather at distinct views on an application (or, as separated concerns of application development).

2.4.5. Reconfiguration. We have so far described how component setups are configured by programming-in-the-large. Configuration is a process that forgoes the launch of the component application. Now, our focus shifts to *dynamic reconfiguration*[5], which describes the process of altering the component setup at runtime. By employing reconfiguration, we can accommodate to behavior that we have not foreseen or did not want to consider when building the initial application. Extending McIlroy's sine function example, reconfiguration might be utilized to switch to a component providing a more suitable sine function, if the former function no longer suffices.

Reconfiguration is a way to achieve adaptivity. It might be arguable if the granularity of components is suitable, but reconfiguration of components offers a number of benefits: First, since configuration is a fairly straightforward concept (discussing component instances, their inter-connections, and possibly their parameters), a reconfiguration plan is also quite straightforward – in essence, it just describes the transition of one configuration to another. Second, reconfiguration can be facilitated by a component framework, which can take care of some of the problems that are given with adaptivity. Reconfiguration is thus a generic approach for building adaptive systems, and because of the clean separation of components, much less troublesome to understand and use than related concepts like self-modifying code [**TY05**].

For pointing out the uses of reconfiguration, let us briefly discuss how this concept is discussed in the literature:

2.4.5.1. *Reconfiguration for Hot Code Update.* Not all reconfiguration is done to provide adaptivity. Early research of component reconfiguration (e.g. [**Blo83, Lim93, BISZ98, TMMS01**]) focused on the problem of hot code deploy, i.e., the updating of the implementation of a running application, and interest has not ceased ever since. For large, distributed systems, the merit is obvious: A shutdown of the entire application, stretching over a number of machines, with lots of components possibly involved in complicated transactions is tedious and also quite prone to problems with partially completed transactions which result in corrupted component states. Instead, the system is only stopped partially where this is unavoidable, and the remainder of the system is allowed to keep running.

Using reconfiguration for component update provides another interesting aspect: Since the old component is replaced rather than modified internally, it can be kept for a little while to see if the new version behaves well [**SRG96**]. If the new version has errors, a further reconfiguration can switch back to the old version, thus providing some fault-tolerance for component updates. Investigating such solutions is what we are interested in in this thesis, independently of hot code update.

2.4.5.2. *Reconfiguration for Architectural Change.* Being able to perform hot code updates requires an infrastructure that usually allows for a different application as well: Instead of updating a component with an improved behavior, the component might be replaced by an altogether different implementation that exhibits a different behavior (cf. [**Med96**]). For example, a component that loads data from

[5]We will use the term *reconfiguration* to describe *dynamic* or *runtime reconfiguration*, unless explicitly noted otherwise throughout this thesis.

a file might become replaced by a component that loads the data from some network source. Going one step further, the application's architecture might become changed beyond a mere replacing of components: Components might be added, removed, and connected differently (the latter being a process often referred to as a "rewiring"). A filter component might be added to an application to further process the communication between two components, or, in the example above, a forking component might be introduced that replays communication to both the new and the old version of a component, judging their response and triggering reconfiguration to either remove a faulty new version or eventually dispose the superfluous old one.

The programming-in-the-large paradigm suggests that explicitly defining an application's architecture can be beneficial in some interesting contexts. Reasoning about architectural reconfiguration on the same, abstract level takes this argument a bit further: Instead of just defining the flow of information statically, possible changes are also considered. The benefits are twofold: First, flexibility is enhanced, since not all possible scenarios the application is subjected to need to be considered when the configuration is devised. This argument is tempting, but a little problematic, since the reconfiguration needs to be envisioned all the same, and be prepared in some way. But second, the separation of roles is amplified by dealing with changed situations on the architectural level rather than on the level of components. If the aforementioned component reading data from a disk is to be made robust against disk failure, it needs to have provisions for obtaining data from the network programmed into its structure. If, on the other hand, the application is to be made robust, reconfiguration can be used to substitute the component reading from the disk by the component utilizing the network. Both the components are written without their actual usage in mind (especially, the component using the network can be employed in a different scenario where it is not a substitute, but a prime choice for obtaining the data). Thus, the load of providing adaptivity is taken from the component implementer and given to the system designer, who might then be able to provide solutions with less effort, since they are described on a level more coarse.

Still, the benefit of reconfiguration needs to be compared to its costs. Components imply a certain level of granularity, which we have described informally as being "not to be divided profitably". This might not be sufficient for a fine-grained adaptivity. And reconfiguration does not come for free: It requires the same planning effort as a custom-written adaptivity requires, it just handles the concern on a different level.

The ability to modify an application's structure at runtime is a special case of *self-adaptivity*. In [**RLS01**], an interesting argument is proposed: It is easier to write a software that responds by adaptively modifying itself to unforeseen problems with its environment than to write a software that foresees every possible situation and change. This argument, if not further refined, is difficult to accept: First, a genuinely unforeseen problem cannot be dealt with by the software, no matter how capable of self-modification – it has to fit some kind of description by the software in order to be treatable (of course, it might be possible to have a software that actually understands *every* possible problem, but then no problem is genuinely unforeseen) – and second, writing self-adaptive software (e.g., self-modifying code) is so tedious and error-prone that a large number of situations can be foreseen when writing the initial code and have a treatment implemented before an adaptive approach will start to pay off. But if the argument is refined by the consideration of a separation of concerns regarding the normal function and adaptivity scenarios, the benefits we might hope to gain from employing reconfiguration become obvious.

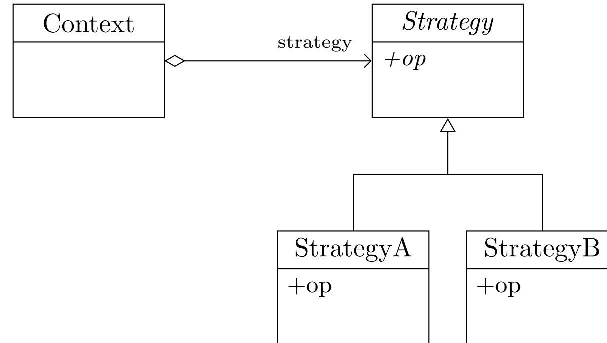


Figure 2.3: Strategy pattern

Reconfiguration hence addresses applications where a number of factors are given:

- There must be foreseeable problems that can be accounted for by a restructuring of the application (e.g., using a different data source if the physical medium fails).
- These problems should require a strategy that transgresses the responsibility of a single component (i.e., the problem should not be foreseeable and treatable within a single component; a component that reads user input should take some precautions against malformed input on its own).
- The problems should be so infrequent that building a special architecture that handles the problems right away should not be required.
- The application itself should require continued operation (e.g., a simple shutdown and restart with a new configuration should not be feasible).

The latter point is not undisputed: Reconfiguration can also describe the process of altering a configuration before conducting a restart [**ALF00**]. But in this thesis, we are mostly concerned with applications that are to become subjected to reconfiguration while they are running.

So far, we have described reconfiguration to be a cure for problems. This is somewhat unfounded, since an anticipated change of the situation might also be deemed as a welcome or a, at least, normal process. Reconfiguration is a general tool to respond to the perceived change. This is best illustrated with the *Strategy pattern* [**GHJV95**].

The Strategy pattern (Fig. 2.3) uses subclasses of a `Strategy` class to provide different implementations of a task. The client obtains an instance of the subclass that provides the functionality that is required. If the requirement changes, a different subclass is instantiated and used. For example, the semantics of a mouse-click in a painting program will differ according to the currently selected painting tool. Hence, we have a `PointMouseClickStrategy` and a `TextMouseClickStrategy`, with the former containing code to draw a point and the latter code to write some text. Depending on which tool is currently selected, one of the strategies is allocated and made available to a mouse click handler. When a mouse click needs to be processed, the current strategy is asked to perform the job, which it does according to its particular definition.

The strategy pattern separates the concern of implementing different strategies from the choice of the suitable strategy. If a case distinction was to be used, both the choice of the strategy and its implementation would be performed at the same

place. By separating these two things, the application gets a cleaner structure; this supports extensibility.

For reconfiguration of component-based systems, a similar separation is envisioned. Using components can be helpful to enhance an application by subsequently replacing components by improved versions, without the need to tamper with the code of other components. With reconfiguration, this benefit is even amplified: Writing adaptive components (i.e., components that are inherently adaptive, maybe by using dynamic code substitution) is hard, but writing two components and have one component substitute the other by the framework is easy, if the framework provides sufficient control over that process.

This comes at the price of being coarse: For practical applications like the CMC model checker described in Chapter 5, components can contain a few hundred lines of code, and the smallest model checker consists of altogether just four components. Replacing any of those four components amounts to a massive modification of the application, and finding such applications is not always easy. This is a remarkable problem: While component frameworks are very well suited for theoretical and practical considerations of reconfiguration because of the ideal granularity that components provide, components appear to be too coarse-grained for many practical applications. We will discuss this in greater detail later in Chapter 8. However, we will place great emphasis on the examples that are provided with the already quite numerous reconfiguration-enabled component frameworks, as good examples might be harder to come by than good component frameworks.

2.4.5.3. *Separation of Roles for Reconfiguration.* In Sect. 2.4.4, we have discussed the two roles involved in building a component application: The component implementer and the system designer. As reconfiguration operates on the system design level, it is reasonable to assume that the separation of roles is also given for reconfiguration: It is the concern of the system designer to devise the reconfiguration and see to the correct execution, whereas the component implementer remains mostly uninvolved. Just as the system designer assembles an application from existing components, reconfiguration modifies an application, possibly adding further (but nevertheless already existing) components.

It is sometimes convenient to introduce a further role, the reconfiguration designer. We will not discuss a distinction to the system designer here. It is nevertheless important to stress that the component designer is even less concerned with reconfiguration as with configuration: Possible reconfiguration scenarios should be completely ignored when devising a component. This is not always possible: In Sect. 6.8, we will see that in order to transfer the state from an old to a new component, some provisions need to be added to the component by the component implementer. It is, however, a major goal of this thesis to keep this necessary provisions as marginal as possible.

This is necessary to retain the major benefit of reconfiguration: To operate on a purely coarse level, as discussed in the last section. If a component implementer was to be tasked with providing numerous reconfiguration-related provisions (unless they are entirely generic), there is little perceivable difference to implementing the adaptivity in the component right away. On the other hand, if the component implementer can remain oblivious to possible reconfiguration scenarios, the task of writing a component is not aggravated by the adaptivity aspects of the application. Since we assume that implementing adaptivity directly into the components is a error-prone and tedious way, we will discuss the necessity to retain a strong repeatedly in this part.

2.5. The Difficulty of Reconfiguration

While reconfiguration is an easy concept, in realizing it we need to cope with different problems, some of which are severe enough to place the utility of reconfiguration in doubt. Given that we wish to utilize reconfiguration in large-scale, concurrent systems, and given that we wish to address infrequent, but demanding problems, reconfiguration can become arbitrarily complex if the framework does not provide sufficient guarantees and steering capabilities that help to control the reconfiguration and ensure the progression of the overall application.

Given the concurrency of the applications we wish to address, the interleaving of reconfiguration with the system's progression needs to be addressed: Reconfiguration takes some time, and the remainder of the system needs to be allowed to continue with as little restrictions as possible. For large-scale, distributed applications, we can hardly enforce a global lock during reconfiguration; instead, we need to provide provisions that allow reconfiguration to commence in parallel to ongoing system activities. These might include sending of messages to components that are just about to become replaced – and if an ongoing reconfiguration is not careful about these messages, some messages might become lost, making the outcome of the reconfiguration very unpredictable and prone to stalling the overall progress of the computation.

We have argued in Sect. 1.2 that realizing adaptivity is hard because it happens only infrequently, while being difficult and error-prone to implement. We hence require the support of the framework to obtain as much guarantees as possible. In this thesis, we will require the framework to guarantee that the reconfiguration is perceived as an atomic step by an outside observer, such that interleaving the reconfiguration process with normal system execution does not produce deadlocks or unanticipated behavior. This has to be achieved with only minimal stopping of components.

Of course, a system cannot be stopped even partially without some precautions, which poses the problem of maneuvering the system into a state where such a partial stop is safe to do. If reconfiguration just interrupts the system at an arbitrary point in time, components that are removed by the reconfiguration might still be in the middle of an ongoing communication. States where such communication is known not to exist have been called *quiescent* [**KM90**], *tranquil* [**VEBD07**] or *safe* [**Weg03**]. The problems associated with reaching such a state actually provide an argument *for* reconfiguration opposed to performing a system restart: It is conceivably easier to reach such a safe state for only a small number of components than it is to reach it for *all* components – which is required for a safe application shutdown. But waiting for quiescent states contradicts the demand for *immediate* reconfiguration – which is required for substituting failed components or coping with errors that will soon lead to an irrevocable failure of the application.

Another problem reconfiguration needs to solve (or, at least, discuss) is the problem of state transfer. As we are interested in stateful components, the reconfiguration process might be required to initialize a component not in the state that would have resulted from an initialization in an initial configuration, but in a state that is calculated from the state of other components. For example, if we replace a store component with a different version (cf. the example of Fig. 1.1), we will often be required to transport the data that has accumulated in the hash table to the new version, in order to make it a genuine substitute. We will later see that this is not hard to do technically; yet it requires some effort to fit consistently into the separation of roles paradigm.

State retainment is an issue that can hardly be avoided when considering hot code updates by reconfiguration, and often required for reconfiguration that changes

an application's behavior. Hot code update by reconfiguration is but one approach to modifying components with an updated implementation – programming languages like ERLANG or LISP support direct code replacement. Using reconfiguration, we build new components and replace the components to be updated by these new versions. Obviously, in order to have the new components act as suitable replacements, they need to copy the data state of the old versions. Such a duplication of a state may appear unnecessary if the state layout does not change, and the old component might just continue to operate on the previous data state. Things change drastically if the state needs to be modified because of the new component implementations using a different state structure: By doing an explicit transfer of the state during reconfiguration, the modifications can be applied in a well-defined way, which, if done in a certain fashion, does not require additional code on behalf of the components. State retainment is investigated in a number of papers [**GJ93**, **Van07**, **Pre07**, **RS07a**], with many of the approaches focusing on an automated approach of associating state elements of the old component version with elements of the new one. This is beyond the scope of this thesis; we expect state retainment to be supplied as an element of the reconfiguration plan. While this provides more flexibility than an automated approach, we still consider the planning of state retainment as one of the most difficult problems during planning a reconfiguration, which is sort of a position statement: Many other works does not consider state retainment at all (cf. Tab. 3.1 on page 41).

This thesis is concerned with two major questions: How to conduct reconfiguration in a way such that it becomes less troublesome to define it, and how to utilize such a reconfiguration for building advanced software. Surprisingly, in some aspects the latter question proved to be somewhat harder to answer.

2.5.1. Ingredients. Fig. 2.4 illustrates the basic ingredients that are required for reconfiguration, as we consider it in this thesis. They split in three large parts:

(1) The detection of a need for reconfiguration. For hot code update, this is not necessary, or rather, entirely provided by the user. Otherwise, the need for reconfiguration has to be found out by monitoring the application and its environment and analyzing the data obtained.
(2) The planning of reconfiguration. Again, this is fairly easy for hot code updates, and becomes challenging for generic reconfiguration. It consists of planning the reconfiguration (by stating which modifications need to be done) and scheduling it. Scheduling needs to provide a point in time when to commence reconfiguration, and, if a choice is provided by the framework, in what order to execute the atomic steps. For multimedia applications, this scheduling might need to consider realtime-constraints so that the reconfiguration can be executed in a way ensuring that the latency stays below some threshold, cf. [**MNCK99**]. In the JCOMP framework, however, the reconfiguration algorithm is fixed, and scheduling is only concerned with determining a suitable point in time.

    Obviously, planning is also connected to the way the need for reconfiguration is detected; if a certain need for reconfiguration is detected in the analysis phase, the planning should strive to satisfy that need. In practice, the opposite dependency also requires consideration: Only those situations that can be mended by a reconfiguration that can actually be planned should be considered during analysis (e.g., if no provisions are provided for handling depletion of memory resources, reconfiguration will hardly be able to save an application that has just encountered an out-of-memory error). Together with the detection of a reconfiguration need, this is discussed in Chapter 8.1

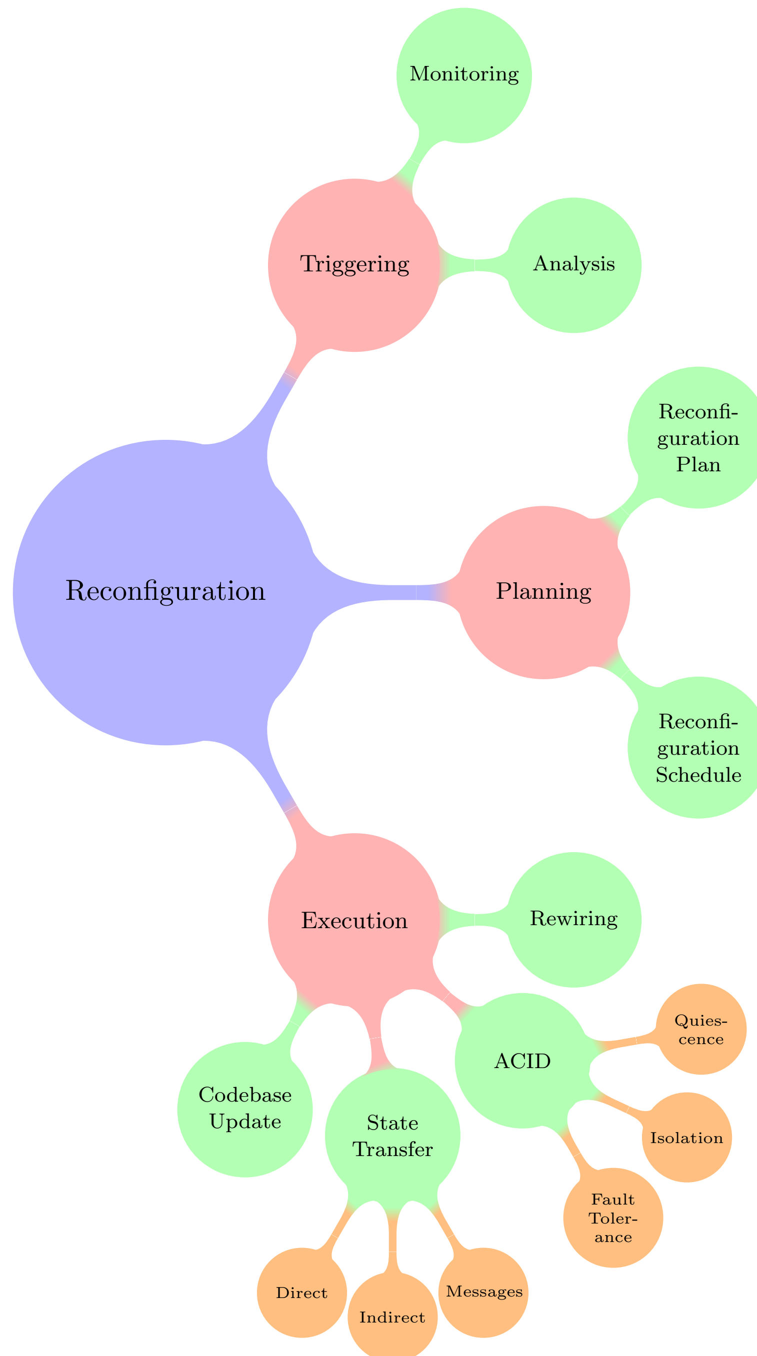


Figure 2.4: Ingredients of reconfiguration

(3) The execution of reconfiguration. This part has the most aspects to it, and the focus of this thesis as well as most other works is placed here (cf. the last two columns of Tab. 3.1). First, the pure technical realization, dubbed "rewiring" (although it also contains addition, removal and possibly updating of components), needs to be provided. This might also require an

update to the code-base (e.g., substitution of underlying data structures like classes), which can become very complex in practice (since many host languages do not provide means to accommodate updated code).

Reconfiguration cannot just be started without considering the remainder of the system. Instead, some precautions have to be taken to provide something akin to the ACID (atomicity, consistency, isolation, durability) of database systems. For reconfiguration, durability is usually less of a concern, unless perhaps for extensive distributed architectures, which we do not consider here. The other three aspects are captured under the term "quiescence" [**KM90**], which is a denomination for a state of a system where reconfiguration will not interfere with ongoing transactions. This is partly to be ensured by the planning phase, but might also require active steps to avoid problems with concurrent components. Isolation is a related concept, but stressing the non-interference of a reconfiguration with other reconfigurations. Finally, fault tolerance stresses the aspect of consistency; since reconfiguration might fail or introduce faulty components into a system, special precautions might be required.

If components are stateful, their state should be retained. There are two ways of doing this [**Van07**]: The *direct* approach, where such a state transferal is part of the reconfiguration plan, and the *indirect* approach, where the components are put in charge of retrieving the relevant state from their precursors. Also, pending messages (and other framework artifacts) need to be retained.

There is quite a consensus about these ingredients of reconfiguration, although different approaches accentuate different aspects, and often various aspects are neglected (in this thesis, we consider the ACID aspect to be handled by the user mostly, while placing emphasis on state transferal, which only few other approaches do).

CHAPTER 3

Related Work

Ein Mann, der recht zu wirken denkt,
Muß auf das beste Werkzeug halten.
Bedenkt, Ihr habet weiches Holz zu spalten.

— Johann Wolfgang Goethe, Faust I

There is an abundance of work on components and reconfiguration, dynamic updates, autonomous adaptivity and related topics. Even a patent for dynamic updates of operation system components [**Mar94**] has been issued. The research area is obviously defined way too broadly to give a really comprehensive overview since many techniques and approaches are related and can spark new ideas. In this chapter, we will first try to give an overview over well-known component frameworks, and then discuss the literature on component frameworks supporting reconfiguration, as well as some frameworks and approaches that are not directly concerned with components, but also relate closely to the approach discussed in this thesis.

A very complete (and frequently revised) overview of component models, covering both commercial and research models, is due to Lau and Wang [**LW05a, LW06, LW07**]. Applying a taxonomy presented in [**LW05b**], a number of component models (note the emphasis on component models instead of frameworks, which is no coincidence – the overview focuses on the model behind the actual framework implementations) is investigated, including most of those presented in the remainder of this chapter. An interesting classification of phases is proposed: The *design phase*, where components are designed and implemented, the *deployment phase*, where binaries are generated and deployed, and the *run-time phase*, where the component binaries are instantiated and initialized. Based on how composite components (i.e., component assemblies) can be created in the three phases, a further taxonomy is presented in [**Lau06**].

In [**NFH+03**], a survey of literature discussing component-based software engineering, techniques and architecture description languages (ADLs) is given. This survey has been made as a part of the SAVE project, sponsored by the Swedish Foundation for Strategic Research – a project that tried to devise ways of systematic development of component-based systems for safety critical systems [**Han01**] – and the papers presented in the survey are chosen by their applicability to that project.

The "Common Component Modelling Example" (CoCoME) [**RRMP08**] offers an interesting insight into various research frameworks and their modelling capability. An example application of a supermarket trading system application was provided [**HKW+08**]. It was then modeled in various component models, including SOFA [**BDH+08**], Fractal [**BBC+08**] and Java/A [**KJH+08**]. Applying different component modelling techniques to a common example sheds an interesting light on their individual capabilities, as well as their common features.

Concerning reconfiguration, there are two often-cited publications: [**KM90**] by Kramer and Magee (and refined in [**MGK96**]), which introduced *quiescent states*;

these are necessary for conducting the reconfiguration in a situation where it does not interrupt active communication. The other one is [Hof93], the dissertation of Christine Hofmeister, which introduced POLYLITH, a reconfiguration framework that considers component state (see also [HP93]). The former publication can be regarded as influential since its definition of quiescence is used by many other works, whereas in the areas discussed by the latter publication, every framework tends to come up with own ideas. Besides these two (early) works, the research field appears very divided – maybe due to the varying incentives to do reconfiguration.

Vandewoude and Berbers [VB02] present an overview of approaches towards dynamically updating component systems, with emphasis placed on embedded systems. Dynamic software updates are concerned with version upgrades of existing components, and the challenges are both technical (like loading the same JAVA class again, which is not supported natively by the virtual machines class-loader) and conceptual, which is mostly concerned with updating the state. Vandewoude and Berbes present a classification of approaches based on their usability, and the requirements, restrictions and suitability for embedded systems. We will not repeat the approaches they investigated here, save for [CD99] and [SRG96]; the mere updating of code tends to be more technical than the focus of this thesis.

Sadjadi et al. [MSKC04b], extending (and making more accessible) the work in [SM03] and building on the taxonomy presented in [MSKC04a], present an overview of middle-ware that supports adaptive behavior. After an overview of traditional middle-ware, which places emphasis on commercial implementations like CORBA or DCOM [CHY+98], they introduce some key paradigms for adaptive systems, which are

- **computational reflection** – allowing a program to obtain information about its own structure,
- **component-based design** – based on the Szyperski definition [Szy98, SGM02], they emphasize the ability of *late binding*, i.e., the ability to link components after application startup,
- **aspect-oriented programming** – in order to allow for composition of *cross-cutting concerns*, AOP is required to build variations of existing middle-ware suitable for adaptation,
- **software design patterns** – in order to make adaptation less expensive in development, the repeated patterns should be identified, like the *virtual component pattern* [CSKO02].

The authors proceed to propose a taxonomy based on the questions "how", "when" and "where" – the former describing the applied pattern (e.g., proxies or aspect weaving), the second describing the point in the application lifetime to do the adaptation (notable, they consider also adaptation at compile or load-time), and the latter describing the layer where the adaptation is conducted. 40 middle-ware frameworks are then categorized according to this taxonomy. The scope of this overview is broad, including the rather generic ASPECTJ [KLL+02] aspect-oriented programming framework as well as BOEING BOLD STROKE [Sha00], a rather comprehensive framework for component-based development of avionics software. Still, this overview is an excellent starting point, and covers many of the related work we will discuss later. Yet the scope on middle-ware systems similar to the heavyweight industrial component frameworks rules out many more experimental component frameworks.

The overview of Bradbury [Bra04], with a shortened version in [BCDW04], focuses on formal means to describe adaptive systems. Based on a distinction of the underlying formalism (graphs, process algebras, logic and others) they present 14 approaches towards describing systems that support reconfiguration. Bradbury

focuses on self-managing systems, giving a concise definition based on four stages of reconfiguration: Initiation, selection, implementation and assessment, and requiring that for a self-managing system, all stages need to be triggered and managed internally. This is only interesting for the first stage, the initiation, as it can be observed that any formalism supporting internal initiation also supports internal progression of the latter stages. Apart from this characterization, Bradbury introduces a taxonomy based on the supported reconfiguration operations, the selection abilities (i.e., how the reconfiguration scheme is chosen among a series of candidates), and whether the reconfiguration is managed centralized or distributed.

Elkhodary and Whittle [**EW07**] present a survey focused on adaptivity with respect to application security. It uses the taxonomy and adaptation requirements of [**MSKC04a**], extends them with goals for security services (authentication, authorization and (fault-)tolerance), and applies them to four adaptive security frameworks. For most of these system, reconfiguration is fairly technical and not within the scope of this thesis (mostly, it is more of a tuning than an actual redesign), with the exception of [**KHW$^+$01b**].

The DySCAS project ("Dynamically Self-Configuring Automotive Systems") provides a deliverable [**Ant07**] that discusses work relevant for their idea of an adaptive system, which is to support self-configurability of embedded systems in the automotive domain. The actual applicability of the concepts is discussed in [**Jah07**]. While not directly concerned with component-based systems, this survey discusses lots of work covering different aspects of adaptivity. Their focus is the field of automotive software, but the broad discussion of control theory and adaptivity-enabled middle-ware provides a very good overview on the techniques and frameworks available. Further relevant areas are also presented: Automotive security, mobile communication and quality of service considerations.

Finally, a recently published survey by Markus Huebscher and Julie McCann [**HM08**] provides a very interesting insight into the various ingredients required for adaptive software. This survey is centered around the *MAPE-K loop*, a five-stage general approach towards adaptivity, and discusses the various approaches and the extent of adaptivity obtained.

3.1. Component Frameworks

In this section, we will introduce some well-known component frameworks. Compared to the listing of reconfiguration-enabled frameworks in Sect. 3.2, this overview is very limited. Still, the various approaches to the topic of components and component-based software engineering supply an impression on the topics that need to be considered, and areas where components can be successfully utilized.

Traditionally, component frameworks are divided into *industrial* frameworks whose primary purpose is to support the development of commercial products, and *research* frameworks that investigate ways to benefit from component-based software engineering as well as research ways to provide the user of the framework with support based on formal reasoning. Obviously, the border between these two fields is blurred, and we will later, in Sect. 3.2, see a number of examples where an industrial framework is used as a basis for researching reconfiguration.

3.1.1. Industrial Component Frameworks. Industrial component frameworks have seen considerable success, mostly due to the activities of Microsoft, which actually built large parts of the Windows operating system and related APIs like ActiveX with components, Sun Microsystems (in the context of Java) and the Object Management Group (OMG), providing the standard of Corba [**OMG08**].

3.1.1.1. CORBA. CORBA is highly influential, also for scientific research of components – as the large number of extensions towards reconfigurability demonstrates (cf. Sect. 3.2.7). Being the first specification the influential Object Management Group (OMG) pursued, it has matured to its current version 3.1 [**OMG08**].

Standard CORBA revolves around the idea of objects, which is not necessarily what we would understand as components. At the heart of the definition is the object request broker (ORB) which handles object location and communication, hiding possible remote access, platform-dependency and programming language issues from the client. Being a specification, a large number of ORBs has been created by different vendors; in [**Pud08**] over 50 different ORBs are listed. Starting with version 2.0, CORBA also allows inter-ORB communication (meaning that a client request issued to its ORB on one machine can be forwarded to a different ORB implementation running on another machine) by means of a special protocol, the GIOP (General Inter-ORB Protocol, cf. [**KL00**]).

ORBs need to provide interfaces to the clients in order to allow them to invoke services (which are then delegated to the proper target). In order to abstract from the underlying programming language, a special interface definition language (IDL) is used. This abstract language is then mapped to a programming language for which a mapping has been provided (which is true for most production-grade languages).

Starting with CORBA 3.0, a component model has been included in the specification: The CORBA Component Model (CCM) Specification 4.0 [**OMG06a**]. This formerly stand-alone specification was included to address some shortcomings of CORBA 2.4, among which are the lack of a specified way of deploying objects in an ORB, leading to ad-hoc solutions and the lack of a standardized object life-cycle management [**WSO01**].

Components use the IDL to provide services and are run in component servers that are placed on top of the ORB. Components define *ports* for communication. A number of port types is defined, with port types fixing the communication means: *Facets* are what we call provided interfaces, *receptacles*, corresponding to required interfaces, *event sources* and *sinks*, which allow for a purely event-based communication with a publish/subscribe scheme and *attributes* which allow the setting of parameters from the outside. All these component features are declared in the IDL. Communication between components can be synchronous or asynchronous.

Using a special declarative Component Implementation Definition Language (CIDL), components can declare parts of their implementation and state, having it persisted at runtime by the component server. This CIDL and the IDL definition can be used to generate implementation skeletons, which can be combined with code written in one of many possible programming languages, resulting in the final executable component. CIDL allows to define a large number of properties of components, like their threading policy. The resulting component model is quite generic and complex, with a multi-stage component creation process.

3.1.1.2. COM. If the commercial success of Microsoft was attributed to the software engineering techniques adopted, COM (the Component Object Model) [**Box98**] would be the most significant success story of components. Summarizing a number of technologies like DCOM, ACTIVEX and COM+, it is part of the backbone architecture of WINDOWS and has led to the widespread use of software components in desktop applications like MICROSOFT WORD, which uses OLE controls (which are defined on top of COM) to insert elements handled by other programs (like an EXCEL sheet) into a document.

At its core, COM is fairly simple, using the notion of *interfaces* and *objects*, the latter implementing the former. Objects are created by an *object server*, which

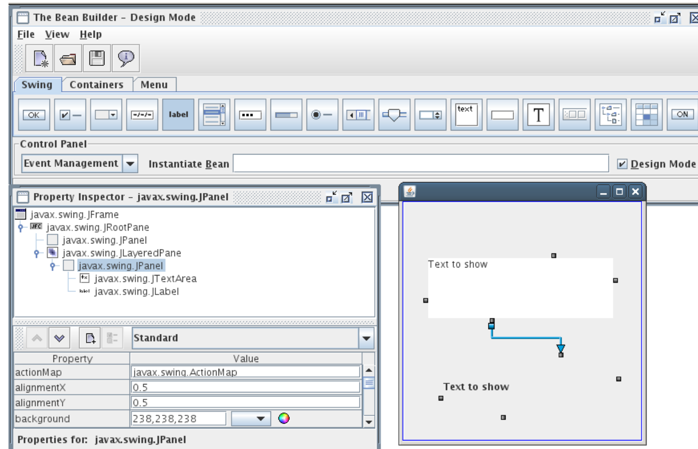


Figure 3.1: Java BeanBuilder screen-shot

comes in the form of an executable or a DLL (dynamic link library). COM encourages the use of *factories*, which, being COM objects themselves, are used to create other COM objects at runtime. The communication of objects is specified by a binary standard (which is proprietary and has subsequently been abandoned for certain domains like web services, where Microsoft now propagates the use of SOAP). COM itself is language independent, though the definition of the communication protocol suggests close relation to C++. Interfaces are described in Microsoft's own interface definition language MIDL. From these definitions, proxy stub code is generated. The assembly of component applications is not considered; instances of components are obtained via calls to the factories, which can be obtained from the dreaded Windows Registry.

COM serves as a basis for various extensions: DCOM, which uses special proxy stubs to enable transparent remote procedure calls (interestingly, this is perceived as a problem, since the generated code is fixed; in [WL98], an extension is proposed that enables customized connectors), COM+ providing enterprise features like distributed transactions, or ActiveX, used for embedding code in documents like web pages or Office documents. The former two extensions are similar to Enterprise Java Beans (see Sect. 3.1.1.3) and Corba, and COM is frequently compared to those [CHY+98]. The concept of using componentized code to embed content from a foreign application in a document is appealing enough to have the two big open-source frameworks Gnome and KDE each work on their own component model, Bonobo and KParts, respectively [Rei07].

3.1.1.3. *Java's Component Models.* Components were introduced in the context of Java by the introduction of Java Beans [Sun97]. The idea is fairly simple: Beans are special Java classes that follow certain conventions, like providing a default (parameter-less) constructor, getter and setter methods for their attributes and being serializable by implementing `java.io.Serializable`. The primary intent of Java Beans is their assembly and editing in graphical tools:

> *"A Java Bean is a reusable software component that can be manipulated visually in a builder tool."* [Sun97]

A reference implementation of such an editor – the Java BeanBuilder – is shown in Fig. 3.1. GUI components are connected such that an event triggered in one component (here, a text input field) triggers an update in another component.

Although the success of such a visual approach was limited, many elements of the SWING GUI library follow the JAVA BEAN conventions.

ENTERPRISE JAVA BEANS are an extension of that idea to enterprise server applications. Being embedded in a host of APIs for building such enterprise applications (e.g., JNDI, the JAVA naming and directory interface, or JSP, the JAVA server pages technology), they provide a component framework that is suitable for developing distributed applications much alike (and, in fact, compatible to) the approach of CORBA.

At the heart of EJB are the beans, similar to the pure JAVA BEANS, but more elaborated. Beans come in different flavors: Session beans, that are used to store a state for a short-termed session, and entity beans that persist data over multiple sessions, providing transactional integrity (the ACID principle known from databases). Message-driven beans facilitate asynchronous communication with legacy systems, and also the utilization of web services can be encapsulated as a bean. Beans are written as JAVA classes and supplied with deployment descriptors, which describe their (implementation-independent) properties in an XML file. ENTERPRISE JAVA BEANS are then deployed in an EJB-container that manages their life-cycle and the persistence of entity beans. The container hides the details of the execution of the beans, their communication and persistence. For example, a suitable container can allow an enterprise application to be distributed, and even redistribute it in case of server failure or server addition.

These EJB containers are notorious for their resource requirements [**CMZ02**]. Often, this is more attributable to misconfiguration than to a genuine problem of the EJB architecture, which offers interesting possibilities for adaptation [**DMM04**].

A vast number of techniques have been developed around the JAVA platform that provide or support adaptivity without explicitly making use of the concept of components, e.g., JINI, a framework for dynamically discovered services [**Arn99**]. JAVA's features like object serialization, platform independence and reflection capabilities, combined with the rather rigid type system (compared to languages like RUBY) and competitive performance, make it an ideal host language for frameworks that support a special kind of computing. Other examples for frameworks based on JAVA that offer similar functionalities as we desire are OSGI [**CH03**] (a framework that offers services which are hot reloadable; recent developments named "declarative services" suggest that OSGI shifts from pure services to the ideas central to components) or JXTA [**Gon01**] (a peer-to-peer networking framework).

3.1.1.4. KOALA. Developed by Philips, KOALA [**Omm98, OLKM00**] is a component model for consumer electronics. Among the industrial framework, it stands out because of its utilization of hierarchical components and its background in the DARWIN ADL [**MDK93**]. KOALA extends DARWIN with the possibility to add glue code outside of components, and an increased emphasis on parameters.

Components in KOALA declare provided and required interfaces. Provided interfaces can become connected to an arbitrary number of required interfaces, which in turn need to be connected to exactly one provided interfaces. *Compound* components are groupings of components (i.e., higher-level hierarchical components), exporting provided and required interfaces. Glue code can be introduced by *modules*, allowing for generic (e.g., multi-cast) communication.

KOALA is a relatively thin wrapper of the C programming language. Multithreading is possible, but is declared on the composition level; component communication is done by the standard C method invocation. Its aims are architectural programming and handling *diversity*, i.e., to provide a structured approach towards component reuse and software product lines. To support diversity, component parameters become important, as to keep components reusable as much as possible.

KOALA supports this by the notion of *diversity interfaces* that require the parameters' values. Also, a component repository is provided, which has some provisions for evolving components in a consistent way.

3.1.1.5. SPRING. The SPRING framework [WB05, NK08] does not declare itself to be concerned with components. Instead of components, the basic entities are "POJOs" (plain old JAVA objects). Spring then performs *dependency injection* by setting the associations of objects from within the framework. Basically, an XML file specifies which POJOs are to be instantiated, and how they are to be connected. The big advantage of SPRING is given by the fact that this is basically all it does (except for provisions for aspect-oriented programming, but again they are quite lean [Col06]). Instead of providing extensive communication mechanisms or object persisting provisions, it provides this lean mechanism of assembling applications, and then proceeds to provide an abundance of wrappers to existing APIs and helper POJOs.

The success of SPRING is encouraging, as it basically revives the idea of the separation of roles for large-scale, enterprise applications. The type of applications targeted by SPRING is governed by large-scale software reuse (in the form of heavyweight APIs like HIBERNATE). SPRING shows that only minimal provisions reminiscent of components are required to facilitate the building of applications. The APIs have been there before, but the making explicit of how they become assembled seems to make the difference.

3.1.2. Research Frameworks.

3.1.2.1. SOFA. The SOFA 2.0 (SOFtware Appliances) component framework [BHP06, BHP07] emphasizes the use of a *hierarchical* component model. Components can be either *primitive*, in which case they are in principle programmed in common JAVA code, or *composite*, i.e., built from other components [BHP06]. Any component is embedded in a *frame*, which provides a black-box view on the component; i.e., only the provided and required interfaces are made visible to the outside world.

SOFA 2.0 also provides a clear definition of *controllers* – small components that are conceptually located in a frame's border and control the execution of components within the frame. This concept is very generic and allows a user-defined treatment of the component's life-cycle and provides a well-defined attachment point for reconfiguration.

SOFA 2.0 supports various means of communication by making *connectors* first-class objects of their component model. This means that connectors, like components, can be custom-written to support different means (like message passing or regular stack-based method invocation) and cardinalities (single target vs. broadcasting) [BMH08].

One of the remarkable things about SOFA 2.0 is that a *software engineering process* is provided, i.e., a description on how an application should be created. The process described in [BHP07] starts with a process of creating composite components and writing primitive ones. The repository is used to find components of finer scale and to store the resulting composite component. The next stage is a top-down approach towards assembling the component, starting with the application frame and subsequently refining it. Finally, the component is deployed, which requires specification of a *deployment plan*. This plan handles the distribution of components on the available nodes.

SOFA 2.0 supports advanced features like reconfiguration [HP06], which we will describe in a dedicated section (cf. Sect. 3.2.8). One outstanding contribution is the support for a definition of interface semantics by modelling them with *frame*

protocols, defined as *extended behavior protocols* [Kol07]. Behavior protocols specify allowable sequences of events received at the component's ports. Syntactically, they are regular expressions with a special operator | for specifying parallelism. Extended behavior protocols enrich this specification approach by introducing features known from programming languages like local variables, event parameter and guarded loops. Model checking of extended behavior protocols can be conducted with JAVA PATHFINDER [PPK06] or SPIN [Kof07].

3.1.2.2. FRACTAL. The FRACTAL [BCL+04, BCL+06] component model is quite similar to SOFA. It emphasizes the use of hierarchical components with sharing – i.e., breaking of the strict tree structure for components providing common services like memory management. Another feature emphasized is introspection of components, and modularized control – i.e., components can be given varying degrees of "intercession capabilities" [BCL+06].

FRACTAL itself is not a component framework, but describes how components should be declared and connected, with the actual implementation (and considerable parts of the semantics, like the interpretation of communication) left open. Like SOFA 2.0, components are encapsulated with control functionality embedded in the component frame, here called a *membrane*. Interfaces are used as communication points to the outside world and the *content* components.

A large number of implementations of the FRACTAL component model exist [CS06], e.g., the reference implementation JULIA [BCL+06] or PROACTIVE [BBC+06]; both are written in JAVA. JULIA's basic concern is the membrane, and the contained controllers; it provides a mix-in mechanism as well as a bytecode generator for interceptors. Optimizations are provided as well, for merging controllers on the bytecode level as well as bypassing unused membrane call interceptors. PROACTIVE, on the other hand, is geared explicitly at Grid systems; it places much emphasis on communication, making futures [CHM08] and one-to-many communication (broadcast for $1 : n$, and gather-cast [NS00] for $n : 1$ connections) available on top of the Active Object pattern [LS96]. The communication styles supported by PROACTIVE are formally described by the ASP calculus [CH05].

FRACTAL is an open and versatile model that can be extended in many directions, like adding reconfiguration capabilities; cf. Sect. 3.2.25.

3.1.2.3. ARCHJAVA. ARCHJAVA [ACN02b, ACN02a] is a JAVA extension for architectural programming with components. The primary concern is to make the architecture of component-based applications more explicit, and to avoid the architectural erosion [PW92] that is encountered if the architecture model is kept in addition (and not as an integral part) of the application.

ARCHJAVA is part of the PhD of John Aldrich [Ald03], and is not updated anymore. It was, however, quite influential for JAVA/A, which in turn was influential for JCOMP. Apart from this influence, ARCHJAVA is interesting to us because it emphasizes a single aspect and builds the entire design around that idea. The resulting framework is very lean and requires only very few additions to its host language JAVA.

ARCHJAVA utilizes bidirectional ports which declare provided and two kinds of required methods: Normal ones, which get invoked by normal JAVA method invocation, and broadcast methods. The architecture is then declared in assembly files, which are components themselves: By using hierarchical components, the enveloping components are responsible for the creation and assembly of the contained components. The component definition, which largely consists of JAVA code with a few keywords added, is then translated to genuine JAVA code that generates ports

and introspection facilities. In an extension of ArchJava, custom connectors are made available that can realize user-defined communication means [**ASCN03**].

Apart from defining the Java language extension with a type system used for ensuring correct component connection [**Ald08**], the papers on ArchJava report on experience with refactoring object-oriented programs to the ArchJava component model. Modest code changing requirements are reported, and a "rediscovery" of the program architecture that is known to the programmer (although in an idealized way that needs to be refined), but not made explicit in the normal Java code.

3.1.2.4. Java/A. Java/A [**Hac04, BHH+06**] is a component framework similar to ArchJava in that it emphasizes architectural programming. It has been influential for JComp, and at the same time is one of the few component models with a throughout formalization [**BHH+06, KJH+08**] based on I/O transition systems [**dAH01**].

Java/A sees itself as an extension of the Java programming language. A central concept, aside from components, are ports. Ports provide and require an interface, and are connected by connectors, which provide a 1:1 connection only. Components are hierarchical, composite components define their structure in *assemblies*. Java/A supports synchronous and asynchronous communication. All these specification elements are realized by special keywords, which are translated by a preprocessor to genuine Java.

One notable aspect of Java/A is the utilization of port *protocols* as a first-class ingredient in the component model (whereas, for the other component models, protocol considerations are usually added to the existing model). These protocols are given by (a textual representation of) UML state machines. Java/A utilizes the Hugo model checker [**KM02**] for checking deadlock-freedom of the protocols.

The component model of Java/A has also been used to model the CoCoME (example) [**KJH+08**].

3.1.3. ADLs. *Architectural description languages* (ADLs) are languages that describe the structure of a software system, i.e., the instantiation and connection of components. ArchJava and Java/A already comprehend language extensions for specifying the architecture within the program code. But many other component frameworks provide just an API for instantiating and connecting components, even if they provide special languages for describing the actual components (e.g., Corba's IDL and CIDL).

ADLs seek to fill this gap by providing a language suitable for defining component setups. Frameworks like EJB or Spring use XML files to instantiate the components and connect them, but here we will focus on special-purpose languages. A classification of such languages, which also provides a comprehensive overview, can be found in [**MT00**]. Here, we will focus on ADLs that are in some way connected to component models that allow for adaptivity.

3.1.3.1. Darwin. Darwin [**MDK93**] is a language that stems from the experience obtained with Conic and is part of the REX project (cf. Sect. 3.2.2). Explicitly referencing the difference between programming-in-the-large and programming-in-the-small [**DK75**], Darwin aims at being a programming language for in-the-large programs.

At its core, Darwin is very much alike a regular programming language that describes a number of processes, which communicate by sending instances of basic data types. As an example, a prime number sieve is given, with a process for each prime number that filters out their multiples. These processes are organized in a chain, which is described in a *component* – here, similar to an assembly or to a

higher-level hierarchical component. Components can allocate their processes to a number of abstract machines, which allows for load balancing on a fine-grained level, suitable for highly parallelized machines like transputers [**WS85**].

Later, the architectural description of components and the constraining of their architecture was emphasized [**GMK02**], with a translation to the ALLOY language [**Jac02**] which provides automated verification capabilities.

DARWIN uses components with provided and required ports, both named and typed, but not related; the connection of provided and required ports is done by a *binding*; there is no additional semantics to the connection in form of a *connector*.

DARWIN provides different views on a component system, including a behavioral view that makes component interaction explicit; this is extended by TRACTA [**GKC99**], which is an approach that allows for compositional verification, where each component is checked in a local context, and the results are combined to eventually verify correctness criteria for the entire system.

3.1.3.2. WRIGHT. WRIGHT [**All97, AG97**] is an ADL that emphasizes the separation of roles even more by making a distinction between *styles*, which describe components, connectors and architectural constraints, and *configurations*, which describe an actual component system. Part of the styles are connectors, which are hence maintained as first-class objects in WRIGHT. Their "glueing", i.e., how they transport communication between components, and the behavior of the components is explicitly described in the "communicating sequential processes" calculus (CSP) [**Hoa83**]. Architectural constraints are defined in first-order logic.

WRIGHT uses ports, but does not distinguish required and provided ports – a port protocol, given in CSP, is used for specifying whether communication is incoming, outgoing, or both; making each port effectively a bidirectional port. Connectors declare *roles* which are connected to ports of components, hence, other connectors than just 1:1-connectors can be realized. Again, the roles get annotated with CSP protocols; together with the glue specification, the semantics of connectors and components is defined. Of course, the multitude of specifications requires consistency verification, which can be automated by model checking [**All97**].

In [**All97**], a discussion on the utility of WRIGHT (compared to CSP alone) is given, which is quite interesting as it addresses, besides technical issues like encapsulation and typing, an objection often encountered: Making components, ports and connectors first-class helps to get a mapping to the informal understanding of component systems, it "elucidates" the architecture more and also uses a common and well-established vocabulary.

3.1.3.3. OLAN. The OLAN configuration language [**BABR96**], categorized as a *module interconnection language*, is given a precise message semantics by means of the ICCS (Interconnected Component Calculus) [**VDBM97**], which is based on the CCS calculus [**Mil89**]. It extends CCS by not only allowing message passing, but also synchronous calls with thread passing (called *activity flow*, cf. the discussion of the CMC framework in Sect. 5.2.4). ICCS distinguishes between *active* and *passive* components; and allows coupling of active and passive components, extending CCS by this special kind of communication. While CCS is capable of simulating such synchronous calls (by explicit locking and callbacks), making synchronous calls first-class calculus operations helps to describe the true behavior of a system in a concise way.

3.1.4. Comparison. The component frameworks and ADLs described here are merely the tip of an iceberg, although those presented here are encountered most often in the literature; especially the scientific literature discussing reconfiguration. But even as the number of component frameworks abounds, some features, most

notably the concept of a component, are mostly undisputed. It appears that component frameworks do not differ so much in their concepts, but rather in the accentuation of different ideas, like the accentuation of communication (highly emphasized in frameworks with first-class connectors like SOFA, and not much considered in KOALA) or the accentuation of the architecture (made first-class in ARCHJAVA and JAVA/A and left to API calls in COM).

A similar variation in the focus is found between industrial and research component frameworks. Research frameworks utilize hierarchical components, which is not found in industrial component frameworks (safe for KOALA). On the other hand, industrial frameworks often place emphasis on interoperability (especially in CORBA) and vendor-supplied implementations of a common model (CORBA, EJB and, with limitations, COM) which only FRACTAL proposes. ADLs do not see widespread use in industry, although some attempts were made (KOALA extending DARWIN).

As a trend, an increased interest in components is observable, e.g., in CORBA's inclusion of the CCM. Similar tendencies can be observed for OSGI; while the commercial success of single-component sales seems not to come to pass, the general idea of components and the separation of roles still seems highly attractive.

3.2. Component Frameworks Supporting Reconfiguration

Mit Eifer hab ich mich der Studien beflissen.
Zwar weiss ich viel, doch will ich alles wissen.

— Johann Wolfgang Goethe, Faust I

Many papers describe the adaptation of existing frameworks [**BISZ98**, **KRL+00**, **BR00**, **TMMS01**, **Weg03**, **BNS+05**, **RAC+02**, **BJC05**, **LLC07**, **KB04**, **CS02**, **JDMV04**], mostly for CORBA [**OMG06a**], or the implementation of an own, dedicated framework [**Blo83**, **KM90**, **MKSD92**, **MDK94**, **Hof93**, **Lim93**, **CHS01**, **KHW+01b**, **BHP06**, **HC03**, **HW04**, **MG04**, **GS02**, **Hac04**, **ASS07**, **Van07**, **RP08**, **IFMW08**], and it is very interesting to note which aspects of reconfiguration are given special attention – sometimes real-time properties are of importance [**MNCK98**], sometimes formal describability is emphasized [**Kni98**]. Additionally, a number of calculi exist [**FZ06**, **CH05**, **SC06**] which highlight the theoretical aspects of reconfiguration.

In this section we will give an overview of adaptive component frameworks. Two things should me mentioned before: First, the order in which the works are presented is roughly chronological, sorting the various frameworks and approaches by the date of their first publication. Sometimes, when it is convenient, we will group frameworks that work in a similar manner (e.g., in Sect. 3.2.7 we will discuss frameworks that extend CORBA). Second, although this overview was started with the intend of giving an exhaustive overview, we are not able to give any guarantee of exhaustiveness. The wealth of research makes it difficult to get an idea of the extent of related work, especially since lots of work is conducted in parallel and, judging from the related work sections, often unaware of concurrent research. Furthermore, there are vast areas of research that are closely related, but are investigated by a different community, e.g., migration of agents [**BU97**, **IKWK00**] or adaptive workflow management [**HSB98**].

Following the taxonomy of Sadjadi et al [**MSKC04a**], we are interested in the "how" question (i.e., the question which means are used to do reconfiguration), fix the "when" question to runtime only, and omit the "where" question, since not all frameworks are made up of distinguishable layers. Instead, in the spirit of [**ZL07**],

we will ask a "what" question, trying to identify the class of applications that is targeted by the framework. This question can be divided into a number of sub-questions:

- The class of problems that are to be solved by applications written in the framework,
- whether an explicit state and its retainment are considered,
- how reconfiguration is triggered and how its goal is described, as investigated in [BCDW04],
- how the reconfiguration is guarded against concurrent operation of uninfluenced components, and
- which examples, if any, are given.

The latter point is of particular interest, since good examples are scarce, as we will discuss in Chapter 8. Tab. 3.1 lists the frameworks under comparison, and gives a quick overview on whether they are state aware, how logical atomicity is ensured, how the reconfiguration is triggered and how it is described. We will next give brief introductions to the frameworks, along with the answers to the aforementioned questions of "how" and "what". For stateful frameworks, further considerations about the state are given in Tab. 3.2.

Framework	State ret.?	Atomicity	Trigger	Reconfiguration description
ARGUS [Blo83]	✓	atomicity is used for all actions, using two-phase commit	external	external
CONIC [KM90]	–	quiescence	external	change specification (own language)
REX [MKSD90, MKSD92]	–	quiescence	programmed	DARWIN [MDK93]
REGIS [MDK94]	–	quiescence	programmed	DARWIN
POLYLITH [Hof93, HP93]	✓	interfaces and components can be blocked	external	CLIPPER [AHP94]
FLEXIPHANT [Lim93, Lim96]	✓	ensured by a pre-calculated schedule	external (hot code update)	series of reconfiguration transitions
SIMPLEX [SRG96]	✓	–	external (hot code update)	hot code update, with user-supplied correctness monitor
HERCULES [CD99]	–	–	external (hot code update)	hot code update, with user-supplied constraint-evaluator
Bidan et al. [BISZ98]	✓	local blocking	external	–

Framework	State ret.?	Atomicity	Trigger	Reconfiguration description
DYNAMICTAO [KRL+00]	–	–	external or triggered by inspection	from a graphical administration interface a reconfiguration agent is generated
LUASPACE [BR00]	–	–	user interaction	user level
ETERNAL SYSTEM [TMMS01]	✓	atomic switchover, quiescence	external (hot code update)	calculated from static analysis
Wegdam et al. [AWSN01, Weg03]	✓	a *safe state* is reached, which is similar to quiescence	external	–
SWAPCIAO [BNS+05]	–	portable object adapter (POA) – CORBA internal, ensures an empty message queue	external	CIDL extension with special `update` directive
SOFA/DCUP [PBJ98]	✓	transactions (discussed)	user-level	programmed
SOFA 2.0 [BHP07]	–	–	user-level (requested by existing components)	component request
OPENCOM [CBCP01]	–	local blocking by special receptables	user-level (third party reconfiguration)	user-level
CACTUS [CHS01]	messages	three-phase protocol	by inspection and fitness evaluation of alternatives	single component replacement only
WILLOW [KHW+01a]	–	–	comparing measurements against finite state machines or human interaction	prioritized, decentralized reconfiguration requests or human interaction
ARCHJAVA [ACN02b]	–	–	programmed	programmed in assembly
GRAVITY [HC03]	–	–	dynamic environment, components “arrive”	local component contracts, automated discovery

Framework	State ret.?	Atomicity	Trigger	Reconfiguration description
ReMMoC [GBS03]	–	–	external	substitution of components, lookup of services
Planit [AHW03]	–	user level	monitoring	planning (major focus)
OpenRec [HW04]	user level	user level	user level	user level
CASA [MG04]	✓	atomic replacement	Monitoring and investigating application contracts	set of alternatives
Rainbow [GCH+04]	–	–	Monitoring by probes and gauges, constraints	programmed strategies
Plastic [BJC05]	–	transactions (briefly mentioned)	programmed (monitoring) or ad-hoc (user-triggered)	defined in extended ACME
DEVS [HZM05]	–	globally synchronized steps (discrete event simulation)	implemented in the components	DSL for adding/removing components and connections
GridKit [GCB+06]	–	blocking of affected subsystems	events	local and global reconfiguration, based on pre-calculated plans
Java/A [BHH+06]	–	–	user level	programmed in Java
Adapta [ASS07]	user level	user level	monitoring, value ranges	user-defined strategy objects
Draco [Van07]	✓	tranquility	hot code update	hot code update
Fractal [PMSD07]	✓	quiescence (ACID)	programmed	programmed or by FPath scripts
Jade [TBB+05]	–	–	control loop, quality of service or error detection	implemented in reconfiguration components
Bark [RAC+02]	–	transactions (but actual atomicity is not discussed)	external invocation	XML scripts

Framework	State ret.?	Atomicity	Trigger	Reconfiguration description
COMPAS [DMM04]	– (discussed)	atomic switchover	control loop with decision policies	single component replacement
Ketfi et al. [KB04, KBC02]	✓	–	hot code update	adaptability relation with scripts
Chen and Simons [CS02]	✓	similar to quiescence, monitoring of communication	reorganization of component distribution, hot code update	–
NeCoMan [JDMV04]	–	quiescence by atomic rewiring and monitoring	–	weaving of aspects
ReDAC [RP08]	✓	local blocking, algorithm similar to quiescence	external	–
CoBRA [IFMW08]	✓	blocking proxies	external	addition, removal and updating of single components
Music [REF+08]	–	global blocking	utility ranking of alternative plans and current situation	plans generated based on functional consistency

Table 3.1: Overview of component frameworks supporting reconfiguration

3.2.1. Argus. Argus [Blo83], a framework by Toby Bloom, aims at enabling replacement of distributed software components. These components are called *guardians*, and their definition is that any guardian resides solely on a single physical machine. Guardians are comprised of objects and processes, and these objects form the state of a guardian. The idea of well-defined interfaces as a prerequisite for reconfiguration is introduced.

Reconfiguration is triggered by the user, e.g., for code replacement or feature extension. There are considerations about the correctness of the reconfiguration, based on abstract states and possible future events. Replacement is allowed if the possible future event sequences are compatible.

As an example a mail system is chosen, and the process of updating the module structure is illustrated, along with transferring the live component state.

3.2.2. Kramer and Magee: Conic → REX → Regis. *Quiescence* [KM90] is undoubtedly the most influential contribution with respect to understanding the theoretical aspects of reaching a reconfigurable state, i.e., a state in which no communication is pending for a reconfiguration candidate node. This will be discussed in Sect. 8.1.4.2.

As a framework, CONIC [**KM90**], and its successors REX [**MKSD90, MKSD92**] and REGIS [**MDK94**], are based on research dubbed "configuration programming", a term similar to architectural programming [**ACN02a**]. All the frameworks are targeted at understanding the necessities of this kind of programming, and the examples are highly artificial: CONIC discusses the well-known example of the "evolving philosophers", a variant of the "drinking philosophers" [**CM84**] due to Chandy and Misra; REX provides an extended example modelling a doctor's office, but the dynamism is not stressed here; instead, great emphasis is placed on the description of the static structure. REGIS provides a badge system example. These examples are definitely non-trivial, but state is not discussed for reconfiguration (although REX does discuss local component state for static structures).

Reconfiguration (called "dynamic configuration") is triggered by the request of other components. In the badge system example of REGIS, `badge` components mirroring the physical arrival of new badges (infrared-enabled devices to be carried around in a building) are instantiated by the code of a master component. The reconfiguration actions are specified in the DARWIN specification language [**MDK93**] (cf. Sect. 3.1.3.1), keeping them separate from the actual component implementation.

3.2.3. POLYLITH. POLYLITH [**Hof93, HP93**] by Hofmeister and Purtilo can be regarded almost as influential as Kramer and Magee's work. Although the work was published 10 years after Bloom's work on ARGUS, POLYLITH is found as the pioneer work's reference in most papers in this field. Like the work of Kramer and Magee, a more theoretical interest is taken; the examples are mostly easy examples, including the evolving philosopher's example of [**KM90**].

POLYLITH operates a *software bus* [**Pur94**], which is similar to CORBA's object request broker, in that it takes the sole responsibility of transporting the communication between components, which can be implemented in different languages. Using a set of fine-grained directives, components can be blocked from communication with this bus, making them available for reconfiguration.

With CLIPPER [**AHP94**], an abstraction language is provided that hides the blocking from the developer (although no implementing algorithm is provided). It provides a `dumpstate` directive that is used to write one component's state to another component using a common interface.

3.2.4. FLEXIPHANT. Alvin Lim's FLEXIPHANT framework [**Lim93, Lim96**] targets complex systems in a heterogeneous environment. The example he introduces is a manufacturing facility, with components representing physical entities or hierarchical groupings of components. His interest is in updating the software of such components for live systems. Finite state machine abstractions are considered, with the abstraction of composite components being the product of the sub-component's abstractions. This gives rise to a definition of *legal paths*, which are used to define correctness criteria for reconfiguration.

Reconfiguration is conducted following a *schedule*, which is calculated before the reconfiguration commences. This schedule ensures correctness of the reconfiguration when executed by a control server.

3.2.5. SIMPLEX. The SIMPLEX system [**SRG96**] provides hot code update for commercial-of-the-shelf (COTS) components (although no specific component model is described). Much emphasis is placed on the *evolution* of systems – i.e., their subsequent improvement by new versions. The important feature of SIMPLEX is that it is fault-tolerant with respect to these new versions by employing a n-version scheme: Instead of *replacing* a component by its update, both are run in parallel for some time, until the safe operation of the update is confirmed. The

criteria for this is given by a user-supplied monitor. This results in a lengthy reconfiguration process, which is eventually concluded by replacing the old component by the new one, only after which the next update may commence.

This work is fairly abstract and only reports on the concepts used (including a brief description of state transferal, which, as it reads, is one of the few examples of guided direct state transferal), but it reports on two examples where hot code update was done in front of an audience that was invited to introduce bugs into the update component's code[1].

3.2.6. Hercules. Being quite similar to Simplex, Hercules [CD99] provides fault-tolerant hot code update for components. The approach differs from Simplex in that a more refined idea on how a component update should work is defined (it should patch errors), and in the supply of the correctness criteria (in the form of a formal specification, although a description of the specification method is not given). The idea is similar: An *arbiter* acts as a facade to a number of different components, and for any request queries each in turn. The responses are then compared to the formal description. Statistics are built for each component, and eventually, an engineer can choose which components to remove and which ones to retain.

As an example, a three-step evolvement of a car steering algorithm is presented.

3.2.7. Corba extensions. Corba [OMG06a] is extended to dynamic reconfiguration capabilities in a number of works. Of course, the utility of extending the leading framework for component-based software development is obvious, yet all works tend to become a description of the Corba-specific problems that need to be overcome. Nevertheless, just the fact that Corba can be extended to adaptive behavior underlines the capability of this model, and maybe component-based programming in general.

3.2.7.1. *Bidan et al.* Their Corba extension [BISZ98] aims at long-running applications, and considers both program-triggered reconfiguration and external maintenance reconfiguration (although this is no further investigated, so we do not represent this in Tab. 3.1. This is the first work (at least in this list) that stresses the importance of separating reconfiguration from application specification. Consistency (i.e., non-observability) of the reconfiguration is considered, and an algorithm employing local blocking of components under reconfiguration is presented. However, no example other than a highly abstract one is given; this example is used for obtaining measurements of real-time behavior of the approach.

3.2.7.2. *dynamicTAO.* The dynamicTAO object request broker (ORB) [KRL+00] try to provide a framework for adapting software when its environment – e.g., the available network bandwidth of a mobile computer – changes. The TAO ORB is then extended to provide reflection and reconfiguration capabilities. A rather massive framework for planning and conducting the reconfiguration is presented, which is mostly concerned with technical aspects like loading the implementation from a repository. No example is provided, but some experience with a very large multimedia streaming system is reported.

[1] In a personal note, this work is my favorite among all those listed here, because of these examples. Also, this work addresses one issue I cannot believe to be not of importance, although nowhere else this issue is discussed: the risk of introducing malformed code during updates. It even has a name: The *upgrade paradox*.

3.2.7.3. LuaSpace. Another Corba extension, defined for the language Lua, is LuaSpace [BR00]. This is a collection of tools, most importantly the LuaOrb object request broker. This ORB is highly dynamic, as it profits from the underlying language Lua. Also, a *generic connector* is used for dynamically choosing components for processing a task. Since this is very dynamic anyway, reconfiguration here only addresses the inclusion of new components, making this approach more service-orientation-like than most other frameworks. In an artificial example discussing "nurse" and "bed-monitor" components, reconfiguration is also described as a way of tuning the dynamic component selection algorithm.

3.2.7.4. *Eternal Systems.* Tewksbury et al. [TMMS01, MMST02] extend Corba with fault tolerance to support "eternal systems" – systems that must never cease to function. The focus is placed on hot code update, but with a remarkable twist: Instead of just enabling the update of existing component's code, they maintain both the old and the new component in order to avoid decreasing system performance or introducing bugs while updating to a new software version. Interestingly, this is one of the few works that does not rely on local blocking, but rather takes the approach of doing the reconfiguration in a preparation and post-reconfiguration phase, with an *atomic switchover* handling the actual rewiring in between. The reconfiguration itself is mostly concerned with deriving the correct updates of the state and the interfaces (even the case where calling code needs to be modified in order to work with a new version is considered, which is an obvious departure from the usual way of regarding the interfaces as fixed). However, even the atomic switchover cannot always be done (or its prerequisites fulfilled), hence quiescence is considered in order to reach a reconfiguration-compatible state. The example, however, is restricted to the update of a counter class.

3.2.7.5. *Wegdam et al.* In this work [AWSN01, Weg03] a framework for highly reliable systems with the ability to be updated without stopping them is described. As for the other Corba extensions, the actual reconfiguration framework remains fairly generic, and the theoretical consideration of consistency preservation carries over to reconfiguration with a broader application range. This work is particularly interesting, as design criteria for dynamic reconfiguration are given:

- Correctness – the system should ensure that code updates indeed work as intended, and that assumptions about the communication (expressed in the definition of *mutual consistent states*) and state invariants are preserved.
- General suitability – a broad applicability of reconfiguration should be provided.
- Minimal impact on execution – reconfiguration should be cheap with respect to computational overhead, and disrupt normal computation as little as possible.
- Maximum transparency – reconfiguration should be of no concern to the application developer.
- Minimum impact on the framework – if, like in this work, a framework is to be extended, the extension should be minimal invasive.

The work proceeds to discuss the architecture that is employed at a practical level and considers the impact on realtime behavior, but does not give an example.

3.2.7.6. SwapCIAO. SwapCIAO [BNS+05] is a Corba extension that establishes quality of service (QoS) functionality. The work introduces a rather extensive case study, where a *inventory tracking system* is modeled. Components are given by software controllers of physical entities, like a conveyor belt, and reconfiguration

follows the physical changes of such a system, e.g., the addition of a machine, or the software update that needs to be done in order to work with an improved version of an existing machine. The paper is very technical, and mostly discusses problems that need to be overcome in order to use their approach within the case study.

3.2.8. SOFA/DCUP → SOFA 2.0. DCUP components are a specialization of SOFA components, which provide a clear interface that can be utilized for reconfiguration [**PBJ97, PBJ98**]. Each component is composed of a permanent and a replaceable part, and the replaceable part (which, hierarchically, can contain other components) is subject to reconfiguration. Much emphasis is placed on the decoupling of various aspects of reconfiguration to different managers, which is illustrated by short example of a bank implementation that is hot updated.

In the SOFA 2.0 component model, whose design is influenced by the experience obtained with SOFA/DCUP (cf. Sect. 3.1.2.1), reconfiguration is inherently made possible [**BHP06**], but restricted by two patterns [**HP06**]: The "Nested Factory pattern" and the "Utility Interface pattern". The purpose of utilizing such patterns is to avoid *architectural erosion* [**BHH+06**], i.e., losing aspects of the component setup design due to repeated introduction of components that do not adhere to it.

The Nested Factory pattern describes how new components and their connections are to be established in the hierarchical component setups of SOFA. The Utility Interface pattern describes the extraction of components from the normal hierarchy, which makes frequently used components available to other components directly. Component removal is also permitted in SOFA 2.0.

As SOFA 2.0 uses first-class connectors [**BHP06, BMH08**], the technical process of reconfiguration is mostly concerned with detaching and attaching these connectors [**BHP07**]. SOFA 2.0 is one of the few frameworks where actual components request reconfiguration, which can be attributed to the strong emphasis of a hierarchical structure. State retainment is not discussed, although the versatile controller structure might be accommodated for that task.

3.2.9. OpenCOM. Building on Microsoft's COM component model, OpenCOM [**CBCP01, CBG+04**] provides provisions for reconfiguration. These include an explicit description of inter-component requirements, and specialized *receptables* that act as connectors between components, providing locking provisions and monitoring facilities. By requiring components to implement special interfaces, the provision of reconfiguration-relevant code in the components is ensured. The component framework is very technical, and most of the problems (e.g., how a connection can be reassigned) stem from peculiarities of the actual implementation.

For ensuring safe reconfiguration, receptables can be blocked; if they are blocked, any request will be denied with an error message. This approach is unique, as most other frameworks simply delay the communication instead of prohibiting it. In [**CBCP01**], some performance measurement is given. [**CBG+04**] introduces OpenCOM v2, which provides a similar approach to reconfigurability (although the blocking mechanism is no longer discussed). In this work, the three-fold separation of roles ("development programmer, system programmer and meta-system programmer") is presented. OpenCOM serves as a starting point for two interesting extensions towards reconfigurability: ReMMoC [**GBS03**] and Plastic [**BJC05**], both discussed below.

3.2.10. Cactus. Cactus [**CHS01**] emphasizes *graceful adaptation* for component systems. Components here are understood as the parts of the framework itself, which is a layered structure for distributed applications. A component might then be responsible for providing reliable message passing in the transport layer,

and should be reconfigurable if a different algorithm is preferred. Reconfiguration is triggered by monitoring (*change detection*); the possible reconfiguration candidates are evaluated using a fitness function and accumulated information about their behavior. If a local decision is made to proceed with a reconfiguration, a global agreement of all nodes in the distributed system needs to be obtained. Finally, the reconfiguration commences, using a three-step process that ensures proper synchronization. Interestingly, this is one of the few works that explicitly discusses the need to transport unprocessed messages to replacing components.

Since the focus of this work is precise, it is straightforward to give an example. Here, a total ordering protocol that ensures the retainment of message sequence ordering is investigated, and a reliable transmission protocol that can be reconfigured to respond either with ACK or NAK (acknowledged or not acknowledged). They report on competitive results.

Overall, this is a remarkable work, if only to show a field where reconfiguration can be employed successfully. The narrowed focus helps to provide an elegant way of checking the need for reconfiguration by calculating the suitability of different setups, an approach that can hardly be generalized.

3.2.11. Willow. The Willow Survivability Architecture [**KHW+01b**, **KHW+01a**] is a component framework that tries to establish security against attacks by *reactive* and *proactive* reconfiguration. This distinction is quite interesting: Proactive reconfiguration, also known as *posturing* reacts to some possible, but not yet immediately effective threat by tightening the security and disabling superfluous services; while reactive reconfiguration tries to remedy security breaches that have already occurred.

Proactive reconfiguration is triggered by human interaction. For reactive reconfiguration, Willow employs a control loop that uses sensors to measure the systems state. a diagnostic component named Raptor maintains a set of state machines that specify valid behavior and compares the measured data with them. Reconfiguration requests are then prioritized to avoid conflicting reconfigurations taking place at the same time – this problem is quite unique to Willow, as most other frameworks use centralized control.

The case study presented is an implementation of the U.S. Air Force's Joint Battlespace Infosphere concept, which consists of a number of coarse-grained components, i.e., applications that store data into a common network, where subscribers can obtain these data. Attacks can be physical or virtual, or they can originate from bugs within the system.

3.2.12. ArchJava. The ArchJava framework [**ACN02b**] provides basic reconfigurability by providing component creation, connection and reconnection commands to the assemblies (cf. Sect. 3.1.2.3). As reconfiguration is not the concern of ArchJava, consideration of atomicity, state retainment or plan description are not given. An interesting aspect of ArchJava's reconfiguration approach is that it does not allow explicit component removal. Instead, components are removed if all their connections have been moved away. This approach is modeled after the garbage collection mechanism of Java. Obviously, such an approach does not cooperate well with state retainment, and ultimately relies on the single-threadedness of ArchJava.

3.2.13. Gravity. The framework Gravity [**HC03**, **CH04**] is situated in the context of service-oriented computing, but is itself component-based, thereby providing an approach to *context-aware computing* [**SAW94**]. The basic idea is to

dynamically use components that become available, and to cope with the disappearance of components. Such changes of availability are triggered by the environment, which is more common to service-oriented computing.

In [CH04], an interesting differentiation between components and services is made, with describing the latter as more dynamic, with changes to the available services at runtime, while components are mostly static. While we fully agree in this thesis, this is a remarkable departure from those works concerned with hot code update, which considers components to be available dynamically; of course, a major difference is that dynamically arriving components will rarely provide genuinely new functionality that can be harnessed.

Gravity is supporting a number of dynamic component reconfiguration scenarios that are triggered by dynamic environment changes (and "new" components are referred to as "arriving"). Like for services, great emphasis is placed on automatically discovering and choosing suitable components. As an example, a web-browser is discussed where plug-ins for different document types can be installed ad-hoc.

Gravity is interesting because it offers a different view on reconfiguration that integrates the well-researched service-oriented programming with component technology. In doing so, some assumptions differ very much from other works, like the existence of explicit communication contracts. In [CH03], the application of Gravity's approach to OSGi is detailed; OSGi supports flexible loading of services already and is further enhanced by the Gravity approach.

3.2.14. ReMMoC. The "Reflective Middle-ware for Mobile Computing" (ReMMoC) [GBS03] extends the OpenCOM framework [CBG+04] by providing adaptive protocol choice for service access for mobile devices. The idea is to reconfigure the service access; e.g., if a service can be used by SOAP calls at one location and by publish-subscribe at another, the software should just reconfigure the communication part. This is done to keep the memory impact of supporting multiple communication means low – an interesting and unusual (yet historically relevant, cf. the now ancient method of overlays [Pan68]) argument for reconfiguration. The service description is obtained by WSDL files, and is used to lookup appropriate components for the available communication means. Experience with an experimental setup is reported, but ReMMoC's most important contribution (as perceived from the viewport of this thesis) is the description of a genuine need to do reconfiguration for a cross-cutting concern – namely that of communication protocols. This is similar to NeCoMan [JDMV04] and Cactus [CHS01], thought the level of abstraction is considerably higher.

3.2.15. Planit. The focus of Planit [AHW03] is placed on planning the reconfiguration rather than conducting it. Its role is seen as a producer of reconfiguration plans, among which one can be chosen for reconfiguration. We nevertheless mention it here, as many other aspects of reconfiguration of component systems are mentioned in [AHW03] as well.

Planit plans the distribution of components on various machines, and conducts a re-planning if components or machines fail. It thus seeks to reestablish the component graph. A plan describing the necessary reconfiguration steps is then calculated. For finding a better plan according to some metrics, further plans are also calculated and compared, until a timeout is reached.

The component model assumed by Planit, with components being restartable after unspecified errors to them, is quite restricted. Also, any consideration of functional properties is missing. The main contribution of this work is to consider automated reconfiguration plan generation.

3.2.16. OpenREC. OpenREC [**HW04**] is a framework for reconfiguring components that offers an interesting twist, as it is itself reconfigurable. Instead of choosing algorithms for the various aspects of reconfiguration – in [**HW04**], the problems of synchronization, state retainment and maintaining system integrity (i.e., planning the reconfiguration) are mentioned – the framework allows users to plug in their own algorithm, and even change it at runtime. OpenREC emphasizes the separation of the reconfiguration trigger (the *change driver*), the *reconfiguration manager* and the *application*. Also, the ability to reflect the component framework, i.e., obtain runtime information on the configuration from within the framework, is pointed out as an important ingredient for adaptive systems.

In [**WSKW06**], a formal verification method for checking the validity of a reconfiguration plan using the Alloy modelling language [**Jac02**] and its associated analyzer [**JV00**], using a similar, yet more automated approach than [**GMK02**].

The example provided in [**HW04**] is limited to simple component replacement and just tries to illustrate the capabilities of the model. In [**WH04**], a case study investigating a component-based router (similar to the Cactus framework mentioned above) is investigated, comparing two reconfiguration algorithms. The contributions of OpenREC are twofold: First, the framework is an interesting testbed to compare reconfiguration algorithms. Second, they point out the common elements of such algorithms, and thus provide an abstract view on the problem.

3.2.17. CASA. The aim of CASA [**MG04**] is to support reconfiguration of components in a dynamic environment. Comparing to frameworks like Cactus, which adapt lower-level services if physical changes require such an adaptation, they aim to adapt on a changed environment, which is perceived to be on a higher level: An example of a tourist guide application that needs to adapt to a changed location (by providing different information) is given. The process of adaptation is triggered by monitoring the environment and consulting *application contracts*. For actually adapting the system, four mechanisms are provided: Changing lower-level services, which is done by integrating with an adaptive middle-ware, aspect-oriented changes, changing attributes (an interesting point which is usually subsumed by component replacement) and component re-composition.

For component reconfiguration, a series of requirements is defined in [**MG05b**], which contain the availability of alternatives to the current configuration, and that any such alternative is consistent with the current configuration. CASA then takes a remarkable departure from the usual approach of doing the actual reconfiguration (which is only discussed as replacement of a single component): Instead of using *lazy* replacement, which lets the component finish the currently executed method, it tries to do *eager* replacement that suspends the component at a safe-point where an alternative component can proceed. The reconfiguration follows a small protocol that uses a *handle* object: Similar to a proxy, it queues the requests during reconfiguration and also serves for redirecting the connections of the retained components to the new one. During the reconfiguration, the state of the component, including the safe-point, is packaged and transferred to the new component.

In [**MG05a**], some experimental results are reported, but no actual example is provided.

3.2.18. Rainbow. The Rainbow framework [**GS02, GCH$^+$04**] considers the reconfiguration of applications built from reusable, but very coarse-grained components (in the example, web server clusters form components). A control loop that monitors and adapts the system from the outside is used, with *probes* measuring data and *gauges* aggregating these data to provide decision criteria for reconfiguration. Reconfiguration is then triggered by rules that are based on the gauges

output. It is carried out by *effectors*, but the actual algorithm remains vague; they are distributed over a number of tools that are integrated in RAINBOW.

For planning reconfiguration, great emphasis is placed on maintaining the *architectural style* [CGS+02]. This can be understood as an architecture classification; in one example the abstract idea of "client-connects-to-server" is called a style. Such a style then offers possibilities of monitoring (like bandwidth of clients) and adaptation (called repair strategies). This is an interesting idea that aims at abstracting from one concrete architecture and offering adaptability for a whole family of architectures (i.e., a style).

The examples provided are mostly examples of evolving network-distributed architectures like a video-conferencing system in [GCH+04] and an abstract client-server example in [GS02, CGS+02]. Other than the use of architectural styles, RAINBOW is an interesting framework in that it combines a large number of tools and languages. However, the resulting coarseness leaves a number of commonly discussed problems (especially the atomicity of reconfiguration) open.

3.2.19. PLASTIC. Combining the ACME architectural description language [GMW00] with the ARMANI extensions [Mon01] and the OPENCOM [CBG+04] framework, PLASTIC [BJC05] offers a complete framework for reconfiguration. ACME ADL is extended with statements that define the trigger of reconfiguration, the removal of elements – creation of elements is covered by standard ACME – and the existence of dependencies; some components cannot exist alone and their addition during reconfiguration needs to be accompanied by the creation of the components it depends on.

Based on a number of extended ACME scripts, a finite state machine is built that reflects the possible configurations and is used to decide how to reconfigure. This is called *programmed* reconfiguration. PLASTIC also supports so-called *ad-hoc* reconfiguration that is not defined at assembly time, but rather controlled by the specification of invariants that must not be violated. Reconfiguration scripts are then fed into the system, validated against the ADL and executed "transitional".

As an example, the replacement of a decoder in a video streaming application is presented. If network bandwidth is insufficient, a different decoder can be utilized (programmed reconfiguration). Also, the user can submit a script that replaces a buffer (ad-hoc reconfiguration).

3.2.20. DEVS. The DISCRETE EVENT SIMULATION (DEVS) framework [HZM05, ZBZ07] is a simulation framework for a variety of models that adhere to the discrete event simulation paradigm (which imposes that the model advances in discrete steps, similar to what is done in model checking, opposing the continuous models often employed for physical simulation). Simulation models are comprised from a number of components, which can become connected and detached during simulation; also, the set of models can be modified. DEVS is itself very technical, providing a number of distribution approaches (e.g., Peer-2-Peer-based [CSPZ04]). A variety of examples of simulation models is given, e.g., a robot scenario where one robot follows another robot in a convoy setup [HZ04].

The work is interesting in the context of this thesis because, being in the regime of a globally synchronized, step-based execution model, many of the usual problems with reconfiguration (especially the synchronization issues) do not have to be considered. Also, the approach of having reconfiguration requests being made by the components itself is quite unusual; it gives an example for a system where the separation of roles is not given, but which still utilizes (hierarchical) components as a means to structure the simulation model.

3.2.21. GridKit. The GridKit framework aims at building adaptive sensor network systems. Sensor networks are ad-hoc networks, and as such need to accommodate to changing environments; this is a vast area of research. Here, reconfiguration is based on a control loop and the modification of component networks. This reconfiguration can remain local, or become global if it stretches over multiple nodes. All nodes involved (respectively the components they host) need to be blocked during reconfiguration for achieving quiescence. Reconfiguration is planned in advance; configurator components are defined to respond to events by selecting a plan and executing a reconfiguration of the component graph. As an example, a sensor network for detecting floods in river valleys is presented [HGP+06], with a practical implementation at model scale. Reconfiguration is used to switch the network topology from a power-saving configuration to a more robust one if a flood is predicted.

3.2.22. Java/A. The component framework Java/A [Hac04, BHH+06] supports reconfiguration by a provided API that offers methods to add or remove connections and components. Reconfiguration can be realized by writing code that invokes this API. In [BHH+06], reconfiguration is described as guarded by a `try/catch`-block, which is notable, as the possibility of reconfiguration failure is usually neglected. The example given is the dynamic addition of new automated teller machines to a bank, which amounts to reconfiguration following physical change.

3.2.23. Adapta. Providing means for implementing adaptive applications in Grid systems, Adapta [ASS07] is a framework that provides monitoring, event notification and reconfiguration. Like OpenREC, Adapta is itself reconfigurable and allows for pluggable reconfiguration algorithms. The monitoring capabilities are focused on the host machine's resources, as changes of these resources require a re-tuning of the Grid application. The reconfiguration is planned by *Adapta Component Configurator* objects, which basically implement the Strategy pattern. Also, the state transferal during reconfiguration is handled by these objects, as is the synchronization. Adapta provides a case study that extends a Grid framework by automating the process of selecting the number of replicas (i.e., duplicates of components on different machines to avoid loss of computation results if a Grid node shuts down) and updating of parameters.

3.2.24. Draco. Vandewoude's Draco [Van07] is a framework for hot code update of components. This work extensively considers how state can be transferred from old to new components. Since the topic of hot code updates narrows the state transferal scenarios, (if a component is updated, fields with the same name can be expected to capture the same data; if a component is replaced, maybe even by more than one component or with a component that has a different purpose, such target identification is much harder.), a number of heuristics, called *harvesters*, are employed to automatize the state transfer. For reaching a safe state, *tranquility* [VEBD07] is used. Draco is part of the Fresco methodology [VB05], which aims at supporting the programmer in devising direct state transferal.

3.2.25. Fractal. One of the most well-known scientific component frameworks, Fractal [BCL+06], also supports reconfiguration [DL06, LLC07, PMSD07].

In [DL06] FScript is presented, which is a domain-specific language for reconfiguring Fractal component graphs. In [LLC07], the process of reconfiguration is investigated. Interestingly, they discuss reconfiguration to be either *structural* (which resembles almost all the reconfiguration approaches discussed here) or *behavioral*. Also, a unique feature is the utilization of the ACID principle, known

from database transactions, not only for ensuring that concurrent communication does not interfere with the reconfiguration process, but also to allow for rollback of failed reconfiguration.

Reconfiguration is composed of a series of *introspection* and *intercession* operations, with the latter modifying the system, and the former merely investigating it, acting as a guard to subsequent intercession operations. Intercession operations are given an inverse operation to undo the effect during rollback. Consistency is ensured by FPath constraints on various levels of detail. For isolation, a two-phase locking protocol is used, with an added deadlock detection. Finally, logging is used to obtain durability, as it is done in database systems.

In [**PMSD07**], experience with reconfiguration using Fractal and FPath is reported. Using Think as the Fractal implementation, an embedded systems operation system is created. Clearly, implementation issues dominate this work, and hardly any report on actual reconfiguration is given. State transfer is mentioned, but barely discussed, other than that it is completely left to the user.

Another approach building on Fractal is the Grid Component Model (GCM) [**BBGH08**], which aims at providing flexibility for Grid systems. The focus of this work is placed on using formal verification of the reconfiguration and possibly rejecting reconfiguration attempts that might lead to problems.

3.2.26. Jade. Another component framework building on the Fractal model is Jade [**TBB+05, PST06**]. Jade aims at building self-manageable systems. For doing this, a control loop is utilized, which distinguishes three stages (cf. Sect. 8.1.1.1 for a discussion of these stages):

(1) The detection of events by means of sensors,
(2) an analysis of the events observed, with a decision on how to reconfigure the system,
(3) and finally, the utilization of actuators to implement the reconfiguration.

In Jade, this control loop is built from Fractal components, while the target of the reconfiguration itself is much coarser. This is why Jade does not need atomicity considerations, since reconfiguration amounts to restarting databases or adding server replicas to a load balancer. As an example, self-management of a J2EE application comprised of a number of servers is discussed; Fractal component wrap these servers and control them. A number of experiments with this setup is discussed, including the consideration of quality of service requirements exemplified in the dynamic growth of a cluster of web servers.

3.2.27. Java extensions. Many reconfiguration-enabled frameworks are built using the Java programming language, e.g., Java/A [**Hac04**]. Java seems a good choice, because it supports introspection and dynamism to some extent, but is rigid enough to restrict reconfiguration abilities to a manageable set of options (e.g., by hiding the state of components so that the state transfer needs to be made explicit). Here, we will discuss some extensions of component frameworks that are already written in Java.

3.2.27.1. Bark. Extending the Java EJB framework, Bark [**RAC+02**]. In [**RAC+02**], an interesting differentiation of *reconfiguration*, *adaptation* and *updating* is proposed: Reconfiguration aims at following changes of the operational requirements and is hence “driven by external pressures”, whereas adaptation tries to maintain the application’s integrity and this is “driven by internal pressures”. Updating refers to installing and running new configurations.

As for many of the reconfiguration frameworks building on existing component frameworks, Bark is mostly concerned with technical problems of fitting reconfiguration into the given structures. An interesting example is given: A directory synchronization software, which has components that are connected both locally and remotely on different servers. The reconfiguration then copes with the arrival of new directories or data channels, which makes for a very fine-grained reconfiguration approach. Also, reconfiguration following software extensions is discussed.

3.2.27.2. COMPAS. Also operating on EJB container, the COMPAS framework adds monitoring functionality to EJB applications. The monitoring aims on finding performance problems, and much work is invested into pinpointing the source of performance loss. The application is then adapted by switching components for other, non-functionally different versions (so-called *redundant components*). Adaptation needs to find a better configuration, which is a nontrivial problem; here this is solved by annotating the replacement candidates with information about their performance (which is obtained from earlier monitoring), and *decision policies*, which are sets of rules, are used to find a suitable substitute. Adaptation then commences by atomic switchover, which utilized the EJB-provided request indirection. This also provides sort of quiescence, since transactions are not interrupted, the authors present this as a warrant not to consider state transferal.

3.2.27.3. *Kefti et al.* Building on the Java module framework OSGi and the BeanBuilder GUI (cf. Sect. 3.1.1.3, Ketfi et al. discuss two reconfiguration frameworks in [**KB04**] and [**KBC02**]. Their approach, however, is limited to simple algorithms on how to reconnect components, not considering any consistency and not providing any examples (thought some example scripts are shown in [**KByC02**]). Their main contribution is the introduction of a replacability relation, which is presented as a graph with its edges annotated with mapping scripts. Still, [**KByC02**] provides an interesting classification of adaptivity.

3.2.27.4. *Chen and Simons.* In [**CS02**], a component model building on Java and its remote method invocation (RMI) technology is described. The reconfiguration supported is either hot code update of components, or the reorganization of distribution of components. This is an interesting field of reconfiguration; especially since state retainment is more straightforward. [**CS02**] discusses the technical difficulties of reconfiguration based on RMI: Finding a quiescent state, and consistently rewiring components. Sadly, neither a discussion of reconfiguration causes nor an example are given.

3.2.27.5. NeCoMan/JBoss. NeCoMan [**JDMV04**] is a framework for dynamic weaving of aspects in distributed system, which was implemented on top of the JBoss EJB application server to provide adaptability [**JTSJ07**]. Distributed aspect weaving is a special form of reconfiguration that is restricted to modifying *cross-cutting concerns* with respect to the remote connections – as an example, they provide the addition of a fragmentation filter that fragments messages on the server and reassembles the parts on the receiver. The challenging task here is to find a safe state for weaving in the aspects such that the communication is not interrupted, i.e., quiescent states.

The actual algorithm consists of a series of steps that turn out to guarantee that a quiescent state is reached and that components are connected such that reconfiguration proceeds without interfering with the running system. Incoming messages are send to the new component, which does not yet process them, such that the old component can finish its current operation and then be removed. Thus, only the rewiring has to be made atomic.

By restricting on a special branch of reconfiguration, any conceivable "do-undo" filter combination (encryption, compression, etc.) serves as an example. This is similar to CACTUS [CHS01], but stresses the separation of concerns even more by using aspect orientation. Other frameworks for distributed AOP exist, e.g. [TT06, TJSJ08], but they fit less well into the perspective of this thesis.

3.2.28. REDAC. REDAC [RP08] is an algorithm that realizes reconfiguration for multi-threaded applications. The authors claim to have developed this in the context of a remote laboratory environment, but do not pick up that example again. The algorithm discusses components (called *capsules*) that are made out of a series of objects. For reconfiguration, a *blocking set* needs to be identified that contains all components that are concerned with the reconfiguration. For this blocking set, a quiescent state needs to be reached, which is done by blocking; transactions extending over multiple messages are not considered.

REDAC is remarkable in two aspects: First, they are the only work we are aware of that discusses model checking of reconfiguration employing a Petri net model checker, although the formal basis is not presented. Also, an extensive discussion of state retainment is presented in [RS07a]. There, an algorithm for automatically transferring the state of capsules is presented; which amounts to a recursion of the object graph. While this is certainly not suitable for generic situations, it is one of the few works where an explicit algorithm is provided.

3.2.29. CoBRA. The CoBRA framework [IFMW08] is described in the context of service orientation, but since their service implementations are actual components, as and they are well-aware of their proximity to component-based reconfiguration, we include this work here.

The reconfiguration is easy and entails only addition, update and removal of single services. This is made up by a profound architecture for adaptation processing, state retainment and consistency preservation. The latter is achieved by introducing *protection proxies* that indirect communication, block adaptation until all communication is over, and block service requests while reconfiguration is pending or underway. Although no example is given, this work contributes an interesting view on reconfiguration if powerful plans are neglected in favor of a clean process. It also considers the state transferal in terms coined by agent programming, and describes the approach (which is, in this regard, identical to all others described here) as *weak state migration* (cf. [IKWK00]), since only the data part of the state is transferred, and the stack, registers etc. are not considered.

3.2.30. MADAM/MUSIC. Building on and extending the MADAM framework [HFS05, AEHS06], MUSIC [REF+08] tries to plan reconfiguration to improve an applications quality of service. The basic idea is to assemble functionally consistent plans and rank their utility. Similar to PLANIT [AHW03], the focus of this work is placed on the plan generation rather than its execution. Reconfiguration execution is discussed nevertheless, providing one of the few approaches where global blocking is employed.

For plan utility reasoning, the domain of service oriented architecture (SOA) is used, most notably by utilizing the idea of service-level agreements (SLAs). In generating the plans, remote services (and their SLAs) are also considered. The description of the actual utility ranking of plans remains fairly vague; but a utility function for a case study is given.

3.2.31. Direct vs. Indirect State Transferal. For those frameworks that do consider the state, two approaches can be distinguished: Either, the reconfiguration manager (i.e., an external entity that controls the reconfiguration process) knows

how to transfer the state, extracts the data from the old component and inserts it into the new one, or the components itself know how to encode or serialize their state, and how to decode it again, possibly from an old version, in order to adopt some other component's state. Following Vandewoude [Van07], the former approach is called the *direct approach*, and consequently the latter the *indirect approach*. Vandewoude uses a slightly different definition that defines an approach where the old component is queried by the new component directly, but does not provide provisions for it, as direct as well:

> *Direct State Transfer: The implementation of the old version is used directly. Either the updating mechanism or the new version is responsible for extracting, interpreting and in most cases converting the information contained in the active version.* [Van07]

Hence, the distinction between a direct and indirect approach is given by the old component and whether it provides provisions for externalizing its state during reconfiguration. In our perception, however, the target component is considered. This is because we perceive it as less likely to have a component provide setter methods for all (relevant) parts of its state than to have a component provide complete read access.

It is sometimes hard to differentiate the two approaches; we will judge all those as indirect where a composite value is passed to the target component and this component is required to understand the semantics.

Framework	**State transferal**	**State transformation**
Argus [Blo83, pp. 109]	indirect	accessor functions for the state, which is returned as a record, user-defined conversion functions can be used for non-1:1-mappings
Polylith [Hof93, pp. 24]	indirect	invocation of a `mh_objectstate_move` method to have the old component send the state over a special connection; the example uses a 1:1 mapping, but theoretically arbitrary mappings can be employed
Lim [Lim96, pp. 191]	indirect	saving states to a state location `loc`, which is accessible by a method `restore(pid, loc, f)`, with `f` being a state transformation function, which needs to be supplied by the designer
Bidan et al. [BISZ98]	indirect	reconfigurable objects need to implement an interface `RO_Object`, which defines methods `writeState` and `readState`, which operate on a `RO_state` instance
SOFA/DCUP [PBJ98]	indirect	component state is externalized to a named location (only briefly discussed)
Tewksbury et al. [TMMS01, pp. 9]	direct, fallback to indirect	a direct transfer based on variables bearing the same name is attempted (at compile time, as the code is generated); if this is insufficient, a user-defined conversion routine is employed

Framework	State transferal	State transformation
Wegdam [**Weg03**, pp. 103, 167]	direct (theory), indirect (implementation)	a *state translation function* is employed for the theoretical consideration, but the actual implementation employs CORBA structures, which amount to the indirect approach of encoding the state in a special object
CASA [**MG05b**, pp. 133]	indirect	`loadState` and `storeState` methods are required, which need to convert the state into a "standard representation"
DRACO [**Van07**, pp. 43, 120, 158]	direct (both variants are discussed)	a series of strategies to identify matching data fields of old and new components is presented. These *harvesters* operate on the class structure of the component in question. The focus is placed on live code updates, so a number of constraints can be defined. The problems of semantical data changes is well discussed. As a backup, user-defined (indirect) state transferal is possible
FRACTAL [**PMSD07**]	indirect	a `StateTransferController` is implemented by the components that grants access to the component's state
Kefti et al. [**KB04**, **KBC02**]	direct	user-defined scripts
Chen and Simons [**CS02**]	indirect	`extractState` and `restoreState` are provided by each component and accessed by the controller using a control interface
REDAC [**RP08**]	direct	a recursive algorithm [**RS07a**] is used for traversing the object graph, and thus extracting the necessary data. a non-1:1 mapping is done by a user-supplied mapping that is not specified further
COBRA [**IFMW08**, pp. 101]	indirect	uses the *Memento pattern* [**GHJV95**] to transfer the state; this pattern basically captures the idea of the indirect approach: The source component packages its non-volatile data into an object of a specially defined Memento class, which is then injected into the target, which may use the Memento's accessor methods to retrieve the state

Table 3.2: State-considering frameworks

Tab. 3.2 lists the approaches of the corresponding frameworks. Those that attempt a direct transferal usually use reflection to get an insight into the component's structures; while most works discuss fallbacks, they are either insufficiently defined or are themselves indirect.

Vandewoude [**Van07**] focuses on the transferal of state for component updates. His work is placed in the context of hot code updates, which gives some constraints on the data that need to be considered; a semi-automatic approach is presented

that employs a series of strategies to copy the state. Both the direct and indirect approaches are discussed, and the direct one is chosen because of its better suitability. The ReDAC algorithm [**RP08**] also employs an elaborate approach towards state transferal. Using an object-graph-traversal algorithm [**RS07a**] and a mapping, they realize a semi-automatic, direct approach; however, the practical applicability is very dependent on the problem at hand.

The indirect approach requires components to implement encode and decode methods. The state can then be obtained from an old component and injected into a new one. Lim [**Lim96**] employs this indirect approach. The work is otherwise similar to ours in that it proposes the employment of *schedules* to guarantee successful reconfiguration. [**BISZ98**] and [**TMMS01**] both enhance Corba with the ability to do stateful dynamic updates. The former uses an indirect approach by requiring state accessor methods in reconfigurable components, the latter work uses a mixture, where the direct approach with a 1:1 mapping of variables is tried first, with a fallback to the indirect approach.

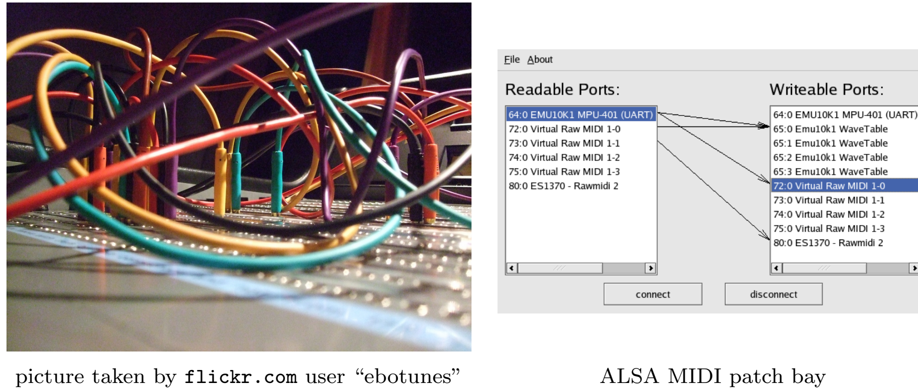
3.3. Formal Approaches to Adaptive Component Systems

Very few component frameworks seek to describe the operation of component applications in a formal way. Work has been done on the communication behavior of components in a framework, e.g., for SOFA [**PV02, AP05**]. Java/A has a strong formal backing [**BHH+06, KJH+08**]. We are not aware, however, of a formal description of reconfiguration in a component framework. The work on quiescence [**KM90**] and tranquility [**VEBD07**] covers an aspect of reconfiguration, and extensive work has been done on describing sound reconfiguration plans [**WF99**]. There are some calculi, however, that cover reconfiguration, but lack a direct connection to a component framework.

The ASP calculus [**Car07, CH05**] (Asynchronous Sequential Processes) is a calculus for asynchronous communication of entities that can be understood as components. Strong emphasis is put on the concept of *futures*. This calculus defines the formal basis of ProActiv, which is an implementation of the Grid Component Model, an extension of Fractal (cf. Sect. 3.1.2.2). These frameworks bear much resemblance to JComp: Mono-threaded components, utilization of the Active Object pattern, no shared memory between components. Reconfiguration only plays a minor role in this calculus, and mostly in the form of *migration*, i.e., the relocation of an active object. Reconfiguration is investigated in the context of process networks, which can be translated to the ASP calculus, thus allowing for reconfiguration on a higher level.

The λ_χ calculus [**SC06**] extends the λ calculus by provisions for defining, instantiating and reconfiguring components. Among those reported here, it uses the notion of components most directly, making communication requirements and supplies explicit by means of interfaces. It is also closely connected to an actual programming language, ComponentJ [**SC02**], which extends Java. The basic calculus of [**SC00**] is modified and extended to the λ_χ calculus in [**SC06, Sec06**]. A type system is provided that ensures atomicity of reconfiguration, meaning that reconfiguration either progresses to completion or does nothing at all. Sadly, the results are not applied to ComponentJ.

For formally describing software updates, a number of formal frameworks have been devised, e.g., the Update calculus [**BHSS03**]. This calculus extends the simply-typed λ calculus; introducing versioned types to reflect different versions of modules. The compatibility of an update can then be reasoned about within the type system.



picture taken by `flickr.com` user "ebotunes" ALSA MIDI patch bay

Figure 3.2: Patch bays, in real life and virtual

The R calculus [**FZ06**, **FZ07**] is an example for module calculi, used for describing modules and their configuration. Reconfiguration is obtained by applying module operators (connection, renaming, etc.) to configurations, thus interleaving normal execution with reconfiguration steps. Building on this, various aspects of reconfiguration can be investigated, e.g., deferred reconfiguration that is only executed if a connection involved is actually used [**AFZ05**, **FZ06**] or the introduction of external module definitions [**FZ07**].

3.4. Special Aspects and Applications of Reconfiguration

Reconfiguration is a very broad field; and many frameworks supporting it are not concerned with components, although the differentiation is often subtle. In this section, we list some frameworks that are concerned with adapting systems at runtime, but are not included into the reconfigurable component framework list; either because they do not directly address components, or use components in a way that is different from ours. This holds particularly true for the first selection of frameworks, which target the domain of real-time systems.

3.4.1. Real-Time Frameworks. Real-time frameworks are built to ensure some real-time properties of an application, e.g., a bounded delivery time for messages or a guaranteed throughput of components. Save for CADDMAS, the frameworks reviewed here are concerned with multimedia, where audio and video signals need to be delivered with a bounded latency and satisfying throughput.

Multimedia is a natural field for reconfiguration, as sources change (physically, or in their being interesting; a camera becomes interesting as soon as something unusual happens in its field of view) and since the data flow is more important than the control flow (cf. [**NGT92**]), which eases the finding of alternatives (it is easier to detect components that can consume a video feed than it is to find components that offer the same interpretation for a set of control commands). Also, the metaphor of "rewiring" is already well-known from multimedia systems, like patch bays in studios (Fig. 3.2 shows an example). Hence, there is no shortage in good examples, and also the triggering and planning of reconfiguration is easy. However, the runtime requirements become much more critical – reconfiguration must take care not to violate the realtime constraints too much. Such real-time requirements are beyond the scope of this thesis, but the frameworks provide interesting examples nevertheless.

3.4.1.1. Cinema. Cinema [**RBH94**, **Hel94**] is a framework for multimedia applications, with rather limited reconfiguration capabilities that amount to filter insertion. Reconfiguration (called "dynamic configuration") is triggered by user interaction, e.g., with a graphical editor [**Bar96**]. Since reconfiguration interrupts a system that needs to adhere to real-time requirements, a "quality" can be associated with reconfiguration. Cinema discusses this under the name of *QoS* (quality of service).

3.4.1.2. CADDMAS. Addressing general real-time signal-processing systems, CADDMAS [**SKB98**] clearly boasts the most impressive examples: Sampling of vibration sensors of the Space Shuttle Main Engine (SSME) beyond rates of 100KHz during engine tests. Although no further insight is provided into how this can be achieved and ensured, this work is remarkable because of an unusual approach towards planning the reconfiguration: Using a sort of model-driven architecture approach, alternatives to the initial model are defined. These alternatives are then grouped in a finite state machine, and state transitions correspond to reconfiguration.

3.4.1.3. Djinn/QoS Dream. Djinn [**MNCK98**, **MNCK99**], a framework that is geared towards "unnoticeable" reconfiguration by staying beyond certain latency times, thus preserving "smoothness". This work also addresses the issue of atomicity and compares it to database transactions. Djinn, which handles media streams only, is combined with a message passing architecture in the QoS Dream framework for adaptive multimedia [**MSBC00**], which provides a very interesting case study of an "intelligent hospital", which allows communication of doctors in a hospital regardless of their physical location (i.e., incoming calls are routed to their present location).

3.4.1.4. RDF. The Reconfigurable Data Flow (RDF) framework [**ZL07**] targets "stateless data flow systems", i.e., component networks where components act as filters on data flow, but without maintaining an own persistent state (e.g., encryption and decryption components operating on data streams).

Reconfiguration is described as having four distinguishable effects on the system:

(1) Functional update – new functions are added as intended,
(2) functional side-effect – problems might be introduced by reconfiguration (e.g., changed message semantics),
(3) logical influence on performance – plan execution might impact the performance of the running system, and
(4) physical influence on performance – computational overhead of the reconfiguration, affecting the overall system's performance.

The framework then seeks to minimize these two impacts, as well as avoid functional side-effects. In discussing how to avoid logical performance impact, the reconfiguration steps are divided in three classes:

- Type I – operations that do not have an impact on the running system, e.g., adding new components.
- Type II – operations which have impact, but are executed almost instantaneous, e.g., reconnecting components.
- Type III – operations that run for extended time and impact the performance, e.g., state transfer.

In order to avoid logical performance impact, no transferal of states is allowed in RDF, and the other reconfiguration steps are planned such that their impact is limited. Avoiding functional side-effect is done by versionizing the data. In the

example, an encryption/decryption framework, data is annotated with the version of the encryption algorithm that was used such that the proper decryption version (either the new or the old one) is used.

RDF is a unique in that considers quite a different reconfiguration execution than most other frameworks, where reconfiguration "sneaks into" the currently operating system without any major locking. This is similar to DJINN, but the application domain is more generic. Its reconfiguration scenarios are, of course, more limited than those considered by performance-unaware frameworks.

3.4.2. Legacy Systems. Legacy systems are systems that are too old to be modified (much) and too important to suffer a lengthy replacement period. Typically, these are systems deeply embedded in a business workflow, with ever increasing operation costs. Component systems can be used to wrap these legacy systems, which is a gentle approach towards evolving the overall IT infrastructure despite their existence [**BLWG99**]. Adding reconfiguration capabilities to such legacy systems may seem like a strange idea (given that legacy systems are perceived to be *unconfigurable* anymore), but the approach of *retrofitting* reconfiguration capabilities is interesting, as it neatly defines what a framework needs to provide in order to support reconfiguration.

3.4.2.1. DARWIN/LAVA. In [**Kni98**], reconfiguration of systems that are not implemented in a special, reconfiguration-enabled framework is discussed. Here, dynamic reconfiguration (or even, load-time reconfiguration of a system that has a fixed wiring) may require to add wrappers to existing components, leading to the so-called *self-problem*: If a wrapper delegates a call to a legacy component, any self-invocation of said component will bypass the wrapper, thus depriving it of a chance to modify the self-calls execution. Solving this problem is surprisingly difficult.

DARWIN[2]/LAVA proceeds to solve these problems. This solution is then carried over to component system, although this is more described as work in progress. We mention this approach here because it offers an interesting view on reconfiguration that is not anticipated at system *and* framework design time.

3.4.2.2. KX. KINESTHICS EXTREME (KX) [**KGK+02**] is a system for retrofitting existing (legacy) systems with adaptability. Large emphasis is placed on obtaining data from the running system, using *probes* (software that gathers data), *gauges* (software that accumulates said data), *controllers* (software that makes reconfiguration decisions based on the gauges output) and *effectors* (software that does the actual reconfiguration). This is quite similar to the RAINBOW [**GCH+04**] system described above, but there is no explicit consideration of the target system being constructed from components. Reconfiguration is used for two effects: Modifying the infrastructure of probes and gauges in order to provide better measurements, and reconfiguration of the system using effectors. While the KX framework is independent of the application, the probes, gauges, etc. are customary chosen for the target application.

In [**VK02**], a large-scale case study is presented. A server farm supporting instant messaging on various channels is instrumented with probes and gauges, and thus enabled to ensure a limited load by automatically activating new servers and reconfiguring the load balancing, should one server exceed a preset load threshold. Compared to component frameworks, the reconfiguration is very coarse, in that entire servers are modified, yet the emphasis of KX is based on the process of determining the need of reconfiguration, which can be applied to components with only minor modifications.

[2] apparently a mere naming coincidence with respect to [**MDK93**]

3.4.2.3. ACT. The ADAPTIVE CORBA TEMPLATE (ACT) [**SM04**] of Sadjadi and McKinley utilizes the *generic interceptor* capability of CORBA to add adaptability in the form of dynamic aspect-orientation to otherwise non-adaptive ORBs. The generic interceptor is part of the CORBA call stack and offers the capability to inspect and possibly modify calls. The work itself is very technical and can be seen as a witness to CORBA's versatility; but it falls short of the possibilities of reconfiguration. As an example, the QUO framework [**ZBS97, VZL+98**] for ensuring quality of service properties of CORBA components is retrofitted to existing systems.

3.4.3. Theory of Reconfiguration Frameworks. Many of the papers listed here are very "narrative": They describe how reconfiguration is to be done, often in a showcase explanation, and often without mentioning the exact circumstances; but usually backed by an implementation. Here, we will report on some work that do not boast an implementation, but rather focus more on the theoretical requirements for reconfiguration.

3.4.3.1. *Warren and Sommerville.* In [**WS96**], Warren and Sommerville discuss how a component framework should be devised in order to maintain integrity. Building on the PCL language [**SD94**], the peculiarities of reconfiguration of stateful, composite components are discussed. Data state retainment is discussed, using an indirect approach of capturing and restoring the state which is based on Herlihy's work on abstract data types [**HL82**]. For transferring state between non-compatible components, *state coercion* is used, which amounts to applying an arbitrary function to the state of the old component. While this idea is not further elaborated, it is interesting because, usually, this problem is not explicitly discussed.

3.4.3.2. WRIGHT. WRIGHT (cf. Sect. 3.1.3.2) also provides provisions for reconfiguration [**AGD97, ADG98**]. Their approach is to describe reconfiguration as part of the CSP calculus used for specifying the component/connector behavior. This is done by fixing a set of possible configurations, and tagging the CSP terms with the configuration they are to operate with. Reconfiguration is then also triggered by CSP terms. In [**ADG98**], a number of checks are provided that can be used to verify the consistency of reconfiguration.

3.4.3.3. CHAM. The CHEMICAL ABSTRACT MACHINE (CHAM) [**IW95**] is a programming model that builds on a chemical metaphor – molecules, solvents (multisets of molecules) and reactions (transformations of solvents). At its heart, CHAM is a term rewriting system. Three kinds of rules are used: Heating rules that decompose molecules into its constituents (refining the view), cooling rules that build larger molecules from other ones (abstracting the view) and reaction rules that actually transform modules. This very general approach allows for specification of many systems, including component systems with reconfiguration [**Wer98**], where the emphasis is placed on finding a consistent algorithm for distributing reconfiguration over a sequence of rules, with possible nondeterministic choice of the order of the rule executions.

3.4.3.4. *Product Line Reconfiguration.* In [**MSTS99**], reconfiguration of product lines is discussed. This does not have anything to do with software, but since the similarities are so striking and since the paper is so much better than many of the others referenced here, we also mention it. Reconfiguration here refers to after-sales operation: A sold product is modified by the manufacturer in accordance to the buyer's request. Providing provisions for this is a challenge to both engineering and project planning. This work provides a comprehensive formal model of reconfiguration, and a refined definition of the reconfiguration task. It is interesting to see

that the purportedly vague field of product design bests most software frameworks in this regard.

3.4.3.5. *Global Consistency – De Palma, Laumay and Bellissard.* In [**PLB01**], the consistency of an application during reconfiguration is investigated. Their work is inspired by distributed transaction systems (cf. their work on OLAN [**BABR96, VDBM97**]); it is very technical (e.g., describing maintaining proper links to components moving to another machine as a primary problem of reconfiguration) and focuses on the redistribution of otherwise static software architectures on multiple nodes. Hence, their concern is about specific artifacts of reconfiguring on a coarse level (migrating application parts rather than components in a dedicated framework, e.g., having to care about messages in transit to a target node while moving away the target component to another node). A timestamp-based optimistic wait-die-algorithm is proposed.

In other work of this group, a reconfiguration protocol is formally specified and model checked [**CGMP01**]. Migration was considered only, and the protocol does not consider the data state of an agent, but this is one of the few works that actually mentions and verifies correctness criteria (ten properties are considered, including absence of deadlocks, termination and correct order of events).

3.4.3.6. SERFS. SERFS [**SPW03**], short for Service Facilities, is a combination of patterns that can be used to decouple software so that dynamic replacement of modules becomes possible. Three patterns are involved: Abstract Factory, Strategy and Memento. Serfs act as sort of a broker that indirects connections to targets. This makes reconfiguration possible for tightly linked systems like OO-based systems, and may also serve as a starting point for implementing reconfiguration-enabled component systems. However, much of the work is concerned with technical difficulties like garbage collection, and an example is missing.

3.4.3.7. *Boinot et al. – Adaptive Components.* In their work on active components [**BMMC00**], Boinot et al. describe how an adaptive component can be built. This work is only loosely related to the frameworks discussed here, in that it seeks to provide a systematic approach towards building JAVA classes that adapt to a changed environment. Although this amounts to an implementation of the Strategy pattern directly in the classes, their work is very interesting since they also describe a three-stage control loop, providing further support for the importance of this concept (which we will discuss in Chapter 8.1).

3.4.4. Physical Systems. A number of paper exist that discuss reconfiguration for physical systems in an abstract way. Chen [**Che02**] describes the requirements for reconfiguration in the context of automotive software. Here, the component application stretches over both the infrastructure (operated by the vehicle manufacturer) and the individual vehicles. The part of the application in the vehicle is operated by a *local configuration manager*. This manager may trigger reconfiguration in order to improve the application's performance on the vehicle. In this very interesting setup, the various problems arising from dynamic reconfiguration – safety issues, planning – are discussed.

In [**MHW04**], the requirements of reconfiguration for robot systems is described. This is quite interesting, because three types of reconfiguration scenarios are described: Reconfiguration for maintaining high reliability (e.g., fault tolerance of the software), reconfiguration for handling physical environment changes, and reconfiguration for coping with hardware faults – however, the latter point is judged as of lesser interest. The work proceeds to discuss various approaches towards maintaining state consistency, which are to be implemented on top of the CORBA

component model. This work might be regarded as the theoretical considerations underlying the OpenREC framework [**HW04, WH04**], or at least as influential (see Sect. 3.2.16).

Kramer and Magee are currently working on an example with robots [**KM07, KM08a**]. This work is highly interesting as they do a "complete revolution" of the loop – i.e., providing solutions for all stages. A three-tier control loop architecture is used; it consists of the component tier, a change management tier, and the top-level goal management tier; this is an established architecture in robotics [**Gat98**]. Here, the monitoring and the execution phases of the MAPE loop are conducted (cf. Sect. 8.1.1.1). The analysis and planning are done in the middle layer, where plans are stored and maintained. These plans are modified by the topmost layer, which tries to reach long-term goals. Kramer and Magee explicitly stress the importance of the separation of concerns; and their work is most valuable in their view on the software engineering requirements for developing adaptive software.

Another interesting example operating with robots is provided by the Runes project [**CLG+06, ÅBD+07**]. They aim at disaster relief in tunnel fire scenarios, obviously sparked by the problems that were observed at the Mont Blanc tunnel fire in 1999. In the event of a disaster, an ad-hoc network is formed to provide connectivity for rescue personnel and delivery of sensor data. This network is built from nodes distributed at regular intervals in the tunnel. Since some of the nodes might be destroyed in the fire, robots are used to provide additional temporary network nodes, such that a network can be reestablished. The Runes project covers a large number of necessary details, building on a component-based middleware [**MZP+05**] and three-staged control loops. It has to be admitted that, for the two previously mentioned projects, the execution of reconfiguration takes only a minor part in the overall framework, and more emphasis is placed on the assessment and planning stages of reconfiguration.

3.4.5. Distributed Systems – the Grid. Grid computing (and peer-to-peer (P2P) systems, which often underlie grid architectures) is an emerging approach towards large-scale computing. Traditional personal computers are used to collaborate on a common computation goal (one of the first and well-known examples being SETI@home [**WCL+01**]). Obviously, such applications need to cope with a very uncertain environment: Nodes and network connections may be taken offline, crash or exhibit an unanticipated load. This is also what distinguishes the idea of grid computing from clusters; although the approach is similar, the nodes in a cluster are under central control and are, to some extent, guaranteed to be available for the duration of the computation, whereas for a grid system, the loss of nodes is somewhat constituent of nominal behavior.

Quite often, grid-enabled algorithms separate computation in another dimension as we do in this thesis: They partition the data, rather than the algorithm (cf. the discussion in Sect. 5.3.2). Of course, it is an altogether different task to reassign a SETI@home data package to another node than to redistribute an algorithm that requires the joint effort of multiple computers to perform various stages required for a computation task.

Some of the aforementioned frameworks explicitly target distributed systems, e.g., Cactus [**CHS01**], the GCM extension of Fractal [**BBGH08**], and ReMMoC [**GBS03**]. These frameworks adapt parts of the communication mechanism of distributed components. If the fixation on components (often merely a wording issue) is relaxed, a vast body of research can be drawn from; exceeding this thesis' scope by far. We will hence only discuss some examples.

vGrid [KHP+03] is a framework for conducting large-scale simulations on a grid. They focus on simulations that can only be parallelized to some extent – as an example, a cellular automaton-like forest fire simulation is proposed. Cellular automata operate on a two-dimensional field, and neighboring cells affect each other. Hence, no "clean" data separation is possible – if the field is split and processed on different nodes, they need to synchronize on the border cell's behavior. Note that while this is the kind of separation we do not address in this thesis, this is an example where multi-stage computing (as discussed in Sect. 7.3) is not feasible at all. Reconfiguration now addresses the relocation of components (here named "computational units") in order to fix a misalignment in computation balance (a self-tuning approach). This is done using a MAPE-K loop; the only example known to the author of an actual use of this form a software control loop. Also, the forest fire example is quite interesting, and subsequently picked up by Accord [PLL+05, LP06], another framework for providing grid applications with self-$\star$ capabilities. Their approach towards adaptivity, however, is merely rule based ("`if isSystemOverLoaded = true then invokeGraphAlgorithm() else invokeBlockAlgorithm();`").

3.4.6. Ad-Hoc Networks and Mobility. Another area of distributed systems that can benefit from reconfigurability is that of ad-hoc networks and device mobility. Ad-hoc networks form spontaneously by the linking of multiple nodes with limited broadcasting capability. For example, cars can try to form temporary networks in order to send traffic volume or road condition information [ESE06]. Also, ad-hoc networks can be built from a large number of cheap sensors/transmitters that collaborate on sending measurements to gateway nodes that eventually process it [CES04]. GridKit [GCB+06] is an example for a framework supporting such ad-hoc networks, as is the Runes project [ÅBD+07].

A related concept is given by mobility, where a mobile device connects to a (usually stationary) service in order to use location-based services. Here, reconfiguration can be used to adapt to changed situation, possibly by substituting their communication means as exemplified by ReMMoC [GBS03]. The Dacia framework [LP00], developed by Radu Litiu in his PhD thesis [Lit00] (which is a profound source of related work covering the various topics that need to be considered) explicitly concentrates on mobility in the form of a user moving to other machines, migrating components in the wake. Also, during absence of a user, components can be *parked*; this maintains their state and allows for quick reconnection.

3.5. An Assessment

When reading the literature presented here, some observations can be made. First of all, a general trend towards using reconfiguration as a programming means can be observed. Obviously, this is influenced by emerging disciplines like service-orientation. Hot code update continues to be of interest, however. This is quite understandable, as the systems that truly require hot code update – large, distributed and heterogeneous systems – are becoming more important.

In Fig. 3.3, a comparison of some work listed in Tab. 3.1 on page 41 is presented. The works are assigned a complexity measure, and the progress over time is shown. This complexity measure was made out of the columns of the table; if a feature is discussed, it counts as a point, and if it is discussed in a way that is convincingly adequate for productive use, it counts as two. Of course, such a comparison is highly subjective and does not reflect the "worthiness" of the research in any way – just consider OpenRec [HW04], which scores a mere 2 points, because state transfer and transactions are considered only. ■ nodes depict the coordinates of frameworks that explicitly discuss hot code update, whereas ● show the coordinates

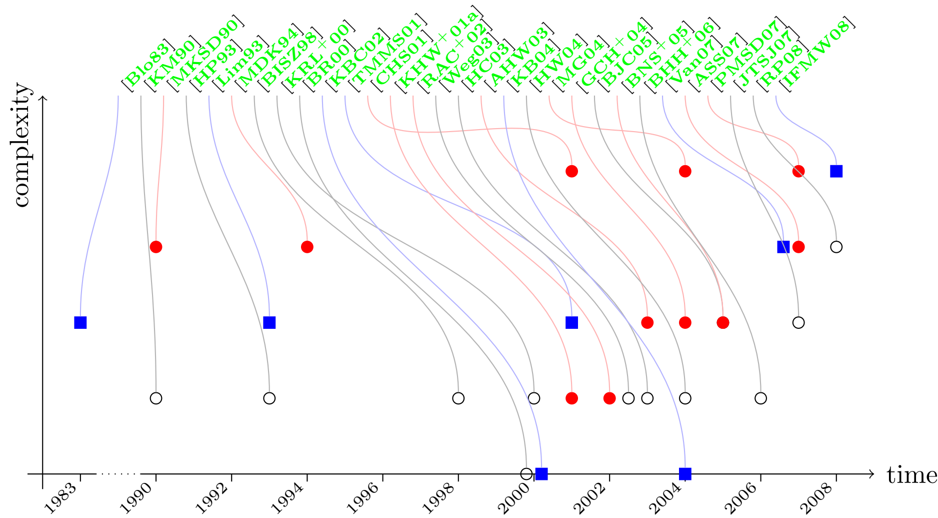


Figure 3.3: Complexity of related work

of frameworks that work with plans, and ○ are the coordinates of frameworks where the intend of reconfiguration is not declared explicitly.

An evaluation of this graphic is of course quite difficult. The "early work" due to Bloom, Kramer and Magee, Hofmeister and Lim scores reasonably well, with Hofmeister's work being more practical and thus scoring low, and [**KM90**] discussing quiescence only. Recent work seems to integrate more technologies and hence scores higher. The four champions – Cactus [**CHS01**], CASA [**MG04**], Fractal [**PMSD07**] and CoBRA [**IFMW08**] – provide credible approaches for enabling reconfiguration. The augmentation of profound work within recent years is raising hopes for a maturing of reconfiguration as a software engineering tool.

The downside of the graphic is the continuing presence of work that is mostly ignorant of problems published by others. This can sometimes be attributed to a narrow problem domain (e.g., NeCoMan [**JTSJ07**]) or a general open architecture that does not provide fixed implementations (e.g., OpenREC [**HW04**] or SOFA 2.0 [**BHP06**]). Quite often, however, many problems are simply not considered [**Par07**]. This especially holds true for frameworks that are built on top of heavyweight component frameworks like Corba.

Assessing state retainment, none of the works provides a truly satisfying solution (which, considering the work done by Vandewoude [**Van07**] on that topic, indicates that there is no such definitive solution). Mostly, the indirect approach is chosen, but the resulting problems – i.e., the replacement component needs to be written in a way that foresees reconfiguration, which is only acceptable in hot code patching – are usually not discussed. Safe for the work of Kefti [**KB04**], a direct approach is only attempted based on (extended) name-based matching – which is a little bit strange, as the utilization of a state transferal script that is external to any component is not that far off.

As for examples provided (note that we only looked at the publications, rather than at available implementations), the situation is bleak. Often, no example is provided at all, or the examples are trivial. For hot code update, this is quite understandable. For generic reconfiguration, this is quite problematic, as an example provides an important clarification of the problem domain addressed. Often,

examples provided for generic reconfiguration are reflection of changes in the physical environment [**MDK94**, **Lim93**, **BR00**, **RAC+02**, **Hac04**], or they describe known techniques like load balancing [**GCH+04**, **TBB+05**] or plug-in installation [**CH04**]. The most convincing examples are provided by those frameworks whose intended application domain is narrowed, most notable CACTUS [**CHS01**] and NECOMAN [**JDMV04**]. Both are concerned with dynamic cross-cutting concerns; this became a great influence for this thesis, as described in Sect. 8.4.

CHAPTER 4

Formal Foundation of Components

Wo ein Tritt tausend Fäden regt,
Die Schifflein herüber hinüber schießen,
Die Fäden ungesehen fließen,
Ein Schlag tausend Verbindungen schlägt.
— Johann Wolfgang Goethe, Faust I

In this chapter, we will define formally what a *component model* is. We have given a basic idea of the concept of components in Sect. 2.1, here, we will formalize these aspects common to the component models we consider. Altogether we try to describe how a *component application* is built. In our context, such an application comprises a set of components and a framework that provides services for connecting and running them. This framework is to be built after a *component model* that formally describes how components execute. In this thesis, we discuss two such component models and their respective frameworks in Chapter 5 and Chapter 6. These models are supporting different application scenarios, but their syntactical foundation as well as the means to describe their semantical behavior are the same.

4.1. Components

We assume a set $\mathscr{C}$ of *component identifiers*, a set $\mathscr{M}$ of *method names*, that are implemented by components, a set $\mathscr{R}$ of *role names*, through which components access other components, and a set $\mathcal{V}$ of *values* which are communicated between components. *Interfaces*, comprised in a set $\mathscr{I}$, are finite subsets of method names. All but the set $\mathcal{V}$ are assumed to be finite (but that is not really important), and $\mathcal{V}$ is either finite or countably infinite.

Role names represent the communication partners known to a component. When writing a component, we cannot know the exact identity of the components we communicate with – these will be determined at configuration time, i.e., when the application starts, or even later during reconfiguration. Instead, the component knows which communication partners it *requires*. These are assigned an interface, taken from $\mathscr{I}$, and are known to the component under a role name. For example, a hash table component might require a hash function component as a communication partner (to query hash values). The actual hash function implementation will be chosen at configuration time, and is not known to the component. All it knows is the presence of a role`hashfunction`, and that any component eventually linked to this role implements an interface `HashFunction` that provides an appropriate hash calculation method (cf. role r in Fig. 4.1).

Let us recall that we are interested in stateful components. We define a set $\mathcal{S}$ (usually of infinite or even uncountable size) that captures the user-accessible state of components. $\mathcal{S}$ needs to store three different kinds of data:

(1) The data state that can be modified by the component's implementation,

(2) the parameter of the current method call, a value from $\mathcal{V}$,

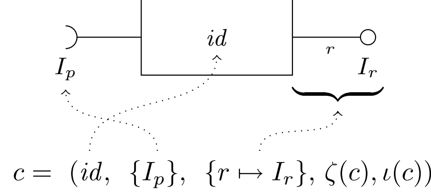


Figure 4.1: Static elements of a component c

(3) and the return value of the last method called, also a value from $\mathcal{V}$.

We hence require three functions: $upd : \mathcal{S} \times \mathcal{S} \to \mathcal{S}$ for updating a state with another state; $prm : \mathcal{S} \times \mathscr{M} \times \mathcal{V} \to \mathcal{S}$ for storing a parameter value for a method name; and $ret : \mathcal{S} \times \mathcal{V} \to \mathcal{S}$ for storing a returned value.

The function *prm* takes the name of the method as a parameter, as this name (which, practically, represents the method's signature) will determine how to store the parameter. It is convenient to just have every method accept a single value $v \in \mathcal{V}$; tuples can be encoded easily, and for methods without parameters, a dummy value can be passed (an example set $\mathcal{V}$ can be found in Sect. 4.6).

Definition 4.1 (Component). *A* component c *is a tuple*

$$(id(c), I_P(c), I_R(c), \zeta(c), \iota(c))$$

with $id(c) \in \mathscr{C}$ *the* component identifier*,* $I_P(c) \subseteq \mathscr{I}$ *a finite set of* provided interfaces *and* $I_R(c) : \mathscr{R} \rightharpoonup \mathscr{I}$ *a partial function with finite domain* requiring interfaces *by role names.* $\zeta(c) : \mathscr{M} \rightharpoonup \mathscr{P}$ *is a* method evaluator*, assigning an implementation to each method the component provides; the set* $\mathscr{P}$ *of* component process terms *will be defined in Sect. 4.2.* $\iota(c) \in \mathcal{S}$ *is the* initial state*.*

The set of required role names $\mathrm{dom}(I_R(c))$ *is denoted by* $\mathscr{R}(c)$*.*

To write the definition of a component c as a tuple $(id(c), I_P(c), I_R(c), \zeta(c), \iota(c))$ is done to abbreviate the notation, and we will use this approach throughout this thesis. It is a compact way to introduce a component $c \in C$ as a tuple (id, P, R, ζ, ι) and then provide methods $id : C \to \mathscr{C}, I_P : C \to \wp(\mathscr{I})$ etc., with $id(c) = id$, $I_P(c) = P$ etc. We denote the universe of components with $\mathcal{C}$.

The statical structure – i.e., the structure of a component prior to its use in a component framework – is thus comprised of a name, which is used to uniquely identify the component (the components themselves should not use it). The set of provided interfaces describes the services offered by the component. As interfaces are sets of methods, the provided interfaces actually describe a set of methods. Often, this set is interpreted by the set of methods a component can "understand", but in our context the view of provided services is more coherent.

The set of required interfaces actually is a set of pairs of role names and interfaces. As discussed before, the role names are the identities of the connections of a component from that component's point of view. They are typed with the interface, meaning that only methods inside the interface can be sent to the component connected to the role (thus effectively prohibiting the sending of non-understood messages). Note that it is quite possible (and quite often required) to have multiple roles with the same interface, e.g., for a load balancing component that can relay a message to multiple roles, each providing the same type.

These three elements comprise the *signature* of a component. Additionally, the method environment $\zeta(c)$ assigns a component process term to each method of the provided interfaces; this term describes how a call of this method will be handled.

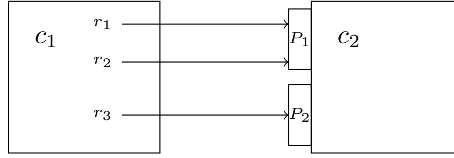


Figure 4.2: Provided interfaces and roles

The initial state $\iota(c)$ determines the component's start-up state. Together, they form the component's *implementation*.

Of course, we need to add some constraints to disallow "irrational" use of this definition. For a set C of components, we require:

- The component identifiers are assigned only once, i.e., $\forall c, c' \in C \,.\, id(c) = id(c') \to c = c'$,
- the method evaluated covers all methods provided by the component:

$$\bigcup_{I \in I_P(c)} I \subseteq \mathrm{dom}(\zeta(c)).$$

From now on, whenever we talk about a set of components, we silently assume that these two constraints hold. The uniqueness of role names within a component is ensured by $I_R(c)$ being a partial function; a uniqueness for all components is not required.

4.1.1. A Brief Discussion of Ports. At this point, one of the most significant design decision of our component model has already been taken, so let us briefly investigate what we have specified so far.

Our definition of components is different from many component definitions, e.g. [**BHH+06**]: $I_P(c)$ and $I_R(c)$ are asymmetric, i.e., do not have the same type. From the point of view taken in this thesis, components have required interfaces, which have a role name and a type, and provided interfaces which have a type only. Hence, a component may be able to send messages to multiple components providing the same interface, but over connections with different role names; and we will later see that calls are indeed addressed to these different role names (instead of to the interfaces, as implemented in Java/A [**Hac04**]). On the other hand, components provide interfaces types instead of ports, and if multiple components connect to an interface, there is no way provided by the framework to determine which component issued a certain message. Fig. 4.2 illustrates this idea: Component c_1 maintains three roles r_1, r_2 and r_3; it may issue messages to each. Component c_2 provides interfaces P_1 and P_2. The types of r_1 and r_2 are both P_1, and the type of r_3 is P_2, therefore a connection as depicted is admissible. c_2 is not aware of the multiple connections of roles typed with P_1; if the source role of a message is to make a difference, this has to be handled by adding a further parameter to the message; the frameworks do not provide a user-accessible source indication for messages. Note that c_1 also does not need to know that all its roles are connected to the same component – messages are addressed to the roles only, and hence the actual target is unknown to the component.

Ports, on the other hand, also provide a role name for incoming connections, and usually group them in pairs with input and output interfaces (e.g., in [**BHH+06**]). Hence, for any received message, it is easy to determine where it was sent from; and inter-component protocols can describe the communication sent and received over one port, which eases the specification of such protocols.

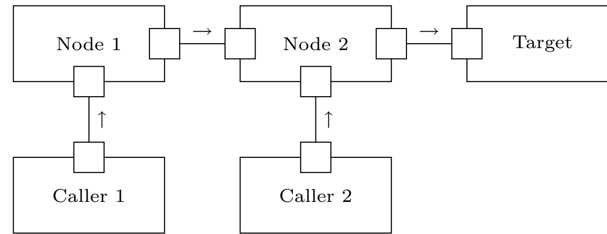


Figure 4.3: "Reverse" Chain of Responsibility pattern using ports (arrows indicating data-flow)

The question whether to use ports or unidirectional connections is hard to answer; and the decision to use the latter was mostly based on the interest how using interfaces only might relate to port-based component models, e.g. [Hac04] or [HJK08].

However, some previous experience with the CMC model checker was also influential: CMC was originally designed with a component system that provided bidirectional ports. Most of the time, however, one of the interfaces (output or input) was empty; usually data flow proved to be unidirectional only, and if a response was required, this was usually provided by the method's return value.

While developing CMC, another problem emerged: In some situations, many components need to send messages to the same, single component. For CMC, this applies to the memory manager (cf. Fig. 5.13 on page 108), a component that manages the allocated memory for new states (just using `malloc` and `free` is too slow by orders of magnitude, since the operating system tries to optimize the allocation). Many components need to connect to the memory manager (cf. Sect. 5.3), but the memory manager itself needs no connection back (i.e., all the provided interfaces of the ports to the memory manager are empty sets of methods), since the allocated objects are passed back as method return values. It would be a breach of the separations of roles paradigm to implement the memory manager with a separate port for each component that connects to it, as a component should be developed independently of its various concrete usage scenarios. It is technically possible to use a "Chain of Responsibility" to gather the data as illustrated in Fig. 4.3 (note that the data flow is inverted here; a more profound discussion of that pattern is given in Sect. 8.3.1). However, there is no benefit given by such a structure, and it slows the application by imposing a large communication overhead. Other examples (cf. Sect. 7.4.2) also have similar configurations. Hence, a practical framework with ports needs to provide multi-ports, which need careful treatment (cf. Sect. 8.3.1.1).

The benefit of ports is their "locality" when specifying components, i.e., with ports, all connections to a component, both incoming and outgoing, are known to the component. For reconfiguration, at some point we need to verify that a component has become completely detached from other components. Using ports, only local knowledge needs to be considered. With interfaces, all components have to be checked in order to see whether any of them is still attached (cf. Sect. 6.6.2).

Also, when considering purely asynchronous systems, it is sometimes necessary to know the identity of a caller. For example, if some component provides a computation service to other components, and is connected to them using a multiport (see Sect. 8.3.1.1), and needs to provide the service's result back to the caller, it needs to be able to identify the correct connection of the multiport to the caller. This is easier with ports, as the outgoing port is the same port where the original

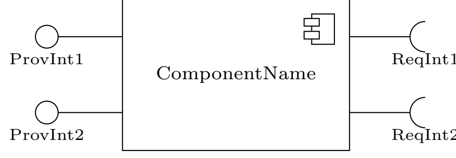


Figure 4.4: A component in a UML component diagram

request was received; but it is possible to provide a complete substitute using just unidirectional connections.

We do not utilize bidirectional ports in this thesis, and so far no problem has emerged in the practical examples. For the sake of specification, however, there is not a real reason why ports cannot be specified on a higher level. Also, there is no technical reason why the identity of the caller should not be made known to the callee. Fig. 4.4 shows the UML notation for components. Here, provided interfaces are also given a role, but the required and provided interfaces are handled distinctly. For obtaining ports, a provided and a required interface can be grouped. Similarly, we can provide ports on a conceptual level even though they are not used in the actual implementation of our model.

4.2. Component Process Terms

We will now define a formal language for describing the behavior of components when processing a method.

Definition 4.2 (Component process term). *The set of* component process terms $\mathcal{P}$ *is defined by*

$$\begin{aligned} P \in \mathscr{P} ::= &\ \text{call}(r, m, v).P \\ &\mid \text{return}(v).P \\ &\mid \text{set}(\sigma).P \\ &\mid \text{choose}((\Sigma_j.P_j)_{j \in J}) \\ &\mid \text{success} \\ &\mid \text{fail} \\ &\mid X \\ &\mid \mu X \Rightarrow P \end{aligned}$$

with $r \in \mathscr{R}$*,* $m \in \mathscr{M}$*,* $v \in \mathcal{V}$*,* $\sigma \in \mathcal{S}$*,* $\Sigma_j \subseteq \mathcal{S}$*.* J *is an index set, and* $X \in \mathcal{X}$ *is a* component process term variable*.*

We will just give the syntax here; the semantics are defined by the component model (defined in Sect. 4.5). However, the informal semantics should be easy to follow: "call(r, m, v)" calls method m on the component that acts as role r with the parameter v, "return(v)" returns value v to the method caller, "set(σ)" updates the component data state to σ, "choose" branches according to the current state (i.e., if the state σ is in Σ_j, then P_j is executed next), and "success" and "fail" terminate the current method. Finally, "$\mu X \Rightarrow P$" is used for defining recursion, e.g., "$\mu X \Rightarrow$ call$(r, m, v).X$" describes a never-ending repetition of method invocations.

Note that "return(v)" can be issued in the middle of a method; this is somewhat like a *co-future* – the calling component will be given the result value, with the called component still working on the method, e.g., for cleaning up. Nevertheless, we do

not encourage such a use of the component process term. Also, a lot of illogical component process terms can be defined, e.g., the term "return(v).return(v').success" or the term "choose($\Sigma_j.P_j)_{j\in\{1,2\}}$" with $\Sigma_1 \cup \Sigma_2 = \mathcal{S}$ and $P_1 \equiv$ return(v).success and $P_2 \equiv$ success. We will later see that such malformed terms lead to lockups in the execution of the system, and assume this to be an error on the user-level: No special restriction is put on the terms. This also applies for method invocations: It is not syntactically enforced to call a method only on a role that actually provides this method; but doing so will lock the execution. Another type of "user level error producing" terms is given by unbound variables $X \in \mathcal{X}$.

For a component process term P, we define $P[Q/Q']$ as the substitution of all subterms Q by Q'. This can be given by a recursive definition, i.e., by

$$(\text{call}(r,m,v).P)[Q/Q'] = \begin{cases} Q'.(P[Q/Q']), & \text{if } Q \equiv \text{call}(r,m,v) \\ \text{call}(r,m,v).(P[Q/Q']), & \text{otherwise} \end{cases}$$

and similar for the other constructors.

We assume a special variable $\mathbf{0} \in \mathcal{X}$. We define the function $\mathit{sub} : \mathscr{P} \to \wp(\mathscr{P})$ recursively as

$$\begin{aligned}
&\mathit{sub}(\text{call}(r,m,v).P) = \{\text{call}(r,m,v).\mathbf{0}\} \cup \mathit{sub}(P),\\
&\mathit{sub}(\text{return}(v).P) = \{\text{return}(v).\mathbf{0}\} \cup \mathit{sub}(P),\\
&\mathit{sub}(\text{set}(\sigma).P) = \{\text{set}(\sigma).\mathbf{0}\} \cup \mathit{sub}(P),\\
&\mathit{sub}(\text{choose}((\Sigma_j.P_j)_{j\in J})) = \{\text{choose}((\Sigma_j,\mathbf{0})_{j\in J})\} \cup \textstyle\bigcup_{j\in J}\mathit{sub}(P_j),\\
&\mathit{sub}(\text{success}) = \{\text{success}\},\\
&\mathit{sub}(\text{fail}) = \{\text{fail}\},\\
&\mathit{sub}(X) = \varnothing,\\
&\mathit{sub}(\mu X \Rightarrow P) = \{\mu X \Rightarrow \mathbf{0}\} \cup \mathit{sub}(P).
\end{aligned}$$

A component process term P is called *nondeterministic* if there exists $P' \in \mathit{sub}(P)$ such that

$$P' \equiv \text{choose}((\Sigma_j,\mathbf{0})_{j\in J}) \quad \text{and} \quad \exists k,l \in J\,.\,k \neq l \wedge \Sigma_k \cap \Sigma_l \neq \varnothing$$

and deterministic otherwise. P is called *recursive* if $\mu Y \Rightarrow \mathbf{0} \in \mathit{sub}(P)$ for some $Y \in \mathcal{X}$. P is called a *query term* (and subsequently any method interpreted by it is called a *query*), if set$(\sigma).\mathbf{0} \notin \mathit{sub}(P)$ for any $\sigma \in \mathcal{S}$.

If the sets J and S are of infinite size, component process terms are obviously Turing-complete. If one of the sets is finite, not even a pushdown automaton can be simulated: An infinite number of data states $\mathcal{S}$ has to be provided to represent the stack configurations. If J is finite, only a finite number of data states can be distinguished using choose, hence only a finite number of pushdown stacks can be distinguished (or, for that matter, written). Hence, if either J or S are finite, only a finite automaton can be simulated. However, finite sets are sufficient for Turing-completeness if a sufficiently strong communication framework is provided (i.e., if the communication can be used to encode a Turing machine tape, cf. Sect. 9.3.2).

4.3. Component Setups

Components are intended for composition. Informally, a composition consists of a finite set of components and a mapping from their roles to other components. Also, an entry point is provided – comprised of a component and one of its provided methods. Of course, a component framework supporting concurrent components might benefit from multiple entry points, but it is conceivably easy to trigger these methods from a single starting point, so we do not generalize this here.

Definition 4.3 (Component setup). *A component setup* (C, M, e) *consists of a set* $C \subset \mathcal{C}$ *of components, a mapping function* $M : C \to \mathscr{R} \to C$ *and an entry point* $e = (c, m)$ *with* $c \in C$ *and* $m \in \bigcup_{i \in I_P(c)} i$.

Again, placing constraints on M is necessary. There are two constraints:

(1) Roles are connected to components that provide the interface required by that role, and
(2) every role is connected.

We call the former requirement being *well-connected* and the latter begin *completely connected*:

Definition 4.4 (Well- and completely connected). *For a set of components* $C \subseteq \mathcal{C}$ *and a component* $c \in C$*, we call a function* $f : \mathscr{R} \to C$ well-connected *for c if*

$$\forall r \in \mathscr{R}(c) \,.\, I_R(c)(r) \in I_P(f(r)).$$

We call f completely connected *for c if*

$$\text{dom}(f) = \mathscr{R}(c).$$

We call M well-connected if $\forall c \in \text{dom}(M) \,.\, M(c)$ is well-connected for c, and we call M completely connected if $\forall c \in \text{dom}(M) \,.\, M(c)$ is completely connected for c. We will usually omit the reference to the underlying set C of components if it is clear from the context. We define the set CS to be the set of all component setups that satisfy these requirements.

Unlike the requirements placed on sets of components (uniqueness of the component identifier), we will always explicitly require the component connections to be well- and completely connected. This is because the component connection is provided by another "person" than the component identifiers (which we assume to be generated by the framework) or the component's roles (which are defined by a component implementer, and are subsequently checked by the compiler before the component can be deployed in a component framework). In this thesis, we are mainly concerned with the problems of said person, which we call the *system designer*. The task of this idealized person is to take the components provided and *assemble* them to become a component application. We will elaborate the responsibilities of this person more in Sect. 9.1.

A component setup forms a *graph*. Within this thesis, we will use all types of graphs (in a sense, a component setup *is* a graph already), with the most generic definition being as follows:

Definition 4.5 (Directed labelled graph). *A* directed labelled graph *for a set of labels* Σ *is a tuple* $G = (V, E, \alpha, \omega, \lambda)$ *with*

- V *and* E *being a finite sets,*
- $\alpha : E \to V$ *and* $\omega : E \to V$,
- $\lambda : V \cup E \rightharpoonup \Sigma$ *the labeling function.*

We write $v \xrightarrow{l} v'$ *to express* $\exists e \in E \,.\, \alpha(e) = v \wedge \omega(e) = v' \wedge \lambda(e) = l$ *and* $v \to v'$ *for* $\exists l \in \lambda \,.\, v \xrightarrow{l} v'$*. We inductively write* $v \to^* v'$ *for* $v = v' \vee \exists v'' \in V \,.\, \exists l \in \lambda \,.\, v \xrightarrow{l} v'' \wedge v'' \to^* v'$*. We write* $v \to^+ v'$ *for* $v \to^* v' \wedge v \neq v'$.

A directed labelled graph is acyclic *if* $\forall v \in V \,.\, \neg(v \to^+ v)$*, otherwise it is* cyclic*. It is* (weakly) connected *if* $\forall v, v' \in V \,.\, v \to^* v' \vee v' \to^* v$*, and* strongly connected *if* $\forall v, v' \in V \,.\, v \to^* v'$.

A subgraph G' *of* G *is a graph* $(V', E', \alpha, \omega, \lambda)$ *with*

- $V' \subseteq V$ *and*
- $E' \subseteq E$ *with* $e \in E' \to \alpha(e) \in V' \wedge \omega(e) \in V'$.

A component setup (C, M, e) with M being completely connected then translates to a graph $(C, E, \alpha, \omega, \lambda)$ over $\Sigma = \mathscr{C} \cup \mathscr{R}$ with

- $E = \bigcup_{c \in C} \bigcup_{r \in \mathscr{R}(c)} \{(c, r)\}$,
- $\alpha((c, r)) = c$,
- $\omega((c, r)) = M(c)(r)$,
- $\lambda(c) = id(c)$ and $\lambda((c, r)) = r$.

Usually, we want component setups to form a connected graph. While, in principle, component applications might also be comprised of disjoint component graphs, this would be not very interesting. We will not require it explicitly, but assume connected component setup graphs tacitly unless it needs to be stated explicitly.

4.4. Concepts for Describing Dynamic Behavior

We have described the definition of components and component setups; these form the static structure of a component application. We will now proceed to define *component models*, which give the dynamic behavior of a component application. The static structure was mapped on the generic notion of a graph; here, we will introduce a similar concept for dynamic behavior: Labelled transition systems.

4.4.1. Labelled Transition Systems. Such a dynamic behavior is a sequence of *states*. We will neither describe how such a state looks like nor how its successors are determined; this information will be provided by the concrete component models. States usually consist of a mapping from a set of components to their component state, which consists of a data state, the currently executed method and the component process term that will be executed next, and some data capturing the communication of that component (e.g., message queues of incoming messages). States evolve by communication of components and execution of component process terms. How this happens exactly will be defined in Sect. 5.2.4 and Sect. 6.3. Right here, we will just describe what a component model provides: A *labelled transition system*, which is another way to write down a graph:

Definition 4.6 (Labelled transition system). *A* labelled transition system (LTS) (S, L, T) *is given by*

- *a set of states* S,
- *a set of labels* L, *and*
- *a set of transitions* $T \subseteq S \times L \times S$.

We write $s \xrightarrow{l} s'$ *for* $(s, l, s') \in T$, *and* $s \longrightarrow s'$ *for* $\exists l \in L \,.\, s \xrightarrow{l} s'$. $s \rightarrow^{+} s'$ *and* $s \rightarrow^{*} s'$ *are defined as the transitive respectively the transitive-reflexive closure of* $\longrightarrow$.

An LTS with initial state (LTSi) (S, L, T, i) *is defined as an LTS* (S, L, T) *with* $i \in S$.

For an LTS (S, L, T), the elements of the label set L are also sometimes called actions. This is used if a more operational view is desired, i.e., a view where a tuple $(s, l, s') \in T$ is to be understood as "s' is the result of applying some changes described by l to the state s".

An LTS is a graph with implicit node and explicit transition labeling. An LTS (S, L, T) can be translated to a graph $(V, E, \alpha, \omega, \lambda)$ over $\Sigma = S \cup L$ by setting

- $V = S$,
- $E = T$,
- $\alpha((s, l, s')) = s$,
- $\omega((s, l, s')) = s'$,

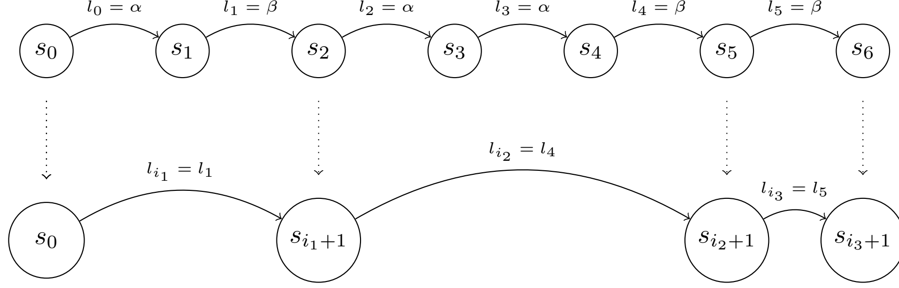


Figure 4.5: Example run reduction with $\varphi = \{\beta\}$

- $\lambda(s) = s$ for $s \in S$ and $\lambda((s, l, s')) = l$.

Note that the notions like $\rightarrow^+$ are preserved, i.e., $s \rightarrow^+ s'$ in the LTS if $s \rightarrow^+ s'$ in the graph.

For a set $A \subseteq L$ of labels, we define $s \xrightarrow{A} s' \equiv \exists l \in A \,.\, s \xrightarrow{l} s'$. We define $s \xrightarrow{A}{}^* s' \equiv s = s' \vee \exists s'' \in S \,.\, s \xrightarrow{A} s'' \wedge s'' \xrightarrow{A}{}^* s'$. Finally, we define $s \xrightarrow[S']{A} s' \equiv s = s' \vee \exists s'' \in S' \,.\, s \xrightarrow{A} s'' \wedge s'' \xrightarrow[S']{A} s'$, i.e., $s \xrightarrow[S']{A} s'$ if there is a path from s to s' using only labels from A and intermediate states from S'.

4.4.2. Runs and Traces. Runs and traces are execution sequences of component systems. They are important means to characterize the behavior of an LTS. A little problem lurks in their finiteness: Sometimes, runs are finite, if we investigate a system that terminates, or if we take a look on the progress of a running system, and sometimes we desire them to be infinite, if the execution of a non-terminating system is to be discussed. It is well known that finite runs can always be made infinite by infinitely repeating the last state with a τ-action (a technique called *stuttering*), but here, we provide distinct definitions for finite and infinite runs:

Definition 4.7 (Run). *A finite* run *(or* execution fragment [**BK08**]*) of an LTS* (S, L, T) *is a sequence* $s_0 l_0 s_1 l_0 \dots l_{n-1} s_n$ *with* $s_i \in S$ *and* $l_i \in L$*, such that, for all* $0 \leq i < n$, $s_i \xrightarrow{l_i} s_{i+1}$ *holds. We write such a run as* $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \dots \xrightarrow{l_{n-1}} s_n$*.*

For an LTSi (S, L, T, I)*, a run* $s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$ *is* initial *if* $s_0 = I$*. A finite run* $s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$ *is called* maximal *if* $(s_n, l, s) \notin T$ *for all* $l \in L, s \in S$*.*

Likewise, we define infinite runs as a sequence $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \dots$ *with* $s_i \xrightarrow{l_i} s_{i+1}$ *for all* $i \geq 0$ *(and, again* $s_0 = I$ *for infinite initial runs).*

Let $\mathcal{L}$ be an LTSi. We are mostly interested in the set $Runs(\mathcal{L}) = Runs_i^m(\mathcal{L}) \cup Runs_i^\omega(\mathcal{L})$ with $Runs_i^m(\mathcal{L})$ the set of initial, maximal runs of $\mathcal{L}$ and $Runs_i^\omega(\mathcal{L})$ the set of initial, infinite runs of $\mathcal{L}$. We call an LTSi *terminating*, if $Runs(\mathcal{L}) = Runs_i^m(\mathcal{L})$ and *non-terminating*, if $Runs(\mathcal{L}) = Runs_i^\omega(\mathcal{L})$.

A run can be *reduced* by removing states and transitions that are not important to us: Let $s = s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$ be a finite run and $\varphi \subseteq L$ a set of *observable* labels. The *run reduction* $s|_\varphi = s_0 \xrightarrow{l_{i_1}} s_{i_1+1} \xrightarrow{l_{i_2}} \dots \xrightarrow{l_{i_m}} s_{i_m+1}$ is the run obtained by the strictly monotonically increasing sequence $i_1, \dots, i_m$ with $i_m < n$ and $l_{i_j} \in \varphi$ and $l_k \notin \varphi$ for all $i_j < k < i_{j+1}$ for all $0 \leq j < n$. An example is given in Fig. 4.5, here, the sequence $\{1, 4, 5\}$ is used. Reduction for infinite runs is defined in the obvious manner, but note that it is not always possible to build an infinite reduced run (if only a finite number of labels is observable, the reduced run will be finite; if

no labels are observable at all, it will consist of the sole state s_0). We extend the reduction to sets of runs by setting $R|_\varphi = \{r|_\varphi \mid r \in R\}$.

Sometimes, we are not interested in the states of a run, but only in its labels. This is especially true for components, where communication is observable, but the data state is not; communication produces transitions, so it is sufficient to just observe them. We hence define *traces*, which are just sequences of labels:

Definition 4.8 (Trace). *The* trace *of a run* $s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$ *is the sequence* $\langle l_0, \dots, l_{n-1}\rangle$ *of labels.*

For an LTS $\mathcal{L}$*, we set* $Traces(\mathcal{L}) = \{t \mid t \text{ is a trace of a run } r \in Runs(\mathcal{L})\}$.

Infinite traces are defined in the obvious manner, as is the reduction $t|_{L''}$ of a trace t for $L'' \subseteq L'$.

Traces hide the states, but sometimes the labels still bear too much information. We hence introduce *trace abstractions*:

Definition 4.9 (Trace abstraction). *Let* A *be a set and* $\alpha : L \to A \cup \{\tau\}$ *be a function. The* α*-trace abstraction of a trace* $\langle l_0, l_1, \dots, l_n\rangle$ *is the trace* $\langle \alpha(l_0), \alpha(l_1), \dots, \alpha(l_n)\rangle|_{A\setminus\{\tau\}}$.

We will use these trace abstractions to throw away all information we are not interested in.

4.4.3. Weak Bisimulation. Weak bisimulation is a concept useful for comparing LTS. Basically, it states that any progress in one LTS can be simulated by the other, and vice versa, allowing for hidden actions in between. These hidden actions are usually required if an LTS is more refined than another, breaking a single step into multiple substeps. Hence, weak bisimulation is defined with respect to a set of labels, which are assumed to be visible.

Technically, weak bisimulation is a relation on states for an LTS (a single one, see below), and defined via the definition of weak *simulation*:

Definition 4.10 (Weak simulation). *For a set* $L' \subseteq L$*, a relation* $\sim \subseteq S \times S$ *is a* weak simulation relation *with respect to* L' *if, for all* $s, s' \in S$ *with* $s \sim s'$ *we have, for all* $t \in S$, $l \in L$*, if* $s \xrightarrow{l} t$*, then*

- *if* $l \in L'$*, then there exists* $t', t'', t''' \in S$ *with* $s' \xrightarrow{L\setminus L'}{}^{*} t'' \xrightarrow{l} t''' \xrightarrow{L\setminus L'}{}^{*} t'$ *and* $t \sim t'$,
- *if* $l \notin L'$*, then there exists* $t' \in S$ *with* $s' \xrightarrow{L\setminus L'}{}^{*} t'$ *and* $t \sim t'$.

Weak simulation hence relates two states if the second state can mimic the moves the first state can do by doing the same move (plus a possible empty sequence of hidden moves) and reaching a relating state again.

Definition 4.11 (Weak bisimulation). *A relation* $\sim \subseteq S \times S$ *is a* weak bisimulation relation *with respect to* $L' \subseteq L$ *if both* $\sim$ *and* $\sim^{-1} = \{(s, s') \in S \times S \mid s' \sim s\}$ *are weak simulation relations with respect to* L'.

Weak bisimulation can be used to compare different LTS. As it is defined on one LTS, we first combine two LTS by building the *disjoint union*:

Definition 4.12 (Disjoint union of LTS). *Let* $\mathcal{L} = (S, L, T)$ *and* $\mathcal{L}' = (S', L, T')$ *be LTS with the same set of labels* L*. The disjoint union* $\mathcal{L}'' = (S'', L, T'')$ *of* $\mathcal{L}$ *and* $\mathcal{L}'$ *is given by:*

- $S'' = (S \times \{1\}) \cup (S' \times \{2\})$,
- $T'' = \{(q, l, q') \in S'' \times L \times S'' \mid \exists (s, l, s') \in T \,.\, q = (s, 1) \wedge q' = (s', 1)\} \cup \{(q, l, q') \in S'' \times L \times S'' \mid \exists (s, l, s') \in T' \,.\, q = (s, 2) \wedge q' = (s', 2)\}$.

By relating states within that disjoint union such that states from S are related to states from S', we can define what it means for two LTS to be weak bisimilar:

DEFINITION 4.13 (Weak bisimilarity of LTS). *Two LTS $\mathcal{L} = (S, L, T)$ and $\mathcal{L}' = (S', L, T')$ are weak bisimilar with respect to $L'' \subseteq L$ if there exists $\sim \subseteq ((S \times \{1\}) \times (S' \times \{2\}) \cup (S' \times \{2\}) \times (S \times \{1\})$ such that $\sim$ is a weak bisimulation relation with respect to L'' of the disjoint union of $\mathcal{L}$ and $\mathcal{L}'$.*

Two LTSi $\mathcal{L} = (S, L, T, i)$ and $\mathcal{L}' = (S', L, T', i')$ are weak bisimilar with respect to $L'' \subseteq L$ if (S, L, T) and (S', L, T') are weak bisimilar with respect to L'' due to the weak bisimulation relation with respect to L'' $\sim$ and $(i, 1) \sim (i', 2)$ as well as $(i', 2) \sim (i, 1)$.

An important kind of weak bisimulation is (strong) bisimulation, which is defined as weak bisimulation with respect to L, i.e., the entire set of labels. For (strong) bisimulation, no hidden moves are allowed, and bisimilar LTS need to run synchronously.

For weak bisimilar LTS, an important property (known as *trace equivalence*) can be shown:

LEMMA 4.1. *Let $\mathcal{L}$ and $\mathcal{L}'$ be weak bisimilar LTSi with respect to L''. Then, $Traces(\mathcal{L})|_{L''} = Traces(\mathcal{L}')|_{L''}$.*

PROOF. We consider an initial run $r = s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \in Runs(\mathcal{L})$ and show the existence of a suitable run $r' = s'_0 \xrightarrow{l_0} s'_1 \xrightarrow{l_1} \ldots \in Runs(\mathcal{L}')$ with $(s_i, 1) \sim (s'_{j_i}, 2)$ for a monotonically increasing sequence $j_0, j_1, j_2, \ldots$ with $j_0 = 0$ by building this r' by an induction on the length i of r:

- $i = 0$, since $s_0 = i$ and $s'_0 = i'$ and by requirement $i \sim i'$ we have $s_0 \sim s'_0$,
- $i \to i + 1$: By induction hypothesis, we have $s_i \sim s'_{j_i}$.
  - If $s_i \xrightarrow{l} s_{i+1}$ for $l \in L''$, then by weak bisimulation requirement, there is an s' with $s_{i+1} \sim s'$ and $s'_{j_i} \xrightarrow{L \setminus L''}{}^* s'' \xrightarrow{l} s''' \xrightarrow{L \setminus L''}{}^* s'$. We can add this sequence to r' and set j_{i+1} to the position of s' in r'. Note that only one $l \in L''$ is used, hence both $s_i \xrightarrow{l} s_{i+1}$ and $s'_{j_i} \to^* s'_{j_{i+1}}$ account for a single entry in the L''-reduction of the traces of r and r' each.
  - If $s_i \xrightarrow{l} s_{i+1}$ for $l \notin L''$, then by weak bisimulation requirement, there is an s' with $s_{i+1} \sim s'$ and $s'_{j_i} \xrightarrow{L \setminus L''} s'$. We add this sequence to r', setting j_{i+1} to the position of s' in r'. Neither l nor any of the labels between position j_i and j_{i+1} show up in the L''-reduction of the traces of r and r'.

Therefore, we obtain $Traces(\mathcal{L})|_{L''} \subseteq Traces(\mathcal{L}')|_{L''}$. The other direction follows from the symmetry of bisimulation. □

This lemma describes why weak bisimilarity is useful for us: If two component systems are weakly bisimilar with respect to an interesting subset of labels, then their sets of traces, reduced to these labels, are the same (for strong bisimilarity, the set of traces is actually the same). If we consider just the communication – which we regard as observable in the context of components – weak bisimilarity with respect to the communication is sufficient to show that two LTS indeed cannot be distinguished by observation, and thus, for all we are interested in, behave alike.

4.4.4. Term Rewriting Systems. Throughout this thesis, we will use a special notation for states and their transition, which is inspired by the MAUDE language [CDE+07]. Let us assume that we are provided with a set $\mathcal{S}$ of component

states (note that this set is not identical to the free accessible data states set $\mathcal{S}$, but captures the entire state of a component, including connections, queues, etc.). For a *state function* $s : \mathcal{C} \rightharpoonup \mathscr{S}$ with finite domain (and all such functions describing component setup states will have a finite domain), we write $c_1, s_1 \parallel \ldots \parallel c_n, s_n$ for $s(c_1) = s_1 \wedge \ldots \wedge s(c_n) = s_n \wedge \mathrm{dom}(s) = \{c_1, \ldots, c_n\}$. This turns the function into a relation, but as long as we make sure that $c_i \neq c_j$ for $i \neq j$, this is equivalent. Thus, the dynamic state of a component setup is written as a sequence of tuples. For the component models we define and use later, the s_i are tuples themselves, and we flatten them, so if $s(c_i) = (s_i^1, \ldots, s_i^n)$ we write $c_i, s_i^1, \ldots, s_i^n$.

It will be part of the component model to precisely state how a dynamic state of a component setup looks like, but in this thesis, it will mostly consist of such a mapping; and we can always write it in a similar fashion, allowing for definition of a uniform way of *advancing* a state.

Let us fix a set V of variables. A *rule* is written as

$$t_1 \to t_2 \quad \text{if } \varphi \tag{N}$$

with N being the rule's name, t_1 a term over V and t_2 a term over $\mathit{free}(t_1)$ and φ a *condition* over $\mathit{free}(t_1)$, where $\mathit{free}(t) \subseteq V$ are the variables of t (actually, the free variables, but there is not quantification involved, so all variables are free).

Given a term t without free variables and a rule N, a *match* is a mapping m of $\mathit{free}(t_1)$ to subterms of t such that

- t_1 with all variables substituted by their mapped subterms equals t, and
- φ with all variables substituted by their mapped subterms evaluates to **true**.

The *result* of a rule matching by m to term t is t_2 with all variables substituted by their mapped subterms.

For example, the set of component states might be given as tuples out of $(\mathcal{P}, \mathbb{N})$, i.e., a state consists of a component process term and a natural number: $\mathscr{S} = \mathcal{P} \times \mathbb{N}$. The set $\mathcal{S}$ of component data states is given by the natural numbers $\mathbb{N}$. We can then define a rule

$$c, \mathrm{set}(n).P, m \parallel C \to c, P, n \parallel C \quad \text{if } \mathbf{true} \tag{EXSET}$$

This rule, which is referenced by the name EXSET ("example set", in order to avoid confusion with rules given later), matches to the state

$$c_1, \mathrm{set}(5).\mathrm{return.success}, 15 \parallel c_2, \mathrm{fail}, 2$$

by setting

$$m = \{c \mapsto c_1, n \mapsto 5, P \mapsto \mathrm{return.success}, m \ \mapsto 15, C \mapsto (c_2, \mathrm{fail}, 2)\}.$$

The result will be a state of the form

$$c_1, \mathrm{return.success}, 5 \parallel c_2, \mathrm{fail}, 2.$$

For brevity – and, ultimately, clarity – we omit all those elements of the state within a rule that do not contribute to the matching. So, even if we are given a much bigger component state tuple, we would still write the rule EXSET like stated above (and omit both the C and the "if **true**" part), since no further state information is decisive for matching the terms the rule can be applied to:

$$c, \mathrm{set}(n).P, m \to c, P, n. \tag{EXSET}$$

Which element of the rule matches to which element of the state term will always be clear from the context (i.e., the elements of a component state have disjoint types, and from the syntax it will always be clear which element is addressed within a rule).

For a given rule and a given state, more than one match might be possible. In our context, this often applies to the choice of the component to move. For example, the rule EXSET has two matches in the state

$$c_1, \text{set}(5).\text{return}.\text{success}, 15 \parallel c_2, \text{set}(7).\text{fail}, 2.$$

To declare which match is to be taken, a rule can be *instantiated*, that is, the variables of the left side of the rule can be bound explicitly to matching elements of the (state) term, thus producing a new state by inserting these variables in the right hand side. For a rule R with free variables $x_1, \ldots, x_n$, we write $\text{R}(x_1 : v_1, \ldots, x_n : v_n)$ to indicate that R is instantiated with variable x_i bound to v_i in the state representation. So, in the example given above, we can write $\text{EXSET}(c : c_1, n : 5, P : \text{return.success}, m : 15)$ to indicate the first of the two possible matches. Sometimes, we wish to talk about sets of rule instantiations, which we denote by underspecified rule instantiations, meaning the set of all rule instantiations for which the non-mentioned (or free in the current context) variables are assigned any value; for example, we write EXSET() for the set $\{\text{EXSET}(c : c_1, n : 5, P : \text{return.success}, m : 15), \text{EXSET}(c : c_2, n : 7, P : \text{fail}, m : 2)\}$. For convenience, we will assume that choosing an element from the set binds the formerly free variables to the values chosen. So we write $l \in \text{EXSET}(c : c_1, \overline{n : N})$ meaning that l is of the form $\text{EXSET}(c : c_1, n : N, P : P', m : M)$ for arbitrary N, P' and M, and we will bind N to the value that is actually used in l. In order to facilitate the reading of such sets, we write $\overline{n : N}$ to indicate that N is supposed to be a free variable that becomes bound to the value used.

Rule instantiations act as labels of an LTS (S, L, T), where we set

- S the set of terms as mandated by the component model,
- L the set of rule instantiations,
- $(s, l, s') \in T$ if s can be transformed to s' by application of rule instantiation l.

Thus, we obtain a concise notion for a rewriting system [DJ90]. However, as we perceive each rule application as a step of the system described, we are not interested in properties like confluence or normal forms. Instead, we are interested in properties that the resulting LTS yields, most importantly its finiteness (which is usually not given) and whether states satisfying a certain property are reachable (violation of a safety property) or can be avoided forever (violation of a liveness property).

The concept of hiding superfluous information in the rule instantiations carries over to the definition of weak bisimilarity of LTS obtained from term rewriting systems. If no information is hidden, only identical systems would be weakly bisimilar, since any difference in a term would yield different rule instantiations. By only discussing rules instantiated with as few variables as required to consistently produce the resulting state, the concept of weak bisimilarity is retained for term rewriting systems.

4.5. Component Models

We can now define what a component model actually is. In the previous sections, we have assembled the ingredients:

DEFINITION 4.14 (Component model). *A* component model *is comprised of*

- *a set* $\mathbb{S}$ *of states (which are to be written as terms),*
- *a set of rules with names* $\{(\text{R}_1), \ldots, (\text{R}_m)\}$,
- *and a function* $\iota : CS \to \mathbb{S}$ *that, given a component setup as defined in Sect. 4.3, produces the initial state.*

A component model produces a labelled transition system $L = (S', L, T)$ by setting

- $S' = \mathbb{S}$,
- L are all instantiations of the rules $\{\mathrm{R}_1, \ldots, \mathrm{R}_m\}$,
- $(s, l, s') \in T$ if s' is the result of applying the rule instantiation l to s.

As an example, the component model mentioned in the previous section defines the component data states by setting $\mathcal{S} = \mathbb{N}$ (neglecting the functions *upd* etc. for the sake of simplicity) and defining the component state tuples as $\mathscr{S} = \{(p, n) \in (\mathcal{P}, \mathbb{N})\}$. The model states $\mathbb{S}$ are then given as mappings $\mathcal{C} \rightharpoonup \mathscr{S}$.

The states of the induced labelled transition system are now all mappings of finite sets of component identifiers to pairs of component process terms and a natural number.

An initial state $I \in S$ is calculated for a component setup (C, M, e) by the function ι. It needs to come up with a state based on the information of the components C (which components are to be included), M (how are they connected) and e (which component is initialized in a special state that can be used to have rules applied to it). Note that no reference to the initial component setup is made by other states of the labelled transition system – this is due to the requirement to accommodate reconfiguration, where the set of components might change. Component models can define arbitrary rules for reconfiguration that can also introduce components.

A component framework is an implementation of a component model. It needs to provide a concise way to implement components (e.g., a special programming language; if this language is based on an existing programming language, the latter is called the *host language*) and a system to run a component setup such that the component execution and communication is conducted in the way described in the model. For some component models (e.g., CORBA [**OMG08**] or FRACTAL [**BCL+06**]), multiple frameworks have been developed for the same component model. In this thesis, a 1:1 mapping is given – each model is implemented in one framework.

Next, we will introduce a concise language for defining the behavior of components that translates to the more abstract component process terms.

4.6. A "Programming Language" for Components

The component process terms presented in Sect. 4.2 are quite generic. Here, we will provide a series of macros that makes the language more readable.

First, let us fix a countable set $\mathscr{N}$ of *variable names*. We assume a function $var(m) : \mathscr{M} \to \mathscr{N}^*$ that assigns a sequence of variable names to a method name . For ease of reading, we assume that the method names are of the form $m(v_1, \ldots, v_n)$ and subsequently use $var(m(v_1, \ldots, v_n)) = \langle v_1, \ldots, v_n \rangle$. The (untyped) parameters are thereby made part of the method name (or signature), as it is done in many programming languages like JAVA.

The set $\mathcal{S}$ of component data states is then a partial mapping of variable names to values:

$$\mathcal{S} = \{\mathscr{N} \rightharpoonup \mathcal{V}\}.$$

A state hence assigns values to variables. To call a method, we need to provide a tuple of parameter values, so we assume $\mathcal{V}$ to be closed under tuple construction, i.e., if $\langle v_1, \ldots, v_n \rangle \in \mathcal{V}^*$, then $\langle v_1, \ldots, v_n \rangle \in \mathcal{V}$.

We then define the function $prm : \mathcal{S} \times \mathscr{M} \times \mathcal{V} \to \mathcal{S}$ as

$$prm(\sigma, m(p_1, \ldots, p_n), (v_1, \ldots, v_n)) = \sigma[p_1 \mapsto v_1] \ldots [p_n \mapsto v_n].$$

For updating of states, define

$$upd(\sigma, \sigma') = \left(v \mapsto \begin{cases} \sigma'(v), & \text{if } v \in \text{dom}(\sigma') \\ \sigma(v), & \text{otherwise} \end{cases} \right).$$

The return value of methods utilizes a special variable $res \in \mathscr{N}$:

$$ret(\sigma, v) = upd(\sigma, (res \mapsto v)).$$

For writing down assignments, we introduce a macro of the form $v \leftarrow t$, with $v \in \mathscr{N}$ and t being a predicate logic term over $\mathscr{N}$ (for brevity, we omit exact specification and generously use common mathematical notations from arithmetic and set theory). We denote the evaluation of a term t under the mapping $\sigma : \mathscr{N} \to \mathcal{V}$ as $[\![t]\!]_\sigma \in \mathcal{V}$. For a predicate φ, we write the evaluation in state $\sigma \in \mathcal{S}$ as $[\![\varphi]\!]_\sigma \in \mathbb{B}$ and $[\![\varphi]\!] = \{\sigma \in \mathcal{S} \mid [\![\varphi]\!]_\sigma = \mathbf{true}\} \subseteq \mathcal{S}$.

Such an assignment translates to $\text{choose}((\varphi_i.\text{set}(\{v \mapsto i\}))_{i \in \mathcal{V}})$ with $\varphi_i \equiv [\![t]\!]_\sigma = i$. Thus, the assignment of variable v is updated to the value $[\![t]\!]_\sigma$ of term t in state σ.

We also introduce a macro $v' \leftarrow \text{call}(r, m, v)$ for $\text{call}(r, m, v).(v' \leftarrow res)$ to actively process the return value of method calls. We introduce $\text{return}(n)$ with $n \in \mathscr{N}$ for $\text{choose}((\varphi_i.\text{return}(i).\text{success})_{i \in \mathcal{V}})$ with $\varphi_i = \{s \in \mathcal{S} \mid s(n) = i\}$. We write return for $\text{return}(\perp)$.

Finally, we introduce a macro for case distinction: $\mathbf{if}\, \varphi\, \mathbf{then}\, P_1\, \mathbf{else}\, P_2\, \mathbf{fi}$ translates to $\text{choose}(([\![\varphi]\!].P_1), ([\![\neg\varphi]\!].P_2))$ and makes things just more readable.

Of course, these definitions allow for a great many of malicious formulations: Method calls parameter tuples might contain less elements than their method requires, or variables might become addressed that are not bound (and thus undefined in the state function, or, worse, defined by a parameter of a previously processed method). We assume, however, that the programmer of a component's method evaluator does not make such (deliberate) errors. Since it is the programmer's responsibility to prove the correctness of a component's specification, any such erroneous specification will only hinder the task.

EXAMPLE 4.1. *Let us assume a component with two roles r_1 and r_2, which both require interfaces that include a method m. We might specify a method $m_1()$ that repeatedly calls this method alternately on the roles:*

$$\begin{aligned} \mu L \Rightarrow (cnt \leftarrow cnt + 1).\mathbf{if}\, (cnt\ mod\ 2 = 0)\, &\mathbf{then} \\ &\quad \text{call}(r_1, m, \langle\rangle).L \\ &\mathbf{else} \\ &\quad \text{call}(r_2, m, \langle\rangle).L \\ &\mathbf{fi} \end{aligned}$$

Another method $m_2(p)$ might check if the parameter p has been passed previously, and call m on r_1 if it has not been seen before, otherwise on r_2:

$$\begin{aligned} \mathbf{if}\, (v \in seen)\, &\mathbf{then} \\ &\quad \text{call}(r_2, m, \langle\rangle) \\ &\mathbf{else} \\ &\quad (seen \leftarrow seen \cup \{v\}).\text{call}(r_1, m, \langle\rangle) \\ &\mathbf{fi}.\text{return} \end{aligned}$$

A third example calls m on both r_1 and r_2 and adds the results:

$$(v_1 \leftarrow \text{call}(r_1, m, \langle\rangle)).(v_2 \leftarrow \text{call}(r_2, m, \langle\rangle)).(v \leftarrow v_1 + v_2).\text{return}(v)$$

We need to calculate the result in v, as we have not defined $\text{return}(v_1 + v_2)$*, but it would be fairly easy to define an appropriate macro.*

Throughout this thesis, we will use this "programming language" liberally, and since its informal intend is so obvious, often use it without explicitly mentioning it.

CHAPTER 5

A Case Study of a Synchronous Component Model: The CMC Model Checker

Wenn ich sechs Hengste zahlen kann,
Sind ihre Kräfte nicht die meine?
Ich renne zu und bin ein rechter Mann,
Als hätt ich vierundzwanzig Beine.
— Johann Wolfgang Goethe, Faust I

Developing the "component-based model checker" (CMC) [HW06a] was an interesting experience which profoundly influenced the view on components taken in this thesis. The success of developing a model checker with fine-grained, "no longer to be profitably divided" components led to adopt this perception of component granularity for the research presented in this thesis. While CMC was never subjected to (runtime) reconfiguration, we still report on the experience obtained, as it justifies many design decision taken later. For a direct understanding of our reconfiguration approach, this chapter is not required, however, and should be considered as a (hopefully interesting) detour.

The idea of developing a component-based model checker stems from the experience obtained by implementing LwaaSpin [HKM05] which is an extension of the Spin model checker [Hol03]. Spin, however, is not a model checker itself, but

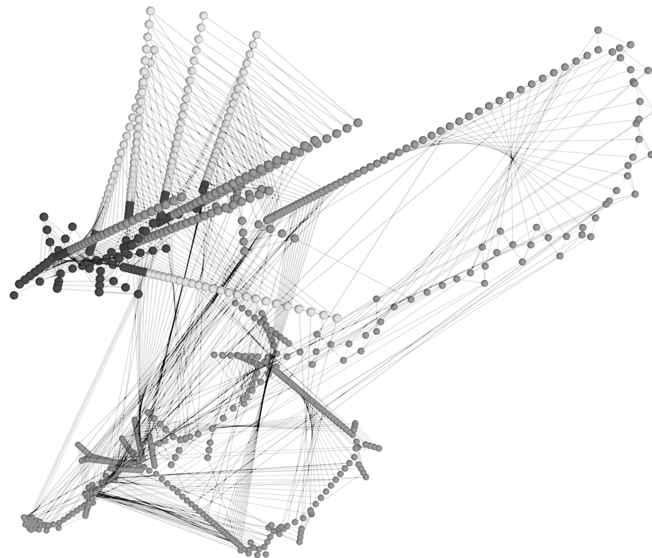


Figure 5.1: State Space Visualization for Peterson's Mutual Exclusion Protocol [Pet81], 3 processes

a code generator. For a given PROMELA input file, source code of a customized model checker is generated. By compiling it, a made-to-measure model checker is obtained, which excels at saving time and memory (since many choices can be made at compile time, e.g., user settings can be compiled into the model checker, making case distinctions unnecessary). Extending such a framework requires a rather hideous amount of hacking, and we hence became interested in developing a model checker capable of quick module replacement; an approach that is of interest to a wide user group (e.g., in personal communication regarding [**SE05**] and [**LSW03**]).

After some initial steps, a quite specialized model checker emerged which utilizes a special technique. This is but one direction this research might have taken. The idea of CMC was well-received by the scientific community [**Eur06**]. For this thesis, the contribution is given by the insights obtained during the development process. Both the benefits (e.g., rapid development of algorithm variations) and the drawbacks (e.g., performance loss) of component-based software development become evident in this rather narrowed view on the subject. Also, the design decisions about the component model sparked ideas that influenced the design of the more mature JCOMP component model, discussed in Sect. 6.2. Other than these immaterial contributions to the overall view on components, and the most beautiful illustrations of this work, this chapter can be considered to be a little detour within this thesis, covering (and explaining) areas otherwise not of particular interest.

Model checking is a technique that has grown to become a well-established approach of formal methods. Basically, it amounts to searching for a node (or, more generally, a subgraph) satisfying a certain property in a finite graph (i.e., a graph with finite V and E). The graph, however, is not given in an explicit description (i.e., by a tuple as introduced in Sect. 4.3). Instead, it is described, e.g., by a term rewriting system as introduced in Sect. 4.4.4; but most often, an (abstract) programming language is used. Obviously, such a description can be vastly more succinct than the final graph.

Explicit model checking then progressively generates the graph from the description, and checks if the sought-after feature is found. For example, the graph might be described by a short program, the formal semantics of the programming language, and an initial state; with the graph representing program states as nodes and statement executions as edges (usually, an LTS is used as the underlying formalism for the graph). The search might then look for a node representing a program state where some exception has been thrown, or where a deadlock is encountered; or it might look for a run in which the program never does something that it is desired to do (e.g., terminate).

The benefit of model checking is that many problems in computer science can be transformed to graphs, with examples provided throughout this thesis. If a matching state is found, a *counter-example* can be provided that shows a way from the initial state to the state just found. Since the sought-after state usually resembles an error, the series of events leading to this error can be assessed. The downside of model checking is its vast memory requirement, since even very short descriptions can produce exuberantly large graphs. From an abstract view, most of the research done on model checking concerns ways to handle such large graphs, and a large number of approaches have been proposed, cf. [**CGP00**]. The CMC model checker described in this chapter offers a solution that combines some of the well-established results, and provides competitive performance.

This chapter starts with a brief explanation of the genesis of the CMC model checker, because it was an interesting experience with components in a field where they are usually not utilized; followed by the formal description of the component model, using the definition given in Chapter 4. We will then describe the CMC model

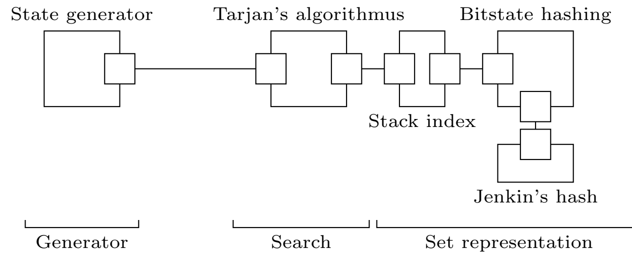


Figure 5.2: First ideas of CMC, with the three abstract components

checker as an example for the synchronous model, and briefly illustrate an extension to parallelize it. For completeness, and for preparing the discussion related to reconfigurability in Sect. 8.4.4, we will finish this chapter with some benchmarking results.

5.1. Evolution of the CMC Model Checker

5.1.1. First Ideas. An explicit model checker basically consists of three abstract components: First, a generator of the graph, as described above. This generator can calculate the set of states directly reachable from a given state, as well as produce the initial state from which the graph exploration starts. Second, a set representation is used to store which states have already been seen, in order to avoid revisits and generating a circle of states over and over again. Third, a search component is required to coordinate these two components by fetching successor states from the generator, querying the set of seen states about which ones are new, and recursively process these. Obviously, a large number of components can (and need to) be defined that refine these three basic components.

We need to mentioned that the set representation (which is almost always done with a hash table, with a notable distinction presented in [**HP99**]) is where. more or less, most of the memory is spent. Ultimately, the memory requirements of this set representation become the limiting factor of the model checker's capabilities.

After pursuing some initial ideas on how a component-based model checker might be built (as shown in Fig. 5.2), a lightweight component framework was programmed that employs the standard method invocation of C++ to perform component communication. After a meeting with Michael Weber, we used the NipsVM [**Web07**] to investigate ideas introduced in [**BLP03**] which amount to not storing every state during a search, but only a few, while deliberately accepting parts of the search-space to be visited twice. This helps to preserve memory, at the cost of running time. But we were not able to reproduce the good results reported in [**BLP03**] (also cf. [**PITZ02**]) and achieved little more than good-looking graphics, generated to understand the structure of state spaces (cf. Fig. 5.1).

We called the hash table that was allowed to forget states a *lossy hash table*. It is an *under-approximation* of a set, i.e., it can tell you that an element is in a set, but it cannot be sure if an element is not included (as it might have been lost earlier). The key idea of CMC is to combine this with another component that provides a *over-approximation* of a set, which is provided by Bloom filters [**Blo70, Hol98, DM04**]. Together, we obtain a three-valued decision algorithm, and for those states that get a "don't know" answer, the (large and cheap) magnetic disk can be used to resolve their state.

```
open ← {initialState}
candidate ← ∅
disk ← ∅

while(true) do
  if open = ∅ then diskLookup() fi
  State s ← open.removeState()
  for s′ ∈ succ(s) do
    if bloom.isUnvisited(s′) then
      addToOpen(s′)
    else
      if ¬lossy.isVisited(s′) then
        candidate ← candidate ∪ {s′}
      fi
    fi
  od
od

funct addToOpen(State s) body
  open ← open ∪ {s}
  bloom.add(s)
  if lossy.isFull() then
    State s′ ← lossy.reclaim()
    disk ← disk ∪ {s′}
  fi
  lossy.add(s)
endfunct

funct diskLookup() body
  for s ∈ disk do
    if s ∈ candidate then
      candidate ← candidate \ {s}
    fi
  od
  if candidate = ∅ then
    terminateSearch()
  fi
  for s ∈ candidate do
    addToOpen(s)
  od
endfunct
```

Figure 5.3: Pseudo-code for the CMC algorithm

Thus, the first version of CMC involved a chain of approximations of sets, each of which could either declare a definitive result, or delegate the check to the next element of the chain. For example, the lossy hash table can give either a "state visited" if the state is found in the hash table, or a "state unvisited" if it is not found and no state has yet been removed from the table to free its memory. If both criteria are not met, the next element in the chain is queried – as can be seen from Fig. 5.13 on page 108, this happens to be a "terminator" component that closes the chain (the exact pattern will be discussed in Sect. 8.3.1). The only purpose of this component is to declare that the approximations all failed, to count the number of times this happens, and to return the call to the search component, which then adds the state to the candidate set. Thus, a series of components act together to implement an algorithm that is given in pseudo-code notation in Fig. 5.3.

5.1.2. Speeding Up the Disk Access. When we use the disk to resolve states that cannot be resolved by the in-memory caches we need to consider the technical constraints imposed by using a magnetic disk: Linear reading is quite fast, while random access becomes prohibitively slow. The usual approach to address this issue is to do a sequential scan of the disk [**SD98**, **BJ05**] and compare the read states to those in memory, the so-called *candidate set*. We also employed this approach, but still found it to be slow – during a disk lookup (done when the candidate set becomes too large to be kept in memory) the CPU is mostly idling. Hence, we had computation time available during the I/O operations, which we spent on compressing the data written to disk using the ZLIB [**GA95**]. This produces not only much smaller disk files, but it also speeds up the entire process as the CPU is now fully involved during a disk lookup (cf. Fig. 5.5).

method	pages		runtime	relative time for	
	written	read		I/O	preparation
HC, 5 byte	896	6872	9:21	7.89%	2.13%
HC, 8 byte	1472	11285	9:48	12.36%	2.02%
zLIB	2880	23205	14:03	4.13%	22.27%
plain disk	56816	438896	47:12	74.84%	—

Table 5.1: I/O effort for verifying the `pftp` protocol (queue size 11, 12.9 million states), as provided with the SPIN model checker. *preparation* describes the part of the algorithm that processes data that is written to or read from disk; for hash compaction (HC, [**WL93**]), this consists of calculating hash values for the state, whereas for the ZLIB-using algorithm, this contains de- and inflating disk pages.

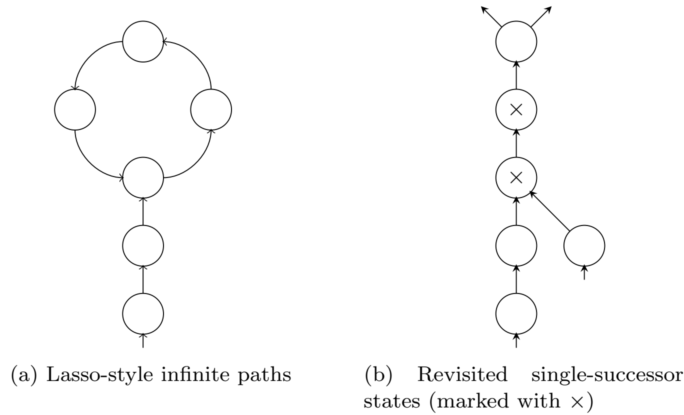
Model	States	non-AA states	Edges	AA revisits	zLIB compr.	avg. SV	SV 0's
HUGO: Hot Failover[1]	4.5M	3.8M	8.5M	5.1M	3.20%	756.6	68%
Peterson, $n = 5$	68.9M	68.9M	378.9M	0.0M	9.06%	148.5	72%
pftp, queuesize=11	14.3M	12.9M	38.7M	5.8M	4.15%	298.0	56%
Lunar, scenario 4(b)	3.3M	0.7M	3.9M	0.9M	5.04%	595.5	42%
Dining Phil., $n = 9$	4.6M	4.5M	12.2M	3.2M	9.95%	95.0	51%
Leader election	8.2M	8.2M	58.4M	0.0M	2.42%	788.0	70%

[1] This is a scaled-down model. For the full model, see Table 5.4

Table 5.2: Some statistics for smaller models checked in CMC. *States* is the total number of states, *non-AA states* the number of states with more than one successor, *Edges* the number of totally generated states (including revisits), *AA revisits* the number of revisits that are made to auto-atomic-filtered states, ZLIB *compr.* the compression ratio achieved by compressing the states written to disk (5% is a twenty-fold compression), *avg. SV* is the average state vector size in bytes, and *SV 0's* the percentage of bytes that are 0 in the state vectors.

Tab. 5.2 shows the compression obtained for some models. A twenty-fold compression is regularly achieved, which is quite comprehensible given the large size of a state vector for large models, and the comparatively limited number of states – if the states did not contain large sequences of zeroes and many repetitions, the state space size obtained for a model with v state vector bytes would be much closer to the theoretical limit of 8^v. For example, scenario 4(b) of the LUNAR protocol suite [**WPP04**] has an average state vector size of 664 bytes and a state space with 248 million states; obviously, large parts of the state vector need to be rather similar, and as the last column of Tab. 5.2 shows, a large fraction of the state vector actually consists of zeroes.

Adding a state compression is made easy by the component model: It just requires the addition of a component, located between the candidate resolving component and the disk query component (⑦ in Fig. 5.7). For a different compression algorithm, one just needs to replace the compression component; given a little care



(a) Lasso-style infinite paths

(b) Revisited single-successor states (marked with ×)

Figure 5.4: Problems with the "Auto-Atomic Filter" improvement of CMC

at the interface design, no surrounding code needs to be modified. Assembling different configurations that compare different approaches as shown in Tab. 5.1 can be done in a matter of a few minutes, once the different components are written.

5.1.3. Further Experiments. Another invention utilized in the CMC model checker is the "auto-atomic filter". This component is plugged between the search and the state generator component. The state generator is asked by the search component to produce a list of successor states for a given state. If there is a single successor state only, the auto-atomic filter queries that state again, until a list with more than one successor state is found; only this list is returned to the search. This is a very easy (and controllable) variation of the ideas we first pursued, building on [**BLP03**].

There are two problems associated with this approach, illustrated in Fig. 5.4: First, the search might get stuck in a lasso-style infinite path. Second, some states might be revisited multiple times, if a state within a single-successor chain is the successor of multiple states. The first problem – which actually exists, e.g., in the GIOP protocol [**KL00**] – can be mended by allowing only a limited length for single-successor chains; when pursuing the lasso, eventually a state will be recognized as revisited. The second problem is deliberately accepted: While a certain number of states is revisited (and this might account for as much as half of the total of visited states) the reduction of the number of states that need to be considered is often worth the effort (cf. Tab 5.2, the difference between the first data column (total number of states) and the second column is the number of states not stored due to the auto-atomic filtering; the forth column gives the number of states superfluously revisited). Even if more states are visited, the number of states that need to be stored is cut down, and their number ultimately becomes the limiting factor regarding the ability to check a given model.

5.2. The Component Framework of CMC

The CMC model checker utilizes the modularity of components, but at the same time tries to be as efficient as justifiable. This is reflected in the choice of its implementation language (which also serves as the component host language), and the design of the component model.

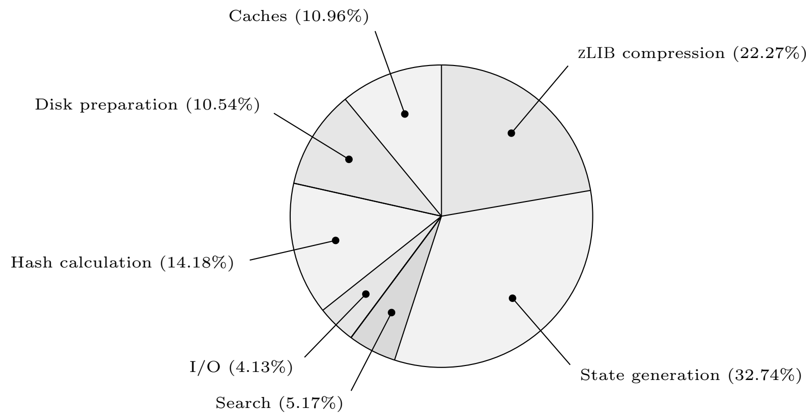
5.2.1. Choice of Programming Language and Communication Paradigm. CMC is written in C++ [Str86]. Most model checkers are written in C, e.g., SPIN [Hol03], NUSMV [CCGR99] and MURφ [DDHY92], or C++, e.g., MAGIC [CCG+03]. Notable exceptions are BLAST [HJMS03], which is written in OCAML, or the JAVA PATHFINDER [VHB+03], which is written in JAVA, but both model checkers are struggling with theoretical limitations more than with actual efficiency concerns. C and C++ are still superior in terms of abilities to save memory, and in the availability of highly optimized components like hash functions, e.g. [Jen97, App08]. We chose C++ since the component model could benefit from the provided class mechanism, but inside the components, the actual language we use is C (i.e., there is no object orientation within the components, save for a few minor examples where structures might have been used as well). Communication is conducted by using the standard C++ method invocation on the required interface; the reference to the providing component is directly injected into the caller component. This is quite similar to the approach of KOALA [Omm98, OLKM00], which also uses direct C method invocation for inter-component communication.

5.2.2. The Component Preprocessor. The component model utilizes code generation for the glue code. The components are defined in a number of header files, with some macros defining their specific properties (e.g., which attributes are to be understood as communication roles and to be bound by the assembly glue code). The assembly is defined in a special file. Given such a set of header files and an assembly, a preprocessor generates component wrappers (i.e., subclasses of the actual components that provide methods for attaching the ports and setting the parameters) and the assembly glue code. All this is then compiled into the model checker. If multiple assembly files are provided, glue code is generated for each, and the actual assembly can be chosen using a runtime switch. However, we never used that feature, since compiling the assembly code becomes time-consuming quickly.

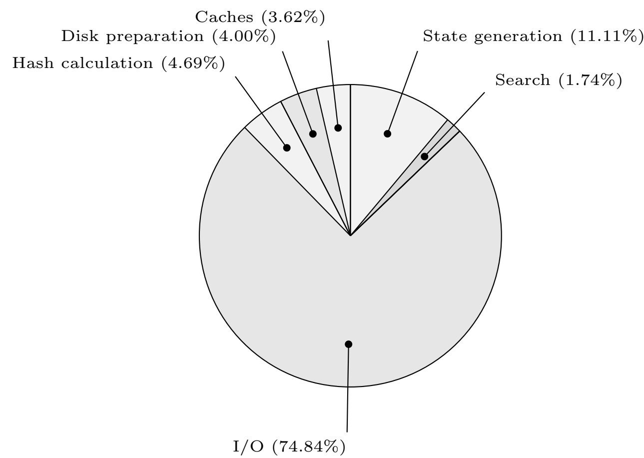
The preprocessor also generates filter code. Using the filter pattern described in Sect. 8.1.2.1, a number of debugging/profiling filters is generated. They can be used to monitor the communication of components. For example, Fig. 5.5 was built using a filter that stored which component the current thread is in (as discussed below, there is only one thread in standard CMC). Using a timer, this value was frequently written to a file, and then accumulated for a set of "tasks", which are shown in the diagrams.

More interesting than the structure are the decisions made about the communication of components. For many component models, it is somewhat silently assumed that components each run in their own thread. For CMC, however, this is infeasible, as the overhead of thread switching is too high, as later proven by experiments with a multi-threaded version (see Sect. 5.3.2). Instead, a single thread "traverses between components", using the usual method call mechanism of C++. So, instead of enqueueing a method object into the target component's queue and wait for the dedicated thread of the target component to dequeue and process it, the current active thread loads the context of the target component and starts processing the method. Hence, all calls are synchronous, with the possibility of synchronous callbacks (which are often forbidden for concurrent components, cf. Sect. 6.2). A nice picture of such a communication is a token that is passed between components, carrying data with it (either parameters or return values) and eventually going back the path it came (cf. [CTTV04] for an interesting approach to model checking such systems). This is known as *activity flow* [VDBM97].

5.2.3. Shared Memory. For performance reasons, the components of CMC need to share memory. This is highly problematic for a generic component framework,



(a) With zLIB compression



(b) No disk page compression

Figure 5.5: Distribution of CPU time for CMC runs

and proved to be a major source of problems in CMC. However, the largest run we did with CMC (Lunar Scenario 4(f) [WPP04]) produced 2.5 billion states, most of them 745 bytes large (1.72 terabyte within 38 hours and 36 minutes – of course, many of these states are duplicates of already visited ones). Passing a 64 bit pointer requires 93 times less memory than serializing and deserializing the state data. Given that each state is passed to at least six components, and possibly to many more, the infeasibility of not using shared memory should be obvious.

During developing CMC, shared memory became a big problem. Many components can conclude that they no longer need to store a state and need to free its memory (cf. Fig. 5.13, all components connected to the memory manager might free states). For example, any state that is put in the open set needs to be stored in the caches – to avoid adding the state to the open set a second time before the first copy is processed. When memory becomes scarce, the lossy hash table chooses some states to be swapped to disk. Depending on the heuristic used, some of these states might also be stored in the open set. Due to the utilization of shared memory,

both the open set and the lossy hash table point to the same state representations. The lossy hash table might then proceed to write the states to disk and remove them from memory, leading to problems when the state is taken from the open set for processing in the search algorithm.

In the better case, a segmentation fault is encountered which indicates that a state was freed by one component, but is still used by another. However, segmentation faults are a risky business compared to a `NullPointerException`, as they are not guaranteed to happen when a problem occurs – if the memory that is falsely addressed is already given to another component, no such error occurs, and even worse, this can trigger other segmentation faults because of misinterpreted data, which is almost impossible to debug (although nowadays, tools like VALGRIND [**NS07**] are available to find the cause of such problems, but the vast size of many models averts their application, requiring a tedious search for a reproduction of the problem for a small model).

Even worse are memory leaks. CMC is all about saving memory – for the run of Lunar Scenario 4(f) (cf. Tab. 5.4), a total of 232.15 gigabyte of state data was stored in the disk/cache combination. As this vastly exceeds available RAM, there must not be a leakage of even a small fraction of states. However, as states are shared, the various buffers need to execute great care not to invalidate data that is still referenced by another component. Finding memory leaks required a lot of code reading and lengthy test runs.

This experience with CMC illustrate the benefits of a clean separation of the memory space of components, as we realized it in JCOMP later. For CMC, however, the focus is different. We refer to this approach as *lightweight components*, as they form a mere wrapper on the host programming language C++ and are not at all restrictive. Components are thus understood as a design concept, and support is provided for their use, but nothing is prohibited or hidden. This has been criticized as being nothing but "mere object orientation" – and of course, we use even less than object orientation usually provides (e.g., no inheritance – although we use inheriting components to avoid duplicate code, we do not use dynamic dispatch other than for port interfaces). Components for CMC are a means for reasoning about algorithms, understanding the data flow, and rearranging them for easy comparisons.

5.2.4. The Formal Model. We will now describe the LTS that defines the dynamic behavior of a component setup in the CMC framework. As mentioned before, components share memory, hence the accessible data is global – all of it, since components are free to communicate a pointer to any data to other components. This approach is possible because we model a *closed system* (cf. [**KS99**]) – there is no "attacker" that might manipulate publicly accessible data. Instead, all components are known, and if there is some misuse of shared memory, this is a fault of the system designer who assembled incompatible components (cf. the discussion in Sect. 9.1.2.2).

However, there is data to be stored for each component that cannot be accessed by the other components: A stack that keeps track of the currently executed method, the method invocation parameters and the component process term to be executed next. Further component-private data are the return value of the last method called on some other component and a function $\gamma(c)$, which tells us how this component is connected to other components. This function needs to be part of the state, as the connection of components might change during runtime due to reconfiguration. Additionally, each component has a flag that indicates whether it is *running*, i.e., currently executing a method, or *blocked* – waiting for becoming called, or for a method call to return.

$$c_1^{\mathrm{r}}, \langle (c_1^1.m_1^1(p_1^1), \mathrm{call}(r,m,v).P), \pi_1\rangle, \gamma_1 \parallel c_2^{\mathrm{b}}, \langle \pi_2 \rangle \rightarrow \qquad (\textsc{call})$$
$$c_1^{\mathrm{b}}, \langle (c_1^1.m_1^1(p_1^1), P), \pi_1\rangle, \gamma_1 \parallel c_2^{\mathrm{r}}, \langle (c_1.m(v), \zeta(c_2)(m)), \pi_2 \rangle$$
$$\text{if } c_2 = \gamma_1(r)$$

$$c_1^{\mathrm{r}}, \langle (c^1.m^1(p^1), \mathrm{call}(r,m,v).P), \pi\rangle, \gamma \rightarrow \qquad (\textsc{scall})$$
$$c_1^{\mathrm{r}}, \langle (c.m(v), \zeta(c)(m)), (c^1.m^1(p^1), P), \pi\rangle, \gamma$$
$$\text{if } c = \gamma(r)$$

$$c_1^{\mathrm{r}}, \langle (c_2.m(p), \mathrm{return}(r).P), \pi_1\rangle \parallel c_2^{\mathrm{b}}, r_2 \rightarrow c_1^{\mathrm{b}}, \langle \pi_1 \rangle \parallel c_2^{\mathrm{r}}, r \qquad (\textsc{ret})$$

$$c_1^{\mathrm{r}}, \langle (c.m(p), \mathrm{return}(r).P), \pi\rangle, r' \rightarrow c_1^{\mathrm{r}}, \langle \pi \rangle, r \qquad (\textsc{sret})$$

$$\sigma \mid c^{\mathrm{r}}, \langle (c_2.m(p), \mathrm{set}(\sigma').P), \pi\rangle \rightarrow \mathit{upd}(\sigma, \sigma') \mid c^{\mathrm{r}}, \langle (c_2.m(p), P), \pi \rangle \qquad (\textsc{set})$$

$$\sigma \mid c^{\mathrm{r}}, \langle (c_2.m(p), \mathrm{choose}((\Sigma_j.P_j)_{j\in J})), \pi\rangle, r \rightarrow c^{\mathrm{r}}, \langle (c_2.m(p), P_i), \pi\rangle, r \qquad (\textsc{if})$$
$$\text{if } \mathit{ret}(\mathit{prm}(\sigma, m, p), r) \in \Sigma_i$$

$$c^{\mathrm{r}}, \langle (c_2.m(p), \mu X.P), \pi\rangle \rightarrow c^{\mathrm{r}}, \langle (c_2.m(p), P[X/(\mu X.P)]), \pi\rangle \qquad (\textsc{loop})$$

Table 5.3: Rules for the CMC component model

Given the universe $\mathcal{C}$ of components and a set $\mathcal{S}$ of component data states, we define the set $\mathbb{S}$ of states to be comprised of tuples (σ, c) with $\sigma \in \mathcal{S}$ and $c : \mathcal{C} \rightharpoonup \mathscr{S}$. $\mathscr{S}$ is the set of component *configurations*, which are tuples of the form

$$(id, f, \pi, r, \gamma)$$

with $f \in \{\mathrm{r}, \mathrm{b}\}$ being the running flag, $\pi \in (\mathcal{C} \cup \{\bot\} \times \mathscr{M} \times \mathcal{V} \times \mathcal{P})^*$ the stack of invocations and component process terms, $r \in \mathcal{V}$ the return value last received, and $\gamma \in (\mathscr{R} \rightharpoonup \mathcal{C})$ the connections of the roles.

The universe $\mathcal{C}$ is intended to capture all defined components – which makes it of infinite size, since components might be generated. The domain of the function c must be finite – and those components for which c is defined are the nodes of the component graph of that state. For each $c_i(c) = (id, f, \pi, r, \gamma)$, we require γ to be well-connected and completely connected for c.

For a more readable state representation, we write

$$\sigma \mid c_1^{f_1}, \langle (c_1^1.m_1^1(p_1^1), P_1^1), \ldots, (c_1^{m_1}.m_1^{m_1}(p_1^{m_1}), P_1^{m_1})\rangle, r_1, \gamma_1 \parallel \ldots \parallel$$
$$c_n^{f_n}, \langle (c_n^1.m_n^1(p_n^1), P_n^1), \ldots, (c_n^{m_n}.m_n^{m_n}(p_n^{m_n}), P_n^{m_n})\rangle, r_n, \gamma_n$$

for a state (σ, c) with $c(c_i) = (f_i, \langle (c_i^1, m_i^1, p_i^1, P_i^1), \ldots, (c_i^{m_i}, m_i^{m_i}, p_i^{m_i}, P_i^{m_i})\rangle, r_i, \gamma_i)$.

Tab. 5.3 displays the rules in the format described in Sect. 4.4.4. CALL performs a method call. It blocks the calling component and pushes a new component process term, which is determined by the environment ζ, on the stack of the target component. The target component is unblocked and may then process the component process term. Only four terms are interpreted: return, the setting of the global data, method calls on roles, branching and looping. Hence, neither success nor fail are used. SCALL works just like the rule CALL, but for a self-invocation of a component; it needs to be handled by a distinct rule since no component becomes blocked. Likewise, RET returns a call to another component, while SRET returns a self-invocation.

The blocking and unblocking of components due to the rules CALL and RET models that the current thread is sent to the new component, where it continues; a procedure called *activity flow* [**VDBM97**] that is known from most imperative programming languages.

RET handles a return statement by setting the last return statement of the calling component to the value given, blocking the called component (and removing its topmost stack element – thus also removing any component process term that is following the return statement) and unblocking the calling component. The calling component's process term is already stripped off the call subterm, so execution continues with the next term.

SET sets the global data value. IF selects a process term according to the current global data state, as well as the (component-local) method parameters and the last return value. (This return value is not put on the stack, as it is overwritten by method return anyway.) Finally, LOOP realizes one iteration of a loop $\mu X \Rightarrow P$ by taking the loop body P and replacing any occurrence of X with another instance of the loop.

Obviously, if a state has exactly one component running, any subsequent state will also only have one component running. The rules do not really support multiple components running, unless they do not interfere – i.e., no component is ever called while it is running itself. We will consider such a scenario in Sect. 5.3.2.

Given a component setup (C, M, e) with $C = (c_1, \ldots, c_n)$ and $e = (c_s, m)$, the initial state $s_s = (\sigma, c)$ is built by setting $\sigma = \mathit{upd}(\ldots \mathit{upd}(\iota(c_1), \iota(c_2)), \ldots, \iota(c_n))$ and

$$c(c_i) = \begin{cases} (\mathrm{b}, \langle\rangle, \bot, M(c_i)), & \text{if } c_i \neq c_s \\ (\mathrm{r}, \langle \bot.m(\bot), \zeta(c_i)(m)\rangle, \bot, M(c_i)), & \text{if } c_i = c_s. \end{cases}$$

Obviously, the method m inserted into the initial component c_s cannot ever be left, as the RET rule is not applicable (there is no component named $\bot$). Hence, either the system never exits the method – e.g., by using a loop somewhere – or it blocks at the final return statement.

Given this initial state, the rules now induce a labelled transition system, with the rule instantiations being the labels. Note that a component setup with no method of any component being interpreted by a nondeterministic component process term produces a transition system that consists of a single path, i.e., if $s \to s'$ and $s \to s''$, then $s' = s''$ for any state s with $s_s \to^* s$.

5.2.4.1. *Formulation of Loops by Self-Invocations.* As it turns out, terms of the form $\mu X \Rightarrow P$ are not truly required – from an *external viewpoint*. They can be substituted by a so-called *self-invocation*: Instead of substituting $\mu X \Rightarrow P$ for every X, we can also substitute a method call to the current component that realizes P: $\mu X \Rightarrow P$ has a comparable effect as $P[X/\mathrm{call}(\mathit{self}, m_P, v).\mathrm{return}(r).\mathrm{success}]$ has.

Of course, at the level of labelled transition systems, a slightly different behavior will be observed: For every iteration using such a self-call, a CALL rule will be applied, whereas for the iteration with substitution, we use the rule LOOP. For the former, the stack will also grow. This is illustrated in Fig. 5.6; the $\rightsquigarrow$ arrows are arbitrary sequences of states, being the same (with respect to the labels executed; the stack obviously differs) in both run variants. But if we take a point of view *distant enough*, the systems act the same: They are weakly bisimilar with respect to a set of labels that does not contain the rule instantiations of LOOP and the self-invocations. For example, we might restrict our interest to truly outgoing communication:

LEMMA 5.1. *Let c be a component with self-invocation capabilities as described above in a component setup (C, M, e). Then the communication behavior of c (excluding self-invocations) remains the same if we substitute, for each method $m \in \bigcup_{i \in I_P(c)} i$, $\mu(c)(m)$ by $\mu(c)(m)[\mu X \Rightarrow P / P[X/r \leftarrow \mathrm{call}(\mathit{self}, m_P, v).\mathrm{return}(r)]$.*

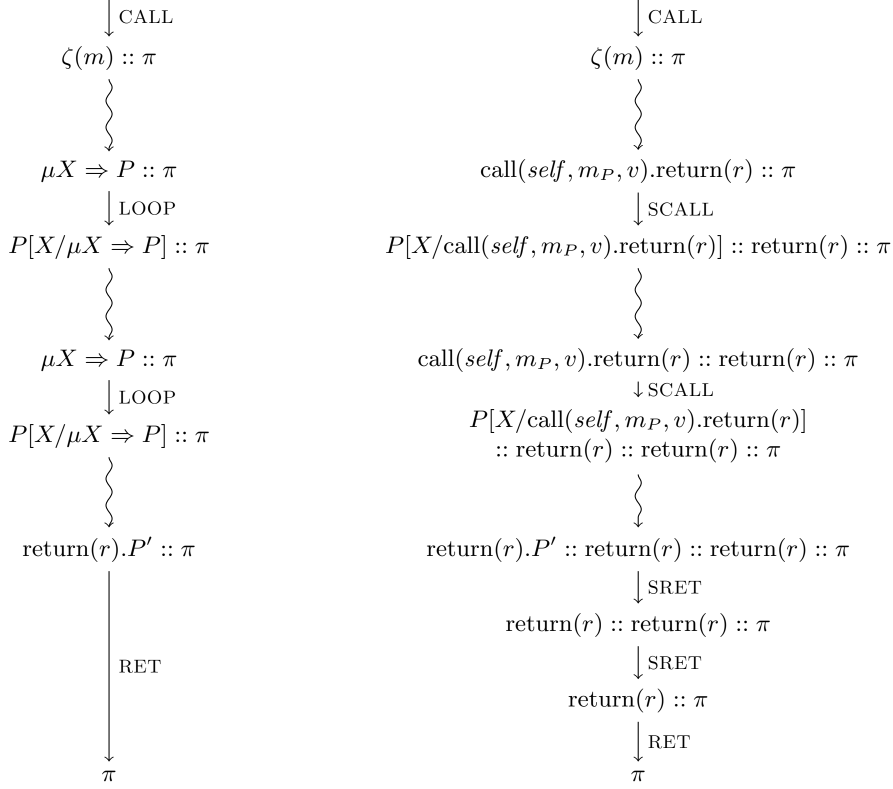


Figure 5.6: Example runs comparing loops and self-invocations

Proof. We show that the LTS $\mathcal{L}$ of (C, M, e) is "almost" weakly bisimilar – with respect to the outgoing communication of c – to the LTS $\mathcal{L}'$ of (C', M, e) with C' being identical to C with exception of the method evaluator of c, which is modified as described. "Almost" refers to the fact that a rule $l \in \textsc{call}(c_1 : c)$ applied in $\mathcal{L}$ cannot be applied directly in $\mathcal{L}'$ – since the stack of c is different. But if $l \in \textsc{call}(c_1 : c, \overline{c_2 : c_2})$ can be applied to a state s of $\mathcal{L}$, then a $l' \in \textsc{call}(c_1 : c, c_2 : c_2)$ can be applied to a state s' of $\mathcal{L}'$ with $s \sim s'$. Hence, by restricting the set of labels of $\mathcal{L}$ and $\mathcal{L}'$ such that the stack is not explicitly mentioned in the labels, we obtain weakly bisimilar systems, and Lemma 4.1 still applies, which is sufficient to show that the communication traces are the same. For an LTS $\mathcal{L}'' = (S, L, T)$ write $\mathcal{L}''|_{comm}$ for the LTS $(S, \{\tau\} \cup \bigcup_{c' \in \mathcal{C}} \textsc{call}(c_1 : c_1, c_2 : c_2), T')$ with

$$T' = \{(s, l, s') \mid (s, l', s') \in T \wedge \\ l = \begin{cases} \textsc{call}(c_1 : c_1, c_2 : c_2), & \text{if } l' \in \textsc{call}(\overline{c_1 : c_1}, \overline{c_2 : c_2}) \\ \tau, & \text{otherwise.} \end{cases} \}$$

We set $L = \bigcup_{c_1, c_2 \in C} \textsc{call}(c_1 : c_1, c_2 : c_2)$. $\mathcal{L}|_{comm}$ and $\mathcal{L}'|_{comm}$ are weak bisimilar with respect to L due to the relation $\sim$, which is the largest relation satisfying: If $s \sim s'$, then $s = \sigma \mid c^f, \pi, r, \gamma \parallel c_1^{f_1}, \pi_1, r_1, \gamma_1 \parallel \ldots \parallel c_n^{f_n}, \pi_n, r_n, \gamma_n$ and $s' = \sigma \mid c^f, \pi', r, \gamma \parallel c_1^{f_1}, \pi_1, r_1, \gamma_1 \parallel \ldots \parallel c_n^{f_n}, \pi_n, r_n, \gamma_n$, and $\pi \approx \pi'$ for a relation $\approx$ that is defined as the smallest relation with

- $\varepsilon \approx \varepsilon$,

- if $\pi = \langle (S, P_1), \pi_1 \rangle$ and $\pi' = \langle (S, P_2), \pi_2 \rangle$ with $P_2 = P_1[\mu X \Rightarrow P'/(P'[X/\text{call}(\textit{self}, m_{P'}, v).\text{return}(r))]]$ and $\pi_1 \approx \pi_2$, then $\pi \approx \pi'$,
- if $\pi = \langle (S, P_1), \pi_1 \rangle$ and

  $\pi' = \langle ((c.m_{P''}(v), P_2), (c.m_{P'''}(v), \text{return}(r).P''), \pi_2 \rangle$

  with $P_2 = P_1[\mu X \Rightarrow P'/(P'[X/\text{call}(\textit{self}, m_{P'}, v).\text{return}(r))]$ and $\pi_1 \approx \pi_2$, then $\pi \approx \pi'$.

For $s = \sigma \mid c_1^{f_1}, \pi_1, r_1, \gamma_1 \parallel \ldots \parallel c_n^{f_n}, \pi_n, r_n, \gamma_n$, we write $\pi(s(c_i))$ for π_i (hence, if $s \sim s'$, we have $\pi(s(c)) \approx \pi(s'(c))$ and $\pi(s(c')) = \pi(s'(c'))$ for all $c' \neq c$).

$\sim$ is indeed a weak bisimulation relation with respect to L:

- $\sim$ is a weak simulation relation: Let $s \sim s'$, and $s \xrightarrow{l} t$. Note that due to $\pi(s(c)) \approx \pi(s'(c))$, the next component process term to execute is the same for s and s', unless the next applicable rule is either LOOP or RET.
  - $l \in L$. Since the component process terms to be consumed next are the same in s and s', and since all rule-relevant data is the same due to $s \sim s'$, we can advance s' directly (with an almost similar rule application that differs only in the stack) to t' with $t \sim t'$.
  - $l = \tau$. For most component process terms, the same reasoning applies. Two special cases need to be considered:
    * The component process term to be executed next in s is a loop construct $\mu X \Rightarrow P$: Since $s \sim s'$, we know that the component process term to be executed next in s' is a self-invocation with m_P. The component process term in t will be $P[X/\mu X \Rightarrow P]$. By advancing s', a new method call will be pushed on the stack, which takes the form $\langle (c.m_P(v), P_2), (c.m_P(v), \text{return}(r).P'), \pi_2 \rangle$ with P_2 as required by substituting loops in P with self-invocations. Since π_1 and π_2 are not modified, we have $\pi_1 \approx \pi_2$ and hence $t \sim t'$.
    * The component process term to be executed next in s is a return statement: the component process term to be executed next in s' is also a return statement, but once it is executed, further return statements might be required to be processed. This is possible since such a return processing is labelled with τ in $\mathcal{L}'|^c_{comm}$. After executing as many additional return statements as the loop had iterations, the method that got terminated in the transition from s to t is also terminated, obtaining a t' with $t \sim t'$.
- $\sim^{-1}$ is a weak simulation relation: Similarly, most of the time the component process terms are the same. They differ only if a self-invocation is made in order to substitute a self-invocation, in which case the unmodified term can execute a LOOP rule instantiation and obtain a matching stack just as described above. They also differ if a number of return statements needs to be executed after a loop has been terminated (by a regular return component process term). Again, we are guaranteed that once sufficiently many SRET rule instantiations are executed, we obtain a suitable stack.

Hence, $\mathcal{L}|_{comm}$ and $\mathcal{L}'|_{comm}$ are weak bisimilar with respect to L, and due to Lemma 4.1, we have $\textit{Traces}(\mathcal{L})|_L = \textit{Traces}(\mathcal{L}')|_L$, and since L covers all communication behavior of c excluding self-invocation, the communication behavior of c in (C, M, e) and (C', M, e) is identical. □

This lemma is interesting because of two aspects: First, self-invocation as a replacement for loops is an important prerequisite for reconfiguration, which usually

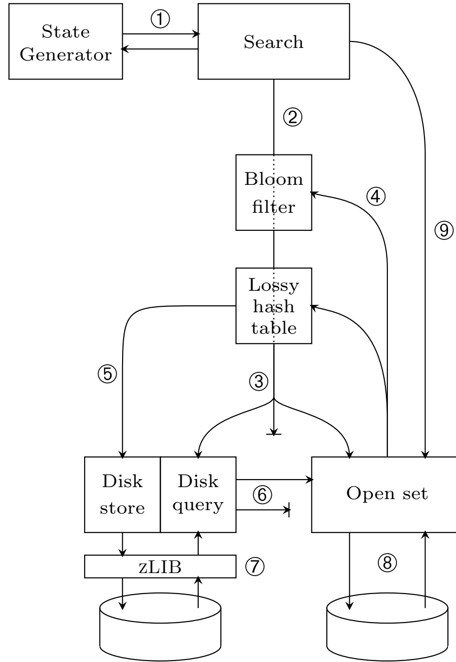


Figure 5.7: Life-cycle of states in CMC; arrows depict the data-flow of states

does not interrupt method execution; it is often necessary to substitute loops by self-invocation to allow for reconfiguration interrupting an otherwise eternal loop (cf. Sect. 7.5.1.1 and reconfiguration examples like in Sect. 8.4). Second, it serves as an example for the separation of roles: If we are just interested in the (outgoing) communication of a component, the implementation detail (genuine loop versus its emulation by self-invocation) does not concern us. Lemma 4.1 asserts us that weak bisimilarity of the systems results in the same observable behavior of components, here given by the sequence of outgoing communication.

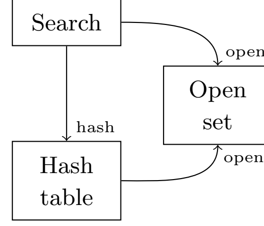
5.3. Data Flow of the CMC Model Checker

In CMC, states are passed between components as indicated in Fig. 5.7. We maintain different sets of states:

- The *open set* consists of all states that have been built, found to be unvisited but not yet processed,
- the *candidate set* is a set of states that need to be checked against the disk to verify that they have not yet been visited, and
- the *reclaim set* is a set of states that are stored in the lossy hashing, but have been scheduled for write-out to disk in order to save memory.

New states are built as either the initial state or successor states to a state taken from the open set. These states are passed to the main search algorithm (① in Fig. 5.7).

The main search then queries the caches (②), with a three-valued outcome: Either the state is visited for sure – as it is found in the lossy hash table. Or it is found to be definitely unvisited – as a bit is missing in the Bloom filter. Or both caches are inconclusive. The further processing depends on that answer (③): Previously visited states are discarded; unvisited states are added to the open set, which immediately inserts the fresh state into the caches so that it is not added to



(a) Open set and non-lossy hash table

`Search::main`	$\equiv$	$\mu L \Rightarrow (\text{call}(open, fetch, *).$ $\text{choose}((\mathcal{S}.\text{call}(hash, check, (s_i)).L)_{s_i \in States}))$
`Openset::fetch`	$\equiv$	$\text{choose}([s_i \in open].$ $(open \leftarrow open \setminus \{s_i\}).\text{return}(s_i).\text{success})_{s_i \in States}$
`Openset::add`(v)	$\equiv$	$(open \leftarrow open \cup \{v\}).\text{success}$
`Hashtable::check`(v)	$\equiv$	**if** $(v \notin hashtable)$ $hashtable \leftarrow hashtable \cup \{v\}.$ $\text{call}(open, add, v)$ **end**.success

(b) Component process terms

Figure 5.8: Specification of the simplified core algorithm

the open set a second time (④), and states without definitive information are added to the candidate set.

By adding new states to the caches, the lossy hashing finally runs full, and a reclaim set is chosen. This set is written to disk (⑤) and states from the set are removed if room is required for new states sent from the open set. Writing states to disk is done in pages, which can be compressed by zLIB (⑦, see Sect. 5.1.2).

Once the open set becomes empty, or the candidate set becomes too large, a disk lookup is triggered. We use the approach of checking the complete set of states written to disk in a linear fashion, comparing each state against an index built from the candidate set; as discussed in Sect. 5.1.2, this circumvents the dreadful performance of magnetic disks for random access requests. Unvisited states are added to the open set, while visited states are discarded (⑥). It is imperative to maintain the invariant of having no duplicates within the open set. This is easily achieved by removing duplicates when building the index.

In order to maintain a definitive upper bound on memory consumption, we still need to avoid uncontrolled memory usage by the open set. Since each state enters the open set only once, this is easy to do by dumping parts of it to disk, should it become too large, and reloading them if the in-memory open set becomes empty (⑧).

Finally, a fresh state is taken from the open set by the search algorithm to have its successors computed (⑨).

5.3.1. Verifying the specification. We can proceed to prove properties of the application itself. For example, a very abstract notion of the use of the open set with a non-lossy hash table is given in Fig. 5.8, with the method specification in Fig. 5.8b.

For simplicity, the `main` method fetches a state and produces another one non-deterministically, without relating the fetched and the produced state. Similarly, the open set takes a random element, removes it from its data state, and returns

it. The interesting part is the hash table implementation and its use: Rather than querying the hash table whether a new state has been visited, the hash table component gets notified to process the state appropriately. If the state was previously visited, it is discarded. Otherwise, the state gets added to the hash table's internal store and inserted into the open set by a call conducted by the hash table.

Obviously, we are requiring that once a state is added to the open set, it is never added again:

$$\forall q \,.\, r = q \rightarrow \Box q \notin \mathit{openset}$$

For showing this, we utilize three rules: The first one, called the *effect rule*, states that if some condition is satisfied that was not satisfied at the initial state, then some method has been called that executed a set statement that satisfied the condition. The *frame rule* states the opposite: If some property is true and no set statement ever changes it, then it remains true. A third rule is that of *control flow*, stating that if some statement was executed, any preceding choose-statement must have been in an appropriate state – this corresponds to a *weakest precondition* reasoning.

We will give the reasoning in very informal terms, utilizing the temporal logic symbols $\Box\varphi$ for "φ holds now and forever", $\Diamond\varphi$ for "φ will eventually hold" and $\Diamond\!\!\!-\,\varphi$ for "φ has been true at some previous point in time". We write $\mathbf{call}(c.r.m(p))$ for the event of component c calling m on role r with parameter p. This is just a brief glimpse of how we can reason about components and their behavior; and how making communication explicit limits the amount of events that need to be considered. We can show the aforementioned property by demonstrating that the inverted formula is not satisfied:

$$\exists q \,.\, r = q \wedge \Diamond q \in \mathit{openset}$$

We prove this using some lemmas:

(1) $q \in \mathit{openset} \rightarrow q \in \mathit{hashtable}$:
- We see that only *add* can modify *openset*.
- Initially, $\mathit{openset} = \varnothing$.
- Hence, by the effect rule, $q \in \mathit{openset} \rightarrow \Diamond\!\!\!-\,\mathbf{call}(\text{Hash table}.\mathit{open}.\mathit{add}(q))$.
- By control flow: $\mathbf{call}(\mathit{ht}.\mathit{open}.\mathit{add}(q)) \rightarrow \Diamond\!\!\!-\,q \in \mathit{hashtable}$.
- By the rule $A \rightarrow \Diamond\!\!\!-\,B, B \rightarrow \Diamond\!\!\!-\,C \models A \rightarrow \Diamond\!\!\!-\,C$ we obtain $q \in \mathit{openset} \rightarrow \Diamond\!\!\!-\,q \in \mathit{hashtable}$.
- By the frame rule, and the observation that nothing is ever removed from the hash table (and the rule $\Box A \rightarrow A$), we obtain $(\Diamond\!\!\!-\,q \in \mathit{hashtable}) \rightarrow q \in \mathit{hashtable}$.
- By logical reasoning, we obtain $q \in \mathit{openset} \rightarrow q \in \mathit{hashtable}$.

(2) $r = q \rightarrow q \in \mathit{hashtable}$:
- By control flow: $r = q \rightarrow \Diamond\!\!\!-\,q \in \mathit{openset}$.
- Hence, with the first lemma (and the rule $A \rightarrow \Diamond\!\!\!-\,B, B \rightarrow C \models A \rightarrow \Diamond\!\!\!-\,C$): $r = q \rightarrow \Diamond\!\!\!-\,(q \in \mathit{hashtable})$.
- By the frame rule (as in the first lemma's proof), we obtain $r = q \rightarrow q \in \mathit{hashtable}$.

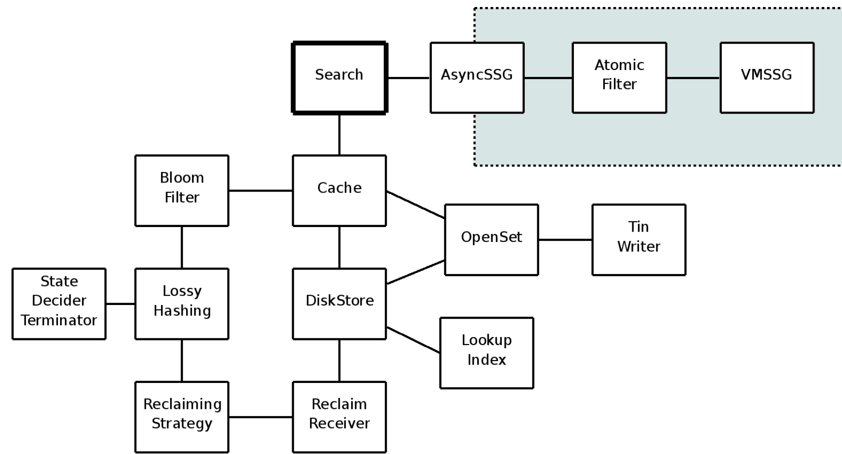
(3) By control flow and induction: $q \in \mathit{hashtable} \rightarrow \Box\neg\mathbf{call}(\mathit{ht}.\mathit{open}.\mathit{add}(q))$.

(4) By lemma 2 and 3: $r = q \rightarrow \Box\neg\mathbf{call}(\mathit{ht}.\mathit{open}.\mathit{add}(q))$.

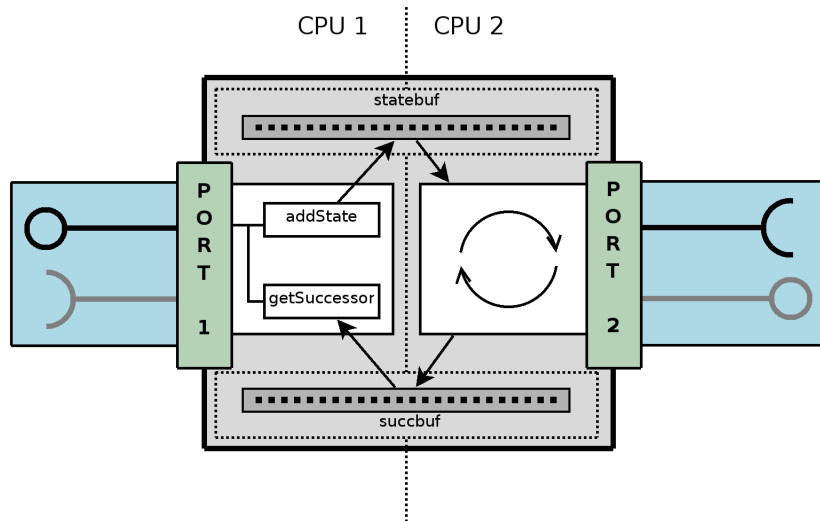
(5) By the frame rule and lemma 4: $r = q \rightarrow \Box q \notin \mathit{openset}$.

Rewriting the last lemma gives $r = q \rightarrow \neg\Diamond q \in \mathit{openset}$, and hence the inversion of the original formula cannot be satisfied.

Note that this is only possible due to the abstraction of the open set being an actual set – in a practical implementation, it is implemented as a list, thus acting as



(a) Separated component setup



(b) CPU link filter component

Figure 5.9: Distributing CMC, pictures courtesy of Philipp Pracht

a multi-set, and the last lemma needs more work; it remains true, but this cannot be deduced from the control flow directly.

5.3.2. Parallelizing the Model Checker. A feature of most modern CPUs is their being equipped with at least two cores. It is always troublesome to see that half the machine's computation power is wasted during an exceptionally long model checking run. Hence, one extension of the CMC model checker was to enable it to run on multiple cores (two, to start with). This requires a parallelization of the former sequential algorithm, providing an interesting case study for an emerging discipline [**PSJT08**, **PT08**].

In the theory of parallel programs, *domain* and *functional decomposition* are distinguished [**Fos95**]. Domain decomposition considers the data and separates it in n equally large packages, handling each on one core (or, more generally, a computation unit). This approach is taken by parallel model checkers [**BLW02**,

LSW03, SD97]. The basic idea is to calculate a hash code for each successor state, which determines the CPU that has to process it. Given a good hash function, each machine has to bear an equal load. By organizing multiple machines in a grid structure, a multiple of memory is made available, at the cost of heavy network traffic.

Functional decomposition, however, considers the algorithm first and identifies various *stages* that can be executed independently (to some extent). This is well-known from modern pipelining architectures of CPUs [OV06], but we are not aware of any attempt to apply this to model checking. The only work that covers multi-core architectures for model checking is based on domain decomposition [HB07].

For CMC, we tried to do functional decomposition [Pra07]. The basic insight here is that successor state generation is to some extent decoupled from the state cache and disk store operation. Any state added to the open set will eventually be handed to the VMSSG virtual machine, so there is no problem in having the successor states calculated right away, in parallel with the revisit checks, as illustrated in Fig. 5.9a.

The interesting thing about such an approach is that parallelizing an application using functional decomposition is a cross-cutting concern, while parallelizing it using domain decomposition changes the core concern (because, ultimately, it changes the data processed by the previous core concern). In practice, handling a cross-cutting concern amounts to adding code that interferes only very little with the former code. For CMC, the "invasiveness" was indeed very limited: Only at the "border" of the parallelization – i.e., between those components that were to be be assigned to different cores – filters had to be added.

Fig. 5.9b illustrates the idea of such a filter. These filter components are the only ones that are ever visited by more than one thread and need to take precautions against race conditions with respect to data access. They implement ring-buffers that decouple the consumption of data between the cores. This decoupling of two sections of the algorithm is very elegant and requires little additional work. In practice, however, things do not work that well: Memory management, a component used by almost every other component, also needs to be provisioned to be queried by more than one thread. This required some less elegant locking.

The results obtained from this parallelized CMC are somewhat discouraging: A speedup was not obtained for most models unless some tedious search for optimal buffer sizes was conducted [Pra07]. Although both cores got reasonable load, an overall speedup was usually not obtained, and never exceeded 30%. This supports the conclusion of [PSJT08]: An automated tuning of the distribution is indeed indispensable. There is ample room for improvement in this regard: Some issues like the calculation of hash codes can be done on either CPU (or even on both, by calculating a free selectable fraction of hash codes on one CPU and the remainder on the other), and the CPU utilization can be modified by changing initial the initial choices. Also, the choices of the parameter values might be adaptively changed, as we will discuss in Sect. 8.5.

5.4. Benchmarking CMC

CMC proved itself to be capable of checking very large models. Tab. 5.4 shows some of the big models that we checked. The GIOP model [KL00] is a specification of the General Inter-ORB Protocol of the CORBA specification, cf. Sect. 3.1.1.1. This model was originally checked in SPIN with bitstate hashing. In our run, we observed 363 thousand failures of the Bloom filter (which actually implements bitstate hashing), and considerably more for the larger models. Each of these failures would have amounted to not visiting a state that has not yet been visited,

Model	States visited	States stored	Edges	Time elapsed	zLIB com.	Uncom. stored	Bloom fail.	Cache fail.
GIOP1	193M	163M	665M	13:34:21	4.81%	79GB	363K	91M
Hugo	556M	205M	865M	15:18:16	3.79%	167GB	128K	32M
Lunar 4(d)	1.3G	248M	1.9G	35:37:29	5.27%	153GB	1.7M	151M
Lunar 4(f)	1.6G	335M	2.6G	38:36:02	5.73%	230GB	13M	387M

Table 5.4: Large models checked in CMC

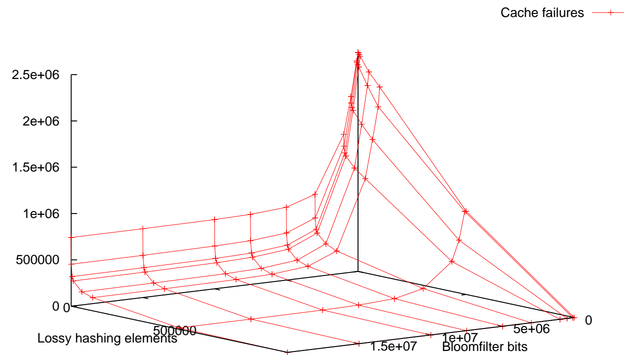
and possible many more solely reachable by that state. Whether this puts the utility of standard Spin for such large models in doubt can hardly be judged. Still, the CMC algorithm shows an approach at checking models of this size without relying on probabilistic algorithms.

Hugo is a model of a hot fail-over protocol [Röl02]. The protocol was modeled in the Hugo/RT model checker [KM02], which generates Promela sources. Lunar [WPP04] is a suite of models for ad-hoc routing protocols. Some of the Lunar scenarios have not been checked exhaustively before.

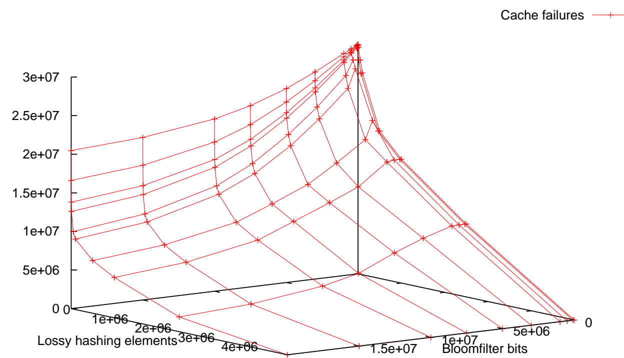
Fig. 5.10 shows two experiments conducted with smaller models by modifying the sizes of the caches between 0 (all requests answered with "don't know") and sufficient size to answer all requests correctly. As can be seen, the number of cache misses (i.e., states that need to be checked on disk) soon diminishes if the caches are given more memory.

Fig. 5.11 shows how the "throughput" of CMC diminishes for the Lunar 4(d) scenario. Note that this scenario is so vast that only at the very beginning the disk is not required. The "smooth degradation" – i.e., the not-too-sudden decrease in throughput as more states are stored – can be attributed to two factors: The decrease of cache efficiency, and the time required for each linear disk check run. This plot also displays the degradation of the caches, which is a little surprising: The caches seem to degrade rather quickly, then even recover a little, before a steady degradation is found (also notice that a decline in the degradation of the throughput and the improvement in cache quality are related). The source of this cache recovery can only be guessed: The heuristic for choosing states for removal from the lossy hash table is based on the assumption of transition locality [PITZ02]. Sometimes, reorganizing the open set (which is also swapped to disk if it grows too big) can devastate the effectiveness of this heuristic. But as one can see, the caches continue to work for the entire run.

Fig. 5.12 shows how various compression levels for the zLIB library have an impact on overall runtime and the compression ratio achieved. After a compression level of 6, not much additional compression is achieved, but the additional effort for compression results in a much larger overall time. Interestingly, a level of 3 also produces a good overall time. It is, however, quite easy to attribute this to the model (in this case, Lunar scenario 4(b)), as a larger model would eventually spend more time during disk look-ups due to the approximately 30% less compression achieved at level 3. The speedup obtained from the zLIB compression is illustrated in Tab. 5.1. HC stands for hash-compaction, a technique used by other disk-based model checkers [PITZ02, BJ05] where, instead of storing the entire state vector, only a hash value is stored [WL93]. While the risk of encountering a collision can be minimized pretty much, the *birthday paradox* problem needs to be considered [BK04]; but most importantly, the state information is lost.



(a) Lunar, scenario 4(b)



(b) Dining Philosophers, $n = 9$

Figure 5.10: Effectiveness of caches with different parameters. Line intersections represent actual measurements. Lower is better, as cache failures result in disk look-ups.

5.5. Benefits of Component-Based Software Engineering for CMC

Using component technology to write a high-performance model checker is a risky endeavor. In benchmarks, CMC performs at approximately 10% of the state-per-second rate of Spin. Approximately half of that slow-down can be attributed to employing the NipsVM virtual machine. The remainder can be expected to be a mixture of missing optimization – often done deliberately, as many optimizations would contradict the clean separation imposed by the components – and component overhead. The component model of CMC is fairly lightweight, making cross-component calls just as expensive as a normal C++ call; however, many optimization techniques, especially those employed by Spin, try to eliminate those calls as much as possible.

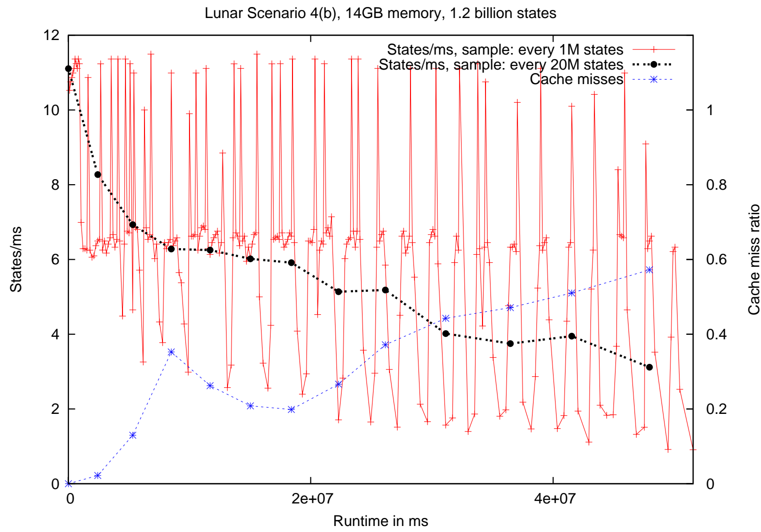


Figure 5.11: Smooth degradation illustrated by the states/millisecond and cache miss/cache success rate development

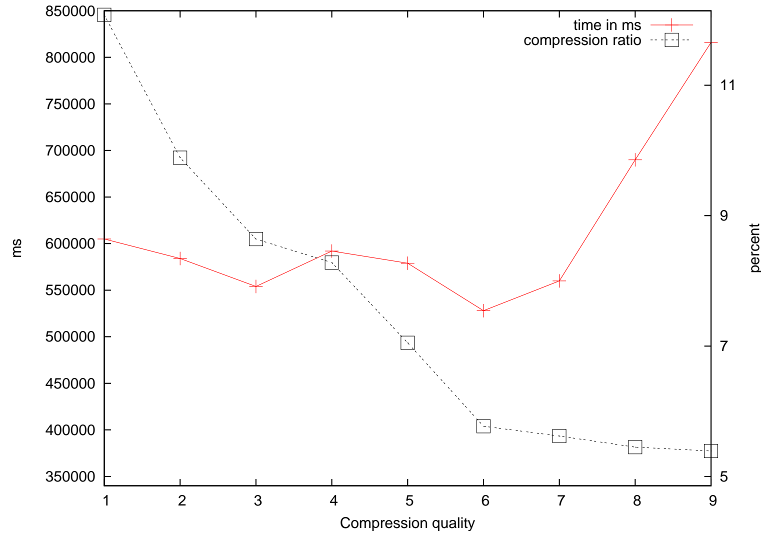


Figure 5.12: Speedup obtained by various ZLIB compression levels

Components provide a good approach towards experimenting with different algorithms and their combination. Adapting SPIN to Tarjan's algorithm [Tar72] in [Ham04] took a few days of hard programming, following approximately a month of code reading and preparation. For CMC, it would only require the definition of an appropriate component, and some changes to the assembly file. But we found other areas, some unexpected, where utilizing components proved helpful:

For example, when rewriting a component (in an attempt to optimize it), it was quite easy to keep the old version and actually write a new one. Not only was the old

Model	LHT	Random	Incoming	Outgoing	Age	Combined
9 Dining Phil.	2%	7,569,757	7,799,983	7,965,160	7,657,372	7,702,798
4,685,071 states	25%	3,282,324	4,913,862	4,366,858	3,251,002	5,143,714
12,234,622 trans.	80%	178,254	429,148	403,077	9,481	9,481
Lunar 4(b)	2%	225,960	233,010	259,499	228,485	194,332
3,335,917 states	25%	70,126	62,733	97,652	59,272	43,221
3,923,209 trans.	80%	8,124	4,861	13,249	64	4

Table 5.5: Comparison of different reclaiming strategies. “Random” removes arbitrary elements. “Incoming” removes the elements with least incoming edges first, whereas “Outgoing” removes elements with fewest outgoing edges. “Age” implements a FIFO strategy that removes the oldest elements. “Combined” describes a strategy where Incoming, Outgoing and Age are combined.

component kept in case the optimization attempt failed, but also the old component assembly could be reconstructed easily for comparison tests. Also, the use of components encouraged experimentation with regard to algorithm comparisons: It was fairly easy to take a few hash functions (e.g., Jenkin’s hash [**Jen97**], SHA1 [**EJ01**] as well as a recently developed hash function called MurmurHash [**App08**]), wrap them in a few lines of component code and plug them into the model checker to compare their performance.

Similarly, we wrote some components that implement different reclaiming heuristics – i.e., heuristics that rank the states in the lossy hash table and try to identify those that can be swapped to disk, because they are unlikely to be revisited. This follows the ideas of [**BLP03**]. In Fig. 5.13, the `ReclaimingStrategy` component provides such a strategy, in the depicted setup an `EasyReclaimingStrategy`, which just uses random choice. The effect of the various heuristics is shown in Tab. 5.5. Indicated are the number of revisited states that got swapped out of a lossy hash table with a size of 2%, 25% and 80% of the total state number. Evidently, no heuristic excels, but the experiment was easy to conduct.

Components also integrate well with the building of a parameter parsing framework: For an experimental model checker like cmc, a large number of parameters need to be made settable by the user. This can be facilitated by the use of components by having them declare the parameters they require. For parameters, the component annotations declare the name and default value of a parameter. The glue code generator gathers the parameters defined by the various components and generates a uniform parameter parsing. This greatly facilitates the provision of just the parameters that are used by the current configuration.

Similarly, logging output is provided by each component, such that at the end of the model checking run, all the relevant data can be output without having to implement a complete traversal of the component graph – and Fig. 5.13 shows that it is indeed a graph, not a tree and not even a DAG.

An extremely interesting application of components was encountered when debugging cmc, as a model checker is pretty hard to debug. Often, errors emerge only after a few million states, because some hash table overflow is handled incorrectly or some weird sequence of events leads to accessing a state vector that just got deleted due to its removal from some set (e.g., the candidate set). Common debuggers fail in such a scenario, and the implementer usually has to revert to code reading. For cmc, another approach was used a few times: Some readily available library was utilized to re-implement the behavior a component (or a set of components) is supposed to provide. For example, the C++ standard hash table

implementation was used to check the cache/disk lookup combination: As soon as a state is added to the open set that has already been visited, an error is found and can be signaled. Now, a trigger can be formulated for this erroneous state, and a common debugger can be used to trace its processing within the model checker, leading to the error. A similar approach has been employed to weed out memory leaks, a persistent plague of a model checker that maintains a large number of state sets.

Large CMC runs take hours, days and even weeks to complete. Quite often, we became interested in the current state of the model checker: Are the caches performing as expected? How much compression is provided by the ZLIB? How much memory will be consumed until the lossy hashing is fully operative? While many data are written to a log file, the number of values usable for observing the model checker's behavior is too large to have all such values logged at frequent intervals. A solution to this problem is provided by two methods every component needs to implement: `listNetworkCommands` and `processNetworkCommand`. The former needs to give a list of strings which the latter one can process. A user may now connect with a running CMC instance using TELNET and a special port. The user can obtain a list of components and the commands they understand. The commands can then be issued to the component, obtaining some number or value in response. Also, some steering functionality can be provided: For example, a "TRAP" command can be sent to the Search component to request that the next state that gets processed is stored, and can then be retrieved by issuing a "SHOW" command. Such a "software-metry" proved a valuable asset to understanding the model checker and its problems, without requiring numerous restarts.

We also were able verify one of the most important properties of component-based software engineering, namely that components encourage discipline with regard to a proper modularization, when including a different state space generator. It has often been proposed that using interfaces encourages discipline with respect to the level of abstraction that is used. Writing a state generator that does many other things as well leads to many short-cuts and internal assumptions. Packaging it into a component that can be linked with arbitrary other components requires a much deeper consideration of the necessary abstraction level. It proved to be quite easy to adapt the CMC model checker to a second virtual machine, μCRL [**GP95, BFG+01**], although this was not planned when the first version was written.

Are components the right paradigm to implement a model checker? Most likely not, if industrial-grade performance is required. But for experimentation, large-scale algorithm comparisons, and optimization research, components provide welcome support. Although we suffer a substantial slowdown with CMC compared to SPIN, a quick look into the source code of SPIN (or MURφ [**DDHY92**], for that matter) will convince anyone that it is far less troublesome to adapt CMC compared to these highly optimized model checkers.

5.6. Future Work

CMC barely scratched the surface of what is possible with disk-based model checking. With CMC being sort of a technology demonstrator, we did not pursue them, but we will discuss some of these ideas here. This is, for most points given here, a complete detour from the topics of this thesis, but these considerations should be listed somewhere.

5.6.1. Mass Storage Media Beyond Magnetic Disks. The reason why we use the magnetic disk in CMC is because it is so much cheaper than RAM memory.

Besides, disk size has less practical limitations than RAM. This is paid for by having to do the disk lookup in a linear fashion – as a true random access is too costly.

But since a few years, flash memory provides an interesting alternative. It is faster than a magnetic disk, and supports random access just as RAM does. It has not yet gained widespread use as a replacement of hard disks because the number of write operations permissible for a single memory location is rather limited. MP3 players and digital cameras have long adopted this new media, despite the ongoing miniaturization of hard disk drives. For our model checker, flash memory is an interesting alternative to magnetic disk, and obviously more suitable since random access is possible. Still, it can be expected that the cost of doing read access to flash memory is considerably slower than RAM-only look-ups, hence the caches are most likely still required. The number of write operations is limited to one per model checking run – unless the flash memory is used as yet another lossy hash table, acting as another cache for the still less expensive magnetic disk.

The utility of being able to do cheap random look-ups becomes immediately obvious if the search algorithm is to be used for depth-first search. The algorithm utilizing the caches relies on a broad search front; the order in which the states are visited is not important for simple reachabilities. Algorithms that find loops or strongly connected components [**SE05**] usually need to maintain an order in which states are visited, and hence require short-termed decisions on whether a state is visited. This would require very frequent disk look-ups, and void the feasibility of the algorithm (we have counted the states where all successor states were answered with "don't know" by both caches, and found that up to 5% of the states would require a disk lookup during an unmodified run – which, for the large runs, can be as much as 60 million states, and each disk traversal can take up to half an hour at the end of the search progression). Hence, non-random access and things like LTL model checking are not reconcilable. But if using flash memory allowed for cheaper random access, large-scale LTL models could also be attempted.

5.6.2. Classical Model Checker Improvements. There is a number of additional improvements that could be added to our model checker in the form of additional components – most notable, state compression, state collapse, partial order and symmetry reduction. State compression compresses each state using a normal data compression algorithm (similar to what we do with the zLIB, but not considering data chunks larger than a single state, thus achieving far worse compression rates [**Hol97**]). When doing state collapse, the state vector for a process is stored separately, and the actual state (which consists of the state vectors of all processes) is just a list of pointers to the state vector table [**Hol97**]. We have not added this to the current cmc model checker because of the unforeseeable memory requirements of the state vector and state store, but of course, it would be interesting to see how this can be achieved with our algorithm, and with the memory retainment techniques employed. Also, a comparison to the zLIB compression (whose efficiency still surprises) would be interesting.

Partial order reduction [**HP94**] and symmetry reduction [**DM07**] as well as slicing [**MT98**] are techniques that aim at removing large parts of the state space without changing the reachability of possible error states. Slicing removes all those variables that do not have a (direct or transitive) effect on those variables that are used in describing the safety or liveness property under check. Partial order reduction seeks to remove state space blowups due to (unobservable) interleaving of concurrent processes. Symmetry reduction tries to avoid symmetric state space parts due to isomorphic process instantiation sequences. We have not employed any of these techniques in the model checking runtime algorithm, although the NipsVM

provides some internal symmetry reduction and provides a path reduction tool that can be used for pre-processing the input file [**Ara06**].

Implementing these improvements is sometimes hard because further dependencies are required between components – e.g., the partial order reduction needs to know about the semantics of state transitions, which up to now have been hidden in the state producer component – i.e., no component other than the NipsVM wrapper considers the state to be anything different than a chunk of binary data. We are not quite sure whether such component-data dependencies (i.e., the set of involved components dictate the required information that needs to be added to the data objects) can be handled beyond the complexity imposed by cmc at the moment. Maybe, at some point in time, special provisions for defining the data objects sent between components based on the assembly's requirements become necessary to keep the component design tractable.

5.6.3. Counter-Examples. The ability to produce a counter-example is one of the arguments for the use of model checking. Depending on the search algorithm, a more or less concise way to the error state is displayed, which can be used for understanding how the error was produced (and, often equally important, whether the property violation is really an error of the system that was modeled). One of the nice properties of cmc is that, more or less, all the states are stored intact at the disk (usually, those are lost in disk-based model checkers that utilize hash compaction [**PITZ02, BJ05**]). But currently, cmc does not produce counter-examples, as states do not store where they got visited from. For depth-first searches, the counter-example can be taken right from the stack, but for our "known-first-search", a stack is no longer used.

Obviously, a counter-example can be reconstructed if each state stored a copy of its parent state. This state could be retrieved from the disk (or memory, if still present), and have its parent state found again, and so on till the initial state is reached. Of course, this is far from cheap, both memory-wise and in terms of runtime requirements to assemble the counter-example. It would be far more efficient to store just a pointer to the parent state; as long as the state is kept in memory, this is sufficient, and as soon as it gets stored to the disk, the disk page (and the entry number) can be referenced, requiring just the reading of a single page for each parent state. However, there is a problem with externalizing states: All the child states need to have their pointers updated, and those might even not be in memory anymore (since most heuristics do not strictly require that parent states are removed before their children). There might be a two-step schema involving state IDs and a lookup table, but then again, this table might grow beyond memory storage capabilities. Note that, however, utilizing the automated-atomic filter poses no problem, as referencing an indirect parent state with a single-successor state chain leading to the current state allows for re-generating that chain and obtaining all missing states.

If the problem of referencing the parent state could be solved, cmc might gain the truly unique property of generating counter-examples while using the disk for obtaining more memory. This property is based on cmc's way of retaining exact state information on the disk, which is the true novelty of the algorithm.

5.6.4. Parallelization. In Michael Jones' words, model checking with magnetic disk is about "getting a larger hash table"[1]. Another way of exceeding one physical machine's memory is to distribute the model checking algorithm [**Web06**]. The basic approach here is to calculate a hash value for each state, take it modulo the number of machines n and send it to the machine designated by the result. This

[1] from the talk presenting [**BJ05**]

results in a lot of network traffic, as most of the states need to be sent to other machines (actually, an expected fraction of $\frac{n-1}{n}$), but each machine only needs to store $\frac{1}{n}$ of the states.

This can be easily combined with our approach as most of the components can be reused. The open set is replaced by a "to-do queue" that is filled by the states sent over the network (and those that are found on the same machine). A state in this set is known to be reachable, but its being visited needs to be checked. Hence, during the search, a state is taken from the to-do queue and sent to the caches. Any state that is found to be unvisited (either directly by a success of the Bloom filter, or later after a disk lookup) has its successor states calculated right away; these successor states are then sent to their target machines where they are added to the aforementioned to-do queue. This "tilts" the algorithm a little, since the open set (where definitely unvisited states are stored) is replaced by the to-do queue (where the states might be visited already), and the successor calculation is placed at the end of the state handling process, not at the beginning. Otherwise, everything can be reused.

5.6.5. Configuration and Reconfiguration. CMC maintains a number of caches and sets, and operates a number of algorithms like the ZLIB which can be tuned by setting a number of parameters. For a standard CMC setup, 14 parameters are understood (plus the input file name), but a number of additional parameters might be found if this was called for. Additionally, the number of component setups is vast, since many components can be combined (e.g., the choice of the hash function, whether to use the automated-atomic filter, the ZLIB and hash value caching all are independent of each other on a functional level). It hence seems reasonable to use some automated reasoning to find good parameter settings. This has been done with considerable success at compile time for the ATLAS linear algebraic library [**WD98**] and for the SPEAR SAT solver [**HBHH07**]. SPEAR is automatically tuned against a set of standard examples, but not specifically modified afterwards; ATLAS is tuned at compile-time. In the context of this thesis, a configuration at assembly time (strictly speaking, this is at compile-time, since CMC is statically built for a component setup) or at runtime (by reconfiguration) would be very interesting. It should be admitted that not much effort was spent on this, for a number of reasons, which, along with some ideas on what might be possible, are presented in Sect. 8.5.

5.7. Conclusions

The experience obtained from writing CMC suggests that using components (or a similar structure) greatly facilitates the development of a complex software. For each idea presented here, a number of ideas got dismissed because they did not work. But for each idea tested, a component got written. If an idea was to be rejected, it was a matter of going back to an old assembly, while the component is retained. The following example illustrates the benefits:

We were wondering if Jenkin's hashing [**Jen97**] is inferior to a cryptographic hash function like SHA1 [**EJ01**]. So we wrote a component that provides SHA1, only to find out that Jenkin's hashing is equal in terms of collision avoidance, but much faster. So we reverted to an assembly which uses Jenkin's hashing, but kept the SHA1 component. Later, when we compared ZLIB compression to hash compaction (cf. Tab. 5.1), we could take this SHA1 component again (because it provided a 128 bit hash value), requiring only modest modification (in order to obtain the full 128 bits instead of the usual 32 bits, as hitherto mandated by the hash function interface).

Playing around with components in such a manner is quite inspiring, and the basic idea of using a double caching was indeed born from such a way of thinking about the application. It should not be claimed, however, that a component model like the one employed by the CMC model checker is without its caveats: Mostly because of shared data, it still requires expert knowledge to assemble components. Most problems emerged from the necessity to perform reference counting for the states; it requires thorough planning on how to avoid premature freeing of a state's memory, as well as introducing memory leaks due to not discarding unreferenced states.

We are confident that CMC's component model is suitable for the task it was designed for; but for more complex software, the framework needs to provide more help to restrict the problems introduced by shared memory. It hence became evident that, for providing a better separation of roles, shared memory should not be allowed.

5.7.1. Influence of the CMC Case Study on Reconfiguration Design. The reason why we include the CMC model checker in this thesis is its influence on the design of reconfiguration, which we will present in the next chapter. CMC has produced a special view on components, and even while CMC itself is unsuitable for reconfiguration (for reasons detailed in Sect. 8.5), this view led to an accentuation of certain aspects of reconfiguration.

The biggest influence regards to statefulness and state retainment requirements of the components. We have seen in Tab. 3.1 that relatively few reconfiguration-enabled component frameworks support state transferal, but stemming from the experience with CMC (where reconfiguration might be used to substitute a component that has exceeded its memory allowance), state transferal is a mandatory requirement for conducting meaningful reconfiguration (if the state is lost, a mere restart would amount to the same as runtime reconfiguration). Also, the prospect of reconfiguring a component under tight memory constraints ruled out the utilization of a reflection-based state transfer [Van07]. In order to save the contents of a hash table and transfer it to a new data structure, a careful treatment of the data is required in order to operate within the memory constraints. Also, data might be modified in very specific ways that require custom-written code (e.g., some states need to be swapped to disk during reconfiguration). This requires the indirect state transfer approach or a direct approach that uses reconfiguration-specific code (we will present a similar approach in Sect. 8.2.2). On the other hand, in the context of CMC, utilizing an indirect state transfer does not seem to be an insufferable violation of the separation of roles paradigm, since component reuse is not important and components are written in close collaboration.

Also, under the impression of the tight collaboration of components in CMC, the prospect of making a system adaptive by reconfiguration provisions after all the components have been completed appeared insurmountable. Instead, just as components helped to come up with a novel design of the model checker, we got more interested in using reconfiguration as a means to design a system than to add unanticipated functionality later. We will discuss the difference throughout the remainder of this thesis, most notably in Sect. 8.4. In taking this view, we spent less time on supporting elaborate reconfiguration planning mechanisms but chose to investigate the execution of reconfiguration.

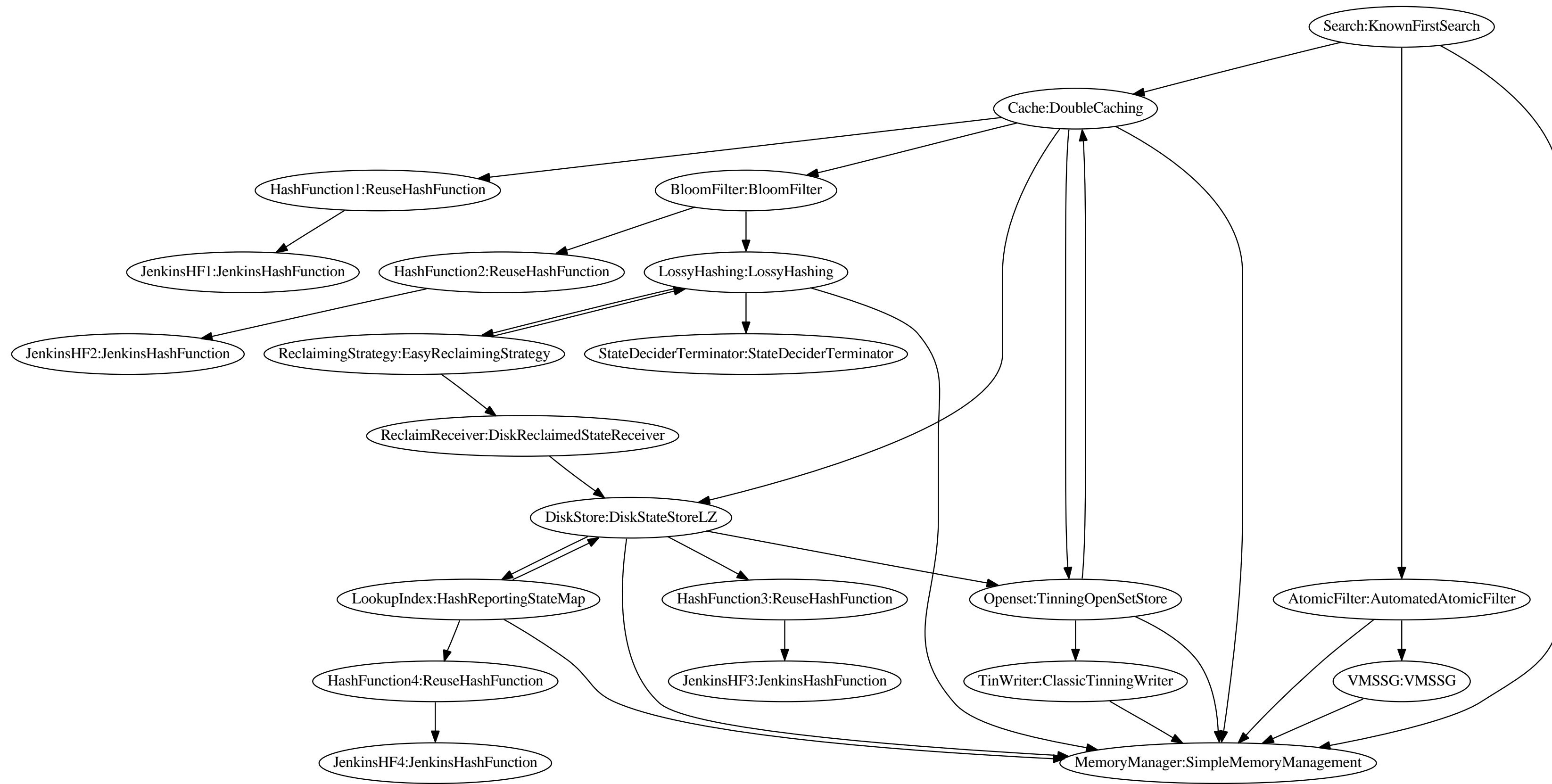


Figure 5.13: Components of the CMC model checker

CHAPTER 6

A Component Model for Reconfiguration: JComp

A designer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.
— Antoine de Saint-Exupery

The CMC component model and framework is *lightweight*, and does not restrict the host programming language at all. Components share memory as they see fit, and the communication is realized directly by the native method invocation means of the host language. This helps to maximize the efficiency of the model checker, but makes the component model quite unsuitable for reconfiguration: first, since parts of the component's state are shared (reflected by the σ part of the state in the component model), removing a component would require an intricate (or user-supplied) detection of the memory that can (and must) be freed, and second, due to the maintenance of a stack, replacing a component might result in the necessity to operate both the old version (for processing of pending stack elements) and the new version for some time.

It hence became interesting to investigate a *heavyweight* component model that imposes restrictions on the component's code by disallowing shared memory and call stacks. Components are forbidden to communicate by any other means than those provided by the framework, making the communication completely observable. Such a rigid framework is required for reconfiguration; if the framework lacks control on its components, it can hardly prevent errors emerging from a reconfiguration done in the wrong moment.

All this is paid for by a severe loss in efficiency. Enforcing the absence of shared memory contradicts the provisions of most host languages, but it is required to avoid "covert channels" that components might use to circumvent the framework-provided communication means. But efficiency is of little concern here: instead, we strive to define a component model that is as simple as possible to ease the reasoning about reconfiguration. In this chapter we present the JComp component model. We first investigate the requirements for a component model capable of supporting reconfiguration, explain the design principles we applied, and introduce a formal component model. By extending this component model, we introduce a formally described way to conduct reconfiguration, and show some of its properties. The implementation of the JComp component model will be described in the next chapter.

6.1. Requirements

A component model is usually devised to support a specific class of applications. The specific properties of the component model – how communication is to be conducted, how components are to be represented in memory – then follow the requirements imposed by that class of applications. For example, if multimedia is to be supported, special media streaming connectors need to be provided [SvdZH08]. Reconfiguration is then included in a way that supports the retainment of these

properties over the reconfiguration process: if the media connectors need to provide some latency control, reconfiguration needs to be designed in a way such that the latency bounds are not violated during reconfiguration [**MNCK99**].

Reconfiguration is, in principle, a vague term. It can be used to describe the restarts that SAT-solvers do [**BLMS01**] or intricate morphing of the application structure towards a goal. Depending on the component model and the required properties, reconfiguration may yield different problems. For example, multimedia reconfiguration needs to preserve real-time constraints, which is why the DJINN framework interleaves reconfiguration as much as possible [**MNCK98, MNCK99**]. This requires a delicate data-flow, as the reconfiguration is entirely non-atomic and parts of the data might be processed by the new component while data still processed by the old component is found in the system. For providing this, DJINN only considers the switching of component chains, i.e., the rerouting of data flow. Quite differently, hot code update does not need strong realtime properties. Instead, the preservation of the components' state is much more important. For example, DRACO [**Van07**] puts much emphasis on how a mapping between the old and new components' state can be derived. But for DRACO, reconfiguration is limited to single components only, and thus does not impose much of a structural problem.

In this thesis, our interest is somewhere in between. In considering reconfiguration as a valuable tool for realizing adaptivity, we try to allow as much as possible as a reconfiguration. However, in the absence of a application scenario requiring their consideration, real-time aspects are ignored, and performance as an abstract idea is only considered in that the reconfiguration tries to be *minimal*: Instead of blocking the entire system during reconfiguration, only the relevant part is blocked, with the remainder being allowed to continue its operation.

In this section, we will formulate requirements we consider important for a component model capable of realizing this concept of reconfiguration. We have discussed situations where reconfiguration can be applied on page 21 in Sect. 2.4.5.2, and provided a taxonomy of features in Sect. 3.2. Also, we have discussed some aspects of reconfiguration in Sect. 2.5.1. Finally, we have discussed the influence of the CMC case study on our reconfiguration approach in 5.7.1.

First, in order to conduct reconfiguration, the component model must yield access to the required properties of the components. Being interested in stateful components and component setups as described in Sect. 4.3, we formulate the following requirements on a component model capable of reconfiguration:

R1.1 *The component model must make it observable if a component can be reconfigured at a given point in time.* We do not want to impose that reconfiguration can be conducted at any point in time, as components might be required to finish some tasks first, but it is necessary to determine whether reconfiguration can be conducted or whether further waiting is necessary.

R1.2 *The components' connections must be readable and modifiable.* During reconfiguration, the connections of existing components are modified, and the component model must guarantee that this is indeed possible.

R1.3 *Components can be created and removed.* Reconfiguration might introduce new components, and discard old components no longer required. The component model must support these addition and removal of components.

R1.4 *The component state must be readable and writable.* Being interested in stateful components, we demand that the component model supports access to the relevant state data. Also, this data must be sufficiently enriched with information such that the reconfiguration algorithm can handle it as

required by the user. For example, if the user wants to retain some messages sent by component A to component B over role r, then messages need to contain information about their source component and role, in order to judge whether the user's request applies to a given message.

These requirements need to be fulfilled by a component model that supports stateful reconfiguration. Most component models will either support these requirements already, or be easily modifiable (e.g., requirement R1.4 will most likely lead to some changes in the component's state, if this is discussed within the component model). However, our interest goes further, and ultimately we want to prove properties of our reconfiguration approach that lead to guarantees given for the actual implementation. In order to transgress from the model to the implementation, we impose two requirements on the reconfiguration algorithm:

R2.1 *Reconfiguration must be conducted by a sequence of conceivably atomic steps.* Instead of just describing the overall effect of the reconfiguration plan, the reconfiguration algorithm needs to be described in a way that can be directly implemented in a regular programming language. As such, we require the reconfiguration algorithm to be implemented by a sequence of fine-grained steps.

R2.2 *Reconfiguration must follow a plan with a precisely defined semantics.* In order to reason about the effects of reconfiguration, the reconfiguration input must be constrained. We hence require that the reconfiguration algorithm operates on a *plan*, which fully describes the reconfiguration (opposed to just providing a number of primitives for reconfiguration, and pushing the responsibility to use them in a consistent way onto the user).

This results in a *small-step semantics* for the reconfiguration plan. The approach of using term rewriting to define the semantics of the component model integrates well with such a requirement, and we are unaware of a component model that has a sufficiently fine-grained semantics to support requirement R2.1.

The implementability of the reconfiguration algorithm is used for obtaining guarantees in the framework implementation that can be proven on the component model. Based on the literature presented in Chapter 3, as well as the experience obtained with CMC and the theoretical considerations of Chapter 2, we require the following guarantees, which need to be ensured by the reconfiguration algorithm:

R3.1 *The results of a reconfiguration must be predictable.* That is, if a given reconfiguration plan is conducted at a known state of the component setup, the outcome should be as deterministic as possible. This requirement can be refined into two sub-requirements:

R3.1.1 *The reconfiguration process should appear as atomic.* Even if the reconfiguration algorithm is conducted by a sequence of fine-grained steps (cf. requirement R2.1), none of the components involved should ever notice that a reconfiguration is underway. This applies to communication especially: no communication must be disrupted or lost due to an ongoing reconfiguration.

R3.1.2 *The reconfiguration should interrupt components only at predictable points.* Since we are interested in state retainment, we need to have some guarantees about the point in time where reconfiguration interrupts ongoing computations of the component. If reconfiguration might interrupt a component at any given moment, little assumptions could be made about the data state at the point in time where reconfiguration commences.

Both these requirements provide a guarantee to both the component implementer, who can remain mostly unaware of reconfiguration scenarios, and the system designer, who can rely on reconfiguration not interfering with ongoing computation.

R3.2 *Reconfiguration should be minimally invasive.* The reconfiguration approach offered by the component model should guarantee that only those components that are strictly required to be modified or temporarily halted during configuration are affected. For large-scale, distributed systems, reconfiguration cannot be allowed to block the entire system to achieve atomicity (for satisfying requirement R3.1.1). Instead, the system should be kept alive as much as possible, allowing components to continue their operation, even if this entails communicating with components under reconfiguration.

R3.3 *A component's implementation must be allowed to stay ignorant of reconfiguration.* As with the requirements of R3.1, we want to have the component model ensure the component implementer that reconfiguration can be conducted with components that have not been especially prepared for being reconfigured.

We would like to require two more guarantees, but they are difficult to formulate, as they depend, to some extent, on the component setup at hand. We hence formulate them as "wishes", and require that the component model does not impede them, should the component setup itself be compatible:

R4.1 *Reconfiguration should be conducted as soon as possible.* Obviously, we required that reconfiguration waits for predictable points to be reached within a component's execution (requirement R3.1.2), and we required these points to be observable (R1.1). But other than that, if reconfiguration is requested, the model should not delay the reconfiguration. But components might behave in a way that inhibits reconfiguration forever.

R4.2 *The reconfiguration algorithm should guarantee that a reconfiguration plan is realized correctly.* A reconfiguration is planned to affect some change on the system, and the state of the system after the reconfiguration is envisioned by the reconfiguration designer. The reconfiguration algorithm should strive to meet these assumptions about the state after the reconfiguration; especially the system should not experience unexpected deadlocks during reconfiguration.

In combination with requirement R2.1, the guarantees we require from our component model are rarely discussed. Some of the guarantees are quite commonplace in the literature, e.g., the necessity to conduct reconfiguration in a way that makes it appear as atomic (cf. the third column of Tab. 3.1). But these guarantees are usually not proven on a small-step granularity, which we wish to do in this thesis. For conducting such formal proofs in the presence of requirement R2.1, the component model has to be given a rigid semantics, even without considering reconfiguration. Therefore, we require a model which fixes communication as much as possible. Closest to realizing the requirements while discussing reconfiguration implementation on a comparable granularity is NeCoMan [**JDMV04**], but it focuses on a very distinct application area (cf. Sect. 3.2.27.5).

In this thesis, we address a general component model that is to support a variety of applications. We do not formulate specific requirements here, but require the component model to be suitable for implementing applications that process data, while not requiring frequent user interaction. Contrary to the CMC model, we do not impose efficiency requirements, however. The supported reconfiguration,

however, should allow for arbitrary changes to the component graph, and not be limited to specific patterns.

6.2. Design Principles

JComp is a component model that is dedicated to the research of reconfiguration. It strives to implement the requirements stated above as concisely as possible, omitting many features found in other component models. This helps to satisfy requirement R3.1, as additional features (e.g., customized communication means) have the tendency to make the result of a reconfiguration less easy to predict. Some requirements (R1.2, R1.3, R3.2) suggest a component model where components are inter-tangled as little as possible, both in means of shared memory and communication. These requirements are incompatible with the cmc model checker's component model, which uses call stacks and shared memory (cf. the discussion in Sect. 8.5). Instead, we found these requirements similar to those for a component model capable of arbitrary distribution (e.g., any component setup must be distributable on a number of machines with an arbitrary assignment, without changing the semantics of the system). In imposing the requirements for doing cross-machine communication on all components, two basic design principles got defined:

(1) JComp uses message passing as the sole means of communication between components,
(2) applications devised with the JComp component model should be distributable without any changes to the components' code.

Interestingly, the resulting JComp framework never really excelled in having applications distributed (mostly because we lacked a proper example scenario) and was eventually even modified to only support distribution on the user level, as discussed in Sect. 7.3. Nevertheless, the implications of the two aforementioned design criteria resulted in JComp becoming rather strict with respect to the design of the components, which in turn enforced the clean separation of components required for clean and predictable reconfiguration.

Being able to distribute a component-based application (i.e., transparently instantiate components on different machines) requires the components to be incapable of using the fact that they operate on the same machine (which minimizes their entanglement). First of all, this means that no shared memory must be used. If two components access shared memory, they cannot be put on different machines (without changing the semantics or without tedious synchronization of the machines' memories). Hence, the first derived design principle for JComp reads:

⇨ Components must not share memory.

Of course, this also applies to other shared resources like the file system, but we will only discuss memory here. If no shared memory is allowed, the only way one component might influence another component is by explicit communication. This is appreciable, as it enforces one of the basic properties of component-based systems, namely observability of communication (cf. Sect. 2.1), which we will eventually use to satisfy both requirements R2.1 and R3.1.1 as well as guarantee requirement R3.1.2.

The first design principle – message-based communication – requires that a caller component runs in a different thread as the receiver. Actually, it does not technically require that (there may be a non-preemptive scheduling that runs one component at a time), but on the conceptual level, this situation is mandatory:

⇨ Components run concurrently.

Concurrency has many well-known dangers, like race conditions, which pose a threat to requirements like R3.1.1. But since all communication is done explicitly between components, racing can only occur for message order. Usually, this can be dealt with. This absence of shared memory greatly facilitates the concurrency of components, because an important invariant is provided: During the execution of a method, none of the accessible data is modified concurrently. This amounts to *local non-concurrency* or *mono-threaded* components. The component implementer is assured that, for the duration of a message processing, no component-local data (which is all the data the component has access to) is modified concurrently. Requirement R3.1.2 (when choosing methods as the basic unit of atomic processing) extends this guarantee to reconfiguration: if a component was able to complete a method's execution in a non-concurrent setup, neither concurrent components nor reconfiguration can interfere and keep it from terminating the method's processing.

Message passing works by delivering messages to components, where they become queued. Sometimes, we might desire a method result, or wait for some method to complete, and thus require *synchronous* messages, i.e., messages that block the calling thread until the message has been processed. Only for synchronous messages, a result value can be provided, although the concept of *futures* provides sort of a compromise [**BH77, CH05, CHM08**]. So we decided to include synchronous messages. There is, however, a problem involved: The problem of *synchronous callbacks*. Component A sends a synchronous message to component B, and waits for its processing to end. During the processing of this message, B decides it needs further information from A and sends a message to A, waiting for this message to be processed. Now, the system deadlocks, as A is currently waiting for B to finish processing of the initial message and thus incapable of processing the message just received.

There are two straightforward solutions to this problem: The first one is to just accept that deadlock as an error and disallow synchronous callbacks. The second one is to provide means to process that callback, i.e., by forking a thread. For JComp, we decided to take the first approach, even if this seems too rigorous: The second approach invalidates the invariant of not having the memory changed during the course of a method, an invariant that is otherwise provided by the components being mono-threaded. Obviously, this is not as troublesome as genuine concurrent modification, as any such modification will happen while waiting for a synchronous message to return, but nevertheless the design decision is:

⇨ Every component is mono-threaded.

Hence we desire a component model that supports components that do not share memory and run in their own thread each, while being mono-threaded internally.

The practical implementations of these design decisions vary from straightforward (e.g., the realization of asynchronous message passing) to downright impossible (e.g., the prohibition of the use of shared resources, a generalization of the requirement to forbid sharing of memory). We will discuss this in Chapter 7. It is, however, very important that the component model presented here is almost directly implementable, such that the properties we prove (and which support the guarantee requirements like R3.1, R3.2 and R3.3) carry over to the resulting component framework. Adopting such an approach is the major contribution of this thesis, and the main purpose of the JComp component model, which makes it stand out from comparable component models.

$$c_1^{\mathrm{r}}, \gamma_1, \mathrm{call}(r,m,v).P \parallel c_2, \pi_2 \to c_1^{\mathrm{b}}, \gamma_1, P \parallel c_2, \pi_2 :: c_1.r.m(v) \quad \text{if } c_2 = \gamma_1(r) \text{ and } \mathrm{sync}(m) \qquad (\textsc{call})$$

$$c_1^{\mathrm{r}}, \gamma_1, \mathrm{send}(r,m,v).P \parallel c_2, \pi_2 \to c_1^{\mathrm{r}}, \gamma_1, P \parallel c_2, \pi_2 :: c_1.r.m(v) \quad \text{if } c_2 = \gamma_1(r) \text{ and } \neg\mathrm{sync}(m) \qquad (\textsc{send})$$

$$c_1^{\mathrm{r}}, \langle c_2.r.m(v')\rangle, \mathrm{return}(v).P \parallel c_2^{\mathrm{b}}, \langle\sigma\rangle \to c_1^{\mathrm{r}}, \langle\bot\rangle, P \parallel c_2^{\mathrm{r}}, \langle ret(\sigma, v)\rangle \qquad (\textsc{ret})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{set}(\sigma').P \to c^{\mathrm{r}}, \langle upd(\sigma,\sigma')\rangle, P \qquad (\textsc{set})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{choose}((\Sigma_j.P_j)_{j\in J}) \to c^{\mathrm{r}}, \langle\sigma\rangle, P_j \quad \text{if } j \in J \text{ and } \sigma \in \Sigma_j \qquad (\textsc{choose})$$

$$c^{\mathrm{r}}, \langle\sigma\rangle, \mathrm{success}, c'.r.m(v) :: \pi \to c^{\mathrm{r}}, \langle c'.r.m(v), prm(\sigma,m,v)\rangle, \mu(c)(m), \pi \qquad (\textsc{deq})$$

$$c^{\mathrm{r}}, \mu\mathrm{X} \Rightarrow \mathrm{P} \to c^{\mathrm{r}}, P[X/\mu X \Rightarrow P] \qquad (\textsc{loop})$$

Table 6.1: Configuration transition rules

6.3. The Component Model

We will now formally describe the JComp component model, based on the definition of Sect. 4.5. We start by defining what a component configuration should be comprised of. Just as the CMC component configurations have to maintain a stack for callbacks, we need to maintain a queue for incoming messages in the component state of JComp components in order to realize communication by message passing.

Definition 6.1 (Component configuration). *A component configuration $\tilde{c}$ of a single component c is of the form*

$$(id(c), f(\tilde{c}), \gamma(\tilde{c}), e(\tilde{c}), \sigma(\tilde{c}), P(\tilde{c}), \pi(\tilde{c}))$$

where f indicates whether the component is running *($f = \mathrm{r}$) or* blocked *($f = \mathrm{b}$); $\gamma : \mathscr{R} \rightharpoonup \mathscr{C}$ is a wiring for the roles of c, telling which role points to which component; e either contains a method call of the form $c'.r'.m(v)$ with $c' \in \mathscr{C}$, $r' \in \mathscr{R}$, $m \in \mathscr{M}$ and $v \in \mathscr{V}$, which c currently executes, or is empty ($\bot$); $\sigma \in \mathscr{S}$ is a component state; $P \in \mathscr{P}$ is a component process term; and $\pi \in (\mathscr{C} \times \mathscr{R} \times \mathscr{M} \times \mathscr{V})^*$ represents the message queue, which is just a sequence of messages. For ease of reading, we write c^f for $id(c), f$, and we group e and σ to form $\langle e, \sigma\rangle$.*

As before, a configuration is a mapping of a set of components to their states, defined for the components instantiated in the current configuration. We denote the set of all component configurations by $\tilde{\mathcal{C}}$.

Definition 6.2 (Configuration). *A* configuration *is a partial function $\mathcal{C} \rightharpoonup \tilde{\mathcal{C}}$ from components to component configurations, for such a configuration $\{c_1 \mapsto \tilde{c}_1, \ldots, c_n \mapsto \tilde{c}_n\}$ we write $\tilde{c}_1 \parallel \cdots \parallel \tilde{c}_n$. $\mathbb{S}$ is the set of all such configurations.*

For processing methods, we need to make the distinction of asynchronous and synchronous methods. This is required to determine how a component should react if its current component process term calls for a message sending to another component. The set $\mathscr{M}$ is hence split into two sets $\mathscr{M}_{sync} \cup \mathscr{M}_{async} = \mathscr{M}$, which are disjoint ($\mathscr{M}_{sync} \cap \mathscr{M}_{async} = \varnothing$). We define the predicate $\mathrm{sync}(m) \Leftrightarrow m \in \mathscr{M}_{sync}$.

The configuration transition rules of our component framework are listed in Tab. 6.1. CALL and SEND treat synchronous and asynchronous invocations differently by blocking respectively continuing the caller. By DEQ a component can start

processing of a new message from its queue. Note that DEQ applies only to running components and therefore prohibits circular synchronous calls: If processing a synchronous call from component c to component c' requires a synchronous call from c' to c, the two components deadlock, as discussed before.

6.3.1. Communication Traces. Often, we are interested in the communication of components. We might thus become interested in runs that are restricted to the communication rules. However, the rules SEND, CALL and DEQ have both the source and the target component as parameter. This is more information than we usually desire: For the sequence of message receptions via DEQ, we do not want to know their origin, and for the sequence of outgoing messages, we do not want to know the (exact) target. The separation of roles paradigm dictates that the exact source of a message should not matter to the receiving component, and thus we should not have to care about it. Things get worse in the context of reconfiguration, where messages can be relocated, and role targets can change.

We hence use traces to abstract from the exact communication. We can either investigate message sending or message dequeueing, these being the two points in time where communication can be made visible to the outside (cf. the communication monitors for JCOMP, described in Sect. 7.2.5.1). For reasons explained later (in Sect. 9.1), we will concentrate on the message dequeueing here.

We thus fix a function (with co-domain $\{\textsc{deq}(c, m, v) \mid c \in \mathcal{C}, m \in \mathscr{M}, v \in \mathcal{V}\}$):

$$\alpha^{\text{DEQ}}(l) = \begin{cases} \textsc{deq}(c', m', v'), & \text{if } l \in \textsc{deq}(\overline{c : c'}, \overline{m : m'}, \overline{v : v'}), \\ \tau, & \text{otherwise.} \end{cases}$$

We denote the α^{DEQ}-trace abstraction of a run r by $\mathit{Traces}^{comm}(r)$. We extend this to sets in the straightforward manner: $\mathit{Traces}^{comm}(R) = \{\mathit{Traces}^{comm}(r) \mid r \in R\}$, and similarly to LTS: $\mathit{Traces}^{comm}(\mathscr{T}) = \mathit{Traces}^{comm}(\mathit{Runs}(\mathscr{T}))$. We call such traces *communication traces.*

If a component is deterministic (i.e., all the component process terms in the range of this component's method evaluator are deterministic), an initial state and a communication trace are sufficient to calculate the component's state at the reduced run's end (should it be finite); while this run cannot be uniquely calculated back (due to interleaving between the components running in parallel), the necessary information can be derived. Accordingly, we say that communication *determines* the behavior of components; basically, this emphasizes the fact that components are not acting spontaneously, but are merely responsive and only become active when requested to process a message – an important distinction to the concept of agents, as we discussed in Sect. 2.2.3.

6.4. The Approach to Reconfiguration

As we have seen in Chapter 3, there are multiple aspects of reconfiguration: How it is triggered, how it is planned and how it is executed. The whole picture is presented in Sect. 8.1. In this chapter, we will focus on the execution of reconfiguration. In accordance with requirement R2.2, we will assume that someone created a *plan* that describes the reconfiguration. The structure of such a plan is described in Sect. 6.5. We then discuss how such a plan can be applied in Sect. 6.7, building on an extension of the basic JCOMP component model as presented in the last section. We then investigate how this process can be relaxed (i.e., how it may interleave with an ongoing execution of the unaffected components, satisfying requirement R3.2) in Sect. 6.7.2. We also show how the JCOMP model satisfies the other guarantees.

We describe reconfiguration in the JComp component model by adding a number of rules [**HK08**]. This is, in principle, quite similar to the approach taken with CHAM in [**Wer98**], but as we also implemented the approach in JComp, our rule granularity is considerably finer, as desired by requirement R2.1.

As mentioned before, the planning is assumed to be finished at the point in time where the considerations of this chapter take place. Nevertheless, the tasks of planning also affect our approach towards reconfiguration: The more we assume the planner to be intelligent, the less precautions we need to take when actually conducting the reconfiguration. This especially holds true for the point in time when the reconfiguration is started (i.e., the *scheduling* of the reconfiguration). Most of the theoretical research focusing on reconfiguration revolves around this problem: Finding (or even actively effecting) a situation where communication has reached a state in which it is not interrupted by the reconfiguration [**KM90, MGK96, AP05, VEBD07**]. In this thesis, this problem is not deemed too important; but we will discuss it briefly in Sect. 8.1.4.2 and Sect. 9.6.1. Our view on components and their reconfiguration is more fine-grained: Instead of waiting for a suitable point in time to do reconfiguration, we try to do the reconfiguration as soon as possible (attempting to satisfy requirement R4.1, but respecting requirement R3.1.2) and have the substituting components take the place of the old ones immediately, possibly picking up transactions right in the middle. Obviously, such a reconfiguration approach is demanding in its own right; and there are situations where this cannot be achieved (e.g., synchronous calls interfering with component removal can deadlock the system if their target is removed). Nevertheless, the approach of reconfiguring in the middle of a (user-defined) transaction has yet to fail us in practice; effectively, it replaces the consideration of transactions by a more elaborate state-transferal, which needs to account for the transaction to become picked up by the new component (we compare the two approaches in Sect. 9.6.2).

State transferal sparks two problems: First, only in very limited situations we can assume that the state of an old component can be mapped directly to a new component. In the field of migration (i.e., the relocation of a software entity to another machine), this is possible, since an exact duplicate of the old component is built [**IKKW01, LS05**]. For more general component substitutions, a mapping needs to be provided, which can become arbitrarily complex. In Sect. 3.2.31, we have discussed various approaches found in the literature. Here, we will not impose a certain solution, but rather allow for both indirect and direct state transfer (as the choice of the suitable approach heavily depends on the kind of application; we will discuss this in Sect. 6.8).

The second problem is that of what actually constitutes the state. Obviously, the data stored in a component is part of it. Also, pending messages need to be included (we will later see that this can become quite problematic for arbitrary reconfiguration plans). We then decided that this is sufficient; for satisfying requirement R3.1.2, we decided that reconfiguration can only commence if no method is executed in a component that is about to become stopped. Hence, the process term is not relevant and does not have to be retained. For components, this seems like a sensible thing to do, and we only know of CASA [**MG04**] trying to do something different by allowing reconfiguration of components in the middle of the execution of a method. The CASA solution (requiring an explicit declaration of "safe-points" within the method's implementation) violates our requirement R3.3, and even if we do not respect this requirement too much, it is still straightforward to emulate safe-points by splitting methods, so the difference to our approach is not that big (cf. the discussion about message postponing in Sect. 7.5.1). But our

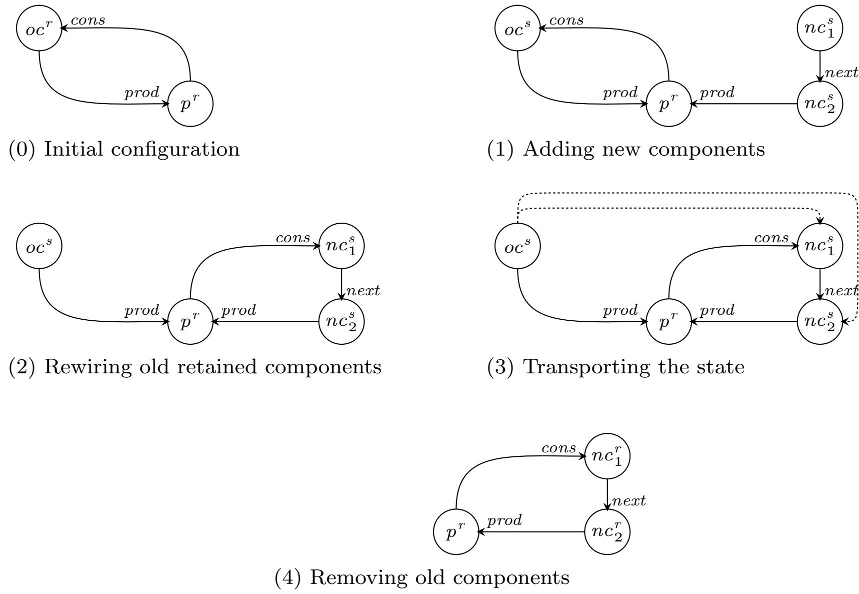


Figure 6.1: Steps of reconfiguration

approach is driven by the need to actually make reconfiguration *invisible* to an observer "distant enough", as mandated by requirement R3.1.1 and R4.2, so a lot of emphasis will be placed on asserting that no message is lost or swapped with messages received before.

6.4.1. The Abstract Idea. Fig. 6.1 depicts the basic idea of the reconfiguration process. It is quite similar to (yet has been developed independently from) the algorithm employed in the NeCoMan/JBoss framework [**JTSJ07**], but extends it to arbitrary components and includes state transfer.

The example of Fig. 6.1 is about a producer-consumer setup, where the consumer *oc* becomes replaced by two consumers nc_1 and nc_2 that are arranged in a chain. Basically, the process ensures that producer component p can be kept running the whole time, and that it may also issue messages to its role *cons* at any time. In (1), the old component *oc* is stopped (indicated by the s superscript); this ensures that its data state is no longer changed, and also that it does not send any messages (although this property is not of importance in this example). Also, the new components are added – in this example, a single consumer component *oc* is replaced by two new consuming components nc_1 and nc_2. New components are added with their wirings already provided, but in a blocked state such that they do not yet start processing messages. In doing so, requirement R2.1 is preserved even if we cannot assume that a component can be added in an atomic step – since it remains inactive and unconnected, executing the rule application by a number of statements will not interfere with concurrent execution, and thus appear as an atomic execution.

In (2), the retained component p is rewired by reconnecting its *cons* role to nc_1. This step is required to be atomic, and it is conceivably easy to implement it in an atomic fashion. Messages sent by p after this point will arrive at the message queue of nc_1, but since nc_1 is blocked, they will not be processed yet. This is important for the next step.

In (3), the state of *oc* is transported to the new components. The data state is copied to both components, depending on their use of the data. The messages are copied to nc_1 and are prepended to the message queue. In doing so, the order of those messages that were sent by p over its role *cons* is preserved, and the point in time when step (2) was conducted becomes irrelevant to the message order. This ensures that p needs not be stopped, which guarantees the minimal invasiveness mandated by requirement R3.2. At the same time, the copying of messages makes sure that the reconfiguration appears as an atomic step, satisfying requirement R3.1.1. Finally, the old component *oc* is removed in step (4).

6.4.2. Peculiarities. There is a "miss-by-one" problem lurking in step (3): Depending on the cause of reconfiguration, the last message that *oc* processed also needs to be retained. For example, this message m_e might have been a task that *oc* was asked to do; and while doing the task, *oc* suddenly fails by arriving at a fail component process term. As we have seen in Sect. 6.3, no rule can process a fail term, so reconfiguration is triggered to substitute *oc*. Now, any messages sent by p are retained in step (3), but the message m_e that caused *oc* to fail never gets processed completely. For p, being an observer "distant enough" (i.e., it is not blocked by the reconfiguration, and since it cannot determine the actual target of its role *cons*, it cannot detect the reconfiguration if everything is consistent), the message m_e then appears to be lost, which makes the reconfiguration perceivable. Thus, the culprit message m_e also needs to be enqueued in nc_1.

Another specific feature of our approach is that we generally do not allow the reconfiguration of connections alone. If a connection that connects the role r of component c to component c' needs to point to c'', c' has to be deleted. This is generally perceived as very limiting. For example, the reconfiguration depicted in Fig. 6.1 might also realize the addition of nc_1 to a ring structure that consists of p and *oc*, with nc_2 becoming an identical copy of *oc*. In this interpretation and from a technical point of view, only the creation of nc_1 is required, and the rewiring of p's role *cons* to nc_1.

This problem with doing a mere rewiring is quite technical, and we will discuss it further in Sect. 6.7.5. Informally, the problem is given by the fact that we only block components that are about to be removed, and if we change just the rewiring, no component is stopped at all. This leads to a problem with the messages sent over the connection that gets rewired. If those are not yet processed, it is unclear whether they should remain in the target component, or taken away and put to the connection's new target; and if so, then interleaving can produce strange results if some messages are processed while others are copied and further more become sent by the source component during reconfiguration. Hence, we opted for just requiring the removal of components that are the target (or source – in the example of Fig. 6.1, both nc_1 and nc_2 can be interpreted as the conceptually added component) of a connection that needs to be rewired. Of course, it is straightforward to optimize such a "removal/addition" such that the state does not have to be genuinely copied in the component framework, so this solution is not as expensive as it may appear at first.

6.5. Reconfiguration Plans

A reconfiguration of a configuration $\tilde{C}$ with $C = \mathrm{dom}(\tilde{C})$ is described by a *reconfiguration plan* as requirement R2.2 mandates:

Definition 6.3 (Reconfiguration plan). *Let* $\tilde{C}$ *be a configuration with* $C = \mathrm{dom}(\tilde{C})$*. A* reconfiguration plan *for* $\tilde{C}$ *is a tuple*

$$\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$$

with $A \subset \mathcal{C}$ *such that* $A \cap C = \varnothing$, $R \subseteq C$, $\alpha : A \to (\mathscr{R} \to (C \setminus R) \cup A)$, $\rho : C \setminus R \times \mathscr{R} \rightharpoonup ((C \setminus R) \cup A$,$\delta : R \times (C \times \mathscr{R}) \rightharpoonup A$ *and* $\varsigma : (R \to \mathcal{S}) \times A \to \mathcal{S}$.

A is a set of components that are to be added; $R \subseteq C$ the set of components to be removed; and α, ρ, δ and ς functions that describe how components are connected (α for components in A, ρ for components in $C \setminus R$) and how the state is preserved (δ handling the messages and ς the data state).

Hence, $\alpha : A \to (\mathscr{R} \to (C \setminus R) \cup A)$ describes the connections of the new components in A, which may be connected to both other new components and components that already exist, but do not get removed. We require, for all $c \in A$, $\alpha(c)$ to be well-connected for c.

The partial function $\rho : C \setminus R \times \mathscr{R} \rightharpoonup ((C \setminus R) \cup A$ describes the rewiring of connections of components that are not removed. It needs to cover those connections that point to components that are about to become removed. Thus, ρ needs to be defined for $(c, r) \in C \setminus R \times \mathscr{R}$ if $\gamma(\tilde{c})(r) \in R$ for $\tilde{c} = \tilde{C}(c)$. We require that $\gamma(\tilde{c})[r \mapsto \rho(c, r)]$, i.e., the connections of c with r pointing to $\rho(c, r)$, is well-connected for c for all r for which $\rho(c, r)$ is defined. Of course, if $\gamma(\tilde{c})$ is completely connected, then so is $\gamma(\tilde{c})[r \mapsto \rho(c, r)]$.

δ and ς are two functions that are used for defining how the state of an old component should be preserved. The partial function $\delta : R \times (C \times \mathscr{R}) \rightharpoonup A$ describes message retainment, i.e., the components that should process previously unprocessed messages of components in R. It is required that messages are moved to components that actually implement the required interface, that is $I_R(c)(r) \in I_P(\delta(c', (c, r)))$ for all $(c', (c, r)) \in \mathrm{dom}(\delta)$. The function $\varsigma : (R \to \mathscr{S}) \times A \to \mathscr{S}$ describes the state of the new components, which is calculated from the state of the old components. This is a very general notion which subsumes more concrete, technical approaches of data state retainment [**Van07**, **RS07a**, **IFMW08**].

Example 6.1. *For giving an example of a plan, let us detail the abstract idea presented in Fig. 6.1. There, we have four components involved:*

- *p, a component that is allowed to run during the reconfiguration process, although it gets modified. We define it as a tuple* $(p, \{P\}, \{cons \mapsto C\}, \zeta(p), \iota(p))$ *(without substantiating the interfaces P and C, as well as* $\zeta(p)$ *and* $\iota(p)$*, as they are unimportant for this example),*
- *oc, the component that gets removed, defined as a tuple* $(oc, \{C\}, \{prod \mapsto P\}, \zeta oc, \iota(oc))$,
- nc_1*, a component that takes the connection of p's role cons, defined by the tuple* $(nc_1, \{C\}, \{next \mapsto N\}, \zeta(nc_1), \iota(nc_1))$,
- *and* nc_2*, defined by the tuple* $(nc_2, \{N\}, \{prod \mapsto P\}, \zeta(nc_2), \iota(nc_2))$.

The interface N may stand for P, C or a newly defined interface.

In order to effect the reconfiguration informally given in Fig. 6.1, we utilize the plan $\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$ *with*

$$
\begin{aligned}
&A = \{nc_1, nc_2\}, \quad R = \{oc\},\\
&\alpha = \{nc_1 \mapsto \{next \mapsto nc_2\}, nc_2 \mapsto \{prod \mapsto p\}\},\\
&\rho = \{(p, cons) \mapsto nc_1\},\\
&\delta = \{(oc, (p, cons)) \mapsto nc_1\},\\
&\varsigma = \{(f, nc_1) \mapsto s_1(f(oc)), (f, nc_2) \mapsto s_2(f(oc))\}.
\end{aligned}
$$

ς *is defined by the means of two functions* $s_1 : \mathscr{S} \to \mathscr{S}$ *and* $s_2 : \mathscr{S} \to \mathscr{S}$ *which take the current state of oc and calculate the new state of* nc_1 *and* nc_2*, respectively. This* ς *is a very high-level abstraction, of course; we will discuss ways to actually implement it in Sect. 6.8.*

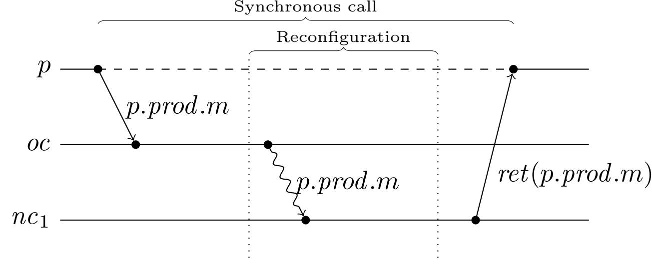


Figure 6.2: Continuation of synchronous calls during reconfiguration

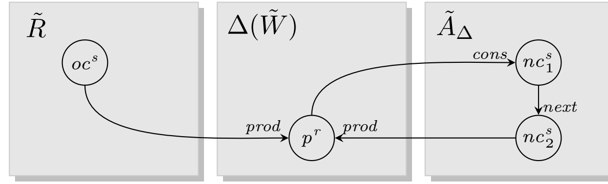


Figure 6.3: Sets of components for the coarse-grained rule

Reconfiguration may only commence if the components of R are not executing a method, i.e., the process term is either success or fail. This is a common approach and guarantees that every method is executed completely 8.1.4.2), and satisfies part of the requirements of *quiescence* [**KM90**], which we will discuss in greater detail in Sect. 8.1.4.2.

6.5.1. Continuation of Synchronous Calls. Fig. 6.2 shows a *space-time diagram* [**Lam78, CBMT96**] that illustrates how a synchronous call $m \in C$ issued by component p over its role *cons* is to be completed, even if the component *oc* it was sent to is replaced by nc_1 during a reconfiguration. Because the message m bears a reference to the component it was sent from, this knowledge is retained when the message is moved from *oc* to *nc* according to δ (indicated by the $\rightsquigarrow$ arrow). Hence, having p blocked does not impose a problem for executing a reconfiguration. Note that the opposite direction – i.e., replacing the sender of a synchronous call – is not possible, as the component can only be stopped when it is not blocked. Such a situation results in a deadlock, which, although being an undesirable condition, alleviates us from updating the sender reference of messages during reconfiguration.

6.6. Rules for Reconfiguration

6.6.1. A Coarse-Grained Rule. When a plan Δ is applied to a configuration $\tilde{C}$, the component set C is partitioned in three sets: The set R of components that get removed, a set $W = \mathrm{dom}(\rho)$ of components that need to have a role rewired (by definition, W is disjoint to R), and the set $C \setminus (R \cup \mathrm{dom}(\rho))$ of components that are not modified at all. After reconfiguration, the set A is added; hence we can describe the effect of reconfiguration by the rule

$$\tilde{R} \parallel \tilde{W} \rightarrow \Delta(\tilde{W}) \parallel \tilde{A}_\Delta \qquad \text{(RECONF)}$$

with all components in R not performing a method, i.e., $\tilde{R} = c_1^{\mathrm{r}}, P_1 \parallel \ldots \parallel c_n^{\mathrm{r}}, P_n$ with $P_i \in \{\mathrm{success}, \mathrm{fail}\}$, $\Delta(\tilde{W})$ describing the effect of the rewiring as defined by ρ, and $\tilde{A}_\Delta$ consisting of new components, which are initialized using α, δ and ς. In

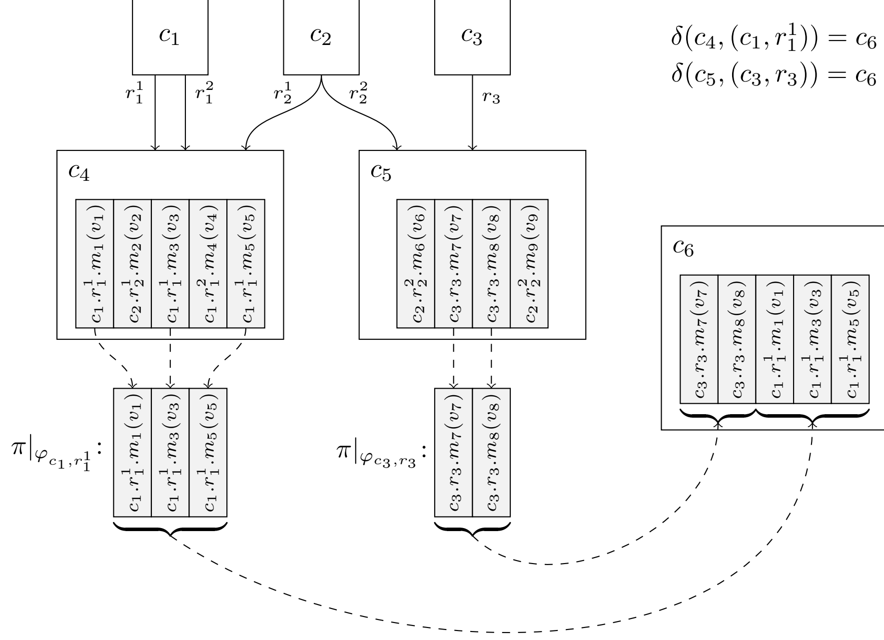


Figure 6.4: Message copying via δ – c_6 is assigned the messages sent by c_1 to c_4 using role r_1^1, and those sent by c_3 to c_5

particular, if $\tilde{W} = c_{n+1}, \gamma_{n+1} \parallel \ldots \parallel c_{n+m}, \gamma_{n+m}$, then $\Delta(\tilde{W}) = c_{n+1}, \gamma'_{n+1} \parallel \ldots \parallel c_{n+m}, \gamma'_{n+m}$ with

$$\gamma'_{n+i}(r) = \begin{cases} \rho(c_{n+i}, r), & \text{if } \rho(c_{n+i}, r) \text{ is defined} \\ \gamma_{n+i}(r), & \text{otherwise.} \end{cases}$$

Thus, $\Delta(\tilde{W})$ is the set of components $\tilde{W}$ with the connections that are defined in the rewiring function ρ modified according to ρ, and otherwise unmodified.

The configuration of new components $\tilde{A}_\Delta$ is

$$\tilde{A}_\Delta = \{c \mapsto c^{\mathrm{r}}, \alpha(c), \langle \bot, \varsigma(\sigma_R, c) \rangle, \text{success}, \pi_c \mid c \in A\} \ .$$

Herein, σ_R is the function capturing the states of components in R, i.e., for $c' \in R$ and thus $\tilde{c}'$ being part of $\tilde{R}$, we have $\sigma_R(c') = \sigma(\tilde{c}')$.

The message queue π_c is a linearization of the parallelization (or shuffling) of message sequences copied by δ: Let $m_e = e(\tilde{c}_t)$ if $P(\tilde{c}_t) = \text{fail}$ and $m_e = \varepsilon$ otherwise. If $(c_t, (c_s, r_i)) \in \delta^{-1}(c)$, then $(m_e :: \pi(\tilde{c}_t))\,|_\varphi$ is a subsequence of π_c, for $\varphi \equiv \{c_s.r_i.m(v) \mid m \in \mathscr{M} \wedge v \in \mathscr{V}\}$, and π_c consists exactly of these subsequences. Note that the order of the subsequences is unspecified, which makes plan application nondeterministic. m_e is the message that produced a fail, and needs to be processed by the substituting component again.

Fig. 6.4 illustrates how messages are reassigned. The assignment is completely arbitrary; any new component can get messages from a component that gets removed, given the ability to process them. This is usually much stronger than required; we will define a more restricted δ in Sect. 6.7.1. The only restriction here is that those messages sent by some component over one of its roles are transferred en bloc; in the absence of timestamps no useful ordering can be provided. This is

problematic in situations where indeed multiple blocks are to be copied, as it might amount to message overtaking – which makes reconfiguration observable. This is a shortcoming that can only be mended by further complicating the component model; for practical examples, we do not use the capability of reconfiguration plans anyway.

The RECONF rule performs the entire task of reconfiguration at once, hence ensuring atomicity of the reconfiguration process. The illustration of the three sets $\tilde{R}$, $\Delta(\tilde{W})$ and $\tilde{A}_\Delta$ in Fig. 6.3 is hence never a valid configuration of the system and depicts sort of an intermediate result merely for the illustration of the sets. We will now refine the rules into a sequence of much finer rules (which can rightfully be assumed to be atomic) and proceed to show that these rules can be applied in a way that is indistinguishable from the effect of the RECONF rule.

6.6.2. Fine-Grained Rules. In order to build fine-grained rules that satisfy requirement R2.1, we extend the set of component running states to $\{\mathrm{n}, \mathrm{i}, \mathrm{r}, \mathrm{b}, \mathrm{s}, \mathrm{c}\}$. The intended state machine for a component can be seen in Fig. 6.5. The new states have the following semantics:

- n: A newly initialized component that needs to be connected to other components.
- i: Once connected, a new component is put into this state, which is used for retrieving data and messages from old components, thus initializing the new component.
- s: Once a component is scheduled for removal, it is put into this state; it remains there until it has become entirely unconnected.
- c: Now that we are assured that no more messages are put into the queue from the outside (all components still connected to this component are in an s or c state), this state is taken, which allows the copying of parts of the message queue and the querying of the component state.

The fine-grained reconfiguration rules are shown in Tab. 6.2. There are two sets of rules: Rules that change the component running state of the components to be added and to be removed, and rules that modify the components' connections, the message queue and the data state. The former set consists of RCADD (adding a new component), RCINIT, RCSTART, RCSTOPS and RCSTOPF, RCCOPY, which can be applied to a stopped component once it is safe for having its state copied to other components, i.e., no active component's role points to the stopped component anymore and finally RCKILL, used to dispose of a component. (Note that RCCOPY needs to consider the entire configuration.)

Since they are much alike, we will frequently write RCSTOP to identify the rules RCSTOPF and RCSTOPS.

The set of rules to change the data state consists of RCWIRE to connect the roles of a recently added component to other components; RCREWIRE to reconnect an existing component's roles such that they point to new instead of to stopped

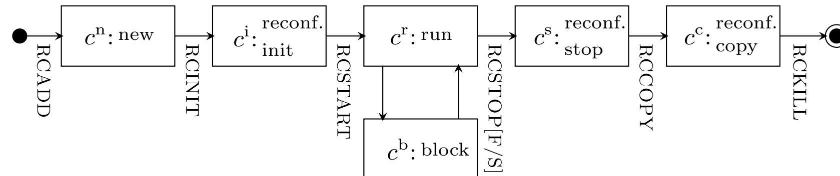


Figure 6.5: State machine of a component, with rules to reach new states

$$\tilde{C} \to \tilde{C} \parallel c^{\mathrm{n}}, \gamma_\perp, \langle \iota(c) \rangle, \mathrm{success}, \varepsilon \quad \text{if } \mathrm{dom}(\gamma_\perp) = \varnothing \tag{RCADD}$$

$$c_1^{\mathrm{n}}, \gamma_1 \parallel c_2 \to c_1^{\mathrm{n}}, \gamma_1[r \mapsto c_2] \parallel c_2 \quad \text{if } \gamma_1[r \mapsto c_2] \text{ is well-connected for } c_1 \tag{RCWIRE}$$

$$c^{\mathrm{n}}, \gamma \to c^{\mathrm{i}}, \gamma \quad \text{if } \gamma \text{ is completely connected} \tag{RCINIT}$$

$$c^{\mathrm{i}} \to c^{\mathrm{r}} \tag{RCSTART}$$

$$c^{\mathrm{r}}, \mathrm{success} \to c^{\mathrm{s}}, \mathrm{success} \tag{RCSTOPS}$$

$$c^{\mathrm{r}}, \langle c_2.r.m(v) \rangle, \mathrm{fail}, \pi \to c^{\mathrm{s}}, \mathrm{fail}, c_2.r.m(v) :: \pi \tag{RCSTOPF}$$

$$c_1^{\mathrm{s}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \to c_1^{\mathrm{c}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \quad \text{if } \forall 2 \leq i \leq n \,.\, c_1 \in \mathrm{ran}(\gamma_i) \to f_i \in \{\mathrm{s}, \mathrm{c}\} \tag{RCCOPY}$$

$$\tilde{C} \parallel c^{\mathrm{c}} \to \tilde{C} \tag{RCKILL}$$

$$c_1, \gamma_1 \parallel c_2 \parallel c_3 \to c_1, \gamma_1[r \mapsto c_3] \parallel c_2 \parallel c_3 \quad \text{for } r \in \mathscr{R} \text{ with } \gamma_1(r) = c_2 \text{ if } \gamma_1[r \mapsto c_3] \text{ is well-connected for } c_1 \tag{RCREWIRE}$$

$$c^{\mathrm{i}}, \langle \sigma \rangle \parallel c_1^{\mathrm{c}}, \langle \sigma_1 \rangle \parallel \ldots \parallel c_n^{\mathrm{c}}, \langle \sigma_n \rangle \to c^{\mathrm{i}}, \langle \varsigma((c_1 \mapsto \sigma_1, \ldots, c_n \mapsto \sigma_n), c) \rangle \parallel c_1^{\mathrm{c}}, \langle \sigma_1 \rangle \parallel \ldots \parallel c_n^{\mathrm{c}}, \langle \sigma_n \rangle \tag{RCSTATE}$$

$$c_1^{\mathrm{i}}, \pi_1 \parallel c_2^{\mathrm{c}}, \pi_2 \parallel c_3 \to c_1^{\mathrm{i}}, \pi_2|_\varphi :: \pi_1 \parallel c_2^{\mathrm{c}}, \pi_2|_{\neg\varphi} \parallel c_3 \quad \text{for } r \in \mathscr{R}(c_3) \text{ and } \varphi \equiv \{c_3.r.m(v) \mid m \in \mathscr{M}, v \in \mathscr{V}\} \tag{RCGETMSG}$$

Table 6.2: Reconfiguration transition rules

components; RCSTATE to take the states of the stopped components and combine them to a state for a new component; and RCGETMSG to transport residual messages of stopped to new components. RCSTATE is an abstraction of a subprotocol that is performed to have the new component query the state of the old component; a description of this process, which requires further component states and special restrictions on the process terms to avoid side-effects; a refined approach will be presented in Sect. 6.8.

RCSTOPF is used to handle a component configuration with a fail process term; only this rule can advance such a configuration. Hence, fail is used to trigger a reconfiguration, which needs to follow a plan which disposes of the failed component. The method that failed must not be lost; maybe some other component is waiting for the return of the method, which would result in a deadlock. Thus, the method that produced the failure is prepended to the message queue, so that during reconfiguration, it can be moved to a new component which is capable of handling it properly.

Expressing such reconfiguration rules illustrates that the JCOMP component model indeed satisfies the requirements R1.1 to R1.4, since the necessary information is stored in the state. The retainment of the source role for messages in the queue π is done exclusively for satisfaction of requirement R1.4. Likewise, the definition of a configuration as a partial function is necessary for satisfying requirement R1.3, as the set of existing components might be changed between configurations. Also note that the rules RCSTOP satisfy requirement R3.1.2, as they only apply to components that are not processing a method. Finally, R3.3 is respected, since the method evaluator is not required to have a special form for any of these rules.

This is attributed to a level of abstraction still present in the rule RCSTATE, we will discuss the problems with data state retainment in Sect. 6.8.

6.7. Reconfiguration Plan Implementation

Given a plan $\Delta = (A, R, \alpha, \rho, \delta, \varsigma)$, we can *implement* Δ using a sequence of rule applications:

(1) For each $c \in R$, RCSTOPS(c) or RCSTOPF(c) is used to stop the component. Note that this requires each $c \in R$ to eventually stop processing the current method. This may lead to deadlocks (if a component $c \in R$ is blocked due to waiting for the processing of a synchronous message, but the message's target has already been stopped, with the synchronous method still residing unprocessed in its queue); in the spirit of Sect. 9.1 we require the person developing the reconfiguration plan to avoid such situations.
(2) For each $c \in A$, we use RCADD(c) to instantiate the component.
(3) For each $c \in A$ and each $r \in \mathscr{R}(c)$, we use RCWIRE$(c, \alpha(c)(r))$, and for each $(c', r) \in \mathrm{dom}(\rho)$, we use RCREWIRE$(c_1 : c', c_3 : \rho(c', r))$. We then use, for each $c \in A$, RCINIT(c). This step connects the new and disconnects the old components.
(4) Then, we use RCCOPY(c) for each $c \in R$. For each $(c', (c, r)) \in \mathrm{dom}(\delta)$, we use RCGETMSG$(c_1 : \delta(c', (c, r)), c_2 : c', c_3 : c, r : r)$, thus copying all messages sent to c' over the role r from component c to a new component.
(5) For each $c \in A$, we use RCSTATE$(c, c_1, \ldots, c_n)$ for $\{c_1, \ldots, c_n\} = R$.
(6) Now, for each $c \in R$, RCKILL(c) is used to remove the component, and, for each $c \in A$, RCSTART(c) is used to start the components.

The order of the rule sequence is important, but rules may be exchanged for a single list item (e.g., the order of the RCADD applications is not important, but they need to commence before any RCWIRE rule is applied). Compared to the coarse-grained rule RECONF, the independence of the RCSTOP rule applications lead to a faster initiation of reconfigurations – we do not have to wait until all components have stopped processing a method, but shut them down as soon as possible. This satisfies requirement R4.1.

EXAMPLE 6.2. *The reconfiguration of Fig. 6.1 can be described by the following actions: From the initial configuration, we reach configuration (1) by applying* RSTOPS$(c : \mathrm{oc})$*; and* RCADD$(c : \mathrm{nc}_1)$, RCADD$(c : \mathrm{nc}_2)$ *and* RCWIRE$(c_1 : \mathrm{nc}_1, r : \mathit{next}, c_2 : \mathrm{nc}_2)$, RCWIRE$(c_1 : \mathrm{nc}_2, r : \mathit{next}, c_2 : \mathrm{p})$ *(in Fig. 6.1 we have identified all reconfiguration component flags with* s*). Configuration (2) is reached by applying* RCREWIRE$(c_1 : \mathrm{p}, c_3 : \mathrm{nc}_1, r : \mathrm{cons})$*. (3) depicts the applications of* RCGETMSG *and* RCSTATE*, and configuration (4) is reached by applying* RCKILL$(c : \mathrm{oc})$ *and* RCSTART$(c : \mathrm{nc}_1)$, RCSTART$(c : \mathrm{nc}_2)$ *for the new components.*

6.7.1. Injective Shallow Reconfiguration Plans. The generic definition of δ is devised to allow for arbitrary message retainment; that is, arbitrary with respect to the choice of the target components for messages to be moved. This can lead to some problems; a reconfiguration may require to remove a large set of components, some of which are only connected from components that also get removed. For those components (e.g., component O_2 in Fig. 6.6), a canonical message redistribution cannot be given; the unprocessed messages might have to be transferred to a new component that acts as a replacement. Since we are interested in a reconfiguration that has minimal impact, we are interested in reconfiguration plans that retain messages in a canonical way; therefore we need to restrict the plans. Most of the time, a single layer of components needs to retain messages, and the replacement

can be found out from ρ – i.e., if a role r of component c is connected to component c' and $\rho(c)(r) = c''$, then any message sent by c over r that is not yet processed in c' should be copied to c''. But since the definition of ρ allows for disconnecting a component that is not in R, we need to restrict ρ first, such that the derived definition of δ adheres to its definition.

We thus define a *shallow reconfiguration plan*:

DEFINITION 6.4 (Shallow reconfiguration plan). *A shallow reconfiguration plan Δ_s is a tuple $(A, R, \alpha, \rho, \varsigma)$ (with the types of the tuple elements as for reconfiguration plans) with ρ subjected to two restrictions:*

(1) $(c, r) \in \text{dom}(\rho) \to M(c)(r) \in R$, *i.e., only those connections that point to a component in R are rewired,*
(2) $(c, r) \in \text{dom}(\rho) \to \rho(c, r) \in A$, *i.e., rewirings always point to new components (thus making A the range of ρ).*

A shallow reconfiguration plan $(A, R, \alpha, \rho, \varsigma)$ translates to a plan $\Delta = (A, R, \alpha, \rho, \delta^\rho, \varsigma)$ with

$$\delta^\rho(c', (c, r)) = \begin{cases} \rho(c, r), & \text{if } c \in C \setminus R \\ \text{undefined}, & \text{if } c \in R. \end{cases}$$

Hence, messages are moved "with the rewiring", which, due to its well-connectedness also ensures that the messages can indeed be processed. Since ρ is defined to just rewire connections that point to a component to be removed to an added component, messages are never moved from or to a live component. As we will see in Sect. 6.7.2, this is a crucial thing to do, since it is a prerequisite for "non-observable" reconfiguration – during reconfiguration, it does not matter if a message is sent before or after the rewiring, as this message will end up in the same component anyway.

However, there is another problem involved if a new component gets connected to by more than one component; the order of the messages moved to the new component cannot be determined deterministically. Fig. 6.6 illustrate this problem, too (also cf. Fig. 6.4): N becomes a replacement for both O_1 and O_3, so it receives the messages sent by C, but the order is arbitrary. In order to avoid this situation, we extend shallow reconfiguration plans to *injective shallow reconfiguration plans*:

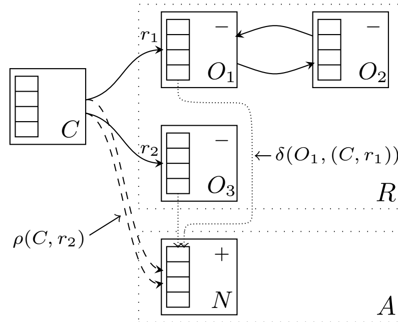


Figure 6.6: A reconfiguration scenario illustrating the problems with message retainment: Preserving message order (O_1,O_3) and finding a destination (O_2). Non-solid arrows represent the functions ρ and δ of a plan $\Delta = (\{N\}, \{O_1, O_2, O_3\}, \{\}, \rho, \delta, \varsigma)$.

Definition 6.5 (Injective shallow reconfiguration plan). *A shallow reconfiguration plan* $(A, R, \alpha, \rho, \varsigma)$ *is called* injective *if* $|\rho^{-1}(c)| \leq 1$ *for all* $c \in A$.

A injective shallow reconfiguration plan is given if the ρ function is injective. Due to the way δ^ρ is derived, δ^ρ will also be injective for an injective ρ. In terms of connections, we require that a new component is only pointed to by one old component at most. This avoids the nondeterminism introduced by the arbitrary order in which rcgetmsg rules are executed.

Example 6.3. *For the message retainment function* δ *and the rewiring function* ρ *of the plan of Example 6.1* $\delta = \delta^\rho$ *holds and also* $|\rho^{-1}(nc_1)| = |\{(p, cons)\}| \leq 1$ *and* $|\rho^{-1}(nc_2)| = |\varnothing| \leq 1$. *Thus the plan can be represented by an injective shallow plan. (We will often drop the word "reconfiguration" if the plan being a reconfiguration plan is clear from the context.)*

Injectiveness is required to keep the reconfiguration outcome *deterministic*. Using non-injective plans we will encounter a nondeterministic reconfiguration outcome, namely if a queue receives messages from more than one source during reconfiguration. In a surprisingly large number of cases, this is no problem, but we will see that, without additional provisions, the central property of reconfiguration – *perceived atomicity* – is not preserved. But let us first investigate how perceived atomicity is preserved for injective shallow reconfiguration plans.

6.7.2. Interleaved Execution of Injective Shallow Reconfiguration Plans. A *plan execution* for a plan Δ is a run that is conducted following the rules of a plan implementation of Δ. An *interleaved* plan execution is a run whose labels are according to both normal and reconfiguration rules, with the steps conducted by reconfiguration rules following the plan implementation of Δ. Such an interleaved plan execution is a run of a component system that gets reconfigured by Δ, while the components not in $A \cup R$ continue their execution.

Example 6.4. *During the interleaved plan execution of the plan of Example 6.1, component* p *may continue to issue asynchronous messages to the component connected to its* cons *role. While reconfiguring, not only the state of* oc *is copied to* nc_1 *and* nc_2*, but also those messages that have not yet been processed by* oc *are moved to* nc_1 *(due to using a shallow plan, they follow the rewiring), including those issued by* p *in the time period between starting the reconfiguration and rewiring the* cons *role. Note that if the reconfiguration was triggered by a process term* fail *in* oc*, the message that caused it would be copied to* nc_1 *also due to the re-enqueueing of the message by the rule* rcstopf *and the subsequent copying by* rcgetmsg*.*

An important property of our approach is to ensure that the reconfiguration remains local; i.e., only a part of the component system is concerned. Thus, while a plan execution may be mixed with arbitrary steps of other components, these other components do not observe the reconfiguration until it is completely finished. As mentioned in the introduction, this "hot reconfiguration" requires some careful treatment of the components' states. For shallow plans, messages are transported in accordance to the rewiring of the retained components. Since this puts the messages sent before and after the application of rcgetmsg to the same component, and since injectivity of ρ prohibits ambiguity with respect to the order of copied messages, we can show that reconfiguration of injective shallow plans is indeed observed as atomic.

In more detail, we prove that an interleaved plan execution of a shallow plan Δ_s can be simulated by another interleaved plan execution of Δ_s in which all reconfiguration actions are grouped together and thus could be performed in a single atomic step (as it would be done by an application of the rule reconf). To

this end, we define $(\tilde{N}, p, q)$ as a triple with $\tilde{N}$ being a configuration in which all $c \in \mathrm{dom}(\tilde{N})$ either are running or blocked; p a sequence of planned reconfiguration actions for Δ_s in the order given by the execution of Δ_s; and q a sequence of non-reconfiguration actions. We say that a component configuration $\tilde{C}$ is *simulated* by a triple $(\tilde{N}, p, q)$, written as $\tilde{C} \preccurlyeq (\tilde{N}, p, q)$, if there exist $\tilde{C}^{(1)}$ and $\tilde{C}^{(0)}$ such that

$$\tilde{C} = \tilde{C}^{(0)} \xleftarrow{q} \tilde{C}^{(1)} \xleftarrow{p} \tilde{N} \ ;$$

note that if $\tilde{C}^{(1)}$ and $\tilde{C}^{(0)}$ exist, they are uniquely determined. The regrouping of reconfiguration actions in a simulating execution is afforded by an LTS with its states given by the triples $(\tilde{N}, p, q)$ as defined above, rule instances as labels, and transitions $(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}', p', q')$ defined by

$$\begin{array}{ll}
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}', p, q) & \text{if } a \text{ is a non-reconfiguration action} \\
& \quad \text{and } p \text{ contains only RCSTOP actions} \\
& \quad \text{and } \tilde{N} \xrightarrow{a} \tilde{N}', \\
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}, p :: a, q) & \text{if } a \text{ is a reconfiguration action,} \\
(\tilde{N}, p, q) \overset{a}{\Longrightarrow} (\tilde{N}, p, q :: a) & \text{if } a \text{ is a non-reconfiguration action and} \\
& \quad p \text{ contains an action other than RCSTOP.}
\end{array}$$

THEOREM 6.1. *Let $\Delta_s = (A, R, \alpha, \rho, \varsigma)$ be an injective shallow reconfiguration plan, and let $\tilde{C}_0 \xrightarrow{a_0} \tilde{C}_1 \xrightarrow{a_1} \dots \xrightarrow{a_{n-1}} \tilde{C}_n$ be an interleaved plan execution of Δ_s. Then there is a sequence $(\tilde{N}_0, p_0, q_0) \overset{a_0}{\Longrightarrow} \dots \overset{a_{n-1}}{\Longrightarrow} (\tilde{N}_n, p_n, q_n)$ such that $\tilde{C}_k \preccurlyeq (\tilde{N}_k, p_k, q_k)$ for all $0 \le k \le n$ with $\tilde{N}_0 = \tilde{C}_0$, $p_0 = \varepsilon$ and $q_0 = \varepsilon$.*

PROOF. First of all, $\tilde{C}_0 \preccurlyeq (\tilde{C}_0, \varepsilon, \varepsilon) = (\tilde{N}_0, p_0, q_0)$. Let the claim hold up to some $0 \le k < n$. In order to show that there is an $(\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ with $(\tilde{N}_k, p_k, q_k) \overset{a_k}{\Longrightarrow} (\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ and $\tilde{C}_{k+1} \preccurlyeq (\tilde{N}_{k+1}, p_{k+1}, q_{k+1})$ we proceed by a case distinction on the action a_k:

If a_k is a non-reconfiguration action and p_k contains an action other than RCSTOP, we may trivially choose $(\tilde{N}_k, p_k, q_k :: a_k)$.

If a_k is a non-reconfiguration action, but p_k only contains RCSTOP actions, we have to provide $\tilde{N}_{k+1}$, $\tilde{C}^{(1)}_{k+1}$, and $\tilde{C}^{(0)}_{k+1}$ such that the following diagram commutes:

$$\begin{array}{ccccccc}
\tilde{C}_k & = & \tilde{C}^{(0)}_k & \xleftarrow{q_k} & \tilde{C}^{(1)}_k & \xleftarrow{p_k} & \tilde{N}_k \\
\downarrow{\scriptstyle a_k} & & \downarrow{\scriptstyle a_k} & & & & \downarrow{\scriptstyle a_k} \\
\tilde{C}_{k+1} & = & \tilde{C}^{(0)}_{k+1} & \xleftarrow{q_k} & \tilde{C}^{(1)}_{k+1} & \xleftarrow{p_k} & \tilde{N}_{k+1}
\end{array}$$

But as p_k only contains RCSTOP actions, q_k is ε; the component conducting a_k is not a parameter of an action in p_k and hence remains unaffected by p_k. Thus $\tilde{N}_{k+1}$ and $\tilde{C}^{(0)}_{k+1} = \tilde{C}^{(1)}_{k+1}$ can be defined as the result of applying a_k to $\tilde{N}_k$ and p_k to $\tilde{N}_{k+1}$.

If a_k is a reconfiguration action, we have to provide $\tilde{C}^{(1)}_{k+1}$ and $\tilde{C}^{(0)}_{k+1}$ such that the following diagram commutes:

$$\begin{array}{ccccccc}
\tilde{C}_k & = & \tilde{C}^{(0)}_k & \xleftarrow{q_k} & \tilde{C}^{(1)}_k & \xleftarrow{p_k} & \tilde{N}_k \\
\downarrow{\scriptstyle a_k} & & \downarrow{\scriptstyle a_k} & & \downarrow{\scriptstyle a_k} & & \\
\tilde{C}_{k+1} & = & \tilde{C}^{(0)}_{k+1} & \xleftarrow{q_k} & \tilde{C}^{(1)}_{k+1} & &
\end{array}$$

All actions in q_k have been invoked for a component in state r or b. Hence all reconfiguration actions applying to a single component only (except RCSTOP) apply

to some other state (n, i, s or c). In particular, the rules RCADD, RCINIT, RCSTART and RCKILL are independent of q_k. Rules RCWIRE and RCCOPY do not modify the other components, and do not rely on parts of the component configuration that can be changed by normal rules. For RCSTOP, we have that $q_k = \varepsilon$, since a plan is executed and we would only have actions in q_k once all $c \in R$ are stopped. RCSTATE also only uses components in a reconfiguration state and only uses the state part of their configuration that is not changed by actions in q_k. This leaves two reconfiguration actions that actually interfere with actions in q_k:

- $a_k \equiv \text{RCREWIRE}(c_1, c_3, r)$: Any message sending from c_1 over role r in q_k will be executed as a message sending to c_3. Using δ^ρ, this is where the messages sent over this role are prepended to, and since $|\rho^{-1}(c_3)| \leq 1$ the order is kept the same. Since the range of ρ is restricted to be a subset of A, messages are not processed until the end of the reconfiguration.
- $a_k \equiv \text{RCGETMSG}(c_1, c_2, c_3, r)$: As the messages are prepended to the queue of c_1 and c_1 is in state i and has not yet dequeued any message, the copying does not interfere with any sending action in q_k. If c_2 is the target of a sending action in q_k, ρ will point $c_3.r$ to c_1, which is where the message would have been copied to according to δ^ρ. Component c_3 is not modified.

Thus we can choose $\tilde{C}^{(1)}_{k+1}$ to be the result of applying a_k to $\tilde{C}^{(1)}_k$ and $\tilde{C}^{(0)}_{k+1}$ as the result of applying q_k to $\tilde{C}^{(0)}_{k+1}$. □

Informally, Theorem 6.1 states that if we have an interleaved execution of an injective, shallow reconfiguration plan Δ, then we can just reorder the steps such that we first advance till the first reconfiguration step other than RCSTOP is observed, then we do all the reconfiguration steps all at once, and proceed to do the remaining, non-reconfiguration steps. The crucial part is that any message sent by a non-reconfigured component to a component that gets reconfigured will be copied by δ^ρ to the place it would have arrived if it had been sent after the reconfiguration. Injectivity is required such that messages do not get mixed up. Eventually, the execution of interleaved reconfiguration and such an atomic reconfiguration will produce the same configuration.

However, the theorem does not say anything about the states in between. And, indeed, they are not alike: For interleaved reconfiguration is is, for example, possible that a state has two components' queues contain messages sent by the same component over the same role, which is not possible with atomic reconfiguration. But if we take a step back, and use traces to hide the details that are dependent on the reconfiguration progression, there is no perceivable difference, which shows that the reconfiguration of JCOMP, for injective shallow reconfiguration plans, satisfies R3.1.1:

COROLLARY 6.1. *Let $\mathscr{T}_a$ be the transition system of a component system that can get reconfigured by atomic execution of an injective shallow plan Δ_s, and let $\mathscr{T}_i$ be the transition system of the same component system that can get reconfigured by interleaved execution of Δ_s. Then $Traces^{comm}(\mathscr{T}_a) = Traces^{comm}(\mathscr{T}_i)$.*

PROOF. $Traces^{comm}(\mathscr{T}_a) \subseteq Traces^{comm}(\mathscr{T}_i)$: Obviously, any run of $\mathscr{T}_a$ is also a run of $\mathscr{T}_i$; hence any communication trace of $\mathscr{T}_a$ is also a communication trace of $\mathscr{T}_i$.
$Traces^{comm}(\mathscr{T}_i) \subseteq Traces^{comm}(\mathscr{T}_a)$: In order to show this, we consider LTS $\mathscr{T}'_a$ respectively $\mathscr{T}'_i$ that differ from the aforementioned LTS in that the components keep a track of their received messages in their data state $\sigma(c)$. To this end, we assume a component setup where each method evaluator $\zeta(c)$ is replaced by an evaluator $\zeta'(c)$ such that $\zeta'(c)(m) = P.\zeta(c)(m)$, and P is a statement that

protocols the method invocation in the component's data state (e.g., $P \equiv comm = comm :: (r, m, v)$, using the "programming language" notation of Sect. 4.6 and an otherwise unused variable *comm* that is initialized with ε). Furthermore, we extend the reconfiguration to include the addition of an unconnected component c_{comm} that stores all the *comm* variables of component to be removed (by using the reconfiguration plan element $\varsigma(f, c_{comm}) = \{id(c) \mapsto f(c)(comm) \mid c \in R\}$).

The transition systems of such a modified component setup $\mathcal{T}_a'$ and $\mathcal{T}_i'$ are weak bisimilar with respect to communication labels DEQ() to $\mathcal{T}_a$ respectively $\mathcal{T}_i$ by means of the relation $\tilde{C} \sim \tilde{C}'$ which we define as

$$\begin{aligned}&(\tilde{c}_1 \parallel \ldots \parallel \tilde{c}_n) \sim (\tilde{c}_1' \parallel \ldots \parallel \tilde{c}_n') \quad \text{iff}\\ &\quad \forall 1 \leq i \leq n \,.\, id(c_i) = id(c_i') \wedge f(\tilde{c}_i) = f(\tilde{c}_i') \wedge \gamma(\tilde{c}_i) = \gamma(\tilde{c}_i') \wedge \\ &\quad\quad e(\tilde{c}_i) = e(\tilde{c}_i') \wedge \sigma(\tilde{c}_i) = \sigma(\tilde{c}_i') \wedge P(\tilde{c}_i) = P(\tilde{c}_i') \wedge \pi(\tilde{c}_i) = \pi(\tilde{c}_i')\end{aligned}$$

Obviously, $\sim$ is a weak bisimulation relation with respect to communication (it is almost a strong bisimulation, but the single added statement that stores the communication, as well as changes to the reconfiguration execution due to the modified plan with the additional component c_{comm} are additional steps in the transition systems $\mathcal{T}_a'$ and $\mathcal{T}_i'$ that are not reflected in the unmodified systems and need to be hidden; but otherwise, all rule executions are directly repeatable in the other system, which effectuates the weak bisimilarity – assuming that the modified state is not considered by the visible actions), and by lemma 4.1 we have $Traces^{comm}(\mathcal{T}_a') = Traces^{comm}(\mathcal{T}_a)$ and $Traces^{comm}(\mathcal{T}_i') = Traces^{comm}(\mathcal{T}_i)$.

For two runs $s_1 = s_1^1 \xrightarrow{l_1^1} \ldots \xrightarrow{l_{n-1}^1} s_n^1$ and $s_2 = s_1^2 \xrightarrow{l_1^2} \ldots \xrightarrow{l_{m-1}^2} s_m^2$ of this LTS $\mathcal{T}_a$ and $\mathcal{T}_i$, we obtain $s_n^1 = s_m^2 \rightarrow Traces^{comm}(s_1) = Traces^{comm}(s_2)$ (since the communication history is stored in the state). We can then combine:

$$\begin{aligned}&t \in Traces^{comm}(\mathcal{T}_i)\\ &\quad \rightarrow t \in Traces^{comm}(\mathcal{T}_i') && \text{(Lemma 4.1)}\\ &\quad \rightarrow \exists r^1 = r_1^1 \xrightarrow{l_1^1} \ldots \xrightarrow{l_{n-1}^1} r_n^1 \xrightarrow{l_n^1} \ldots \in Runs(\mathcal{T}_i') \,.\, Traces^{comm}(r^1) = t\\ &\quad \rightarrow \exists r^2 = r_1^2 \xrightarrow{l_1^2} \ldots \xrightarrow{l_{m-1}^2} r_n^1 \xrightarrow{l_n^1} \ldots \in Runs(\mathcal{T}_a') \,.\, Traces^{comm}(r^2) = t\\ &&& \text{(Theorem 6.1)}\\ &\quad \rightarrow \exists r^2 \in Runs(\mathcal{T}_a') \,.\, Traces^{comm}(r^2) = t\\ &\quad \rightarrow t \in Traces^{comm}(\mathcal{T}_a')\\ &\quad \rightarrow t \in Traces^{comm}(\mathcal{T}_a) && \text{(Lemma 4.1)}\end{aligned}$$

□

Hence, for the restricted class of injective shallow reconfiguration plans, an outside observer (i.e., a component that monitors the communication that it receives) cannot distinguish between atomic and non-atomic reconfiguration. This property is important, as the programmer only needs to consider the effect of the atomic reconfiguration, and does not have to consider various interleaving possibilities which are prone to problems. Next, we will show that, in addition to the perceived atomicity, reconfiguration can also be completely undetectable if it does not introduce components with a changed behavior.

6.7.3. Transparency of Reconfiguration. Besides atomicity, reconfiguration should also be somewhat "natural" to the system. That is, even if the system is modified by reconfiguration, it should not exhibit behavior that can be attributed

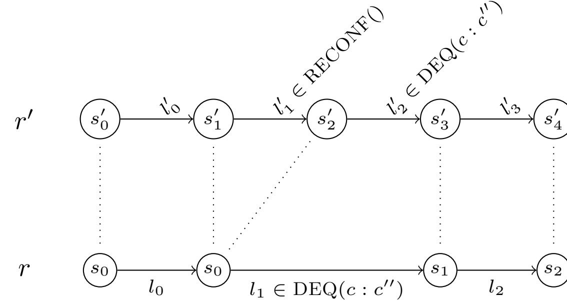


Figure 6.7: Example for embedding of reconfiguration as in lemma 6.2

to the reconfiguration process, which is mandated by requirement R4.2. This is intricate to define: Obviously, reconfiguration is intended to change the behavior of a system by substituting components with different communication. But other than this changed communication behavior, nothing should be observable. Obviously, no messages should be "made up" by the reconfiguration process itself, and if a component is substituted by another component that exhibits exactly the same communication behavior (and the state is transported in a consistent way) then no external observer should be able to see whether a reconfiguration took place at all.

As a litmus test for requirement R4.2, we show that component update is *transparent* to the system, meaning that if a component is updated such that its communication behavior remains the same, then no other components can judge whether a reconfiguration actually took place: Their communication traces do not change.

THEOREM 6.2. *Let $\mathscr{T} = (S, L, T)$ be a transition system of a component setup, and c and c' be components with $I_P(c) = I_P(c')$, $I_R(c) = I_R(c')$ and $\zeta(c) = \zeta(c')$. We require $c \in \mathrm{dom}(\tilde{C})$ and $c' \notin \mathrm{dom}(\tilde{C})$ for all $\tilde{C} \in S$. For $s \in S$ with $s = (\tilde{c} \parallel \tilde{c}_1 \ldots \parallel \tilde{c}_n)$ with $c_i \neq c'$ for all $1 \leq i \leq n$, we define $\Delta_s = (A, R, \alpha, \rho, \varsigma)$ as the shallow reconfiguration plan with $A = \{c'\}$, $R = \{c\}$, $\alpha = \{c \mapsto \{r \mapsto \gamma(\tilde{c})(r) \mid r \in \mathrm{dom}(I_R(c))\}\}$, $\rho = \{(c_i, r) \mapsto c' \mid \gamma(\tilde{c}_i)(r) = c\}$, $\varsigma = \{(f, c') \mapsto f(c)\}$.*

Let $\mathscr{T}_{\Delta_s}$ be the transition system of the component setup of $\mathscr{T}$ that gets reconfigured by interleaved plan execution of Δ_s in state s. Let $L = \{\mathrm{DEQ}(c'', m) \mid c'' \in C \setminus \{c, c'\} \wedge m \in \mathscr{M}\}$. Then $Traces^{comm}(\mathscr{T})|_L = Traces^{comm}(\mathscr{T}_{\Delta_s})|_L$.

PROOF. Due to corollary 6.1, we only need to consider atomic reconfiguration and the corresponding LTS $\mathscr{T}^a_{\Delta_s}$.

Note that, actually, a much stronger property holds: Since Δ_s is, in effect, a mere renaming of c to c', and even the runs are the same, except for the name change. Note that the LTS $\mathscr{T}^a_{\Delta_s}$ and $\mathscr{T}$ are not bisimilar, as $\mathscr{T}^a_{\Delta_s}$ uses different labels (rules instantiated with c' instead of c). Further note that, even if the relevant parts of the state are renamed, some references to c remain in the message queue.

For the term that results from substituting all occurrences of c in a term t with c', we write $t[c \mapsto c']$.

We first show that Δ_s amounts to a *partial* renaming $[c \mapsto c']$ by showing that the following diagram commutes:

$$\begin{array}{ccc} s & \xrightarrow{\text{RECONF}} & s' \\ \Vert & & \downarrow{\scriptstyle [c \mapsto c']} \\ s & \xmapsto[{[c \mapsto c']}]{} & s'' \end{array}$$

Let $s = (\tilde{c} \parallel \tilde{c}_1 \parallel \ldots \parallel \tilde{c}_i \parallel \ldots \tilde{c}_n)$. Since the reconfiguration plan Δ_s is injective, we can deduce that there is only one component c_i with $c \in \text{ran}(\gamma(\tilde{c}_i))$.

According to rule RECONF (cf. page 121), c is not performing a method (in particularly, not waiting for a pending synchronous call), and we have $s' = (\tilde{c'} \parallel \tilde{c}_1 \parallel \ldots \parallel \tilde{c'}_i \parallel \ldots \parallel \tilde{c}_n)$ with $\tilde{c'} = (c'^{\mathrm{r}}, \alpha(c'), \langle \bot, \sigma(\tilde{c}) \rangle, \text{success}, \pi(\tilde{c})) = \tilde{c}[c \mapsto c']$ (note that our definition of α rules out self-invocation loops for c and c'). $\gamma(\tilde{c'}_i) = \gamma(\tilde{c}_i)[r \mapsto c']$ for $r \in \mathscr{R}$ with $\gamma(\tilde{c}_i)(r) = c$. Hence $s'[c \mapsto c'] = s[c \mapsto c']$. Note that s' still contains c as a subterm, namely in the queues and the current method state part. But since c is not in a pending synchronous call, this is not problematic – no reference to c is ever used again, and we can proceed to show that any further action of the reconfigured system corresponds to an action of the renamed one:

$$\begin{array}{ccccccccc} s & \xrightarrow{\text{RECONF}} & s'_1 & \xrightarrow{l'_1} & s'_2 & \cdots\rightarrow & s'_{n-1} & \xrightarrow{l'_{n-1}} & s'_n \\ \Vert & & \downarrow{\scriptstyle [c \mapsto c']} & & \downarrow{\scriptstyle [c \mapsto c']} & & \downarrow{\scriptstyle [c \mapsto c']} & & \downarrow{\scriptstyle [c \mapsto c']} \\ s & \xmapsto[{[c \mapsto c']}]{} & s''_1 & \xrightarrow[l''_1]{} & s''_2 & \cdots\rightarrow & s''_{n-1} & \xrightarrow[l''_{n-1}]{} & s''_n \end{array}$$

with l''_i a label such that $l''_i = l'_i[c \mapsto c']$ if $l'_i \in \text{DEQ}()$ and $l''_i = l'_i$ otherwise. We elements of the state s'_i by $(\tilde{c}^{s'_i} \parallel \tilde{c}_1^{s'_i} \parallel \ldots \parallel \tilde{c}_n^{s'_i})$. By induction on the length of the runs i:

$i = 1$: We have $s'_1[c \mapsto c'] = s''_1$, and c is only subterm of $\pi(\tilde{c}''^{s'_1})$ and $e(\tilde{c}''^{s'_1})$ for all $c'' \in \text{dom}(s'_1)$.

$i \rightarrow i+1$: Assume $s'_i[c \mapsto c'] = s''_i$ and c is only subterm of $\pi(\tilde{c}''^{s'_i})$ and $e(\tilde{c}''^{s'_i})$ for $c'' \in \text{dom}(s'_i)$.

- $l'_i, l''_i \in \text{DEQ}()$: Since c is only part of $\pi(\tilde{c}''^{s'_i})$ and $e(\tilde{c}''^{s'_i})$ for $c'' \in \text{dom}(s'_i)$, it cannot be identified as the acting component c in the rule instantiation, but only as c'. Hence, if we instantiate the rule as $\text{DEQ}(\overline{c : c''}, c' \; : \; c, \overline{r : r'}, \overline{m : m'}, \overline{v : v'}, \overline{\pi : \pi'})$, we obtain $e(\tilde{c}''^{s_{i+1}}) = c.r'.m'(v')$, and $\pi(\tilde{c}''^{s_{i+1}}) = \pi'$, and since c does not become referenced in any other part, we retain the property of c being subterm only of $\pi(c''^{s'_{i+1}})$ and $e(\tilde{c}''^{s'_{i+1}})$ for $c'' \in \text{dom}(s'_{i+1})$.

  Since c is renamed to c' in s''_i, we need to consider the renamed rule $l''_i = l'_i[c \mapsto c']$, and obtain $\pi(\tilde{c}''^{s''_{i+1}}) = \pi(\tilde{c}''^{s'_{i+1}})[c \mapsto c']$ and $e(\tilde{c}''^{s''_{i+1}}) = e(\tilde{c}''^{s'_{i+1}})[c \mapsto c']$, hence $s''_{i+1} = s'_{i+1}[c \mapsto c']$.

  Note that, as c is only used for the rule element c', c will not show up in $\mathit{Traces}^{comm}(s'_1 \xrightarrow{l'_1} \ldots \xrightarrow{l'_{n-1}} s'_n)$.
- $l'_i, l''_i \in \text{RET}()$: Since c was not in a synchronous call when becoming replaced by c', all the occurrences of c in $\pi(\tilde{c}''^{s_i})$ and $e(\tilde{c}''^{s_i})$ are asynchronous calls. Hence, c never is instantiated in $l'_i \in \text{RET}()$. We can hence treat this case like the next:

- $l'_i, l''_i \notin$ DEQ(): Since c is not subterm of l'_i, and since $l''_i = l'_i$, we rewrite only identical parts of s'_i and s''_i and hence obtain $s'_{i+1}[c \mapsto c'] = s''_{i+1}$. Also, as c is not used in l'_i, we retain the property of c being only a subterm of $\pi(\tilde{c}''^{s'_{i+1}})$ and $e(\tilde{c}''^{s'_{i+1}})$ for $c'' \in \text{dom}(s'_1)$.

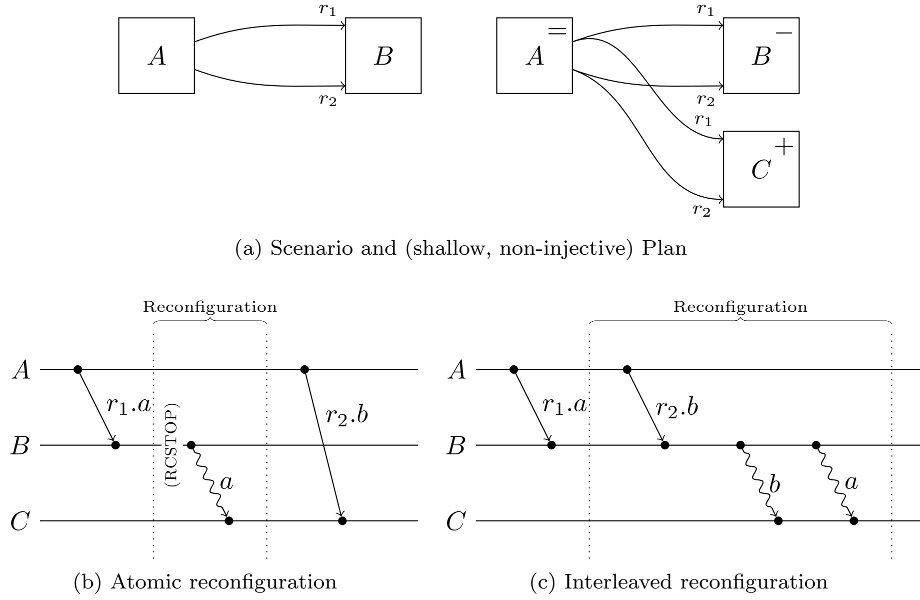
Hence, a run r that is reconfigured according to Δ_s acts like a run r' that is modified by a renaming $[c \mapsto c']$ at the same state s. In communication traces, we only consider components as the dequeueing component, and hence obtain $\mathit{Traces}^{comm}(r)|_L = \mathit{Traces}^{comm}(r')|_L$. □

The proof illustrates a subtle problem with the storing of message sources in within the messages, used for moving them during reconfiguration, and for returns of synchronous calls. In our approach, they do not get updated if the component that sends them gets removed (and keep a reference to a component no longer present in the system). This is not problematic for normal operation, since none of these messages is a synchronous call (the source component cannot become removed while one of its synchronous call is pending). We *can* get into problems if multiple reconfiguration is used; in this case, we would have to let δ define the transferal of the old messages as well, requiring non-shallow reconfiguration plans and running into the problems with message orderings discussed in Sect. 6.7.1. Alternatively, the plans might be extended to allow for a reassignment of the source components for messages in the queues of active components. We will later discuss (Sect. 8.3 and Sect. 10.2) that multiple reconfiguration is tricky. This problem can be seen as another argument for using ports (cf. Sect. 4.1.1), since maintaining global information in component-local data is prone to this kind of problems.

6.7.4. Non-Injective Shallow Reconfiguration Plans. Sometimes (e.g., in section 8.2.2) it is not possible to employ injective shallow reconfiguration plans. If multiple components are to be combined into a single component, a single-stage reconfiguration plan will not be injective. There is a workaround using (temporary or permanent) intermediate filter components, but this workaround cannot solve the problem of message ordering: Since $|\rho^{-}1(c)| > 1$ for at least one component, δ^{ρ} is not injective, hence ambiguity with respect to the ordering of the messages copied will arise.

Fig. 6.8 illustrates a problematic situation: Component A is connected to component B via two roles, r_1 and r_2. Reconfiguration replaces component B by a component C, rewiring both roles to this new component. If the plan is shallow, C takes all messages sent to B by A, but as they are copied according to the incoming role, their order might change – and exactly this happens in the plan execution depicted in Fig. 6.8c. According to proposition 6.1, a reordering is achievable that results in the same state. But if we just take a reordering as shown in Fig. 6.8b, a different state is obtained – a state where C's queue content is ab, whereas the queue content was ba before.

If we have a non-injective ρ (and hence a non-injective δ^{ρ}), another means for ordering messages deterministically has to be given, and the most obvious solution is to use message IDs. Any message created due to a call statement is given a unique, monotonically increasing message number. Using these IDs, the ordering of two messages sent by component c_1 to component c_2 and c_3, respectively, is determined by the order in which they got sent by component c_1. However, we found the scenarios in which the reordering of messages truly becomes a problem quite rare; the message order is only relevant for transactions extending over multiple components, and we will discuss in Chapter 9.4 that transactions are, in practice, often very limited. Often, a reordering of messages, especially those received from

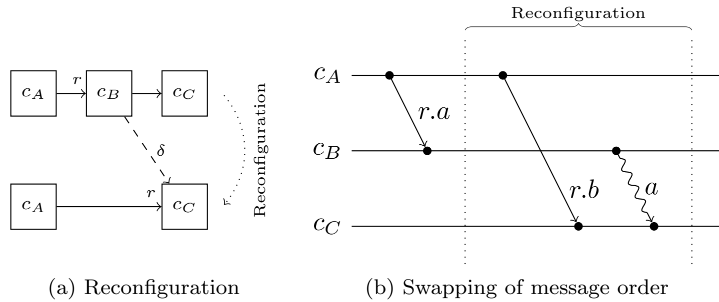


(a) Scenario and (shallow, non-injective) Plan

(b) Atomic reconfiguration

(c) Interleaved reconfiguration

Figure 6.8: A problem with non-injective shallow reconfiguration plans



(a) Reconfiguration

(b) Swapping of message order

Figure 6.9: Problems with an image intersecting $C \setminus R$ for ρ

different components, will not lead to problems; we have yet to find a practical example where a non-injective reconfiguration plan leads to problems.

6.7.5. The Restrictions of ρ. The range of ρ is restricted to be a subset of A for shallow reconfiguration plans. This might seem limiting at first, since there might be situations where a component is removed and its incoming connections are to be set to other, retained components. For example, a component might be removed from a Chain of Responsibility pattern, as depicted in Fig. 6.9. This example also illustrates the problem of message retainment with a less restricted rewiring – a ρ with a range that intersects $C \setminus R$ violates Theorem 6.1. Since we keep c_A and c_C running, c_C can receive (and, more importantly, process) messages from c_A as soon as the rewiring, according to $\rho = \{(c_A, r) \mapsto c_C\}$, has been effected. Since this happens before the message reassignment, the message a previously sent from

c_A to c_B is perceived to be overtaken by b, something that is impossible to achieve with an atomic reconfiguration step.

Obviously, this problem can be avoided by temporarily blocking c_A or c_C. Instead of extending our plans, we decided to implement such a blocking by the means that we already have: Temporarily stopping a component is produced by removing and re-adding it. State transferal is necessary to make the re-added component an exact duplicate of its old instance. It is straightforward to retain the component's representation in the component framework and avoid the tedious state transfer in such a situation. In the filter-removal example, c_C might become removed and re-added. Note that using a shallow reconfiguration plan will result in c_C taking the messages of c_B, but losing its own messages; if they need to be retained, a reconfiguration plan with a matched ρ needs to be employed, leading to the message ordering problems discussed before.

A similar situation is encountered when adding a filter component to a connection using a shallow reconfiguration plan: Since the domain of ρ is restricted to include only those component/role-pairs that point to a component in R, we again need to remove the target component in order to allow a reassignment of the role. For a filter component insertion, we would run into problems if the messages in the queue of the target component would have to be reassigned to the filter component; if this cannot be done in an atomic step, the target component again needs to be locked.

Using a restricted ρ and a derived δ^ρ within a shallow reconfiguration plans leads to plans that have a controllable effect on the application, but require detours for scenarios not supported directly. On the framework level, much can be remedied by providing automated optimizations; but eventually, the reconfiguration plan capabilities need to be chosen for the reconfiguration scenarios that are to be supported. We will discuss this for some examples of Chapter 8, where we need to add and remove filter components. We will sometimes revert to the full capabilities of ρ, but loose the atomicity property of the reconfiguration. Incidentally, while non-injective plans did not result in problems, we indeed experienced problems when dealing with Chain of Responsibility pattern situations. This was observed with the example presented in Sect. 8.3.2, where a number of chain components got removed; communication relayed down the chain can became lost. In such situations, an extended consideration of the communication behavior is required, as we will discuss in Sect. 8.1.4.2.

6.8. State Transferal

Now. . . where was I?
— Christopher Nolan, Memento

Following the definition of Vandewoude [Van07], data state transferal can be done using the indirect or the direct approach. The indirect approach requires the components to implement means for transferring the states. Usually this amounts to an implementation of the *Memento pattern*: The old component builds a memento object that encapsulates the state in a way that can be read by the new component. The direct approach, on the other hand, does not require components to be prepared for state transferal; instead, the reconfiguration manager handles the state copying by other means provided by the component framework.

It should be noted that the Memento pattern, as described in [GHJV95], is supposed to store and restore the state of a single object (e.g., for storing an "undo" history), instead of transporting the state to another object. However, some works

explicitly mention the pattern [**SPW03, IFMW08, EMS+08**], and since the departure from the original pattern is not too grave, we follow their naming. It should not be forgotten, however, that the question of how to restore the contents of a Memento object in an arbitrary component is not covered by the original pattern definition.

In Sect. 3.2.31, we have discussed the various frameworks that support state transferal, and seen that most of them take the indirect approach. In JComp, the indirect approach is utilized, too. The execution of the state transferal, however, employs the usual message passing mechanisms. There are two things required: First, temporary reconfiguration edges need to be defined in the plan and established during reconfiguration, and second, the new and the old components need to enter a distinct phase where a protocol for state retainment can be executed. In JComp, this protocol is triggered by calling a method on the new component that can then send queries to the old components using the temporary reconfiguration edges.

Yet, implementing such a protocol within a component is a breach of the separation of roles paradigm and requirement R3.3: The component writer has to be aware of future reconfiguration scenarios, which severely limits the variety of reconfiguration plans for a given set of components. This is an inherent problem of the indirect approach [**Van07**, pp. 44].

A pure direct approach, while offering the direct solution towards implementing ς and remaining in accordance to the requirements of Sect. 6.1, also breaks with important paradigms of component-based development. The problem is that the data stored within the component needs to be accessed during reconfiguration, e.g., by some script that implements the ς function of a reconfiguration plan. However, usually a component will not provide full access to its data. For example, a map component might only provide a method to check if an element is stored, but not an access to all its elements. Obviously such a method is insufficient for accessing the components internal state (given that the domain of values stored is reasonably large).

There are two solutions to this problem: Either enforce a component to provide access to all the relevant data, or access the data by other means (e.g., by traversing the object graph found within the component [**Van07, RS07a**]). Both are problematic: Providing explicit access easily becomes a breach of the separation of roles paradigm and requirement R3.3 again, as the component programmer needs to judge which data is relevant for later reconfiguration. External access constitutes a breach of the encapsulation paradigm of components. Furthermore, it requires detailed knowledge about the component's internals, and even if this is not ruled as another violation of the separation of roles paradigm, it prohibits evolvement of components and generic reconfiguration that can be applied in a variety of situations.

It should be noted that sometimes, the direct approach is applicable; e.g., if all the data that needs to be retained is guaranteed to be accessible. We will discuss this in Sect. 6.8.3, and give an example in Sect. 8.4.1. The scenario presented there, however, is too restricted for extending it to a general approach. Instead, we will investigate an approach that limits the use of indirect state transfer protocols as much as possible, while still not requiring explicit data accessor methods – we call this a "hybrid" approach, and discuss it in Sect. 6.8.2.

6.8.1. Indirect State Transferal. As mentioned before, this approach requires temporary roles. In the actual JComp implementation, these roles are just like ordinary roles, but are annotated as "configuration edges" that are to be connected (and used) only during reconfiguration. Obviously, this collides with our idea of

components being completely connected during normal operation. We hence consider these temporary edges in the extended reconfiguration plan only, and extend the definition of a component being completely connected. To this end, we split the set of role names into two sets $\mathscr{R} = \mathscr{R}_{perm} \cup \mathscr{R}_{temp}$ with $\mathscr{R}_{perm} \cap \mathscr{R}_{temp} = \varnothing$ and redefine "completely connected" (cf. definition 4.4 on page 72) to be

$$\mathrm{dom}(\gamma(c)) = \mathscr{R}(c) \cap \mathscr{R}_{perm}.$$

Also, we need to restrict ρ and α to address only permanent roles:

- $(c, r) \in \mathrm{dom}(\rho) \rightarrow r \in \mathscr{R}_{perm}$,
- for each $a \in A$, $r \in \mathrm{dom}(\alpha(a)) \rightarrow r \in \mathscr{R}_{perm}$.

Other than that, a state transferal method m_{copy} is required to be implemented by each component in A that performs the state copying protocol. It is to perform a sequence of synchronous calls over the temporary roles, but not on any other roles. Likewise, the methods called on the old component during this state transferal protocol are not to perform any other communication.

We define the *call set* of a component process term P inductively as:

$$\mathit{calls}(\mathrm{call}(r, m, v).P) = \{r\} \cup \mathit{calls}(P)$$
$$\mathit{calls}(\mathrm{return}(v).P) = \mathit{calls}(P)$$
$$\mathit{calls}(\mathrm{set}(\sigma).P) = \mathit{calls}(P)$$
$$\mathit{calls}(\mathrm{choose}((\Sigma_j.P_j)_{j \in J}) = \bigcup_{j \in J} \mathit{calls}(P_j)$$
$$\mathit{calls}(\mathrm{success}) = \varnothing$$
$$\mathit{calls}(\mathrm{fail}) = \varnothing$$
$$\mathit{calls}(X) = \varnothing$$
$$\mathit{calls}(\mu X \Rightarrow P) = \mathit{calls}(P)$$

The call set is the set of all roles communication might be done over during a method execution. We can then require the call set of the reconfiguration copy message to be comprised of temporary edges only, and the call set of the old component's getter methods to be empty.

For the temporary edges, we extend a reconfiguration plan $(A, R, \alpha, \rho, \delta, \varsigma)$ (and, at the same time, drop the ς element) with an element $\tau : A \rightarrow (\mathscr{R}_{temp} \rightharpoonup R)$. τ needs to connect all temporary required interfaces:

$$\forall a \in A \,.\, \mathrm{dom}(\tau(a)) = \mathscr{R}(a) \cap \mathscr{R}_{temp}$$

or, put differently, $\tau(a) \cup \gamma(\tilde{a})$ needs to be completely connected in the classical sense for the configuration $\tilde{C}$ with $\tilde{C}(a) = \tilde{a}$ the reconfiguration commences from. Of course, we require $\tau(a)$ to be well-connected for a. Such a reconfiguration plan $\Delta^\tau = (A, R, \alpha, \rho, \delta, \tau)$ is called an *extended reconfiguration plan*. Shallow extended reconfiguration plans are defined similarly.

These interfaces of the range of $\tau(a)$ are intended for reconfiguration only (although it is technically not forbidden to connect to them from other components using normal roles), hence we require that

$$\forall a \in A \,.\, \forall r \in \mathscr{R}(a) \,.\, r \in \mathscr{R}_{temp} \rightarrow \forall m \in I_R(a)(r) \,.\, \mathit{calls}(\zeta(a)(m)) = \varnothing.$$

This requirement state that all methods of reconfiguration interfaces do not do communication on their own. Additionally, we require all these methods to be queries.

We expect each component to provide a *reconfiguration service interface* I_{Rec} which consists of the single method m_{copy}. We require that only temporary edges are used for communication within this method:

$$\forall a \in A \,.\, \mathit{calls}(\zeta(a)(m_{copy})) \subseteq \mathscr{R}(a) \cap \mathscr{R}_{temp}$$

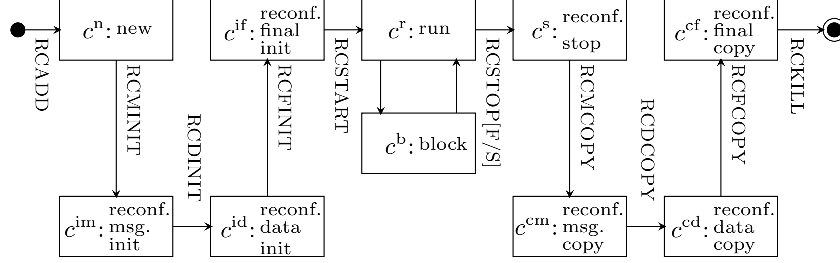


Figure 6.10: Extended component state machine

$$c^{\mathrm{n}}, \gamma \rightarrow c^{\mathrm{im}}, \gamma \quad \text{if } \gamma \text{ is completely connected} \tag{RCMINIT}$$

$$c^{\mathrm{im}}, \text{success} \rightarrow c^{\mathrm{id}}, \zeta(c)(m_{copy}) \tag{RCDINIT}$$

$$c^{\mathrm{id}}, \text{success} \rightarrow c^{\mathrm{if}}, \text{success} \tag{RCFINIT}$$

$$c^{\mathrm{if}}, \gamma \rightarrow c^{\mathrm{r}}, \gamma \quad \text{if } \mathrm{dom}(\gamma) \subseteq \mathscr{R}_{perm} \tag{RCSTART}$$

$$c_1^{\mathrm{s}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \rightarrow c_1^{\mathrm{cm}}, \gamma_1 \parallel c_2^{f_2}, \gamma_2 \parallel \ldots \parallel c_n^{f_n}, \gamma_n \quad \text{if } \forall 2 \leq i \leq n \,.\, c_1 \in \mathrm{ran}(\gamma_i) \rightarrow f_i \in \{\mathrm{s}, \mathrm{c}\} \tag{RCMCOPY}$$

$$c^{\mathrm{cm}}, \pi \rightarrow c^{\mathrm{cd}}, \langle\rangle \tag{RCDCOPY}$$

$$c^{\mathrm{cd}}, \langle\rangle \rightarrow c^{\mathrm{cf}}, \langle\rangle \tag{RCFCOPY}$$

$$\text{RCKILL}\tilde{C} \parallel c^{\mathrm{cf}} \rightarrow \tilde{C}$$

$$c_1^{\mathrm{n}}, \gamma \parallel c_2^{\mathrm{s}} \rightarrow c_1^{\mathrm{n}}, \gamma[r \mapsto c_2] \quad \text{if } r \in \mathscr{R}_{temp} \text{ and } \gamma[r \mapsto c_2] \text{ is well-connected for } c_1 \tag{RCWIRETMP}$$

$$c_1^{\mathrm{im}}, \pi_1 \parallel c_2^{\mathrm{cm}}, \pi_2 \parallel c_3 \rightarrow c_1^{\mathrm{im}}, \pi_2|_{\varphi} :: \pi_1 \parallel c_2^{\mathrm{cm}}, \pi_2|_{\neg\varphi} \parallel c_3 \quad \text{for } r \in \mathscr{R}(c_3) \text{ and } \varphi \equiv \{c_3.r.m(v) \mid m \in \mathscr{M}, v \in \mathscr{V}\} \tag{RCGETMSG}$$

$$c_1^{\mathrm{if}}, \gamma[r \mapsto c_2] \parallel c_2^{\mathrm{cf}} \rightarrow c_1^{\mathrm{if}}, \gamma \parallel c_2^{\mathrm{cf}} \tag{RCUNWIRE}$$

Figure 6.11: Rules for indirect state transferal

Finally, we disallow any calls over temporary edges by requiring that

$$\forall c \in C \,.\, \forall m \in \bigcup_{I \in I_P(c)} I \,.\, m \neq m_{copy} \rightarrow \mathit{calls}(\zeta(c)(m)) \subseteq \mathscr{R}_{perm}.$$

Given an *extended reconfiguration plan* $\Delta^i = (A, R, \alpha, \rho, \delta, \tau)$, reconfiguration commences with the same idea as described. However, four new states are added to the component state machine, as shown in Fig. 6.10. The “copy” state of Fig. 6.5 is split into a “message copy”, a “data copy” and a “finalize copy” state – this is required so that the message queue can be utilized for handling the reconfiguration subprotocol, and that the temporary connections can be removed again. To reflect this, the “init” state is also split into a “message init”, “data init” and a “finalize init” state.

Fig. 6.11 shows the reconfiguration rules that are required for the indirect state transferal reconfiguration, in addition to the rules RCADD, RCWIRE, RCSTOPS, RCSTOPF and RCREWIRE. Furthermore, the rules DEQ, RET[1] and CHOOSE need to be copied to work for components in state cd (by requirement, the rules for sending messages and changing the state are not required), and the rules CALL, SEND, SET and CHOOSE need to be copied to work for components in state id.

For the reconfiguration plan implementation of an indirect state-transfer reconfiguration plan $\Delta = (A, R, \alpha, \rho, \delta, \tau)$, we redefine step (3) and (4) such that RCMINIT and RCMCOPY are used. We add a step (3.1) after step 3:

3.1 for each $a \in A$ and each $r \in \mathrm{dom}(\tau(a))$, we use RCWIRETMP$(c_1 : a, c_2 : \tau(a)(r), r : r)$, thus establishing the temporary connections.

We then replace step (5) by the following sequence:

5.1 for each $a \in A$, we use RCDINIT(a), and for each $r \in R$, we use RCDCOPY(r). This ensures that the message queues of the old components are empty, and that the process term for the m_{copy} message is loaded into the components in A.
5.2 both the components in A and R perform a subprotocol phase that allows regular processing of component process terms.
5.3 once all components in A and R have reached a component process term of success, for each $a \in A$ we use RCFINIT(a) and for each $r \in R$ we use RCFCOPY(r).
5.4 for each $a \in A$ and each $r \in \mathrm{dom}(\tau(a))$, we use RCUNWIRE$(c_1 : a, c_2 : \tau(c_1)(r), r : r)$.

Obviously, this kind of state transferal protocol is quite limited, because it only admits one single method for a new component; this method must be written in a way that is aware of the situation it is about to be deployed in. If it is to replace two components, it has to provide two temporary edges, and the method m_{copy} has to be interpreted in a way that both edges are used for querying those two components' states. However, quite often, this is perfectly sufficient, especially if a component just replaces some other component; a scenario that can be expected to be not all too uncommon.

Especially, the Memento pattern is covered by this kind of technique: In the execution of the m_{copy} method, the Memento objects are obtained over the temporary edges, and then used to form the new component's state. This corresponds to a ς function that can written as

$$\{(s, a) \mapsto t(s(c)) \mid s \in (R \to \mathcal{S}) \wedge a \in A \wedge c \in R \wedge t : \mathcal{S} \to \mathcal{S}\}$$

(i.e., each new component's state is calculated from the state of a single component from R by a translation function $t : \mathcal{S} \to \mathcal{S}$). We can implement this ς by using a Memento interface $I_{Memento} = \{m_{Memento}\}$, and extend each component c with an $r_{Memento} \in \mathscr{R}(c) \cap \mathscr{R}_{temp}$ such that $I_R(c)(r_{Memento}) = I_{Memento}$. In the τ plan element, each new component is then connected to the one it replaces.

For building the Memento, we assume an injective function $w_{Memento} : \mathcal{S} \to \mathcal{V}$. We then set

$$\begin{aligned} \zeta(r)(m_{Memento}) &\equiv \mathrm{choose}((\{\sigma\}, \mathrm{return}(w_{Memento}(\sigma)))_{\sigma \in \mathcal{S}}.\mathrm{success}) \\ \zeta(a)(m_{copy}) &\equiv \mathrm{call}(r_{Memento}, m_{Memento}, *).\mathrm{choose}((R_v, S_v)_{v \in \mathcal{V}}) \\ &\quad \text{with } R_v \equiv \{\mathit{ret}(v, \sigma) \mid \sigma \in \mathcal{S}\} \\ &\quad \text{and } S_v \equiv \mathrm{set}(t(w^{-1}_{Memento}(v))).\mathrm{success}. \end{aligned}$$

[1]Note that this is the sole place where we actually require synchronous methods; if they were not allowed, bidirectional temporary edges would be required.

Obviously, this approach cannot cover a situation where the state of a new component is comprised of more than one old component. Furthermore, the state translation function t is "hard-wired" into the component. This is especially troublesome if reconfiguration is to be planned after the component has been written – as the separation of roles paradigm would require. That problem is inherent to the indirect state transferal approach, however: Since components are tasked with building their initial state (from a Memento, or by repeated queries), their applicability in reconfiguration is restricted to those scenarios that have been foreseen at the time the component was built.

6.8.2. Hybrid State Transferal. As mentioned previously, the problem of the indirect state transferal outlined in the previous section is the breach of the separation of roles and requirement R3.3. The component implementer needs to declare the temporary roles that get connected during reconfiguration, and the method m_{copy} needs to be implemented. While the latter might make use of an elaborated Memento pattern implementation (maybe using a Memento object that carries the semantics of its contents along, such that a wide range of possible source components can be accounted for), the definition of temporary roles needs to make some assumptions about the source of the state data. It is especially problematic to condense the state of multiple components into a single component.

As discussed before, the direct approach requires a breach of encapsulation: In order to populate a component's state with data copied from another component, it needs to perform write operations on data that should remain hidden[2], unless an exhaustive set of writing operations is provided by some interface. The existence of such writing operations is unlikely (unless they are enforced, again leading to problems with a separation of roles), hence the state transferal mechanism (e.g., a plan-provided script executed by the assembly) needs to inject data directly into the component. But this requires knowledge (even if obtained at runtime by reflection) that should not be passed to the system designer, but remain a concern of the component implementer only.

Since component data encapsulation is virtually an axiom in this thesis, we opt to stick with the indirect approach and deal with the breach of separation of roles paradigm, i.e., the problem that the reconfiguration scenarios need to be considered at component creation time. A mitigation of this problem is to postpone the time the new component is built as far as possible. Instead of directly adding a component that consumes the states of the old components and then continues to operate, we can use a two-stage reconfiguration and generate an intermediate component for just a single reconfiguration plan.

Since the reconfiguration plan is fully known at the time the intermediate component is added to the system, this intermediate component can be devised to require exactly the temporary roles necessitated to implement ς, and doing so will be in accordance with requirement R3.3. This component can then query the old components, and assemble a Memento object that carries the combined state. Any necessary state transformation can be done in this stage. Once this is completed, a second reconfiguration can be conducted to copy the Memento object to the final, new component. Hence, all regular components just need to provide facilities for the Memento pattern (which is still a departure from the direct approach) but the remainder is handled by the intermediate component.

---

[2]Note that requirement R1.4 calls for the ability of the component to do exactly this. For message retainment, we fully support this requirement. For the data state, which is much less constrained and mostly defined by the component implementer, the component model needs to favor other concerns.

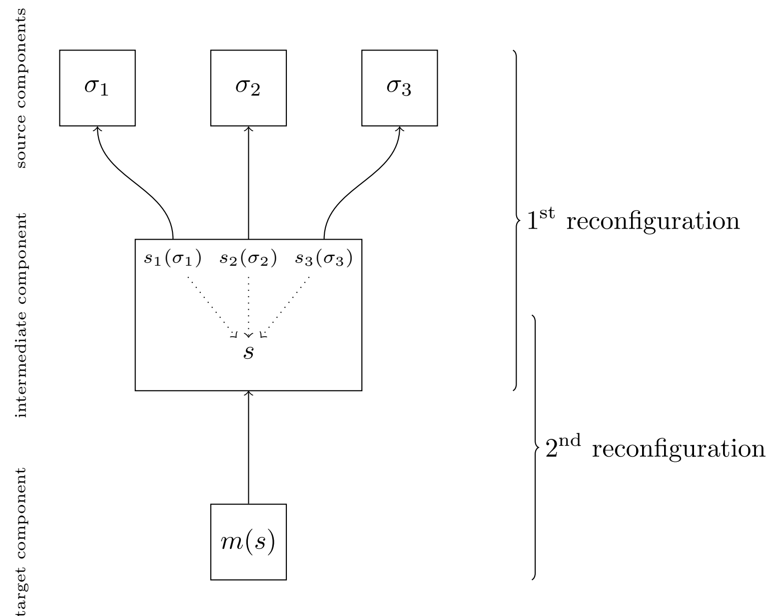


Figure 6.12: Hybrid state transfer

Fig. 6.12 illustrates the hybrid approach. All arrows between components are temporary edges; in the first reconfiguration, a copy (not necessarily an identical image, hence the s_i functions) of the state of each component is queried by the temporary component. Obviously, the interface of the temporary edges needs to provide accessor methods for the relevant parts of the source components' state, but, as argued before, their existence can be assumed. From these state copies, a combined state (which can be regarded as the Memento object already) is built and stored. During the second reconfiguration, this Memento object is copied to the target component. Some wrapping, indicated by the m function, is required.

The utilization of the Memento pattern in this way preserves the separation of roles issue: The implementer of the target component can choose freely how this Memento should look like; it is hence entirely dependent on the target component and thus entirely within the concern of the component implementer. Only the fact that a Memento object has to become consumed hints at the possibility of future reconfiguration. The reconfiguration designer then needs to ensure that the data of the old components is transformed in a way that can be used to populate the Memento object.

Using the Memento pattern helps to keep the reconfiguration lean. Implementing the Memento consumption is not different from implementing state setter methods, but this would require a completely different reconfiguration algorithm that requires the execution of the state transfer protocol (i.e., the retrieving of the state from the source components, any necessary conversion and the depositing of the result in the target component) within the assembly. Using the hybrid approach, all can be handled by the components using only the indirect approach as described before. This helps in keeping things flexible – the means to describe the generation of the Memento object can be chosen by the intermediate component and are not fixed by the assembly's implementation.

Obviously, the generation of the intermediate component requires the ς function to be given in a way that enables automated code generation. This could be provided by a *domain specific language* (DSL). In the JComp framework implementation, we utilize JavaScript to describe the relevant parts of the ς function. An example is given in Sect. 8.2.2.

6.8.3. Direct State Transferal. Direct state transferal without violating important paradigms is not entirely impossible if ς has a limited scope. In Sect. 7.2.3, we have discussed the use of *component parameters*. The important property of this parameter is that they remain constant during the lifetime of a component, and that they can be assumed to be visible (as they are part of the component definition rather than its internal data state). This aspect of the component's state can therefore be accessed by the plan creator, and directly injected into the new component.

The parameters are not made first-class entities in our formal model, leaving two ways to include them: Either they become a part of the component identifier, indicating their static nature; or they are made part of the component state, to be set by ι initially. This integrates them with choose, but, obviously, also subjects them to change by set. For the purpose of direct state transfer, the second option is more favorable; by using a restricted ς plan element, the parameter state part of new components can be set directly. An example will be given in Sect. 8.3.3; there, component parameter are used to parametrize components with responsibilities'; during reconfiguration, direct state transferal is used to retain these information.

CHAPTER 7

Implementing the JComp Model in a Component Framework

Ja, mach nur einen Plan
Sei nur ein großes Licht!
Und mach dann noch 'nen zweiten Plan
Geh' n tun sie beide nicht.

— Bertolt Brecht

In this chapter, we describe how the JComp component model, introduced in the previous chapter, is implemented in the JComp framework. The JComp framework is written in Java, and also uses Java as the host language for the components. Java is particularly suitable for our purpose, as it is restricted enough to be controllable (e.g., no runtime changes are allowed to the component implementations), yet quite versatile and capable of elaborate reflection. In our implementation, we try to realize the formal JComp component model as closely as possible, in order to retain the properties we have proven for its reconfiguration mechanism. For the basic component framework, two major issues need to be solved: the realization of the message-based communication paradigm, and the prohibition of sharing of memory. The inclusion of reconfiguration capabilities is then rather easy to achieve, but extended features are required for detecting the need for reconfiguration by monitoring the communication and occurrence of errors.

7.1. The Active Object Pattern

The *Active Object pattern* [LS96] describes how message passing can be achieved by common objects in a language with multi-threading support (such as Java). Basically, the pattern calls for adding a queue to the active object, which conducts a thread switch before the method execution; the caller enqueues a *method request* instead of directly invoking the method.

Fig. 7.1 illustrates the dynamical progression of the Active Object pattern. A *client* wants to execute a method that is ultimately implemented by a *servant*, but only visible to the client via a *proxy*. This proxy builds a *method request* object that gets enqueued in a queue. If a return value is required, a *future* is returned to the client. All this is done by the client's thread. The remainder of the processing now happens in a special thread that belongs to a *scheduler*. This scheduler is tasked with selecting messages to execute on the *servant* in such a manner that some abstract criteria (e.g., deadlock-free executability) is met. To this end, the scheduler calls the method request object with a method `guard`, which checks the executability. Once this executability is given, the method request object is removed from the queue and dispatched. This is done by invoking `call` on the method request object, which in turn invokes the requested method on the servant using a Double-Dispatch pattern. Finally, the return value is propagated to the future.

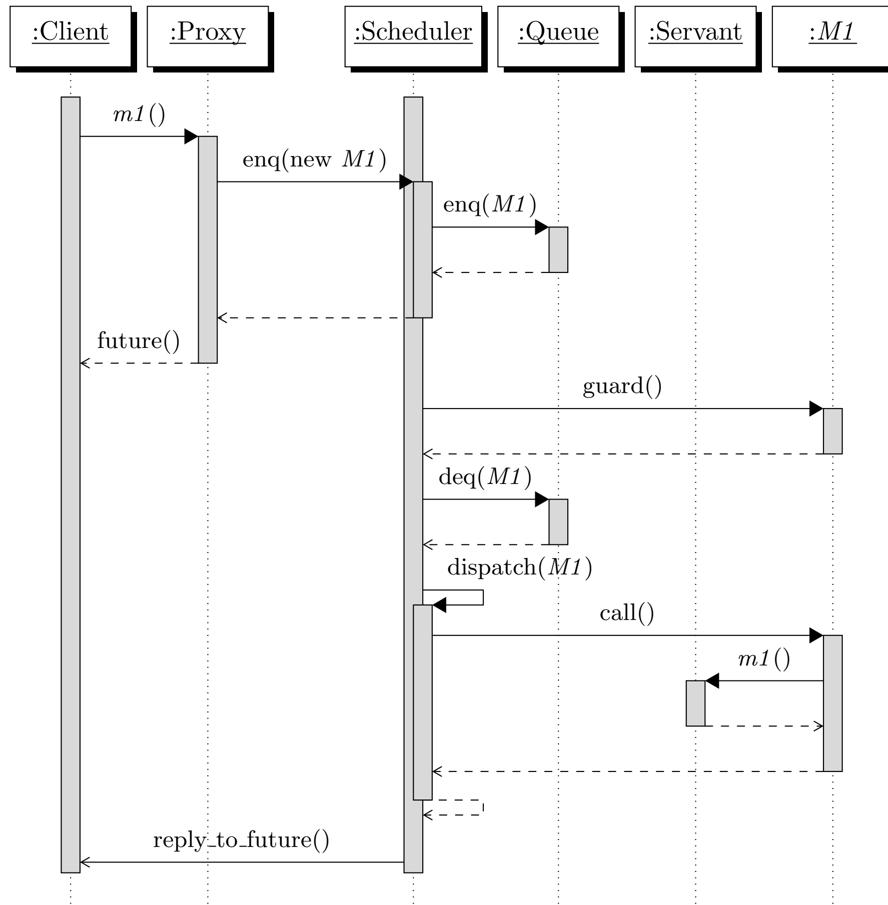


Figure 7.1: Active Object pattern, as described in [LS96]

Basically, the Active Object pattern implements a producer-consumer scheme for method processing: The proxy produces method request objects, which are consumed by the scheduler. The pattern refines this idea by detailing how the method request object is to be consumed. Like any producer-consumer example, a buffer is required to transport jobs between the threads; here this buffer is implemented by the queue, although the queue designation is a bit misleading since the Active Object pattern allows for arbitrary choice of the method request to be consumed next. This queue is the only data structure that might be accessed by both threads simultaneously; hence its operations need to be atomic.

A different perspective on the Active Object pattern is given by the observation that it provides message passing between two threads. This view is closer to the intent of the pattern: It makes an object *active*, i.e., has it running in its own thread. By using the queue for communicating the method request to the recipient's own thread, message passing supporting asynchronous calls is achieved.

7.1.1. Using the Active Object pattern for JComp. The Active Object pattern describes the basic approach towards implementing message passing in JComp, as it is done in the Julia reference implementation of the Fractal component model [BCL+06]. The actual use of the pattern in the JComp framework does not exploit all its power, however. Especially the role of the scheduler, with its

capability to change the message processing order, is not used in JComp. Instead, message execution order is the same as the order of message reception.

Fig. 7.2 illustrates the utilization of the Active Object pattern in the JComp framework to support two ways of doing communication: Asynchronous calls, which are not providing return values (we evaluated the use of futures, but have not found a good example [**KD99**]), and synchronous calls, which block the caller until the return value is obtained. The role of the scheduler and the servant is now merged in the *component* implementation, which both maintains the queue and finally executes the methods. For synchronous calls, blocking is handled by the proxy. The actual implementation of these ingredients of the Active Object pattern will be discussed in the next section.

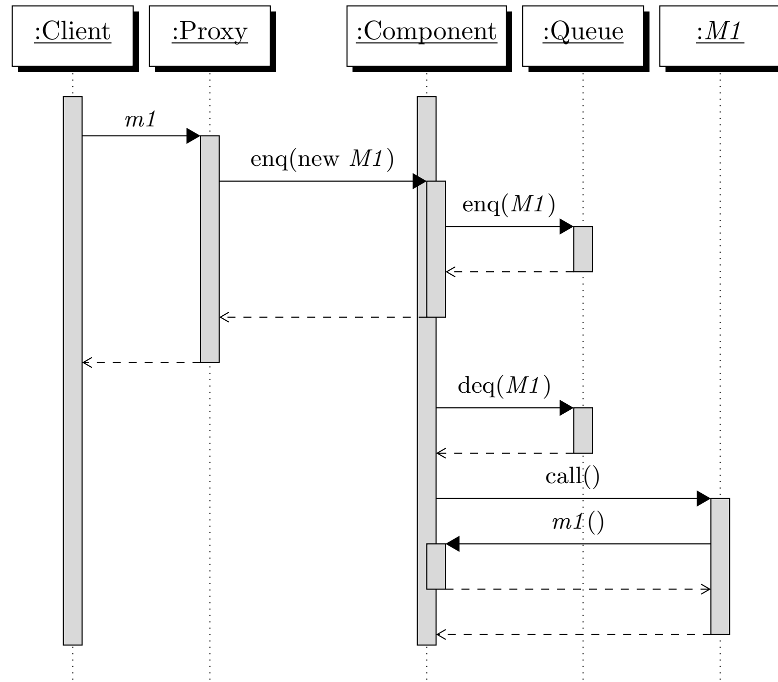
7.2. Implementation

The JComp framework is a fairly slim framework (since the model focuses on only a few features) that is written in the Java programming language, which also serves as the component host language. In using Java, some very benign features are obtained, like dynamic class loading and multi-threading. These are sufficient to allow for wrapping component implementations in a lightweight layer that makes them a component, following the Active Object design pattern as discussed above. On the other hand, Java allows shared memory by using references for objects. Since communication should be able to carry objects as parameters, a conflict arises: By passing references to objects, a covert channel is built – one component might communicate with some other component using the shared memory, and thus bypass any observer. We have discussed in Sect. 6.2 that sharing memory between components needs to be impeded, and subsequently only included a component-local state in the JComp component model. If the component framework is to represent the component model, we have to avoid memory becoming shared by passing references between components.

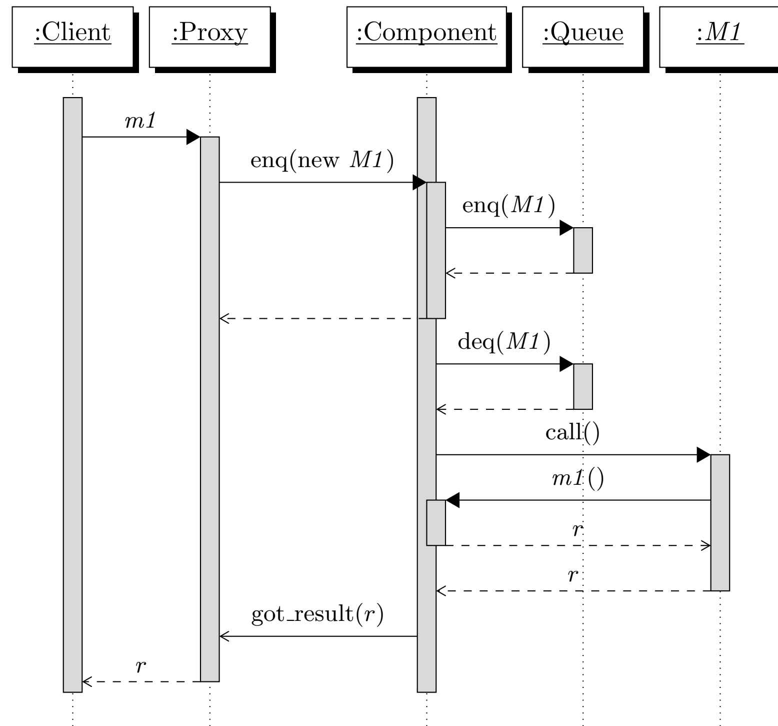
7.2.1. Suppressing Shared Memory. The JComp framework solves this problem by mandatory serialization of parameters. Technically, all parameter classes need to implement the `java.io.Serializable` interface, since the Java serialization mechanism is used for obtaining deep copies (i.e., a complete copy without shared parts).

This slows the communication down, but by using static bytecode analysis, the mandatory copying can be suspended for immutable objects (like `java.lang.String`) that cannot be modified after their initialization. Fig. 7.3 shows the throughput obtained with different communication means. Two components c_1 and c_2 call each other i times, using asynchronous calls (synchronous calls would deadlock immediately). As a reference, a direct Java implementation is given (though, for large initial i, the stack overflows). Message passing, which requires building a message request and various calls to listeners described in the next sections, is much slower, of course. Using a primitive `Integer` variable to represent i is fastest, but using an immutable object that creates a new copy to store the result of the operation $i - 1$ is comparable. Using such an object is more than three times faster than using a generic parameter copying approach, where the parameter is wrapped in an object and $i - 1$ is implemented as a modification to that object's state.

Still, there is ample room for improvement (we have done some experimentation with static bytecode analysis for detecting immutable objects, which do not have to be copied), and obtaining high efficiency is not a primary concern of the JComp model and framework, but providing a clean communication paradigm implementation is; ultimately for obtaining guarantees for reconfiguration. A much

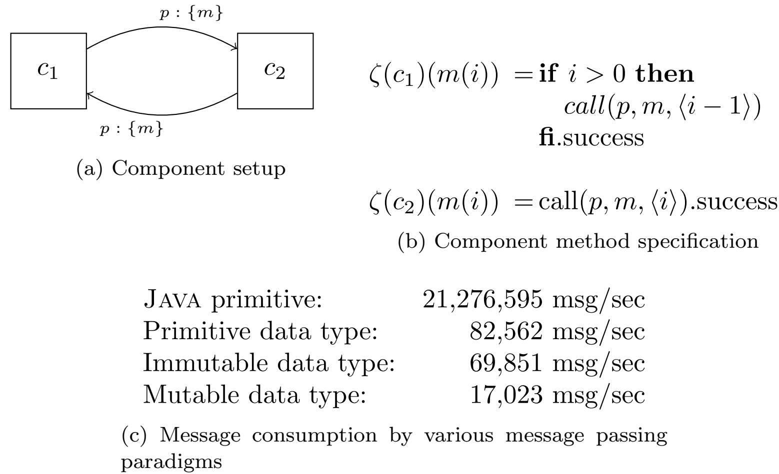


(a) Asynchronous call



(b) Synchronous call

Figure 7.2: Active Object pattern, as realized in JComp



(a) Component setup

(b) Component method specification

Java primitive:	21,276,595 msg/sec
Primitive data type:	82,562 msg/sec
Immutable data type:	69,851 msg/sec
Mutable data type:	17,023 msg/sec

(c) Message consumption by various message passing paradigms

Figure 7.3: Message passing cost

bigger problem is given by the fact that it is downright impossible to avoid covert channels with Java. First of all, `static` variables can always be used to transport data between components (if they are on the same virtual machine). Other than that, shared resources like files might breach the explicit communication paradigm.

The latter can be mended by utilizing the Java security model and disallowing access to all problematic resources. Prohibiting access to static variables, on the other hand, would require bytecode analysis, which is prone to disallowing legitimate behavior if the analysis is not exact enough – e.g., a read access of `Math.PI` should not be prohibited, but a read access to `Singleton.INSTANCE` should be prevented if – and, preferably, only if – the class `Singleton` offers mutator methods. It is quite unclear if read access to `System.out` can be allowed. The JComp framework does not prohibit static variable access; instead, static variables act as a substitute for what is known as *Utility Interfaces* in SOFA [BHP07]; we will see some examples in Sect. 7.3.4 and Sect. 8.3.3. For a complete Java-based component framework that enforces explicit communication, however, static variable access needs to be accounted for.

7.2.2. Role and Communication Implementation. Implementing the communication is greatly facilitated by the fact that Java uses interfaces in quite the same fashion as we use them for roles: As sets of method declarations. Hence, it is straightforward to have the Java implementation of a component be a class that implements the provided interfaces and has attributes that represent the required roles.

Component interfaces are implemented by their Java counterparts. All necessary information is included, except the message synchronicity. We therefore provide two annotations `@AsynchronousCall` and `@SynchronousCall` that are used to annotate a method's call type. To allow for consistent interpretation of method calls, some constraints need to be obeyed:

- If a method is declared to by asynchronous, the return type must be `void` and no exceptions can be declared,

- all parameter types and, for synchronous non-`void` methods, the return types must either be primitive types or implement `java.io.Serializable`.

A component is implemented in a JAVA class that extends `jcomp.core.AbstractComponent`. It has to implement the interfaces it provides and implement their methods. Only two elements within such a component class need annotation: The roles and the component parameters. Roles are declared by adding `@RequiredInterface` annotations to attributes. The annotation can be provided with a role name, but if this is omitted, the name of the JAVA attribute is used. Furthermore, it is not required to provide a setter method for the attribute or grant a visibility beyond `private` – the JCOMP framework utilizes a security manager that allows the direct setting of the attribute. Avoiding redundant information (e.g., maintaining a role name both in the annotation and the JAVA attribute) and requiring as little as possible from the component implementation code helps to keep the component applications slim and easy to understand.

Roles can also be given a scope, which can be either permanent or temporary, with permanent being the default. Temporary roles are used for transferring data during reconfiguration as described in Sect. 6.8.1. Given an annotated attribute that is declared to be a permanent role, the JCOMP framework needs to provide a link to a target component as specified by the component setup. This is done at component instantiation time; a proxy is used to translate the JAVA method invocation to the message passing as prescribed by the JCOMP communication model. In JCOMP, this proxy is instantiated from code that is generated for the required interface using the VELOCITY engine [**Apa08**][1].

Once invoked by having the component implementation call a method on the role's attribute, the proxy builds a method request object that stores the method call name, deep copies of the parameter objects, and information required for calculating statistics and performing reconfiguration. This method request object is now passed to the target component by invoking a method defined in `jcomp.core.AbstractComponent` which enqueues this method request in the queue of the target component. Eventually, this method object will be dequeued and executed, by calling a method `perform` that invokes the appropriate method on the target component in a scheme similar to double dispatch.

If the method call is synchronous, the method request waits for completion of the method and stores the result – if any; the result might either be a value or an exception. The proxy object, itself waiting for the target method's completion, maintains a JAVA-provided monitor (obtained by invoking `Object.wait()`) on the method call object, and is notified on the return value's setting. This communicates the method call completion back to the caller component.

7.2.3. Defining and Launching Component Setups. Building a component setup with the JCOMP framework is basically done in three steps:

(1) instantiation of a `jcomp.assembly.Assembly` instance,
(2) declaring the component graph by calling methods of the `Assembly` instance,
(3) establishing the components and launching them by calling `start()` on the `Assembly` instance.

[1]Note that JAVA 1.6 provides the `java.lang.reflect.Proxy` class that facilitates the same approach with minimal overhead, but this was not known to the author, who was very influenced by code generation done by some well-known model checking software at that time.

`getComponentForClass(String id, Class c)`	build a component descriptor for a user-supplied class deriving `AbstractComponent` and a unique, user-supplied ID
`getComponentForClass(Class c)`	build a component descriptor for a user-supplied class deriving `AbstractComponent`, with a generated ID
`addComponent(ComponentDescriptor c, ComponentParameter[] p)`	add a component to the network, with user-supplied parameter settings
`addComponent(ComponentDescriptor c)`	add a component described by a component descriptor to the network, with an empty parameters set
`linkComponents(ComponentDescriptor src, ComponentDescriptor tgt, String role)`	link the role of component `src` to component `tgt`

Table 7.1: Component network creation commands in JComp, provided by the class `jcomp.assembly.Assembly`

The `Assembly` instance maintains a graph representation of the component setup. The basic methods provided for establishing a component network are shown in Tab. 7.1. The reference to components is given by `ComponentDescriptor` instances, and they can be obtained from the assembly by calls to `getComponentForClass`. Since these descriptors will also be required for planning reconfiguration, they need to be added to the component setup by a distinct call to `addComponent`. This call also takes an array of component parameter which are essentially name-value-pairs and which are to be injected into the component before it is launched. Parameters are useful to obtain a higher diversity of components [OLKM00], in the JComp component model, they can be represented by a read-only part of the data state that is set by ι. Calls to `linkComponents` are used to establish the connections of roles to other components. The assembly also provides methods for adding the various listeners described in Sect. 7.2.5, for obtaining the component graph for inspection, and for preparing and executing reconfigurations.

Once the setup is complete, the entire system is started by a call to `Assembly.start()`. First, the assembly checks the well- and completely-connectedness of the component setup, and checks the various implementation-related constraints on the components and interfaces (e.g., no asynchronous method must have a return type other than `void` declared). The assembly then builds the required code for the communication proxies and instantiates the actual components, injecting the communication endpoints and finally starting each component's own thread. Finally, all components providing the `MainInterface`, which consists of a single asynchronous method `start()`, get a `start` method object added to their queue. Those components are called *initial components*; their processing of the `start` method launches the component application.

7.2.4. An Example. We present a small producer-consumer example to illustrate how a component setup is realized in the JComp framework. The component setup is illustrated in Fig. 7.4. It consists of two components: *cc* and *sc* – a "client" component (that produces tasks) and a "store" component (that stores data). For

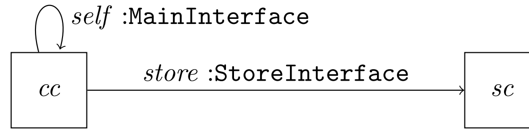


Figure 7.4: Component setup for the producer/consumer example

defining this example, we require two interfaces: the `MainInterface` which we introduced in the last section, and an interface `StoreInterface` that contains a single asynchronous method `store(String s)`.

We can thus define the components

$$C = (cc, \{\texttt{MainInterface}\}, \{\texttt{self} \mapsto \texttt{MainInterface}, \\ \texttt{store} \mapsto \texttt{StoreInterface}\}, \mu(C), \{\texttt{i} \mapsto 0\}) \qquad \text{and}$$

$$S = (sc, \{\texttt{StoreInterface}\}, \{\}, \mu(S), \{\texttt{entries} \mapsto \langle\rangle\}).$$

The method evaluators are defined as

$$\mu(C)(\texttt{start}) = (\texttt{i} \leftarrow \texttt{i} + 1).\text{call}(\texttt{store}, \texttt{store}, \langle\text{"}\texttt{i}\text{"}\rangle). \\ \text{call}(\texttt{self}, \texttt{main}, \langle\rangle).\text{success} \qquad \text{and}$$

$$\mu(S)(\texttt{store(s)}) = (\texttt{entries} \leftarrow \texttt{entries} :: \texttt{s}).\text{success}$$

The component setup is formally defined as

$$(\{C, S\}, \{C \mapsto \{\texttt{self} \mapsto C, \texttt{store} \mapsto S\}, S \mapsto \varnothing\}, (C, \texttt{start})).$$

Implementing these elements in JAVA is straightforward. First, we implement the `StoreInterface` interface, which is a regular JAVA interface with annotations used to describe how communication is conducted:

```
1  public interface StoreInterface {
2      @AsynchronousCall
3      public void store(String s);
4  }
```

Writing the components is equally straightforward:

```
1  public class ClientComponent extends AbstractComponent
2                               implements MainInterface {
3      @RequiredInterface
4      private StoreInterface store;
5      @RequiredInterface
6      private MainInterface self;
7
8      private int i;
9
10     @Override
11     public void start() {
12         i++;
13         store.store(Integer.toString(i));
14         self.start();
15     }
16 }
```

```
1  public class StoreComponent extends AbstractComponent
2                              implements StoreInterface {
3      private LinkedList<String> entries = new LinkedList<String>();
4
5      public void store(String s) {
```

```
6         entries.add(s);
7     }
8 }
```

Note that these two components are actually component types that need to be instantiated; as we discussed in Sect. 2.3, a distinction is usually omitted.

For building the component setup and starting the application, a regular Java `main` method is used. The component setup is built by invoking the methods presented in Tab. 7.1 (or, rather, shortcuts that omit unnecessary parameters) as described in Sect. 7.2.3:

```
1 public static void main(String[] args) {
2     a = new Assembly();
3     cc = a.addComponent(a.getComponentForClass(ClientComponent.class));
4     sc = a.addComponent(a.getComponentForClass(StoreComponent.class));
5     a.linkComponents(cc, sc, "store");
6     a.linkComponents(cc, cc, "self");
7     a.start();
8 }
```

`cc` and `sc` are static variables; we will later use them for building a reconfiguration plan. The JComp framework implementation supports the loading of component setups from graph files, but we usually build component setups just as shown, by directly calling the appropriate methods on the assembly.

7.2.5. Monitoring Facilities. The JComp framework needs to support a variety of monitoring facilities in order to allow for the triggering of reconfiguration, as we will discuss in Sect. 8.1.2. While monitoring can be added to most systems externally [**Sca00**] or at component code level by introducing filter components (cf. Sect. 8.1.2.1), having direct provisions in the framework facilitates the implementation of lightweight monitors and thus reduces the impact monitoring has on the concept and the runtime load of a component application.

7.2.5.1. *Communication Monitoring.* The most important kind of monitoring is about inter-component communication. An observable communication (for an outside observer) is one of the distinguishing properties of a component framework, hence monitoring capabilities are an integral part.

For message monitoring, monitors can subscribe to the assembly using the Observer pattern [**GHJV95**]. They are then informed on different stages of message processing, listed in Tab. 7.2. Such a fine-grained distinction of points in time is required, as reconfiguration needs to be able to preempt message processing (see Sect. 8.1.2). It also gives rise to a number of different "combined" monitoring approaches, including non-functional evaluations like the number of messages that are stored in the queue (the difference of stage 2 and stage 3 events) or the time of a method execution (time elapsed between a stage 3 and the corresponding stage 4 event).

Apart from the single information that a message stage has been (or will be) conducted, the JComp implementation supports the retrieval of further information on the messages sent by utilizing the JVMTI interface [**PRRL04**]. JVMTI is a C API for querying the Java virtual machine about various data, in this case about the size of a message parameter. This allows for an (almost) efficient calculation of message sizes, which consist of the message header and the sizes of the message parameters objects. Such information is useful for determining "heavy use" connections, which should not be divided during a distribution of the application, or possible problems with a component division too fine-grained. We will utilize such JVMTI-based measurements in Sect. 7.3.3.2.

Stage	**Description**	**Thread**
1 – message send	after the message was sent by the target component. The message object has been built, but not yet enqueued. This stage is a relict of times where JComp was intended to provide distribution on framework-level.	source component
2 – message received	after enqueueing the message in the target component's queue. At this point in time, it is actively available to the target component.	source component
3 – start processing	after dequeueing the message from the target component's queue. This is just prior to invoking the message on the target component. The time difference to stage 2 is the queue residence time.	target component
4 – end processing	after the method execution completed on the target component. The time difference to stage 3 is the actual method execution time, the difference to stage 2 the time the invoker of a synchronous call had to wait.	target component
5 – message return	just prior to returning to the source component in a synchronous call. Omitted for asynchronous calls. This stage should occur just after stage 4 with only negligible delay, but the thread switch is required.	source component
6 – message postponed	if the message is postponed (see section 7.5.1), this is called prior to reenqueueing the message.	target component

Table 7.2: Observable stages of message processing in JComp

7.2.5.2. *Exception Monitoring.* The JComp framework also allows for monitoring of exceptions. This is very similar to communication monitoring, as an exception completes the execution of a method. However, in the JComp component framework, exceptions cannot be recovered from without external intervention, thus implementing the fail behavior of the JComp component model. Hence, monitoring exceptions is made distinct from monitoring communication, as the former is much more important for an application, and it might be reasonable to assume that every application should contain an exception monitor, whereas only very special applications need to have their communication monitored – additionally, monitoring communication is very expensive (as the five stages need to be processed for

every method call). Also, exception monitors are required to communicate a way to handle the exception, which decides how the application proceeds.

Technically, the component's thread is used to notify the exception listeners (again employing the Observer pattern). The monitors are required to return a value indicating their proposal on how to proceed:

- `DIE`, indicating that this particular monitor does not know of any way to handle this particular exception,
- `RECONFIGURE_AND_SKIP`, indicating that the monitor knows a way to handle the situation by reconfiguration, but cannot handle the faulty message (which is, actually, not covered by the rule RCSTOPF),
- `RECONFIGURE_AND_KEEP`, indicating that the monitor knows how to conduct a reconfiguration and that the problematic message should be prepended to the queue prior to the reconfiguration,
- `CONTINUE`, indicating that the monitor can vouch for the error being no problem at all; the system may continue unabated.

If multiple monitors are present, the most favorable suggestion is used. Depending on the type of method and the best suggestion made, the application proceeds in different ways which are detailed in Sect. 7.5.3. If none of the monitors suggested a reconfiguration or continuation of the application, and if the method whose execution caused the exception is synchronous and the exception is declared in the method's signature's `throws` part, the exception is then thrown at the calling component, which has to cope with the exception by means of JAVA's exception handling mechanism. If the exception has not been declared (i.e., is a `RuntimeException` or an `Error`), and if none of the exception listeners claimed the exception, the system is terminated, as the exception cannot be dealt with in a consistent way.

Extending the example of Sect. 7.2.4, we might add a check after line 4 of the component `StoreComponent` that reads

```
if (entries.size() > 100) throw new RuntimeException("Capacity exceeded");
```

Once such a line is added, the component application will soon fail. Adding an exception monitor is achieved by implementing the JAVA interface `jcomp.monitoring.ExceptionMonitor` and announcing it as a listener to the component S by adding the line

```
a.monitorExceptions(sc, new ExceptionMonitorImpl());
```

after `a.start();` in the `main` method. We will use such a monitor to trigger a reconfiguration to mend the "capacity problem" in Sect. 7.5.4.

7.2.5.3. *Component Graph Change Monitoring.* Finally, JCOMP supports monitoring changes to the component graph, which may happen due to two event classes: System startup and reconfiguration. This is not very prominently used, but it is helpful for system-wide logging and observation. We will discuss the benefits of hierarchical components in Sect. 10.1.3.2; this component graph change monitoring provision is a first step into that direction.

7.3. Distributing JCOMP

7.3.1. Abstract Idea. The JCOMP model was developed with distribution in mind, which proved to enforce exactly the level of component confinement required for reconfiguration. It was therefore rational to investigate how components might be distributed on a network of machines (called *nodes* in this thesis), which was investigated in [RS07b]. Distributing a component framework is done in two steps: First, a central node is asked to provide a partitioning for a given component graph, i.e., a labeling of the graph nodes which describes on which of the available network nodes they are to run. Then, these machines, with each operating a so-called

local assembly, instantiate the components. Components that happen to be located on the same node can be connected using the regular means, described in Sect. 7.2. For components residing on different nodes, a more elaborate communication mechanism needs to be employed: The component sends its message to an interface provider (just as the Active Object pattern proposes), which in turn has a socket connection to the remote node. On the remote node, a special thread is dispatched immediately to handle the message – as incoming synchronous calls that need to wait for method processing completion must not block the network interface of the node itself, as it might receive messages from multiple components. This thread – obviously, one per incoming connection is required – then enqueues the message in the target component's queue. If the method call is synchronous, it waits for method processing completion and sends a response message back to the caller's node.

7.3.2. A First Approach – Extending the Component Framework. In [**RS07b**], the JComp framework was extended to provide distribution and remote communication facilities in the framework. This resulted in two problems: First, it made the component framework quite bulky. The assembly needs to be split into a global and local part, and even for non-distributed applications, both have to be instantiated and linked. Resolving a component's reference requires a two-step lookup: First, the component descriptor that is made known outside of the assembly needs to be resolved to a *component locator*, which stores the node the component is actually running on, and second, the actual component has to be found on its host node.

But far more problematic is the problem of having extensibility of distribution severely limited due to having all the functionality fixed within the framework. Any change to the remote communication method (which is far more elaborate than local communication) entails a change to the framework – e.g., if the messages need to become encrypted, the framework has to be changed. And, worst of all, fixing the communication means in the framework deprives us from one of the finest examples for reconfiguration, which will be discussed in Sect. 8.4.

7.3.3. Another Approach – Distribution on the User-Level. There are two ways to remedy that situation: Communication, especially remote communication, might be modularized and hence be made subject to user-supplied changes. Such an approach is taken by Corba [**OMG06a**], where multiple object request brokers (ORB) can be provided that then can support different ways of doing communication (obviously, these communication ways have to adhere to the Corba standard defined in the "general inter-ORB protocol" (GIOP) (cf. [**KL00**]), but other decisions such as performance issues are left to the implementer, cf. [**CB01**]). We propose a similar approach in the REFLECT middle-ware [**SvdZH08**], where *user-defined connectors* [**LEW05, BMH08**] can be used to provide different ways of doing remote communication.

In this thesis, however, we take a different approach that places distribution on the user level. Basically, this amounts to having a number of component applications running on various nodes. Each application is self-contained; the component framework instances on the different nodes are unaware of the fact that only a fraction of the entire application is running under its supervision. Communication that needs to be sent to a remote node is handled by proxy components (in reference to the Proxy pattern [**GHJV95**]) that maintain a network connection and delegate the message requests by means of their choice. Basically, these proxy components are just what the framework would have provided as an interface provider, but they are now user-supplied and thus fully changeable without touching the component

framework itself. Obviously, the "user" is different from the user that writes the application that gets distributed; such a hierarchy of users is often found in this thesis and will be discussed further in Sect. 8.3.1.1.

7.3.3.1. *Phase 1: Setting up the Network.* For distributing an application, three phases are required: In an initial phase, all available nodes subscribe to a server. This could be done using some peer-to-peer software, but in the example implementation, a minimal socket-based implementation and a lookup using user-supplied IP addresses are used. Also, this initial network configuration might already utilize the code later used for doing inter-component communication, but such a bootstrapping process seemed to provide little gain [**KD99**]. Instead, a proprietary protocol is used to have some client send a component setup request to the server, which describes the layout of the entire application to be instantiated.

7.3.3.2. *Phase 2: Finding a Good Partitioning.* The server now decides on how the components should be assigned to the available nodes.

Formally, a *graph n-partitioning* for a graph $(V, E, \alpha, \omega, \lambda)$ over Σ is a function $V \rightarrow \{1, \dots, n\}$. Given such an n-partitioning function P, a *partitioned graph* $(V, E, \alpha, \omega, \lambda')$ can be defined, using the extended alphabet $\Sigma' = \Sigma \cup \{(\sigma, n) \mid \sigma \in \Sigma \wedge n \in \{1, \dots, n\}\}$, by defining λ' as

$$\lambda'(o) = \begin{cases} \lambda(o), & \text{if } o \in E \\ (\lambda(o), p), & \text{if } o \in V \text{ and } P(o) = p \end{cases}$$

Given $\lambda'(v) = (l, i)$, we write $l(v)$ for l and $p(v)$ for i .

Often, graphs are *weighted*, meaning that they have "costs" associated with nodes and edges. Formally, this is a function $\mathit{cost} : \Sigma \rightarrow \mathbb{R}^+$. The *edge cost* of a partitioned graph $G = (V, E, \alpha, \omega, \lambda')$ is given by the sum of the weight of edges crossing partitions:

$$\mathit{cost}_e(G) = \sum_{e \in E} \begin{cases} \mathit{cost}(\lambda(e)), & \text{if } p(\alpha(e)) \neq p(\omega(e)) \\ 0, & \text{otherwise.} \end{cases}$$

The cost of a partition i is the sum of the cost of its nodes:

$$\mathit{cost}_p(i) = \sum_{\{v \in V \mid p(v) = i\}} \mathit{cost}(l(v)).$$

The *balance* of a partition is the inverse of the difference between the cheapest and most expensive node (other means to calculate it, like the standard deviation, can also be used). A graph partitioning should both have a high balance as well as a low edge cost. For component applications, the edge cost is given by the amount of communication sent over this edge, and the node cost is given by the CPU time required for method processing by the associated component.

Finding such a good partitioning is a problem that is known as a "multi-constraint graph partitioning problem", and very good heuristics exist. We employed the METIS partitioning tool-set [**KK98**], and partitioned the component graph, whose weights were retrieved by communication monitoring and JVMTI-based profiling [**PRRL04**] as described in Sect. 7.2.5.

Fig. 7.5 illustrates such a weighted graph; edges are annotated with the megabytes of sent and returned messages (for synchronous calls) flowing over the edge, and their respective count. Components are annotated with the CPU time consumed, the number of messages processed, and the memory required (in bytes). The example is an image processing application, where two filters are applied to an image, which is then recombined with the original source in order to mix the filter effect. Since most of the communication is asynchronous, the

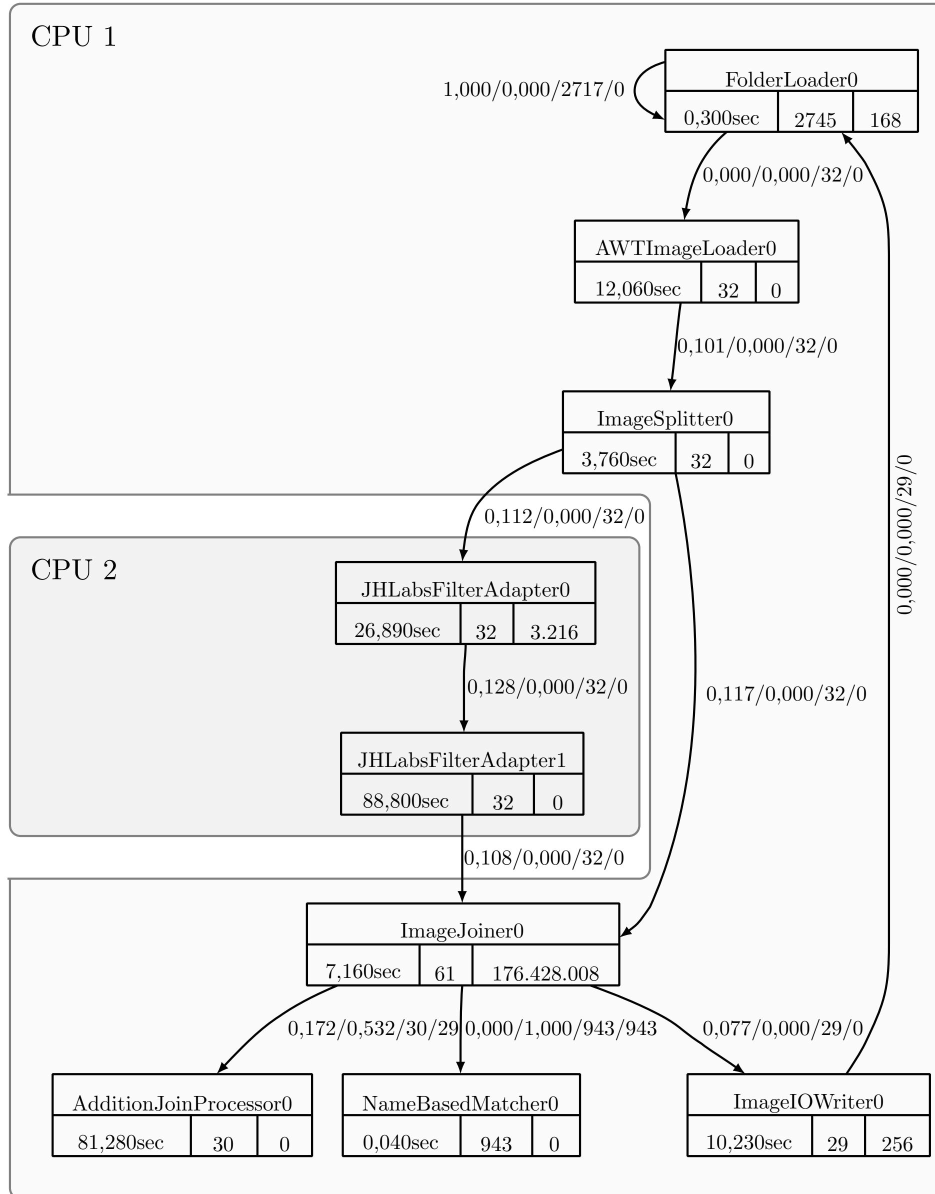


Figure 7.5: Component graph with edge/node weights and distribution on two CPUs

`ImageJoiner0` component needs to wait for both images to arrive, find out that they belong together (asking the `NameBasedMatcher0`) and join them (which is delegated to the `AdditionJoinProcessor0` component, thus implementing a Strategy pattern [**GHJV95**]). The exceptional memory requirements of `ImageJoiner0` can be explained by the fact that arriving images need to be cached by the component until the corresponding, filter-processed image copy has arrived. This was a benchmarking example we used for [**RS07b**]. Using METIS, we obtained a partitioning that outperformed a "naive" round-robin partitioning noticeable.

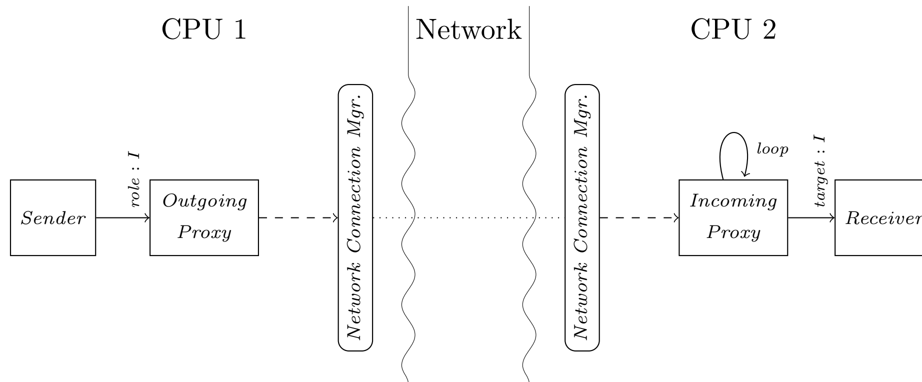


Figure 7.6: Proxy component for network connections

7.3.3.3. *Phase 3: Establishing the Component Connections.* Given a partitioning, the server then informs all clients about the local component subgraph that they have to establish, as well as the necessary inter-node connections to other components. The clients build the required proxy components, assemble the local component setup, and launch it.

Proxy components are initial components, and start right away to establish the network connections that they require to do remote communication. The target nodes' addresses have been provided by the server during the transmission of the local component subgraph. As soon as the communication is started, the normal operation phase may commence.

7.3.3.4. *Phase 4: Starting the Application.* By starting the regular (application-provided) initial components on the nodes they have been assigned to, the distributed application starts. There is a minor delay at the beginning until the proxy components have succeeded in establishing their connections, but any request sent to a remote component in the meantime by a regular component will be cached in the proxy component's queue.

7.3.4. The Proxy Components. Fig. 7.6 illustrates the concept of the proxy components for a single connection. The setup resembles a single connection of the *Sender* to the *Receiver* component. The proxy components wrap the network access; in theory, they might be linked directly using a JAVA `Socket` instance. Due to resource limitations, however, another indirection is added, which resides outside the component model (it could be made a component, using multi-ports to communicate with the proxies attached; see Sect. 8.3.1.1 for a discussion on how this can be achieved.) For each connection from one node to another, a *Network Connection Manager* is built on each of the nodes; it encapsulates the actual network communication. The proxy components then subscribe to this manager.

An *Outgoing Proxy* assembles a method call object similar to those used for implementing the Active Object pattern in the JCOMP framework. It is then passed to the network connection manager, which sends it over the network to its counterpart. This recipient network connection manager maintains a list of *Incoming Proxy* instances, and selects the one that is responsible for delivering the message to the recipient component. It enqueues the message in a special queue of this *Incoming Proxy* component.

The *Incoming Proxy* component is actively running, using a loop implemented by self-invocation (see Sect. 7.5.1.1). Hence, a method is called over and over again.

This method takes a look into the queue the network connection manager uses for adding messages received from remote nodes to, and processes any pending message by calling it on the recipient component. For synchronous methods, this call will return a result which is communicated back to the *Outgoing Proxy* using similar means.

In the actual implementation, the network connection manager is connected to by utilizing the Singleton pattern [**GHJV95**], which amounts to calling a static method for obtaining a reference to a shared instance. As discussed in Sect. 7.2.1, this is a breach of component encapsulation. Still, we feel this is no problem since the access to the shared object is limited to the proxy components, which are supplied by a framework extension. But nevertheless, the need for breaching encapsulation in such a way hints at the necessity of providing a framework-endorsed extension to shared resources, similar to the Utility Interface pattern of SOFA [**BHP07**].

7.3.5. Discussion. The decision to place the distribution code in the user level has proven to be the better approach compared to adding this code to the JComp core framework. First, the framework is kept lean, and the majority of applications that are not distributed is not required to handle the artifacts of distribution (like having to instantiate both a local assembly and a global server). Second, the flexibility is retained.

This is offset by an increase in message cost: Instead of handling a message sent to a remote component directly, it is first passed to the stub component (for which the data is copied) and then over the network (requiring another serialization of the data). It is quite easy to build provisions into the framework that allow for suspending the mandatory message data copying for such stub components, but that would violate the claim that the entire remoting framework is placed in the user level.

Still, given the cost of network communication, this drawback does not have a big impact. Finally, using regular stub components gives rise to an interesting approach of reorganizing a distribution by reconfiguration, detailed in Sect. 8.4.1.

7.4. Case Studies

Here, we will briefly present two larger projects implemented with the JComp framework. They have sparked some ideas about reconfiguration, which we will discuss in Chapter 8. We first present a classical component that gathers data from various sources and integrates these data. The other example uses the concurrency of JComp's components to implement an efficient web crawler with minimal overhead.

7.4.1. News Condensed. NewsCondensed is an example that was implemented for two purposes: Gathering experience with component-based design, and obtain a data base for evaluating *concept drift*, which is further described in Sect. 8.1.3.2. The basic idea is to gather news from a series of RSS feeds [**Boa06**] of newspaper websites, extract the words and generate a "most active words" list. This list is then used to rank the messages obtained. The highest ranking messages, presumably covering the most important topics of that day, are then displayed in generated HTML code, as shown in Fig. 7.7. Additionally, a number of static web resources (like weather charts) is downloaded for archiving purposes.

The component design, illustrated in Fig. 7.8, utilizes two chains of components, an often utilized pattern discussed in Sect. 8.3.1. These chains are responsible for querying the various data sources (the left chain of Fig. 7.8 retrieves messages from RSS feeds, the right chain fetches binary data, both from RSS feeds (e.g., comic

Donnerstag, den 28.11.2006, 20:12 Uhr

Top News

- Regierung gegen Tempolimit 120 auf deutschen Autobahnen (Newsticker - WELT.de)
- Mogadischu unter Kontrolle der Übergangsregierung (Newsticker - WELT.de)
- Keine 120 km/h für das Klima: Gabriel gegen Tempolimit (n-tv.de - Topmeldungen)
- Siemens und IBM erhalten "Herkules"-Auftrag Siemens und IBM haben den Zuschlag für die Modernisierung der IT-Systeme der Bundeswehr erhalten. Für insgesamt 7,1 Milliarden Euro sollen die beiden Konzerne in den kommenden zehn Jahren 140.000 PC, 7000 Großrechner, 300.000 Telefone und 15.000 Handys auf den neusten technischen Stand bringen. (tagesschau.de - Die Nachrichten der ARD)
- Siemens erhält größten Auftrag aller Zeiten Siemens und der US-Konzern IBM haben einen milliardenschweren Auftrag zur Modernisierung der Kommunikationstechnik der Bundeswehr erhalten. Für Siemens ist es der größte Auftrag aller Zeiten. Die Nachricht ließ auch die Anleger nicht kalt. (Financial Times Deutschland)
- Fünftägige Trauerfeierlichkeiten für früheren US-Präsidenten Ford (Newsticker - WELT.de)
- Koalition lehnt generelles Tempolimit auf Autobahnen ab Die schwarz-rote Koalition hat dem Vorschlag des Umweltbundesamtes, aus Klimaschutzgründen ein allgemeines Tempolimit auf deutschen Autobahnen einzuführen, abgelehnt. Klimaschädliche Abgase könnten mit Innovationen bei Kraftstoffen und Antrieben verringert werden, so Verkehrsminister Tiefensee. (tagesschau.de - Die Nachrichten der ARD)
- Weihnachtsgeschäft besser als im Vorjahr (Newsticker - WELT.de)
- Regierung übernimmt Kontrolle über Mogadischu Die somalische Übergangsregierung hat die Kontrolle über die Hauptstadt Mogadischu übernommen, die seit Juni von den islamistischen Milizen beherrscht worden war. Unterdessen warnen Hilfsorganisationen vor Chaos und Anarchie. (ZDFheute Nachrichten)
- "Herkules"-Auftrag für Siemens und IBM Die Verhandlungen für das größte IT-Projekt der Bundeswehr sind mit der Unterzeichnung von milliardenschweren Verträgen beendet worden. Die Bundeswehr besiegelte mit Siemens und IBM das Vertragswerk für die Erneuerung der Kommunikationstechnik. (ZDFheute Nachrichten)

Topwords tempolimit, autobahnen, geht, ibm, mekka, programm, hauptstadt, mogadischu, donnerstag, kandidatur

Figure 7.7: Sample output of NewsCondensed

strips) and static URL locations). These calls are synchronous; a list of results is assembled, which encapsulates all the downloaded data. The `RSSKnownHandler` instances at the beginning of both chains then prune the list of elements that have already been processed during previous runs. The `NewsOperator` component – which is the central instance and also the initial component – then uses the data to extract the words, evaluate them, rank the news, and assemble an HTML page.

7.4.1.1. *Design Decisions.* Utilizing chains of components for representing tasks of the same kind (i.e., different RSS feeds to query) is a design decision that has its benefits and drawbacks: Obviously, such an approach is only feasible if the number of tasks is limited (if a few thousand RSS feeds had to be polled, the approach would soon become too resource-wasting). Also, the system designer is put in charge of choosing which RSS feeds to download (as the configuration needs to reflect this choice). A user-provided list needs to be processed outside of the component framework – in the actual NewsCondensed application, this is done by dynamically instantiating components for a list of RSS addresses during system configuration by custom-written assembly code.

The benefits of employing chains is mostly given by the added flexibility. Each component "cares" about one RSS feed and stores information about it, e.g., if the RSS feed was reachable at the last attempt. This corresponds to building an object per RSS thread. As more functionality is provided, the importance of such an object grows – e.g., if multiple ways of fetching an RSS feed are implemented, the object would have to store a description of the algorithm to be used, and possibly even execute it. Hence, from a component-development point of view, representing different tasks by components is favorable. This will become quite important in the context of reconfiguration, as explicitly shown in Sect. 8.3.3.

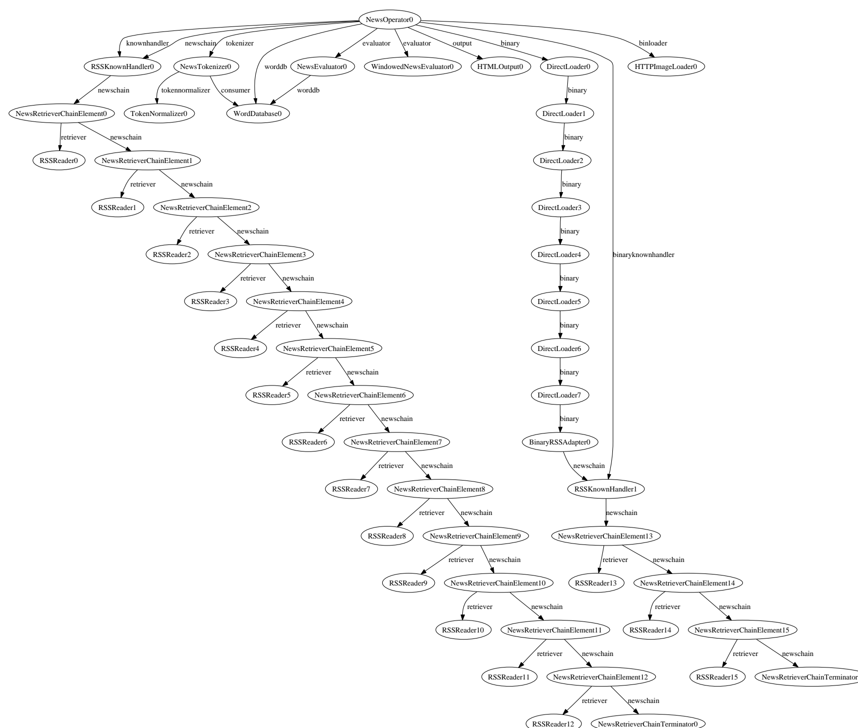


Figure 7.8: Components of NewsCondensed

7.4.2. A Web Crawler. NewsCondensed needs to judge the importance of a message based on the frequency of the words with respect to other messages. If a word (e.g., at the time of writing this, "Obama" would have made a good example) is found in many messages, it is deemed to hint at these messages covering a "hot topic". Obviously, this needs to be offset by the overall likelihood to encounter the word in a message. In order to build a frequency table of words of the German language, a web crawler was devised for crawling the German version of WikiPedia. This web crawler was to run a number of parallel threads in order to maximize the throughput.

This example is interesting because it took very little effort to implement. Actually, it was written in a few hours, at a conference, before breakfast. Afterwards, more work was added, but only for things like doing proper umlaut handling. This was possible since the multi-threading is all handled by the component framework. Fig. 7.9 shows the components involved. The `LoadDistributor` component is connected to a number of `PageLoader` components which download WikiPedia pages. They extract the links found and send them back to the `LoadDistributor`, who queries a `LinkFilter` if these links should be pursued, and if so, issues the request to one of the `PageLoader` components (based on the hash code of the URL). The `PageLoader` maintains a `PageSeenDatabase` which prohibits revisits to pages already processed. Also, it is connected to a `BracketStripper` that removes markup code. In theory, a number of such stripping components can be used to remove unwanted chunks of the page, so the next component is a `StripperTerminator` terminating the chain (cf. Sect. 8.3.1). This component then sends the data to a `WordDatabaseBuilder` which extracts the words and builds a database, using a `TokenNormalizer` to build normal forms (i.e., only lower-case letters and no blanks,

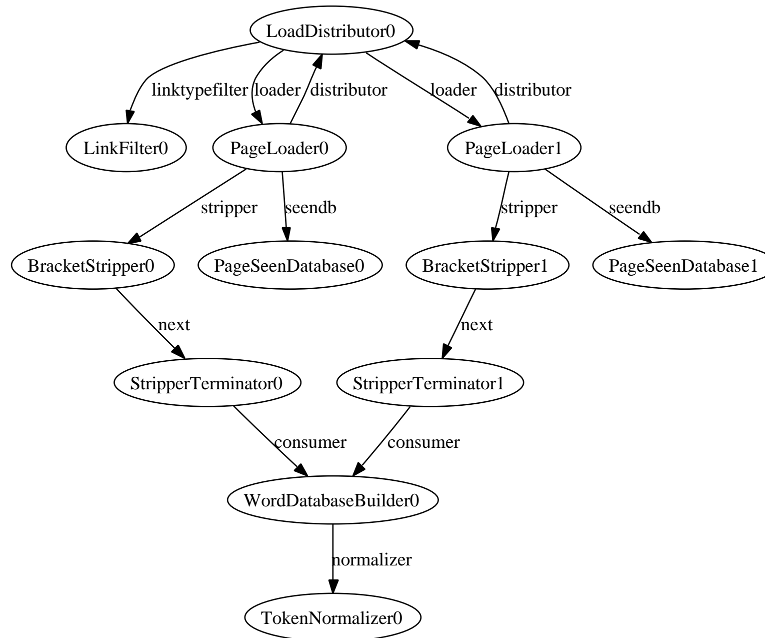


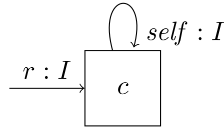
Figure 7.9: Components of the web crawler

as well as umlauts replaced in a way compatible with the NEWSCONDENSED application which eventually uses the database).

This example uses a *domain decomposition* (cf. Sect. 5.3.2) to achieve parallelization: By using a number (two in Fig. 7.9, but we used more in the practical runs) of identical components, with each handling a fraction of the pages to be processed, the impact of network latency is diminished. At the same time, *functional decomposition* is achieved by conducting the page-processing in a number of stages. The latter form of parallelization is provided for free by the component framework, which runs all components in their own thread. Domain decomposition needs to be established by the assembly, but the components only need very limited modification (i.e., the only place where domain decomposition is made known to the components is given within the `LoadDistributor` component, which needs to know the number of partitions in order to distribute the URLs to load). Hence, providing both kinds of parallelization is made easy with the JCOMP model. The de-serialization takes place in the message queues, which act as buffers between producers and consumers. The downside to this approach is that the distribution is hidden from the component programmer, and controllable only to some extend by the system designer. In Sect. 8.4.2, we will discuss a problem encountered with this approach.

7.5. Extensions of the JCOMP Component Model for Supporting Reconfiguration

In this section, we will discuss some extensions to the JCOMP model (and their implementation in the framework) that enable architectures and executions that are otherwise difficult to obtain, or impossible at all. The JCOMP model is rigid in some aspects: Concurrent method execution within the same component is impeded, even for synchronous messages, which can pose difficult problems with reconfiguration;



$$I \equiv \{m\},\ \zeta(c)(m) = P.\text{call}(\mathit{self}, m, \langle\rangle).\text{success}$$

Figure 7.10: Component with self-invocation capability

hence we extended JCOMP with the ability to interrupt synchronous messages by re-inserting them into the queue. We proceed to discuss component blocking, an important provision for multi-stage reconfiguration, and error handling.

7.5.1. Message Postponing. Message postponing is required in some cases where a message is recognized to be non-processable right now, or if its further execution needs to be deferred. Basically, the requirement to not interrupt the method processing by reconfiguration (R3.1.2) requires the component implementer to avoid methods to be running for too long, in order to meet requirement R4.1. One way to achieve this is to divide the method into a number of sub-methods, with each method calling the next one, thus providing intervention points for reconfiguration. For asynchronous calls, this can be achieved solely on the user level by utilizing the *Self-Invocation pattern*. For synchronous calls, message postponing is required.

7.5.1.1. *The Self-Invocation Pattern.* As a result of requirement R3.1.2, reconfiguration can only commence if all components in R are not executing a method. This becomes problematic if a component is forever executing a method because this method keeps the component running in a loop. In order to provide points in time where reconfiguration can commence, the *Self-Invocation pattern* can be applied. Basically, it substitutes a loop by a single iteration of the loop's body, followed by an invocation to itself. Obviously, the necessity to use such a loop substitution violates the separation of roles, but as long as no substitute by the framework is provided (and the only framework we are aware of to interrupt ongoing message processing for reconfiguration is CASA [**MG04**]), this needs to be done. Hence, part of the planning of a component application that can be modified by reconfiguration needs to be spend on choosing components that are suitable for reconfiguration by substituting loops by self-invocations.

Often, components need to remain active, i.e., not merely respond to future communication, but run actively for unlimited time. Components that observe some external resource will necessitate such a behavior. Component process terms provide a means for (unbounded) loops by defining, for a component process term fragment P, $\mu X \Rightarrow P.X$. Both CMC and JCOMP support this by their respective rules LOOP. In Sect. 5.2.4.1, we have seen that CMC can emulate this rule by self-calls. A similar approach can also be used for the JCOMP model, but asynchronous calls have to be used. Since the processing of the self-invocation may be interspersed by the execution of other messages, the behavior of both approaches are not the same, even if an action-reduction to communication actions without self-invocations is considered.

Fig. 7.10 shows the basic layout of a component that uses self-invocation as a replacement for a loop. An interface I, comprised solely of a method m, is provided; a required role r of type I is connected to the same component. Upon being triggered by an outside component (obviously, this should happen only once),

the loop's body P is executed once, followed by the self-invocation. Quite often, this is also done with the `start` method defined in the `MainInterface`, which is invoked by the framework; in this case, no other component should be connected to the component C requiring interface I. An example for this are the *Incoming Proxy* components presented in Sect. 7.3.4, as well as the example of Sect. 7.2.4.

7.5.1.2. *Dividing Synchronous Calls by Message Postponing.* There is a technical problem involved if the message in question is a synchronous call. A calling client is blocked until completion of the synchronous messages' execution, but not any longer. Hence, such a self-looping scheme cannot work for synchronous calls. The solution of JCOMP is to introduce *message postponing*, which allows a message's handler implementer to have the current method call object reinserted at the end of the queue. Both asynchronous and synchronous methods are allowed to be postponed. For asynchronous methods, the postponement is equivalent to sending the current method call to a self-loop, and immediately terminating the current method's execution afterwards. For synchronous methods, this feature cannot be substituted.

Besides breaking atomicity, message postponing can be used to implement a synchronicity-to-asynchronicity converter. This is an adapter component that provides an interface with some synchronous methods. Upon receiving such a method, it relays the request to a role that provides a similar interface, but defines the method as asynchronous and sends the result back by an asynchronous method call itself. The synchronous call is then postponed, and thus eventually processed again. It can then look in some table if the result has yet been sent by the asynchronous result method; if so, it can return it, and otherwise continue postponing itself.

Formally, we can introduce message postponing in our component model by adding postpone as a component process term, and use the following rule:

$$c_1^{\mathrm{r}}, \langle c_2.r.m(v)\rangle, \mathrm{postpone}, \pi \rightarrow c_1^{\mathrm{r}}, \langle \bot \rangle, \mathrm{success}, \pi :: c_2.r.m(v) \quad (\textsc{postpone})$$

postpone can be used to terminate component process terms; upon executing it, the current message will be enqueued again. It is up to the component's implementation to store loop-internal data in the component's state. Obviously, if the method is a synchronous invocation by component c_2, c_2 will remain blocked until the method's execution is terminated by a return.

In JCOMP, message postponing is implemented by allowing method implementations to throw a `CallPostponedException`. This method is caught by the framework, the communication monitors are notified (cf. Sect. 7.2.5) and the message is re-inserted in the queue.

7.5.2. Component Blocking. Sometimes, we require a component that does nothing at all. Of course, it needs to accept messages, but it should not process any. For a static component system, this is pretty useless, of course. When doing reconfiguration, however, sometimes such components are required. This is because many reconfiguration scenarios need to be conducted by a series of reconfiguration steps, e.g., for conducting an extended protocol between an old and a new component between adding the new and removing the old component, as described in Sect. 6.8.2. In such settings, proxy components are required that just cache the communication that happens between the reconfiguration steps such that it does not get lost, but they are not intended to actually handle the messages received. Instead, they are to block until the final reconfiguration transports the accumulated messages to another component that can properly handle them.

With standard means, a component can be made to block immediately after the beginning of processing a message. It can be useful to block at a later time, e.g., if a situation is detected where the only possible way of progressing is reconfiguration.

Reconfiguration is the only way to resolve a blocked component, which needs to be removed during reconfiguration. We already have a provision for that: The fail component process term. None of the rules shown in Tab. 6.1 can consume a fail component process term, but reconfiguration rules, shown in Tab. 6.2 in Sect. 6.6 will be able to do so.

There is a subtle point to be made: fail, by its informal meaning, indicates a failure of the component; it is unrecoverable and the system needs to be repaired. On the other hand, blocking is fully anticipated and often the only outcome of a method being invoked on a placeholder component. Therefore, we would like to use a different term, like block. In order to keep the component process terms slim, however, we refrain from adding such a term; for the remainder of this thesis, it will always be clear from the context whether a genuine error or a blocking is intended by issuing fail.

Another point worth mentioning is the use of the component running flag c^{b}. This flag blocks the component; so it might be feasible to use this one to signal that a component is blocked (since it has no pending synchronous calls, it will not become unblocked unless a reconfiguration rule is applied). The problem here is that the running flags are intended to describe communication behavior[2], whereas the issuing of fail or block is done by the user and outside the responsibilities of the component model. We hence rely on a component process term that cannot ever become consumed by means other than reconfiguration.

7.5.3. Error Handling. Errors inevitably occur; in the JCOMP model this is reflected by reaching a state c^r, fail. Since none of the rules presented so far matches to such a component state, the component is locked forever; and, once it has become locked like this, only reconfiguration can be used to progress it again. Having a component in such a failed state leads to further deadlocks or a stalled system; hence a correct response to an error might even be to shut down the entire component application [BP07]. In any case, errors and their treatment need to be considered by the component framework.

Aside from using errors as a trigger for reconfiguration, as we will do in Sect. 7.5.4.2, there are other ways to cope with an error. Suitable solutions need to take the kind of error into account: It can be a response to a changed environment (i.e., loss of a hardware device, faulty data read from a file etc.), which can (and, by virtue of the JAVA exception handling paradigm, also need to) be handled by the component's code. An error might also result from a malformed query of one component to another; i.e., due to a violation of a precondition. Or an error might come up due to an unexpected event like running out of memory (if having sufficient memory is not in doubt at component design time).

All three types of errors might be communicated to the framework by the component, instead of handling it internally. For a contract violation (e.g., receiving a message that cannot be processed in the current component's state), a system failure should be signaled, since the component setup is malformed and needs repair [BP07]. An unexpected error might not be curable by a component, and will usually render its state corrupted and the component unusable. Whether this warrants a shutdown depends on the capabilities of the component application: Reconfiguration can be used to reestablish a working version of the component. Even an anticipated error might be communicated to the environment if the resolution strategy is not clear at component design time (e.g., it is possible to anticipate that a required file might not be found, but the treatment of such an error might

[2]It was once intended to allow for a more generic definition of component models, providing arbitrary communication means; this was given up in favor of the first-class connectors now utilized in REFLECT [SvdZH08].

heavily depend on the actual utilization of the component, which is only known at assembly time).

Hence, we need to provide components with provisions for signaling errors, and support the handling of errors by the system. In Sect. 7.2.5.2, we have discussed how exceptions can be monitored. Here, we will investigate different ways to handle an exception.

7.5.3.1. *Error Handling by System Shutdown.* In the concurrent EIFFEL extension SCOOP [AENV06], shutting down the system in response to an error of a single thread is the default behavior [BP07]. This is justified by the assumption that an error should never occur in a well-assembled system, and hence the occurrence of an error hints at a malformed system design, which needs to be repaired. We can express such a system-wide shutdown by removing all components:

$$c_c^r, \text{fail} \mid c_1^{\xi_1}, P_1 \mid \ldots \mid c_n^{\xi_n}, P_n \rightarrow \tag{ERR}$$

Alternatively, we can just block all components and thus prohibit any further progress:

$$c_c^r, \text{fail} \mid c_1^{\xi_1}, P_1 \mid \ldots \mid c_n^{\xi_n}, P_n \rightarrow c_c^r, \text{fail} \mid c_1^{\xi_1}, \text{fail} \mid \ldots c_n^{\xi_n}, \text{fail} \tag{ERR'}$$

The benefit of the second variant is that reconfiguration might now be used to restart the system. This is, however, far from likely, and in the JCOMP implementation, if all exception monitors suggest `DIE` (or if no monitor is given), the entire application is just terminated with a call to `java.lang.System.exit()`.

7.5.3.2. *Error Handling by Delegation.* For synchronous methods, JCOMP allows the declaration of exceptions. Since the invoking component is blocked until the call has finished execution at the target component, an exception can be transported back and invoked in time on the target component. Thus, the JAVA exception handling can be reused to enforce proper exception treatment within the calling component's code. As the experience with JCOMP broadened, however, this kind of error treatment became used less often; and most of the programs presented in Chapter 8 do not use synchronous calls at all, particularly not with exception declarations. For our purpose, this kind of error handling can be considered to be a special kind of return value of a method; but it was so seldom used in the examples that little seems to be gained from explicitly adding provisions to the component model, unless such provisions can be extended to work with asynchronous method invocations also (cf. Sect. 7.5.3.5). Delegating exceptions is, however, the behavior of JCOMP if an exception is encountered during the processing of a synchronous call, with no exception monitor suggesting anything other than `DIE`.

7.5.3.3. *Error Handling by Reconfiguration.* Handling an error by reconfiguration is a delegation of the concern of handling the error from the component designer level to the system designer. If error handling is perceived as a concern of the component designer, the components need to be written in a way such that they become capable of handling the errors they may encounter. The example of a missing file required by a component illustrates the problem: If this file is mandatory, and the component does not know an alternative strategy to obtain its contents, it needs to delegate the error and thus shut down the system, as discussed before. This happens regardless of the importance of the component to the overall system. It is unknown to the component (or, rather, beyond its implementer's concern) whether the system can run without it, or whether the system commands a suitable alternative. Hence, it is more suitable to recover such an error on the system level (or decide that it is unrecoverable) and make error handling a concern of the system designer.

Hence, error handling is a trigger for reconfiguration. An example for such an error-triggered reconfiguration will be given in Sect. 8.4.3. Since issuing a fail component subterm blocks the component until it is reconfigured, no special rule is required in the calculus. A component is placed in that fail state by the framework if the monitors agreed on either `RECONFIGURE_AND_SKIP` or `RECONFIGURE_AND_KEEP`. The difference between these two suggestions relates to the treatment of the message that caused the exception: the former suggestion has this message discarded, whereas the second suggestion prepends it to the message queue of the failed component, properly reflecting the RCSTOPF rule. Using the δ plan element, it will eventually be moved to a new component and become executed anew.

An example of a reconfiguration triggered by exception monitoring is given in Sect. 7.5.4. From a technical point of view, the framework needs to be carefully crafted to support such a reconfiguration, as it must be precisely timed: no message must be lost, and no further messages must be consumed. This is one of the reasons why building a dedicated framework is useful: retrofitting such provisions to an existing framework that does not anticipate a special "failed and about to become reconfigured" state for its components will be cumbersome and prone to colliding with existing mechanisms.

7.5.3.4. *Error Handling by Ignoring.* Obviously, an error might also just be ignored. For example, a component that reads a file and inputs the data into a test framework might experience failure when reading some lines without compromising the entire application's purpose. By issuing a `CONTINUE` suggestion, the exception monitors can achieve such a functionality. This might also be used for delaying a necessary reconfiguration a little bit, if the error does not require immediate attention. In the small-scale examples exhibited in this thesis, we have not found a necessity for such a strategy, and it is included in the JComp framework for completeness only, without extending the component model.

7.5.3.5. *Future Work.* Other interesting approaches towards handling errors exist; most notably the approach taken by Erlang. Erlang [**RA03**] is a language developed for telecommunication switches. One of the many distinct features of this language is its unique error handling, which cascades any uncaught error through the system, shutting down all processes that are connected to processes experiencing an error and are unable to resolve it, until either the entire system is shut down, or the error remains contained. Since the language is geared towards hardware-intensive systems (where errors might be nondeterministic from the point of view of the software), killing processes is a very convenient way to resolve errors: Any process that fails to handle the error is considered corrupted and gets terminated. Such an approach is a major distinction from exception handling systems found in programming languages like Java; here, objects that throw an exception may be left in a corrupted state. The "resolve or die" approach of Erlang prevents processes from operating on a broken state.

In our setting, such an approach can be included like this: A component producing an error is removed. All connected components are notified by a special message, and they are free to take action as desired. If they do not take action, i.e., they block the method call telling them to do so by interpreting it with fail, they fail as well; thus the error propagates through the system until all components either have taken appropriate action, are shut down, or are not directly connected to a failing component. This is, however, more realistic in a hardware-oriented environment, since the Erlang approach usually is to just restart corrupted processes, which in a software-only environment are bound to just reproduce much of the errors again. The approach is nevertheless very interesting, and can be considered

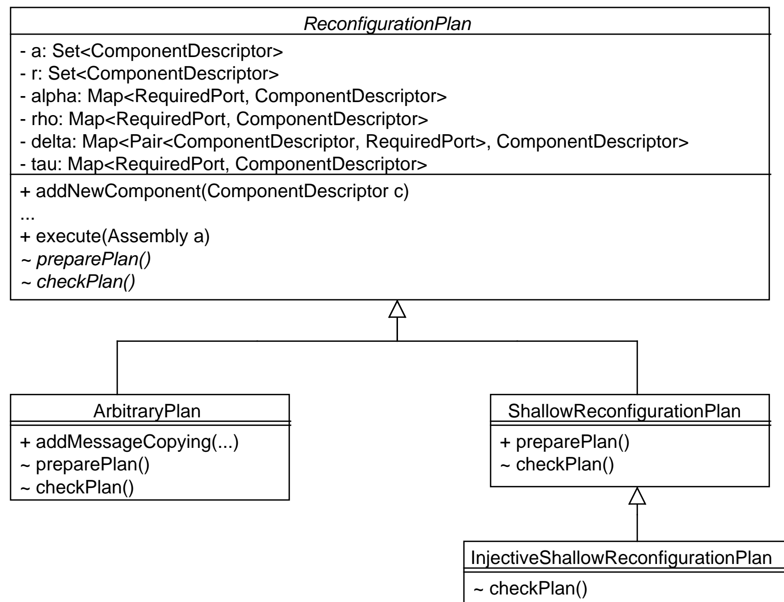


Figure 7.11: UML class diagram for reconfiguration plan classes

as promising future work; both for its capability to add robustness to a system and for its provision of a reconfiguration scenario.

7.5.4. Reconfiguration in the JComp Framework. The reconfiguration approach of the JComp model is exactly implemented by the JComp framework, using the indirect approach for state transferal. In doing so, the theoretical results of Chapter 6 provide the reconfiguration implementer with useful guarantees that ease the planning of reconfiguration. At the center of the implementation are the reconfiguration plans, which can be directly created or be generated from various extensions (e.g., a graph transformation engine described in Sect. 8.1.4.1).

7.5.4.1. *The Reconfiguration Plans.* Fig. 7.11 shows a UML class diagram of the reconfiguration plan hierarchy as implemented in the JComp framework. Most of the functionality and all of the relevant data is located in the abstract `ReconfigurationPlan` class. This class offers methods to populate all plan elements except δ. The subclasses just differ in their treatment of δ and their check for consistency: the `ArbitraryReconfigurationPlan` class offers a way to directly add rewiring requests to the plan, whereas the `ShallowReconfigurationPlan` class calculates δ within the `preparePlan()` method. The `InjectiveShallowReconfigurationPlan` class just extends the check conducted within the `checkPlan()` method, which needs to verify that the plan indeed describes a shallow reconfiguration plan.

Reconfiguration is affected by the method `execute(Assembly a)`, which conducts a plan implementation as described in Sect. 6.7. These rules are executed by calling appropriate methods on the assembly. The initial rules RCSTOPS and RCSTOPF are implemented by prepending a special configuration message to the target component's queue. This way, it is ensured that the component finishes its current method, then dequeues the reconfiguration requests and in executing it

becomes stopped for reconfiguration. Failed components (that experienced an exception, followed by an exception monitor suggesting reconfiguration) wait exactly for that reconfiguration message, and cease to dequeue any other message. For this reason, and for avoiding delays of the reconfiguration, the reconfiguration request is prepended to the queue, and not appended like regular messages are.

7.5.4.2. *The Example Continued.* The example of Sect. 7.2.4, extended with the capacity check presented in Sect. 7.2.5.2, can now be mended by reconfiguration. We assume that we have a component implementation `NewStoreComponent` that does not suffer from any capacity constraint. For this example, the actual reconfiguration planning is conducted within the exception monitor:

```
1  public ExceptionMonitorResult uncaughtExceptionThrown(AbstractComponent src,
2                                                          Call call) {
3      InjectiveShallowReconfigurationPlan p =
4          new InjectiveShallowReconfigurationPlan();
5      p.addRemoveComponent(sc);
6      ComponentDescriptor nsc = a.getComponentForClass(NewStoreComponent.class);
7      p.addNewComponent(nsc);
8      p.addRewireConnection(cc, "store", nsc);
9
10     ReconfigurationTools.runReconfigurationConcurrently(a, p);
11     return ExceptionMonitorResult.RECONFIGURE_AND_KEEP;
12 }
```

We use an injective shallow reconfiguration plan, so we do not have to worry about δ. Instead, we just declare which components are to be removed and to be added, and use the method `addRewireConnection` to add an element to ρ. α is not required for this example, and for now, we omit state transferal with τ. Note that we stored the components in static variables which we can now use. For larger reconfigurations, this is infeasible, and the component graph needs to be traversed in order to obtain the actual reconfiguration plan; Chapter 7.3 will provide many examples.

The reconfiguration is executed by a call to `Assembly.reconfigure(Plan p)`, but as we need to ensure a thread switch (the thread that called the exception monitor and subsequently built the reconfiguration plan is the failed component's own thread, so it cannot dequeue the reconfiguration message unless the exception monitor terminates), a helper method is used that forks a dedicated thread for the reconfiguration. By returning the exception monitor with `RECONFIGURE_AND_KEEP`, the message that caused the exception will be re-enqueued, and the `NewStoreComponent` instance will receive it as the first message after completion of the reconfiguration.

7.5.4.2.1. *State Transfer.* In order to add state transfer, we need to modify the components a little:

(1) First, we have to add provisions for reading the contents of the old component. In this example, which heavily utilizes the indirect approach to state transfer, we introduce a new interface `StoreRetrieveInterface`:

```
1  public interface StoreRetrieveInterface {
2      @SynchronousCall
3      public LinkedList<String> getEntries();
4  }
```

This interface is implemented by the `StoreComponent`, which just returns the `entries` attribute in the implementation of the `getEntries()` method. Note that the interface has to use a

java.util.LinkedList, as the java.util.List interface does not inherit from java.io.Serializable, as it is required for return types of synchronous methods in the JCOMP framework.

(2) Next, we implement the NewStoreComponent in a way that supports the state transfer during reconfiguration. In this example, we retrieve the list of entries stored in the old StoreComponent, and write them to a file, which is where the NewStoreComponent will store the data it receives after the reconfiguration:

```
1  public class NewStoreComponent extends AbstractComponent
2                                implements StoreInterface {
3      @RequiredInterface(binding=InterfaceBindingPolicy.RECONF)
4      private StoreRetrieveInterface oldcomp;
5
6      private PrintWriter outfile;
7
8      public NewStoreComponent() throws IOException {
9          outfile = new PrintWriter(new FileWriter("store.log"));
10     }
11
12     public void store(String s) {
13         outfile.println(s);
14     }
15
16     public void reconfigurationCopyEvent() {
17         List<String> l = oldcomp.getEntries();
18         for (String s : l) {
19             outfile.println(s);
20         }
21     }
22 }
```

The @RequiredInterface-annotated attribute is only bound during reconfiguration, and will not be allowed (or required) to be connected at any other time. The actual state transfer is realized in the reconfigurationCopyEvent() implementation, which is triggered by adding a message to the message queue of the NewStoreComponent prior to allowing it to process regular messages (thus implementing the c^c state).

(3) Finally, the temporary connection has to be realized in the reconfiguration plan. This is achieved by adding the line
p.addTemporaryConnection(nsc, "oldcomp", sc);
to the main method.

From the implementers point of view, the state transferal is implemented as a sub-protocol of the component, differing only very little from regular communication. However, it becomes evident that the indirect approach requires substantial modification of the components. Still, this seems more tractable than allowing direct access to the component's internal attributes by the framework, requiring the reconfiguration designer to gain detailed knowledge about the component's implementation. For more complicated state transfer, the hybrid approach seems to offer a compromise that restricts the complexity of implementing the reconfigurationCopyEvent() method to a minimum. We will see an example for the hybrid approach in Sect. 8.2.2.

CHAPTER 8

Applications of Reconfiguration

In response to a query from the computer, each black box could answer in turn: "I am well", with all parameters within limits. A particular box might answer, "I am sick", with one or more parameters outside the safe zone. Similarly a box might respond, "I am about to get sick", with a parameter drifting toward danger. Further queries from the computer then could identify the bad parameters and permit cures.

— T. A. Heppenheimer – The Space Shuttle Decision

This chapter used to be called "examples", and actually, it will present some examples on how reconfiguration as we have realized it in the JCOMP model and framework can be put to use in real applications. While writing the related work section in Chapter 3, however, a different view on the utility of reconfiguration examples emerged. The best examples are provided by work that does not consider generic reconfiguration, but restrains itself to just a subset of the problem domain, e.g., by just reconfiguring network connection code as done in CACTUS [**CHS01**] or NECOMAN [**JTSJ07**]. Here, we take the opposite direction: Instead of providing just examples, we try to investigate various scenarios where reconfiguration can be beneficial, and exemplify them with a report on their implementation with the JCOMP component framework.

Finding good examples for reconfiguration is not easy. This is due to two problems: First, reconfiguration cannot do anything that could not be done otherwise; from the actual application execution, it is barely more than a big case distinction. The *separation of concerns* (in the form of the separation of roles) is what matters: Possible reconfiguration scenarios should not be considered when devising a component – save for the need to provide data accessing methods for state retainment. It is, however, not easy to find domains where such a separation of concern can be exemplified by a small application, written by a single person.

For example, we considered developing a computer chess program. During a match, a chess program runs through a number of distinct phases: It starts with an opening book table, which is a pre-calculated graph annotated with move feasibility. Once a situation is obtained that is no longer covered by this graph, iterative deepening search with alpha-beta-pruning and lots of interesting heuristics is employed to find good moves [**Hei99**]. Eventually, when only a few pieces are left on the field, an end-game table is employed again. Recognizing these phases and adapting the program accordingly appeared as a good example for reconfiguration.

However, most likely it is not. There is no real separation of concerns: Anyone devising an opening table component is fully aware that it will eventually be replaced by the usual search. There is little to be gained if the reconfiguration from open book play to mid-game play is planned externally; any forking component will do. Of course, the program structure stays more tidy if the open book database is removed once it is no longer required, but this argument is severely offset by the added complexity introduced by the reconfiguration algorithm.

Hence, one reason why reconfiguration examples are hard to come by is that separation of concerns is difficult to find. The second reason is that reconfiguration is often too "binary".

Consider the chess program again. Basically, the mid-game search consists of three components: A search component conducting the iterative deepening search with alpha-beta-pruning; a heuristic evaluation of moves, used to rank moves to search the better ones first, and get a narrow alpha-beta-window fast; and a situation evaluation component that ranks boards according to their "utility" for winning the game. The latter two components obviously require tuning, heuristics and all kinds of tricks and tweaks to make a program competitive. Of course, all this has to be modified according to the situation (e.g., the rule-of-thumb "a Queen is worth two Rooks" needs to be modified according to stage of the game). All these adaptation sounds like an interesting field for reconfiguration.

Alas, however, not for the kind of reconfiguration described in this thesis. If not for drastic developments, all the different components involved in calculating a heuristic move evaluation are run, and their output is combined in a different way depending on the game situation; but rarely will we be able to introduce new components or remove old ones. While not strictly impossible, it seems questionable if anything can be gained from introducing or removing components.

Hence, an example for reconfiguration will have to provide two things:

(1) It has to be credible with respect to doing an actual separation of concerns, and
(2) it has to work with entire components, not their internal configuration.

Nevertheless, there are some examples where these two conditions are met. They cannot really be claimed to be profoundly realistic, or that there is no other way to obtain the results, but we consider them to be genuine examples of situations where reconfiguration can be beneficial.

First, we will illustrate how reconfiguration can be used to effect hot code updates. Besides a simple example for simple component updates, a more elaborate example with a change of the component graph structure is presented. These examples, however, share that they do not require monitoring and assessment by the software; the reconfiguration is started by a user's explicit request.

We proceed to discuss an example which provides two things: An illustration of integration of various reconfiguration possibilities, and the illustration how reconfiguration can be used to adapt to external changes, e.g., the loss of network connectivity. This example builds on the NewsCondensed example of Sect. 7.4.1.

We then investigate a promising area for reconfiguration: cross-cutting concerns. Aside from the broad approach of Kramer and Magee [**KM08a**] and the domain of hot code updates, those examples are the most convincing ones from the literature [**CHS01, JDMV04**]. Cross-cutting concerns cover application distribution, logging and maintenance of non-functional properties. We also report on an example providing fault tolerance to a system only partially prepared for this.

We finish this list of examples by discussing reconfiguration possibilities for the cmc model checker, which proved surprisingly resistant to runtime reconfiguration. Instead, we propose a reconfiguration at compile time, effectively building a new version of cmc if the old version turns out to be insufficient to handle a particular model.

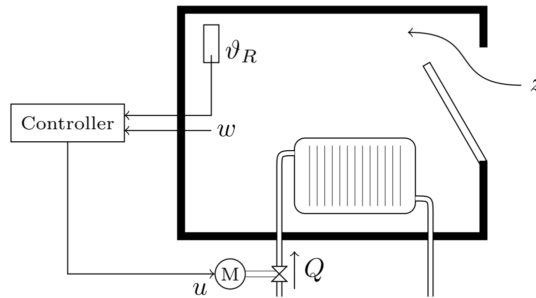


Figure 8.1: Closed control loop

8.1. MAPE

"I've tried A! I've tried B!
I've tried C! I've tried D!
Tell me what else I can try!"
— Tom Wolfe, The Right Stuff

So far, we have only discussed the application of reconfiguration once a plan is obtained. For providing meaningful examples, a bigger picture needs to be drawn. In this section, we will discuss the stages preceding the reconfiguration plan execution that lead to the generation of the reconfiguration plan. There is quite an interesting consensus in the literature that these stages need to be organized in a closed control loop, an approach adopted from control theory [**And06**], and mostly, only the shifted focus of the various works make a difference.

8.1.1. Control Loops. The control loops we investigate here have two distinctive properties: They are *staged*, meaning that they consist of a repeated application of distinct algorithms, and they are closed, meaning that they investigate the very data they eventually modify. Such control loops are by no means an invention of computer science. The actual four stages we consider in this thesis, namely *Monitoring*, *Assessing*, *Planning* and *Execution*, abbreviated as *MAPE* within this thesis, are introduced in a US Air Force doctrine document [**USAF05**]:

- "Monitoring *involves the processes of collecting, storing, maintaining, and tracking of data.*
- Assessing *results in the ability to determine the nature and impact of conditions and events on force capabilities and commander's intent. It involves the processes of analyzing and evaluating along with modeling and simulation to describe situational awareness and alternative solutions.*
- Planning *is how we support the operational objectives; develop, evaluate, and select courses of actions; generate force lists (capabilities) and force movement requirements; and detail the timing of sequential actions. Planning is essentially a description and prioritization of how to achieve stated mission goals.*
- Execution *is the overall dissemination and action of the plan to ensure successful mission accomplishment."*

Control loops, or feedback loops, originate in control theory. In control theory, a physical environment is affected by some actuator, in order to steer it towards

some goal. A well-known example is a room heating system [Sch05]: The actual temperature is measured, and depending on some rules (that describe the *need* for change, which corresponds to the assessment, and the *method* of performing the change, which constitutes the planning) the environment is affected by turning the radiator up or down.

This example, shown in Fig. 8.1, is about keeping the room temperature ϑ_R within as close as possible to a user-defined value w. The temperature changes over time due to a disturbance z. Also, the radiator flow Q affects the room temperature. This radiator flow can be affected by the means of an actuator M that operates a valve. The controller can send a command u to the actuator. Thus, a closed control loop is established: The room temperature is monitored, and an action u is calculated (which might well be to retain the old flow value, if the temperature is progressing as intended). The action is effected by the means of the valve modifying the flow Q. The effect is then measured by a change to the room temperature ϑ_R; making this setup a closed loop. In contrast, an open control loop does not measure the effect of the actions; instead, it would measure the outside temperature z and operate M based on rules without ever comparing the outcome ϑ_R to the intentions w.

The example shows three important concepts of control theory, as described in [And06]:

(1) *Feedback*: By measuring the room temperature ϑ_R over and over, the effect of changes to the flow rate Q can be assessed. Generally, feedback is given if the state of the system is used for computing control output.
(2) *Fluctuations*: Since a control loop is responsive and will suffer some latency time, it can only be employed in a system where a certain amount of fluctuation is admissible. In the room temperature example, a certain range of temperature values is certainly acceptable.
(3) *Optimization*: Control theory is about optimizing a system, which makes it necessary to have some measurement of quality.

When considering software, it immediately becomes obvious that these concepts need to be adapted. Feedback is easy to obtain; in fact, it does not make sense to consider a software system that is unaware of its state. Fluctuations and optimization, however, do not carry over from physical systems to software as easily. There are certain domains, e.g., in load balancing, where the analogy works well, but as soon as qualitative errors come into play (e.g., throwing of exceptions), neither fluctuation nor optimization considerations are applicable.

8.1.1.1. *Control Loops for Software.* While this might indicate that the MAPE loop works only for a certain domain of systems, where quantitative parameters are of concern, its basic idea can also be employed for qualitative properties. Obviously, all four stages need to be interpreted in a very different way.

In the literature, the difference between quantitative and qualitative properties is usually not discussed; although there are some frameworks that employ software control loops for ensuring quality of service, e.g., SWiFT [GSPW98], the TAPAS middle-ware [Fer02] or ControlWare [ZLAS02]. There is a certain anticipation that, once a sufficient level of abstraction has been reached, both will become the same anyway. If a system is built to fail smoothly (i.e., go through a series of degraded operation modes before stopping to work altogether), there is little difference between an under-performing and a faulty system. In this spirit, Kephart and Chess from IBM (who adopted the MAPE loop for software, here being an acronym for "monitor, analyze, plan, execute") talk about self-optimization (where

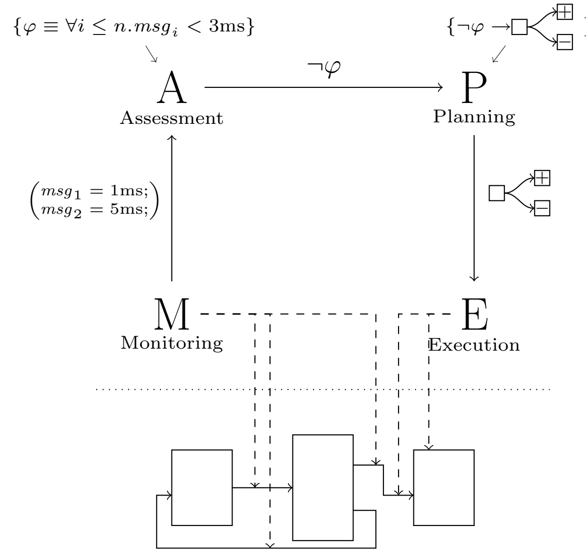


Figure 8.2: MAPE loop

improving the performance is considered) and self-healing (finding and removing faulty components) in the same context [KC03].

When considering the four stages in the context of reconfiguration as described in this thesis, we get a slightly different view on the control loop. Illustrated in Fig. 8.2, the loop consists of these phases:

- *Monitoring* is concerned with obtaining data about the component application. This part is usually the easiest one, but can be extended to very elaborate architectures (e.g., probes and gauges [GCH+04]). Component-based applications are especially suitable for monitoring, as the communication is very visible. Component introspection, however, is usually ruled out; if a component's state needs to be monitored, the component implementer has to provide means to externalize the relevant data (cf. Sect. 5.5 for the description of providing monitoring of the component's state for CMC – while this monitoring is provided for human access, it can as well be a source of data for an automatized control loop).
- *Assessing* the situation is closely tied to the subsequent step of planning the adaptation. In the context of reconfiguration, assessment is mostly concerned with detecting the need for reconfiguration, which consists of two almost separate considerations: First, is the situation degraded in a way that requires reconfiguration, or at least makes it desirable, and second, do we have means to improve the situation?
- *Planning* reconfiguration has been given much research in the context of architectural styles [GS93]. Again, two distinct considerations need to be made: First, what is the configuration that should be achieved by reconfiguration, and second, how should the reconfiguration be conducted? The plans introduced in Sect. 6.5 are provided as the result of this phase.
- *Execution* finally realizes the plan, as discussed in Sect. 6.7.2.

These stages are found in the work of Garlan and Schmerl [GS02], which explicitly mention a closed control loop and describe the stages of monitoring, interpretation, resolution and adaptation, which directly correspond to the MAPE

stages. In the survey of Bradbury [**Bra04**], the first two MAPE stages are combined (monitoring/analysis), followed by planning, execution and assessment, which stresses the closedness of the control loop. Arshad and Heimbigner combine the middle stages and describe three stages: Sense, plan and act [**AH05**].

"Sense-plan-act" is known as the SPA paradigm in robotics [**KM07**], where it is regarded as kind of old-fashioned because of its being monolithic; it was subsequently replaced by decentralized architectures (nevertheless employing variants of the MAPE loop) as early as 1985 [**Bro85**]; an adoption of these insights is carrying over to software control loops only recently [**KM07, SvdZH08**]. A slightly varied three-stage loop is found in the work of Boinot et al. [**BMMC00**], where the stages are named introspection, control and installation. Jade [**TBB+05**] also uses three stages, called "Detection", "Analysis and Decision" and "Action". In GridKit [**GCB+06**], such a three-stage loop is dubbed the "Event-Condition-Action pattern" (obviously, the event needs to be generated by an external introspection). The underlying pattern of first detecting the need for adaptation, then planning and executing it is retained in all these works.

The MAPE loop plays a central role in the design of autonomous systems, which are systems (often, robots) that run unsupervised for an extended period of time and are capable of handling changing environments. Dobson et al. [**DDF+06**] name the loop's stages "collect, decide, plan, act". In the survey of Huebscher and McCann [**HM08**] the loop is described as "MAPE-K", a loop with an "internal feedback" – knowledge "K", the fifth element of this loop – that is used for optimizing the loop itself. Examples for this knowledge representation are utility considerations and reinforcement learning.

Building on the MAPE definition of a control loop, Kramer and Magee advocate the use of control loops for robust software systems, and directly address the parallels to robotics [**KM07, KM08a**]. They extend this approach by introducing a layered architecture, where each layer is subjected to a MAPE loop, providing adaptivity on various levels – and incorporating the knowledge element by the ability of higher-level loops to modify the lower-level ones. Considering their and other projects' efforts (e.g., [**GSPW98, ZLAS02**]), it might be safe to say that the utilization of control loops has become a common denominator of adaptive software research. Of course, not all systems discussed in the context of "reconfiguration" are amenable to control loops – most notable hot code updates, where the first three stages are entirely left to the user. (Although, if one wants to, the phases still can be recognized: The user monitors the software while using it, assesses this observation and eventually decides that a code update is required, has this update planned by the system designer and finally executes the update by installing the rewritten software – this traditional software maintenance cycle is described as a "very first control loop" in [**EMS+08**]).

Anticipating the example of Sect. 8.4.3, we can illustrate the phases in the context of fault tolerance: A (part of a) software system needs to work with a component that is known to fail sometimes. First, in order to provide some fault tolerance, the component needs to be monitored for an occurrence of such a failure. In the JComp implementation, exception listeners can be registered for each component, so that an exceptional end of a method processing can be detected. Alternatively, a watchdog component might do repeated health check calls. Ultimately, monitoring needs to gather sufficient data such that a failure can be detected.

Monitoring, however, does not actually detect an error; only the data is provided. The actual detection of an error condition is done by assessing the monitored data in the next stage of the MAPE loop. Sometimes, as for the exception listener, this is fairly trivial, but it can become quite difficult to judge erroneous behavior

of a component from communication observations (cf. [Bau05], which uses a SAT solver for a similar task). Not only the fact that an error has occurred needs to be recognized, but also the extent of damage caused so far needs to be approximated, and provided as an input to the planning stage. This is necessary because, depending on the nature of the error, data might have been corrupted or invalid communication behavior might have commenced; the plan needs to contain provisions that mend these problems.

In the planning stage, the recovery of the system needs to be planned. The corrupted data needs to be taken care of, and maybe the faulty component needs to be replaced or protected against further, similar errors. Obviously, this is pretty difficult, however, given a careful design of the system, it is not impossible. We will give a very restricted solution (that requires very little planning) in Sect. 8.4.3, but more elaborate systems exist for hot code update [SRG96].

Finally, the reconfiguration plan needs to be executed. Here, a lurking threat is the possibility of deadlocking the system or interrupting communication transactions because a *quiescent state* has not been reached. Avoiding this problem is part of the planning phase, but the framework usually needs a provision that allows for sufficient control over the reconfiguration process.

The distinction of multiple stages is a separation of concerns: one concern is to extract the relevant data from the application, another one is to come to a reconfiguration decision based on a stream of data, and so on. Using MAPE-style loops hence enforces a distinct consideration of each of the stages, facilitating the detection of reuse scenarios and providing a tidy application structure.

We will now investigate the first three phases in greater detail. The fourth phase – execution – is a straightforward implementation of the algorithm described in Chapter 6.4.

8.1.2. Monitoring. Monitoring of software often describes the process of interpreting a sequence of messages and finding out whether they indicate faulty behavior [BGHS04, BLS06]. In our context, this will be part of the assessment phase. Instead, monitoring describes the process of producing said sequence of messages, or, more broadly, data about the situation at hand. A wealth of means for doing so exists on various levels of granularity, from logging frameworks to elaborate introspection APIs like the JVMTI [PRRL04].

For monitoring, two dimensions need to be considered: The granularity and the immediacy of information. The former may range from very coarse system state change messages (e.g., system shutdown requested) down to bytecode-level information (as can be obtained by stepping through the program with a debugger). The latter can range from delayed – maybe by caching large portions of messages before writing them out, or by only reporting every nth occurrence of an event – to real-time synchronization, i.e., if the monitored information can be fully processed before the system state changes again, the example again being a debugger in step mode.

For components, the granularity should be fine-grained enough to identify the source of problems (i.e., should be at least at component level, and better at message level), but it should respect the component being a black box. Monitoring the communication at message level can be achieved either on the user-level by introducing *filter components* – an approach that was successfully employed in the CMC model checker; or alternatively, monitoring can be enabled by the component framework, an approach that we discussed in Sect. 7.2.5.1. The latter has the benefit that the immediacy can be improved, and in the JComp framework, communication observers are informed of a series of events concerning a single message, producing a fine granularity.

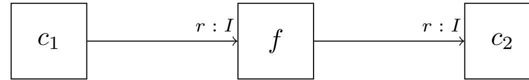


Figure 8.3: Filter component

8.1.2.1. *Filter Components.* Filter components enable monitoring entirely on the user level, but the pattern itself can be used in many more situations. A filter component is placed between two components and relays messages, possibly changing the message contents, suppressing some, or changing its own data state in the process. Syntactically, it is a component where one role has the type that is also provided; Fig. 8.3 illustrates this concept. It is perfectly acceptable to let filters have additional required roles, or additional provided interfaces: A component c is called a *filter component* if $\exists I \in I_P(c) \,.\, \exists r \in \mathscr{R}(c) \,.\, I_R(c)(r) = I$.

8.1.2.1.1. *Cross-Cutting Concerns.* Filter components are used to introduce *cross-cutting concerns* into a component system, similar to the concept of a *point-cut* in aspect-oriented programming [**KLM+97**]. A cross-cutting concern is a required functionality of a system (part) that is independent of its core functionality. A well-known example is logging (which, from the implementer point of view, is little different to monitoring): An application that needs to log some events tends to get interspersed with logging statements, regardless of the encapsulation chosen. Any change to the logging mechanism requires a modification of virtually all components[1]. Aspect-oriented programming (AOP) tries to solve those problems by weaving aspects, each handling a different concern [**KLM+97**], thus satisfying the separation of concerns paradigm. This is widely regarded as useful, though apparently the concept is more appealing in technical domains (like adding functionality for reconfiguration, as described in [**MSKC04b**]) than in normal programming. We have utilized AOP for artificially slowing down parts of generic Java programs in [**HKMR08**] and used filter components to do the same for the CMC model checker, where the parts to be slowed are given by the components.

For components, a cross-cutting concern like logging can be handled by a filter component. In Fig. 8.3, component c_1 might send data to component c_2, which consumes it. The core concern of the application – operating a producer/consumer scenario – is satisfied by the direct connection. However, if the data (or the mere occurrence of messages, which is more likely) needs to be logged, a filter component F can be introduced to handle this concern. Thus, a filter can be a substitute for a monitor as described in Sect. 7.2.5, but remains a component that can be subjected to reconfiguration. An interesting report on using filters, independently of components, can be found in [**BA01**].

In applications based on the JComp component framework, filters are used for logging (cf. Sect. 8.2.2), distribution (cf. Sect. 7.3) and user-level monitoring. We will frequently encounter filter components in Sect. 8.4, which is about responding to changed cross-cutting concerns by reconfiguration.

Note that the syntactical concept of a filter is not necessarily tied to a cross-cutting concern. Often enough, the implementation of the core concern

[1]This is hard to overestimate. With Java, a common mistake is to conduct a logging call like this: `logger.fine("Processing entry " + e);`. This results in building a new `String` object for each logging entry, regardless of whether the `fine` level is actually used. The correct code reads `if (logger.isLoggable(Level.FINE)) logger.fine("Processing entry " + e);`, and one easily gets the idea of what "cross-cutting" is all about if this has to be changed in an industrial application made up from a few dozen packages, with a few ten-thousands lines of code each.

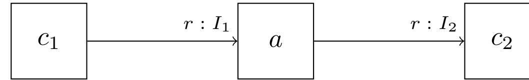


Figure 8.4: Adapter component

of an application utilizes filters, e.g., for applying effects to images (cf. the `JHLabsFilterAdapter` components in Fig. 7.5, which are components that wrap an image processing filter in a component). CMC utilizes application-specific filters frequently: The automated-atomic filter presented in Sect. 5.1.3, a hash value filter that can be used to store the hash value in the state representation and only calculate the hash value if it is not yet known, or the ZLIB compression presented in Sect. 5.1.2. However, such filters also address a concern that is slightly orthogonal to the core concern, and usually relates to performance optimizations.

We have discussed the message monitoring arrangements provided by the JComp framework in Sect. 7.2.5 and provided the various stages in Tab. 7.2. Having a precise information about which thread invokes the monitoring code is crucial, as for the duration of executing the monitor the component that yields the thread is blocked. If the monitored event leads to a reconfiguration, the planning phase might need precise knowledge about the state of the component that gets reconfigured. If this planning is done by the same thread that executed the monitor's code, the component's state cannot have changed in between (due to the mono-threadedness of JComp's components). Reconfiguration planning might now proceed to tailor a reconfiguration to exactly the state observed, and proceed to trigger the execution of the reconfiguration, all within the component's own thread.

The precision of this approach is limited, however. In accordance with the formal description, the JComp framework can only reconfigure a component that is not currently executing a method. Hence, even while the first three stages of the MAPE loop may run in the component's thread, the execution needs to have a thread of its own. This also means that the current method that, depending on the stage the monitoring was conducted in, will invariably be executed. Only if the check is done early enough (stage 1), reconfiguration can intervene before the message is processed. If this is not sufficient, reconfiguration adapter components can help – i.e., filter components that relay messages after checking their integrity, either within the adapter component's implementation or in the framework-provided monitoring, providing sufficient interception points to schedule a reconfiguration.

8.1.2.2. *Adapter Components.* A related concept are *adapter* components. The idea, illustrated in Fig. 8.4, is rather straightforward: In order to attach a component providing an interface I_2 to a component that requires a slightly different interface I_1 for one role, a component is built that provides I_1 and requires I_2. The actual implementation of this adapter component depends on how I_1 and I_2 differ. In easy cases, a simple renaming is sufficient. Sometimes, additional parameters need to be supplied. More complex adapters need to change the parameter values, or even change the communication protocol. Such complex components are sometimes referred to as *proxies*; they guard a component incapable of performing a certain complex protocol. We will encounter such proxy components in Sect. 9.2, where they will add concurrency control to a component that is ignorant of being accessed by multiple clients.

Generating such adapters, preferably automatically, is an interesting research topic, which is addressed by frameworks like UniFrame [**CBZ+02**], and plays an important role in the context of web services [**BCG+05**]. Here, we can use adapter

components to provide the necessary decoupling for reconfiguration. An example of such an adapter is presented in the example in Sect. 8.4.3.

8.1.3. Assessment. From the monitoring, we expect a stream of measurements. The assessment phase now needs to judge the necessity or utility of reconfiguration based on that stream. In many cases, this is easy to do – e.g., in JComp, if an exception is witnessed, reconfiguration needs to be done. Things get harder if runtime behavior is to be optimized, as the threshold where reconfiguration should be done is often dependent on the estimated future – usually, the question boils down to "will reconfiguration save enough resources during the remaining computation to warrant its expenses?". Often, this is viewed from a different perspective by considering "quality of service" (QoS) [**VZL⁺98, WSG⁺03, DP07**].

Different problems emerge for applications that try to maintain functional properties by reconfiguration. Here, the question is "did the recent behavior violate my properties? And if so, should we reconfigure, or should we just hope that these things will not happen again?"

Here, we will outline some means to answer these questions. These are examples for ways to handle the assessment phase, and are by no means exhaustive. Especially, in JComp, genuine physical sensors are not considered, which often require cleaning (e.g., smoothing or outlier removal) of the sensor readings in the assessment phase [**JAF⁺06**].

8.1.3.1. *Predicting the Runtime.* A reconfiguration is feasible if the improvement achieved outweighs the cost of the reconfiguration. But how do we calculate the "quantity" of improvement? Given a system that runs forever, any improvement no matter how small will save an infinite amount of time, so no matter how costly the reconfiguration is, it will eventually pay off. This will not apply to real systems for two reasons:

- The system will likely not run forever.
- Even if it does, it might change its characteristics and thus render the effect of reconfiguration useless.

Hence, an estimation of the remaining time or at least of the stability of a system is required. That this is difficult to do should be evident to everyone who ever compared the progression of a progress bar (for a file download or installation progress) with wall time.

The problem of estimating the remaining time of a software task is studied in the context of the "shortest remaining processing time" job scheduling paradigm, which suffers from the same problem (cf. [**HBSBA03**] for an example of a domain where the estimation is working well). In our experiments with the cmc model checker, we found that the size of the open set (i.e., those state that have not been visited but need to be) provides a rough estimation for the remaining time; Fig. 8.5 shows a "normal" model and the progression of the set sizes. After approximately 25% of the total states the open set peaks, it then declines rather slowly, until the final states are progressed rather quickly. Observing the open set peak provided us with a very rough estimation of the problem size; if we found the open set to be increasing for too long, we would judge the model to be infeasible. This was, however, severely sabotaged by a seemingly simple model which involved a simple counter that was nondeterministically reset to half its value. This resulted in a very small search front (two states), but the state space was still vast, and proved to be uncheckable due to the cache degradation.

8.1.3.2. *Concept Drift and Software Phases.* Concept drift [**SG86, NFM07**] is an idea from machine learning; it describes how a system can adopt to a changing

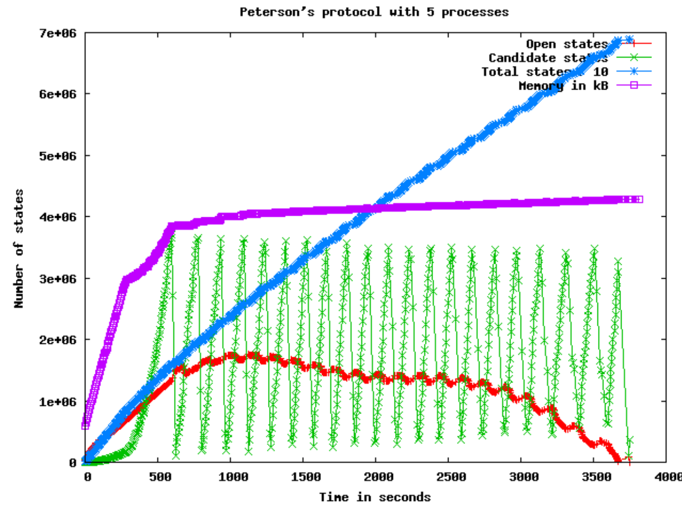


Figure 8.5: Progression of the CMC run of Peterson's mutual exclusion example, $n = 5$

world. If a learning system neglects the possibility of a changing world, it will respond to a perceived change by adjusting its world representation, and thus take very long to fully accept the changed reality. With concept drift, this lagging is reduced as changes are anticipated and incorporated into the representation rather quickly. Such a change in the world representation is not considered an adaptation to a hitherto imprecise believe, but rather an integration of a change.

Consider a SPAM filter that tries to adapt to new disguises of SPAM. When neglecting the concept drift, the filter would be required to learn that mails carrying a single image are most likely SPAM, and would need to reclassify those mails previously not detected to be SPAM as such, if they were subjected to another classification run. In this sense, the decision base of the filter is extended to include the new disguise of SPAM, and the justification for this extension would be that the criteria learned so far are insufficient. When considering concept drift, however, it is accepted that a formerly respectable form of email is now considered SPAM, due to a change in the approach taken by junk mail senders. The knowledge base used to classify mails can thus be *shifted* rather than extended: It may also "forget" about former SPAM criteria which have not been used for some time.

Whether to regard unexpected inputs to a classification algorithm – like a junk mail message disguised in a form that has hitherto been accepted as unsuspicious of being SPAM – as an error of the previous knowledge base (which then has to be extended) or as a hint towards a changing concept in the observed world (where the knowledge base needs to be shifted) is an interesting question in machine learning and believe revision (cf. [**KM91**]). In our context, where we want to adapt our system to a change in the environment, concept drift is the key approach towards describing perceived change.

We make a basic assumption here: At least for some problems, software operates in *phases*. Phases are supposed to be distinguished by a (hopefully rapid) change in the communication behavior of the components. Consider, for example, a database that is used to store the products sold by a company. Most of the time, this database will be queried. Sometimes, we can expect minor changes to a small amount of data (e.g., changing prices, inserting a small number of products

or product variations...). Every once in a while, the database will undergo a rather severe change when the company decides to release a new product line. A bulk of data is inserted in a rather short amount of time, during which the normal usage of the database is neglected, if not disabled. For many large relational databases, such a bulk insert – which we can regard as a distinct phase different from the normal phases – will require severe modifications to the database, like disabling consistency checks, indexing and trigger evaluation. This is usually done by the database administrator, and is required for performance reasons (which, in practice, can become a real problem, especially since the database cannot process regular queries during the bulk insert phase, or at least cannot handle them efficiently).

The assumption of phases may not apply to most data-processing algorithms, like search algorithms, with one exception: Quite often, such algorithms have distinct preparation and execution phases. During the preparation phase, data is read and placed into in-memory data structures. Adequate lookup tables are built. Eventually, the main algorithm is triggered, which presumably handles the data in quite a different fashion. Usually, the programmer takes care of making both phases efficient, but in the wake of reusable software components, this might become neglected (i.e., the same data structures are used for preparation and execution, and they are usually chosen to allow for an efficient execution). Sect. 8.4.4 describes an approach towards improving this situation by adaptively choosing suitable data structures for different software phases. Such an approach can utilize the theory of concept drift to identify the beginning of a new phase.

Let us assume that we operate a component setup and observe the messages received by the components. By aggregating them over a time window, we obtain a distribution of message frequency. Let us further assume that we have some estimate of the time consumed by the the processing of a message on a given component. (We will present an example of such an estimate in Sect. 8.4.4). We are then interested in calculating the future cost of the system *if it continues to perform as currently observed*, as well as the cost of the reconfiguration alternatives we have at our disposal. If the difference between the expected cost for the current setup is higher than the cost of a reconfigured setup plus the expense of the reconfiguration process itself (when considering only a limited interval of the future – otherwise, reconfiguration benefits will always outweigh the cost of reconfiguration under said assumptions), reconfiguration should commence.

There are two things to consider:

- The algorithm must not “jump to conclusions”, i.e., quickly initiate a reconfiguration with little gain, if the drift is not permanent (also known as *virtual drift* [**LVB04**]). More specifically, the expected benefit of reconfiguration should not be calculated for a constant time frame, but rather for a period that is modified according to observations.
- On the other hand, the algorithm must not delay a reconfiguration that is clearly useful. Some “freak message” might even require a reconfiguration prior to processing the first received message of a certain type, if that message type is too costly for its target component’s current type. This is especially true if the benefit of executing the single message in a reconfigured setup outweighs the cost of reconfiguration.

Thus, the choice to perform reconfiguration based on some recent observations is affected by the following parameters:

- The *window size*, i.e., the history taken into consideration when predicting the behavior of the system – it needs to be large enough to avoid being influenced by virtual concept drift too much, but small enough not to lag the reconfiguration decision.

- The predicted *stability*, i.e., the time that we expect the system to be within the currently observed phase – if chosen too high, even the slightest change in message distribution will trigger a reconfiguration, if chosen too low, no drift will ever produce a reconfiguration.
- The expected cost of reconfiguration, possibly including the impact on other components that are blocked due to waiting for synchronous messages to be processed – in a reactive component setup that waits for input most of the time, reconfiguration can be done in periods of less activity without impact on the system's performance, whereas in a heavy-duty setup reconfiguration might stall the system for an unacceptable period of time.
- The expected benefit of reconfiguration, which needs to consider the current state and the expected usage for the expected phase stability.

All these parameters can be pre-calculated (or set manually), but at least the window size and the stability should be adapted to the experienced behavior. Adapting the window size is performed by learners such as Flora2 [**WK92**]. The basic idea here is to decrease the window size if concept drift is suspected, while enlarging it if the data is inconclusive (and further data might be beneficial to decision processes). The idea is further refined in [**LVB04**], where multiple windows (actually, three) are employed to become more sensible towards different types of concept drift.

Stability can be derived from the frequency of reconfiguration decisions. It is usually not researched in the context of concept drift, which aims at learning facts rather than predicting the future outcome of an action, but there are some related concepts, like detecting a recurrent context [**WK96**].

Considering software phases can be used to adapt an application to a changed setting, allowing it to use the available resources in an efficient way. The definition of concept drift aims at situations where the behavior can be improved; in order to remain robust against virtual drift, some latency has to be accepted. This makes this approach less suitable for applications where some behavior needs to be prevented – even if this behavior is described with non-functional properties. We will now discuss a means to provide a assessment that is more direct and can response to functional and non-functional property violations as soon as they occur.

8.1.3.3. *Temporal Logic-based Assessment.* In some situations, detecting communication errors can be achieved using a simple state machine. In Chapter 9.1.2, we will discuss how formal component protocols can be established, the basic idea being that some messages can only be processed in some state, and that processing messages might change that state. Monitoring the communication and comparing it with the protocol can be used to assess whether a component behaves as required by the protocol. This amounts to monitoring a safety property; and these can be expressed in linear temporal logic (LTL) [**Pnu77, KM08b**]. Thus, we might formulate that receiving an `init` message is prohibited after we received a `terminate` message:

$$\texttt{terminate} \rightarrow \Box(\neg\texttt{init})$$

Such a property can be checked by storing whether `terminate` was received and then throwing an error if `init` is received afterwards.

But LTL can do more, and we might be tempted to formulate properties like

$$\texttt{getBalance} \rightarrow (\neg\texttt{getBalance})\ \textbf{until}\ \texttt{withdraw}$$

which is supposed to mean that every call of `getBalance` is followed by a `withdraw` message before a second call to `getBalance` is admissible. This is a liveness property that demands that the system eventually progresses in a desired way.

This leads directly to a problem with LTL: Although the formula makes perfectly sense for our protocol, LTL's semantics of $\varphi\,\mathbf{until}\,\psi$ requires that ψ is eventually made true. Obviously, we can never find out whether this aspect is violated; we can only check if φ holds as long as ψ does not hold. While such an examination is still useful, the formula $\Diamond\psi \equiv \mathbf{true}\,\mathbf{until}\,\psi$ ceases to make any sense in this approach.

Still, it might be useful to specify liveness properties for use in the assessment phase. For example, a component that requires that any `getBalance` is followed by an `withdraw` might also like to specify that this should only take a given amount of time. This introduces a non-functional requirement and requires real-time properties, and while we usually stay away from real-time considerations in this thesis (especially since early experimentation clearly indicated that calamity will result from imposing real-time requirements on components running in a framework that is itself oblivious to real-time, while allowing for concurrency), it might be interesting to specify how an assessment can be formulated in a uniform way.

In [**BLS06**], Leucker et al. propose a three-valued monitoring of LTL formulas, where the formula might either be judged with **true**, meaning that the witnessed communication sequence already proves the formula being true, or **false**, or ?, if the formula's holding or being false cannot be assessed from the sequence witnessed so far. They extend this approach (which, by the way, proved to be the vital inspiration to the CMC model checker's basic caching structure) to real time properties. In [**Bui08**], some effort was made to simplify the language used in order to facilitate monitoring with real-time properties.

Temporal logic-based assessment is a useful and very generic tool, especially if combined with real-time properties. However, in the context of this thesis, we again suffer from examples too artificial and too much governed by the reconfiguration execution; those that consider quantitative properties (Sect. 8.4.2 and Sect. 8.4.4) have their assessment hard-coded. For large-scale QoS considerations, real-time temporal logic will be an important ingredient in an attempt to further separate the concerns; but in this thesis, they do not play an important role.

8.1.4. Planning. Following the assessment phase, where the need for reconfiguration is determined, the planning phase commences. Based on information about the kind of problem at hand, a solution needs to be found. This phase can be regarded as the most difficult of the MAPE loop, if a generic approach is to be taken. It is easy to construct an argument that generic planning is entirely infeasible – e.g., by pointing out that replacing an algorithm with a suitable substitute necessitates a decision procedure for algorithm equivalence, which is known not to exist.

Reconfiguration can be seen as configuration with further constraints – especially constraints that rule out the situation that was just assessed as problematic. Planning a configuration, however, requires a precise semantical description of the component's communication requirements and strong reasoning capabilities. It has been attempted, and success has been achieved (e.g., in the AMPHION project [**SWL+94**], cf. Sect. 2.4.1). But the effort required is vast, and most likely only very specific domains can be addressed.

Instead, planning of configuration and reconfiguration often resorts to predefined strategies that are easy to attain. For example, the fault tolerance of ERLANG, as discussed in Sect. 7.5.3.5, requires little planning, because it just entails shutting down components, spreading the error and checking if the components can cope

with it. Also, every example we provide in this thesis is rather simplistic with respect to planning. Usually, a fixed set of "cures" is pre-calculated – even more, it is carefully crafted with an effort often far exceeding the effort of writing the initial application. The assessment of an error then usually determines the plan that needs to be chosen. There is also some work that aims at generating plans from a fixed set of component substitutions, e.g., in [**PCDJ08**] an approach towards generating plans from atomic component substitutions is proposed.

At the granularity level considered in this thesis, which is considerably lower than that of many other works, real planning seems abundantly complicated when considering the necessity to adhere to the problems of state transferal, quiescent states and the like. It appears that predetermining the reconfiguration possibilities is the best we can do for now. It is, however, still a challenging task to write down these alternatives and generalize them so that they become more like patterns, instead of "single-situation only" plans. Architecture Description Languages (ADLs) can be extended for this purpose [**Med96, MBC03, MBC04**], or domain specific languages can be defined [**WLF01**], but we utilize the more prevalent means of graph transformations [**DD98, WF99**].

Another interesting field of planning is the choice of an optimal plan among a set of alternatives [**Ars03, AHW03, CEM03, KM07, REF+08**] or the selection of a suitable single component substitute [**DMM04**]. This is sometimes complemented with the choice of values for the component parameters, e.g., with a setting of a "sending power" parameter that affects the dependent variable "power consumption" for a broadcasting device [**WDT+07**]. Finding an optimal (or, and often preferably, sufficiently suitable) solution can be done by using microeconomic mechanisms like auctioning schemes [**CEM03**] or by soft constraint solving [**WDT+07, HMW08**].

The suitable planning approach needs to be chosen for the kind of problem that needs to be solved by reconfiguration. In this thesis, planning is not given much elaboration; instead, the examples presented in this chapter usually employ only simple plans that are constructed explicitly. Here, we will proceed to introduce graph transformations, which offer a means to concisely denote reconfiguration plans.

8.1.4.1. *Graph Transformation.* Graph transformations [**EEPT06**] have two properties that make them interesting for defining reconfiguration plans: First, they use a visual metaphor and are easy to understand. Second, they can be defined so that they apply to more than one given situation: A graph transformation might be as abstract as "insert a filter component between a component satisfying property φ and a component with property ψ". If the assessment looks out for situations where a filter component needs to be added, we can use this graph transformation to plan the reconfiguration for any scenario. Thus, plans can be devices for a variety of situations, and might even apply to applications not yet designed (cf. [**DD98**]).

Using graph transformations is especially useful if cross-cutting concerns are to be addressed: If, for example, a logging filter component is to be added between a producer and a consumer component, we might not be particularly interested in the remainder of the component graph, or how many such producer-consumer setups are found in the application.

A graph transformation consists of two parts:

- A *left-hand side*, which defines to which part of a graph the rule matches, and
- a *right-hand side*, which describes how the matched part should be altered.

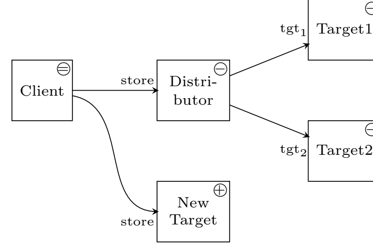


Figure 8.6: SPO graph transformation rule example

It is technically feasible to handle both the left- and right-hand side of a graph in the same data structure. For defining graph transformations, we utilize *single push-out* (SPO) rules, as utilized by the GROOVE tool [**Ren03, Ren04**]. Instead of using two rules that describe addition and removal of nodes, a single rule is used that *attributes* the graph with “keep” or “remove” annotations.

Fig. 8.6 shows an example of such a rule. The nodes are annotated with $\circledequal$ (match), $\ominus$ (remove) and $\oplus$ (add). The rule matches a graph where the $\circledequal$- and $\ominus$-annotated nodes are present and connected as described. It produces a graph in which

- all the $\circledequal$- and $\oplus$-annotated nodes are present,
- none of the $\ominus$-annotated nodes is present,
- all connections originating or ending in a $\ominus$-annotated node are removed,
- all other connections are present.

More formally, an SPO rule is a graph $R = (V, E, \alpha, \omega, \lambda)$ over $\Sigma = VL \cup \mathscr{R}$ for $VL = \{(C, F) \mid C \subseteq \mathscr{C} \wedge F \in \{\oplus, \ominus, \circledequal\}\}$ with $\forall v \in V \,.\, \lambda(v) \in VL \wedge \forall e \in E \,.\, \lambda(e) \in \mathscr{R}$. For $l = (C, F) \in VL$ and $c \in V$ with $\lambda(c) = l$, we write $C(c)$ for C and $F(c)$ for F. The set VL of labels uses pairs of sets of component identifier and a node annotation; the set of component identifiers is used to define the set of components the graph rule node can be matched to.

We require that $\forall e \in E \,.\, F(\alpha(e)) = \ominus \rightarrow F(\omega(e)) \neq \oplus$, i.e., no edge may connect a $\ominus$- and a $\oplus$-labelled node – the opposite, i.e., connecting a $\oplus$-labelled node with a $\ominus$-labelled node, is acceptable only if the role is a temporary role (cf. Sect. 6.8.1). This is the only way temporary roles can be connected:

$$\forall e \in E \,.\, \lambda(e) \in \mathscr{R}_{temp} \rightarrow (F(\alpha(e)) = \oplus \wedge F(\omega(e)) = \ominus)\,, \tag{R1}$$

$$\forall e \in E \,.\, \lambda(e) \in \mathscr{R}_{perm} \rightarrow (F(\alpha(e)) \neq \oplus \vee F(\omega(e)) \neq \ominus)\,. \tag{R2}$$

Hence, a regular role may connect $(\oplus, \circledequal)$, $(\oplus, \oplus)$, $(\circledequal, \oplus)$, $(\circledequal, \circledequal)$, $(\circledequal, \ominus)$, $(\ominus, \circledequal)$ and $(\ominus, \ominus)$-labelled pairs of nodes. The first three will result in edge addition, $(\circledequal, \circledequal)$ is just a match and will not change the graph, and the last three will result in edge removal. $(\ominus, \oplus)$ is ruled out. $(\oplus, \ominus)$-labelled pairs can be connected by temporary edges.

We also require that the plan preserves well- and completely-connectedness:

- $\forall e \in E \,.\, \lambda(e) \in \mathscr{R}(C(\alpha(e))) \wedge I_R(C(\alpha(e))(\lambda(e)) \in I_P(C(\omega(e)))$, i.e., edges connect existing roles to provided interfaces,
- $\forall v \in V \,.\, F(v) = \oplus \rightarrow \forall r \in \mathscr{R}(C(v)) \,.\, \exists e \in E \,.\, \alpha(e) = v \wedge \lambda(e) = r$, i.e., for $\oplus$-labelled nodes, all (permanent and temporary) roles are connected,
- If an edge points to a component that is to be removed, another edge with the same role label exists that points to a new one (it has to be a new

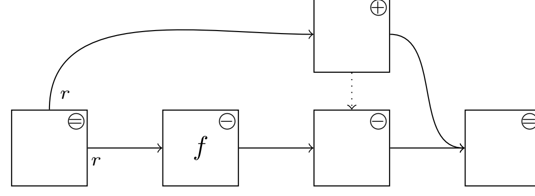


Figure 8.7: Graph transformation rule for filter removal

one, due to the definition of ρ in the reconfiguration plans):

$$\forall v \in V \,.\, F(v) = \circeq \rightarrow \forall e \in E \,.\, (\alpha(e) = v \wedge F(\omega(e)) = \ominus \\ \rightarrow \exists e' \in E \,.\, \alpha(e') = v \wedge F(\omega(e)) = \oplus \wedge \lambda(e) = \lambda(e')) \tag{R3}$$

- $\forall v \in V \,.\, \forall e \in E \,.\, \alpha(e) = v \rightarrow \forall e' \in E \,.\, \alpha(e') = v \wedge \lambda(e) = \lambda(e') \rightarrow (e \neq e' \rightarrow F(\omega(e)) \neq F(\omega(e')))$, i.e., every role is connected only once to a target node with the same annotation.

Given a rule $(V, E, \alpha, \omega, \lambda)$ and a component setup (C', M', e'), a *match* is a (total) function $m : V \rightarrow C'$ such that

- $\forall v \in V \,.\, F(v) \in \{\circeq, \ominus\} \rightarrow m(v) \in \lambda(v)$,
- $\forall e \in E \,.\, (F(\alpha(e)) \neq \oplus \wedge F(\omega(e)) \neq \oplus) \rightarrow M'(m(\alpha(e)))(\lambda(e)) = m(\omega(e))$,
- $\forall v \in V \,.\, F(v) = \ominus \rightarrow \forall c' \in C' \,.\, \forall r \in \mathscr{R}_{perm} \,.\, r \in \mathscr{R}(c') \wedge M'(c')(r) = m(v) \rightarrow \exists e \in E \,.\, \omega(e) = v \wedge C(\alpha(e)) = c' \wedge \lambda(e) = r$,
- $\forall c \in C' \,.\, \left|\bigcup_{v \in m^{-1}(c)} F(v)\right| = 1$.

The last requirement states that if a component matches more than one rule node (which is perfectly acceptable), the annotation has to be the same for all rule nodes matched. Otherwise, a $\ominus$- and a $\circeq$-labelled rule node might become matched to the same component. The third requirement states that if a node is to be matched to a $\ominus$-labelled node, then all the incoming connections need to be explicitly mentioned in the graph rule – in order to preserve completely-connectedness of other components which are not mentioned in the plan. The other requirements state that all $\circeq$- and $\ominus$-labelled nodes are matched, and that the edges are present as required.

Given a rule $(V, E, \alpha, \omega, \lambda)$ and a match m, we can build an extended reconfiguration plan $\Delta = (A, R, \alpha', \rho, \delta, \varsigma)$ by setting

- $A = \{m(v) \mid F(v) = \oplus\}$,
- $R = \{m(v) \mid F(v) = \ominus\}$,
- $\alpha' = \{c \mapsto \{r \mapsto c' \mid \exists e \in E \,.\, m(\alpha(e)) = c \wedge m(\omega(e)) = c' \wedge \lambda(e) = r\} \mid c \in A\}$,
- $\rho = \{(c, r) \mapsto c' \mid \exists e \in E \,.\, m(\alpha(e)) = c \wedge F(\alpha(e)) = \circeq \wedge m(\omega(e)) = c' \wedge F(\omega(e)) = \ominus \wedge \lambda(e) = r\}$. Note that no edges representing temporary roles are considered, due to the requirement R1 and that all permanent roles are covered due to requirement R3,
- $\tau = \{(c, r) \mapsto c' \mid \exists e \in E \,.\, m(\alpha(e)) = c \wedge F(\alpha(e)) = \oplus \wedge m(\omega(e)) = c' \wedge F(\omega(e)) = \ominus \wedge \lambda(e) = r\}$. All edges considered represent temporary roles due to requirement R2.

Graph transformation rules are appealing because their informal meaning is easy to understand, and it can be used to describe reconfiguration for a variety of situations. For example, the rule shown in Fig. 8.7 can be used to remove any filter component from a set $f \subseteq C'$ from the connection they instrument. By

repeatedly searching for matches and applying the resulted plan, all the existing filter components of the set f are removed. This “relaxing” of the reconfiguration description (i.e., being able to describe a reconfiguration such that it applies to a large number of situations, and not just a single one) is an important prerequisite for building systems that use repeated and varied reconfiguration, as we will discuss in Sect. 8.3.

Note that a design decision of the JCOMP model reconfiguration is reflected by graph transformations as we defined them: The co-domain of ρ is A, and the domain is $(C \setminus R) \times \mathscr{R}$ with the requirement that each pair ρ is defined for points to a component in R. The same requirement is made by the definition of the graph transformation rule. If ρ was to be given an extended co-domain, we would have to provide labels for the graph transformation edges, too: If an edge between two $\ominus$-labelled nodes was to be redirected (e.g., for the insertion of a filter without removing the target node), it would have to bear a special label. Although we suffer from the same necessity to remove and re-add more nodes than strictly necessary, the graph transformation remains more tractable in its definition this way.

It should be noted that our examples in this chapter do not use graph transformations to define the reconfiguration plans; this should be attributed to the limited scope of the examples. Many of the examples could benefit from graph transformation rules, if they were to be considered in a large-scale component application that is subjected to various reconfigurations that try to maintain the different concerns.

8.1.4.2. *Planning and Quiescence.* Besides defining *how* reconfiguration should modify the component graph, planning also needs to determine *when* the reconfiguration should commence. We have seen that the characteristics of the plans we use make reconfiguration quite robust against concurrent changes. This ceases to hold if we do not use injective shallow reconfiguration plans (an example is given in Sect. 8.3.2) or if the state transferal is insufficient for picking up a transaction at precisely the point where the old component left it when becoming detached.

8.1.4.2.1. *Quiescence.* Transactions are the key concern of the concept of *quiescence*, the most influential theoretical consideration of reconfiguration [**KM90**, **MGK96**]. In [**KM90**] Kramer and Magee, building on a general solution to multi-way synchronization due to Chandy and Misra [**CM84**], formulate four conditions that need to be given for a component to be in a quiescent state:

(1) *It is not currently engaged in a transaction that it initiated,*
(2) *it will not initiate new transactions,*
(3) *it is not currently engaged in servicing a transaction, and*
(4) *no transactions have been or will be initiated by other nodes which require service from this node.*

Once a component (or, using Kramer's and Magee's notion, a node) has reached a quiescent state, it can be reconfigured without running into problems with concurrent communication, since it is guaranteed that no such communication will take place. Obviously, a component cannot guarantee being quiescent on its own, since the “other nodes” that might initiate a transaction need to be considered also. The solution presented in [**KM90**] is to calculate a *lock set* of components that need to be passivated (i.e., blocked in a way that disallows the initiation of transaction, but allows the handling of transaction requests issued by other components). Obviously, such a lock set can be quite large, and, in extreme cases, contain all components of the application.

8.1.4.2.2. *Tranquility.* Vandewoude et al. proceed to provide the weaker concept of *tranquility* [**VEBD07**] (in the sense that quiescence implies tranquility), which

aims at component substitution (intended to support hot code updates). The last two requirements of quiescence are replaced as follows:

(3) *it is not actively processing a request,*
(4) *none of its adjacent nodes are engaged in a transaction in which this node has already participated and might still participate in the future.*

Tranquility does not require that components connected to the component under observation do not start new transactions. These new transactions, however, cannot be participated in, but the processing must be delayed until after the reconfiguration – which is possible as the messages are retained in Vandewoude's model, as it is (by the δ plan element) in our approach. Hence, tranquility is sufficient for updatability, as a transaction that is initiated after the reconfiguration will continue with the updated component; only an interleaving of a transaction and a component update is ruled out.

Our approach ensures the second and third requirement of tranquility: A new transaction cannot be initiated, and no request is processed, since reconfiguration can only commence if the component is not executing a method invocation. The first and forth point are required to ensure that reconfiguration does not interfere with an ongoing transaction, since this might lead to inconsistent behavior: If the transaction-local data is lost, an ongoing transaction cannot be completed, or the response might be inconsistent.

As an example, consider a component that wraps a cryptographic service. It requires an initialization (which is used to set the keys) and then services encryption/decryption requests. A transaction hence begins with an initialization and contains a number of cryptographic requests, after which it is concluded by a logout message. If this component becomes updated while still servicing transactions, and the transaction-local data (i.e., the keys) is lost, subsequent cryptographic requests will appear as "out-of-protocol" to the updated version, as it is unaware of the foregoing initialization call. This problem is addressed by the forth property of tranquility (and also covered by the forth property of quiescence, although it is more restrictive than required here – for reaching quiescence, all clients need to be made passive, whereas, for tranquility, only the absence of ongoing transactions needs to be ensured). The first property of quiescence/tranquility is required to prevent clients that are in the middle of a transaction from becoming reconfigured and subsequently starting a new transaction without ever having sent a logout message. Obviously, state transferal can mend both problems: If the keys of ongoing sessions are retained during an update of the cryptographic-service component, it can pick up the transactions transparently (cf. Theorem 6.2; we will discuss state retainment as a replacement for tranquility in Sect. 9.6.2). Likewise, if the client retains information about being in the middle of a transaction, it will not start a new one. Whether such a state retainment is possible and feasible is very situation dependent; we would like to point out that having transaction information and the ability to wait for a tranquil point in time (or, as a fallback, to a quiescent state, if a tranquil state cannot be attained [**Van07**]) is also not likely; whereas state retainment needs to be done for transaction-independent data anyway.

If transaction-local state transferal is not desired or possible, however, the assessment and planning phases need to ensure that reconfiguration happens at a suitable point in time, which possibly needs to be procured. For doing so, the component communication behavior needs to be made explicit, as discussed in Chapter 9.1. We have not explicitly defined transactions so far, but they can be included easily, either by explicit declaration (i.e., extending the method specifications by an annotation about which transactions are begun and which are ended),

or by using implicit information, like the requirement of receiving a future message, which we will introduce with the *Eve* element in Sect. 9.1.2.1.

We will further investigate this problem in Sect. 9.6.1. As far as the planning phase of a MAPE loop is concerned, it needs to make sure that reconfiguration is conducted in a way that preserves the communication expectations of all components. In our framework, the means to achieve this are not fixed; the reconfiguration might provide a sufficient state transferal, or wait for a quiescent or tranquil state. As usual, the choice of the best approach depends on the problem at hand. It might even be possible to ignore transactions, if they can be shown not to interfere with reconfiguration, as the following approach suggests.

8.1.4.2.3. *Protocol Analysis.* In [**AP05**], static analysis of protocols, combined with locking, is proposed as a way to guard reconfiguration against interfering communication: By analyzing component communication protocols, the atomicity of component updating is ensured. This is made possible by including the component update requests (that trigger the reconfiguration) in the communication protocol, which can then be checked for a given setup. If the reception of a message during processing of a reconfiguration is possible, the component update cannot be guaranteed to be atomic, requiring additional locking.

Message reception is no issue for our approach, as it is no issue for tranquility. The fourth property of tranquility only works if a component is impeded from picking up the servicing of another transaction, which means that the component issuing this transaction needs to wait. Hence, a framework supporting tranquility needs to provide means for message retainment; quiescence is required if messages are allowed to be lost during reconfiguration. The approach of static protocol analysis can be extended to see whether tranquility considerations need to take place (cf. Sect. 9.6.1). Another prospect is to relax the reconfiguration if no interfering communication is possible: The reconfiguration framework might then do steps in parallel or mix them up. It might become necessary to look into such possibilities once reconfiguration needs to adhere to real-time constraints (i.e., finish within a given latency time).

8.2. Hot Code Updates

In the remainder of this chapter, various examples for conducting reconfiguration within the MAPE framework described above are presented. The first examples are about *live* or *hot code updates*. This is a well-researched and relevant use of reconfiguration, and usually suffices to motivate the provisions for reconfiguration in a component framework [**Hof93, Van07**]. It is also an area where the necessity for MAPE-style reconfiguration preparation is not required, since it is entirely user-triggered. Hence, hot code update investigates the pure execution of reconfiguration, making it an important benchmark example for the capabilities of a component model capable of reconfiguration.

Hot updating of components is well understood, and often a design goal for a programming language (e.g., Erlang [**Arm07**] or CLOS), but hot code update is also researched for programming languages that do not directly support it (e.g., C [**NHSO06**] or Java [**AR00**]; cf. [**HG98**] for an overview). While results are often encouraging, the process of hot code updating is dangerous and in the long run infeasible without explicit language or framework support [**EVDB05**]. It should be mentioned that the reconfiguration of the JComp model was first developed for the necessities of hot code update, which explains some of the particular choices made, especially the definition of the reconfiguration plan element ρ.

In utilizing a component framework for hot code updates, it can be hoped that the granularity is kept at a more manageable level than when modifying source

code directly. Furthermore, the greater independence of components is useful for explicit state retainment and more elaborate features like fault-tolerance [SRG96].

8.2.1. Single Component Hot Code Patching. In this first example, we will show how components can be updated by an administrator. This is rather straightforward, as the plan can be computed automatically, and reconfiguration can commence any time.

8.2.1.1. *Monitoring and Assessment.* For hot code update, the assessment is completely left to the user, who has to decide whether a code update is required. Obviously, such an assessment needs to be based on an observation of the system, requiring monitoring facilities that are either provided by the framework as described in Sect. 7.2.5, added at assembly time in the form of filter components as described in Sect. 8.1.2.1 or dynamically introduced at runtime by reconfiguration. Nevertheless, as far as the code update by reconfiguration is concerned, no assessment on behalf of the system is required.

8.2.1.2. *Planning.* This example only requires two parameter values: The identity id of the component to be updated, and the code to update it with. Let $c = (id, I_P, I_R, \zeta, \iota)$ be the component to be updated, and $c' = (id', I_P, I'_R, \zeta', \iota')$ be a new component that has the same provided interfaces, but differs in the implementation ζ' (it may also differ in ι', but this is not important here). The permanently required interfaces of c' are the same, but c' requires an additional temporary role $r_\tau \in \mathscr{R}_{temp}$, i.e., $I'_R = I_R \cup \{r_\tau \mapsto I_{data}\}$ for an interface $I_{data} \in I_P$. This interface is devised to obtain a copy of the state during hot reconfiguration, so the component c needs to have been prepared with suitable state accessor methods.

Given a component setup (C, M, e) with $c \in C$ and $c' \notin C$, we can calculate an extended shallow reconfiguration plan $\Delta = (A, R, \alpha, \rho, \tau)$ by setting

- $A = \{c'\}$,
- $R = \{c\}$,
- $\alpha = \bigcup_{r \in \mathscr{R}(c')} \{(c', r) \mapsto M(c)(r)\}$,
- $\rho = \bigcup_{s \in C} \bigcup_{r \in \{r' \in \mathscr{R} \mid M(s)(r') = c\}} \{(s, r) \mapsto c'\}$, and
- $\tau = \{c' \mapsto \{r_\tau \mapsto c\}\}$.

This plan is straightforward: The roles of c' are connected to those components that were connected to by the same roles of c, and those roles pointing to c from other components are changed to point to c'. Note that this plan Δ is not an injective shallow reconfiguration plan, as ρ is not injective in general. ρ obeys the shallow plan restrictions, however.

If the state retainment saves the entire state of c, and if the communication specification is not changed in c', reconfiguration can be started anytime. This is because possible the state of unfinished protocol sessions are retained, too, and c' can pick up the communication where c left it. If the communication protocol changes in c', inconsistencies cannot be ruled out, and c' needs to be in a tranquil state to be updated safely.

8.2.1.3. *Execution.* As the plan is not injective, message IDs are required to obtain an identical ordering of the queue (actually, the ordering is not guaranteed to be identical if assignment of a message ID and its reception from the queue is not done in an atomic step, but no-one will be able to tell the difference). However, only in situations we consider as artificial an actual problem will emerge from method reordering – as the order of the message sent over the same role is not changed. During reconfiguration, the state is retained by executing the m_{copy} method, which needs to be implemented by the component author of c'. Since we may assume that the necessity of introducing this component into the system by hot code update

reconfiguration is known to this person, no genuine breach of separation of roles is given.

8.2.1.4. *Experience and Discussion.* In JComp, hot code update is not directly built into the framework. Instead, a small example project was designed to show the capabilities of hot code update. Two challenges need to be solved:

(1) The administrator requires a means to trigger the reconfiguration. This was solved by adding a small server functionality that is accessible by Telnet (after this proved to be a benign approach with cmc, see Sect. 5.5). The administrator can use simple commands to query the components present in the system and issue update commands.
(2) The bytecode of the new component version needs to be loaded. Java allows for reloading class definitions, if a different class loader is used for the new class. This is possible due to the loose coupling of components: The actual class of a component is never referenced, other than within the setup generation code (that establishes the component setup by doing calls to the assembly). Hence, the component class can be loaded on the fly and used to generate a new component for the reconfiguration plan:

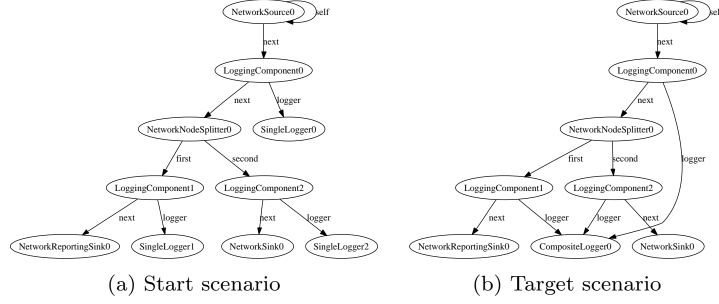
```
1 Class<?> ncc = new HotCodeUpdateClassLoader().load(compclass);
2 ComponentDescriptor nc = assembly.getComponentForClass(ncc);
3 plan.addNewComponent(nc);
```

State transferal needs to be implemented by the programmer of the replacement component. If the requirement to do hot code updates was known at the implementation time of the old component, it is quite easy to have this old component store its state in a Memento object and retrieve this object in the implementation of m_{copy}. If such a Memento provision is not given, the new component needs to utilize the available getter methods. Since they might not give sufficient access to the state (a method might change parts of the state that are not accessible by getter methods and hence not allowed to be read during reconfiguration), it is advisable to employ this technique only with components that are prepared for the task. A direct state retainment as employed in [**RS07a**] or [**Van07**] is not attempted, since it would breach the encapsulation of components and require additions to the JComp framework that would violate its basic design principles.

It is beyond the scope of this thesis to discuss the utility of hot code updates in component-based applications; we have no experience with applications that truly require hot code patching. However, it is well-documented that there are application domains like telecommunication (cf. [**PS07**], which provides an interesting story on how hot code update was used to maliciously wiretap telecommunication equipment) that require hot code update, and it is quite likely that such applications might benefit from a reconfiguration-based approach like the one presented here.

8.2.2. Architecture Update. Sometimes, it is not only required to update the code of a component, but also change the surrounding architecture to accommodate necessary changes to the required or provided interfaces. For example, the updated code might rely on an additional role to which part of the former code's functionality has been exported. Such updates, however, are easily handled by our reconfiguration mechanism.

It gets a little more tricky if a code update stretching over multiple components mandates a *consolidation* of functionality. An example for such a scenario is shown in Fig. 8.8a. The component setup depicted is a graph of components that keep sending asynchronous messages (neither the kind of messages sent nor the fact that



(a) Start scenario (b) Target scenario

Figure 8.8: Hybrid reconfiguration – logging example

the graph is acyclic are important). The edges of the original graph have been instrumented with logging components, which report each message to log store components.

Now, a system update might amount to replacing all these single log store component with a combined log store component. For example, the single log components might all have been storing the log entries in memory, thus depleting it eventually. An improvement of this system might amount to just have one component write to the log file, requiring that all log messages are stored in this component. (For the sake of the example, the actual benefit of such a replacement should be taken for granted here.)

Like for the previous example, monitoring and assessment are handled by the user.

8.2.2.1. *Planning.* For this reconfiguration example, the user has to provide a shallow reconfiguration plan $\Delta = (A, R, \alpha, \rho, \tau)$. Since we are going to employ hybrid state transferal, the τ element of the plan does not have to correspond to real temporary roles of the components in $A - \tau$ rather describes the roles that are to be generated for the temporary connection (cf. Sect. 6.8.2). Since these roles are not found in the components of A, we need a function $I_\tau : A \times \mathscr{R} \to \mathscr{I}$ that tells us for which interfaces the temporary roles should be generated. Additionally, for each $a \in A$, a script defining the creation of the Memento object needs to be provided. In the actual JComp implementation, we use JavaScript as it integrates well with recent versions of Java.

Hybrid state transfer requires two stages: In the first stage, the user-supplied plan is executed, but instead of adding the components $\{a_1, \ldots, a_n\} = A$ directly, temporary components $t_{a_1}, \ldots, t_{a_n}$ are generated and added in their place. It is the purpose of a temporary component t_{a_i} to run the script that queries the old component (as defined by $\tau(a)$) and builds a Memento object suitable for consumption by the actual component a_i. The temporary components do not have permanent roles, but may have a number of temporary connections that an be utilized by the Memento builder script. Also, the temporary components need to provide the interfaces the component they precede provides, but they interpret all methods by fail and thus block themselves immediately as described in Sect. 7.5.2. Thus, for the reconfiguration plan Δ and a component $a \in A$, we can build the component t_a as the tuple

$$(id_{t_a}, I_P(a), \{r \mapsto I_\tau(a, r) \mid r \in \mathrm{dom}(\tau(a))\}, \{m \mapsto \mathrm{fail} \mid m \in \mathscr{M}\}, *)$$

for an arbitrary $* \in \mathcal{S}$.

In the second stage, each temporary component t_{a_i} is replaced by the actual component a_i, which needs to fetch the Memento object from t_{a_i} over a temporary role r_τ and consume it during the execution of its m_{copy} method; since the definition of t_{a_i} is known at the time the Memento building script is written, this is not very hard to implement. Note that the first stage utilizes direct state transferal, whereas the second stage uses the indirect approach.

The actual plans of these two stages are similar, and can be derived from the user-supplied extended shallow reconfiguration plan $\Delta = (A, R, \alpha, \rho, \tau)$ as $\Delta_1 = (A_1, R_1, \alpha_1, \rho_1, \tau_1)$ and $\Delta_2 = (A_2, R_2, \alpha_2, \rho_2, \tau_2)$ by setting

- $A_1 = \{t_a \mid a \in A\}$ and $A_2 = A$,
- $R_1 = R$ and $R_2 = A_1$,
- $\alpha_1 = \{\}$ and $\alpha_2 = \alpha$,
- $\rho_1 = \{(c, r) \mapsto t_a \mid \rho(c, r) = a\}$ and $\rho_2 = \rho$,
- $\tau_1 = \{t_a \mapsto \{r \mapsto c \mid \tau(a)(r) = c\} \mid a \in \mathrm{dom}(\tau)\}$ and $\tau_2 = \{a \mapsto \{r_\tau \mapsto t_a\} \mid a \in A\}$.

Both plans are identical, except that for the first plan, the components of the set A are replaced by temporary components, and that the second plan uses only temporary connections between the new components and their temporary forerunners.

8.2.2.2. *Execution.* A two stage reconfiguration with hybrid state transfer is employed to reconfigure the system to the final configuration shown in Fig. 8.8b. The interesting work is done in the first reconfiguration, where the execution of the m_{copy} message executes the user-supplied JavaScript to build the Memento object. This Memento is stored in the temporary components. The second reconfiguration merely copies this Memento to its destination component, which sets its state according to its contents.

Since the plan is not injective, message ordering needs to be done. In this particular example, this is not really required – the order of the non-processed log messages is not exactly important.

A problem similar to the message ordering requirement emerges during the time-span between the first and the second reconfiguration step. During this time, the temporary replacement components must not consume messages, since they are mere placeholders and cannot give a meaningful semantics to the messages (in theory, they might, but then the automated generation of these placeholders would only be possible by adding much specification – which is irreconcilable to the separation of concerns paradigm). In this example, we can use message postponing as described in Sect. 7.5.1; as the order of the messages does not really matter to the logger, they can become shuffled if need be. The requirement to restrain the component from message processing is a typical artifact of multi-stage reconfiguration, however, with many examples not admitting a loss of message order. This is why component blocking, as described in Sect. 7.5.2 was introduced.

8.2.2.3. *Experience and Discussion.* The example, as displayed in Fig. 8.8, was implemented in JComp. Technically, the challenges are not too vast. The temporary component can be built from the plan by declaring required roles for each connection defined by τ_1. In the m_{copy} implementation, each of the temporary connections is made available to the JavaScript script. For the example of Fig. 8.8, the script displayed in Fig. 8.9 was used. Here, the data objects obtained from the old components (lists of log entries) are modified to match the requirements of the new component (i.e., having all log entries in chronological ordering). The `memento`

```
1 importPackage(Packages.jcompexamples.reconfiguration.hybrid.data)
2
3 m = new LogEntryMemento()
4 m.addEntries(access0.getLogMessages())
5 m.addEntries(access1.getLogMessages())
6 m.addEntries(access2.getLogMessages())
7
8 java.util.Collections.sort(m.getLogEntries(), LogEntryComp.getInstance())
9 memento.setContent(m)
```

Figure 8.9: Example JavaScript code for state retainment

variable points to a wrapper object, whose content can be set by the script as required. The choice of the `LogEntryMemento` instance as content of the Memento is governed by knowledge about the requirements of the target component.

This example (which was made up as an example for non-injective reconfiguration plans) suffers from similar problems as the previous one: While illustrating capabilities of the component model, it cannot give a credible report on its utility. Nevertheless, the ability to support such user-triggered system architecture updates is a prerequisite for doing more automated adaptations.

Also, the hybrid state transferal appears like a good compromise between the separation of concerns and the technical requirements. Separation of concerns is largely preserved: The old component only needs to provide sufficient getter methods, and the new component needs to provide facilities to consume a Memento instance. The content of the Memento is defined by the new component alone, and the responsibility to create a suitable object is given to the reconfiguration designer. This Memento requirements might even be specified in a way that allows the reconfiguration designer to built the Memento without knowledge about the internals of the new component. At the same time, all this is practically implementable in a reasonable component model, without any extensions required.

Fig. 8.10 illustrates an interesting aspect of reconfiguration that is otherwise neglected in this thesis: The runtime impact of reconfiguration execution. The plot shows the difference between between the time of the monitoring of a message and the time of the processing of the corresponding log entry in the log store component on the y-axis. On the x-axis, the reception time at the log store component is shown. At 2 seconds, reconfiguration commences; it takes roughly two seconds, in which time no messages are received by the log store component. Afterwards, messages transported during reconfiguration are processed; the throughput of the log store component is quite good and pending messages get consumed at a higher rate as new messages are received. Since no message ordering is done, the queues of the temporary components are copied one after the other, so that messages added to the first block just before reconfiguration have a higher latency as messages added at the end of the reconfiguration to the last block, although they are processed later; this explains the spikes seen from 4 to approximately 4.5 seconds. After that, the messages that have been send to the composite log store component directly are processed, until at 5.3 seconds, the queue is depleted and messages are consumed with no additional latency.

Keeping a bound on this latency is discussed in works like Cinema [**RBH94**] or Djinn [**MNCK99**], but they use different communication paradigms. In the component applications we tried with the JComp framework, reconfiguration often consumes a multiple of the time required for normal message passing, often because components or framework-related classes need to be generated. In this example,

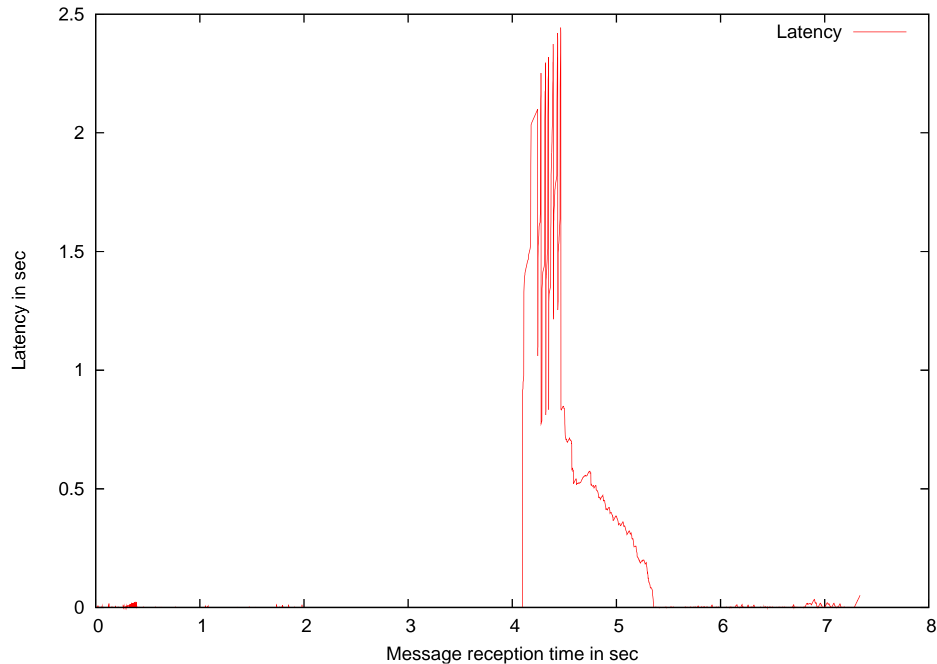


Figure 8.10: Impact of reconfiguration on an asynchronous system's message latency

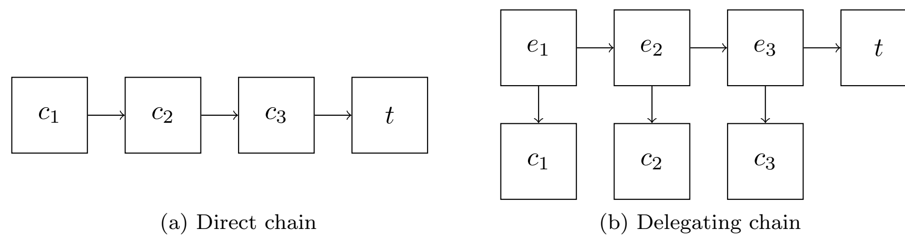
the generation of the temporary components and the execution of the JavaScript is far from being cheap, and we assume that the aim of keeping reconfiguration as atomic as possible conflicts with latency reduction too much to consider both.

8.3. Handling External Changes

Responding to external events that signal a change of the application's environment provides an interesting application of reconfiguration and is found as an example in many component frameworks [**MDK94**, **Lim96**, **SKB98**, **BNS+05**]. For example, a component might be used to wrap access to a physical device like a sensor connected by USB; if the availability of this device deteriorates (because it becomes unplugged or fails), reconfiguration is required to accommodate the replacement of the component with a suitable substituting configuration (e.g., if the sensor is disconnected, the wrapping component might need to be removed from a list of sensors).

In this section, we will enhance the NewsCondensed example first introduced in Sect. 7.4.1. Again, we will read RSS feeds from the internet and process them. But contrary to the original implementation, we will also provide provisions for handling network failures and configuration changes. This would be required to upgrade the batch-processing original application (which, when started by the cron daemon, queries each RSS source once and processes the data, then terminates) to a long-running server application that frequently queries the feeds and is not expected to stop.

We will discuss two kinds of reconfiguration here: First, we will discuss a reconfiguration of the Chain of Responsibility pattern that organizes the list of feeds; we will do this in order to respond to on-line changes to the feed list, which is provided by the user. While there is no genuine need for reconfiguration here (the list will presumably not change too often), it is interesting to see how this



(a) Direct chain (b) Delegating chain

Figure 8.11: Chain of Command pattern

reconfiguration interferes with the second kind of reconfiguration: the response to network and data problems.

8.3.1. Chain of Responsibility Pattern. The "Chain of Responsibility pattern" is a way to send a message to an ordered list of components without using multi-ports. In this pattern, the target components are formed into a linked list. A message is then sent to the first component, processed and sent to the next component of the list. In a direct approach, this is utilized in the cache sequence of the CMC model checker, as shown in Fig. 5.7 on page 94. There, components are implemented as parts of a Chain of Responsibility, and contain a "`next`" role pointing to the next element in the chain. Fig. 5.13 on page 108 shows the actual components, where the `StateDeciderTerminator` is used to satisfy the required role of the last cache in the chain. An example of such a Chain of Responsibility is found in the web crawler described in Sect. 7.4.2.

In the NewsCondensed example, described in Sect. 7.4.1, a different form of a Chain of Responsibility is used. Here, the chain is external to the target components. As shown in Fig. 7.8 on page 160, the actual chain is formed by `NewsRetrieverChain` components, which delegate the query to the actual RSS reader. Thus, the actual RSS reader implementation does not have to be aware of being part of a Chain of Responsibility. Instead, an off-the-shelf component can be used, which is wired by an external chain (which is easy to implement). As before, a terminator component is used to end the chain. Both variants are depicted in Fig. 8.11. All the roles depicted have the same type.

The actual implementation of the chain is depending on the use of the Chain of Responsibility. The chain might just relay an asynchronous call to the connected components. It may also do synchronous calls and assemble the return values (this is what is done in NewsCondensed). It may also do synchronous calls until some return value is obtained (this is often done in event handler architectures).

8.3.1.1. *Substitution of Multi-ports.* The asymmetric nature of provided interfaces and roles in JComp allows to connect an unlimited number of components to one component, but only allows for a fixed number of components to which a component can be connected. The Chain of Responsibility pattern is useful for enabling $1 : n$ communication, with more flexibility than a framework-provided multiport provision (where an arbitrary number of components can be connected to one role). Here, we will only consider asynchronous communication and external chains.

For a *broadcast*, a chain element sends the message to both the attached client component as well as the next element in the chain. Thus, all clients attached to the chain are notified. On an abstract level, this acts as a broadcast over a multiport.

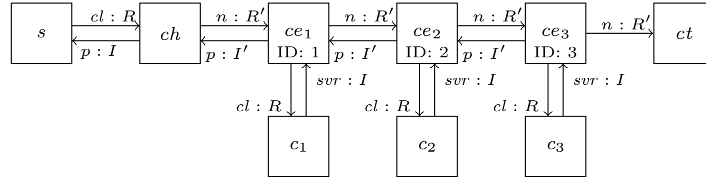


Figure 8.12: Substitution of a multiport by a Chain of Responsibility

Multi-ports can also be used to support a server-client-schema with a varying number of clients. In standard JCOMP, a number of client components can be attached to a server component and send requests. If the requests are handled synchronously, a response can be sent to the client in the return value of the message. If, however, the requests are to be answered by an asynchronous call-back message, the recipient cannot be addressed. Obviously, it is easy to address the proper recipient in a Chain of Responsibility if a unique ID is sent with the client's request. This is a little problematic, however, since the requirement to send such an ID needs more consideration on behalf of the client component's programmer than is necessary (or justified by the separation of roles paradigm): The component needs to be aware that its access to the server component will not be exclusive, but shared with other components. This is not a real problem, but we can do better.

If the server component does not maintain sessions, it does not have to know the identity of the client's component for any other reason except to address a response correctly. Multi-ports can support this by providing local component IDs (that can be used to reference a component within a multiport instance) and by telling the server from which component the currently processed method was received via passing the ID along. Note that the JCOMP component model stores the origin of a message during its execution in the state of a component: $e(\tilde{c})$ does exactly that. We do not want to provide direct access to that information, however; components should not know their environment in order to facilitate transparent reconfiguration. Instead, we can use local IDs that can be retained during component replacement (cf. Fig. 8.12). This can also be used by the Chain of Responsibility pattern: The chain elements can provide an interface I containing a method $req(p_1, \ldots, p_n)$ to the client components, and require an interface I' with a method $req'(p_1, \ldots, p_n, id)$ which is connected to the server for the head of the chain, and to the previous chain element for the other chain elements (hence, the chain elements also need to provide I'). The chain element can now be given an ID as a component parameter (this ID is chosen at assembly time, and can be kept consistent during reconfiguration by the reconfiguration designer). Upon reception of a *req* method, it relays this method up the chain using the method req' with the last parameter set to its ID. This way, the interface provided to the client does not impose the treatment of a (unique) ID, but the server can address its message.

We can even relax the server from having to treat IDs if we assume a well-defined behavior: Each request is immediately answered by a response. Then, we can build the chain just as above, but strip the ID from the request to the server. An additional "chain head" component needs to be used, which translates the method req' back to a method *req* for the server. This chain head component also maintains a queue of IDs and can hence assign the proper addressee to a response from the server.

What is actually done here is the implementation of a *connector* with components and their native communication means. Using the Chain of Responsibility pattern allows for creation of an $1 : n$ connector capable of broadcast and server-client communication. Other techniques can also be provided, e.g., gathercast [**NS00**] or broadcast of synchronous messages with user-defined processing of the result values. For example, in Fig. 8.12, the chain element components ce_i might each query their client with a synchronous call, putting the result in a list which is eventually processed by ch to a form that can be consumed by the server component s.

JComp does have multi-ports implemented in the framework; and the `LoadDistributor` component of the web crawler example in Sect. 7.4.2 utilizes it to address the `PageLoader` components. As with parallelization discussed in Sect. 7.3, this adds additional bulk to the framework as well as to the theoretical background (which needs to cope with $1 : n$ connections instead of just $1 : 1$ connections). We hence prefer to avoid discussing multi-ports as a first-class concept of the component framework, and rather rely on the fact that multi-ports are not genuinely necessary.

8.3.2. Adapting a Chain of Responsibility. The Chain of Responsibility pattern, as described in Sect. 8.3.1, realizes $1 : n$ connections of components. Such a connection, with the exact number of connected components being defined at assembly time, is more likely to be subject to change than a $1 : 1$ connection is. For example, a web server might use components to represent each of the connected clients, requiring frequent reconfiguration to handle connection and disconnection of clients. Representing each client by a component on its own might seem wasteful, but it offers an interesting prospect: Changes to the connection to the component (e.g., mode of encryption or check-sum checking) can again be attained by reconfiguration (a dedicated example is given in Sect. 8.4.1).

Here, we will use this kind of reconfiguration to respond to user-triggered, online changes to the RSS feed list.

8.3.2.1. *Monitoring and Assessment.* The list of RSS feeds is an actual file within the local file system, and this file can be monitored for changes. Again, we have to be precise about who does this monitoring; in this case, we opted to have a genuine component monitor the file and notify a Utility Interface to do the reconfiguration, if need be. As detailed in Sect. 7.2.1, we use `static` access to transgress the boundary of component and assembly code.

The monitoring component itself operates in a self-invocation loop (cf. Sect. 7.5.1.1). Every second, it checks for changes to the RSS list file; if a change is detected, the entire file is parsed and the resulting list given to the reconfiguration manager, whose instance is obtained by a Singleton pattern implementation (which is a practical implementation of the Utility Interface pattern).

We also use this approach to establish the initial configuration of RSS readers. Contrary to the classic NewsCondensed application, no RSS reader components are added initially, but they are reconfigured in right after application startup; this helps to avoid redundant code.

8.3.2.2. *Planning.* The reconfiguration manager needs to be pretty flexible, as various kinds of reconfiguration can be requested. This can be achieved by assuming as little as possible about the component graph. In this example, we need to compare the existing RSS readers to those just found in the configuration file, adding them to and removing them from their chain. Since this reconfiguration is to be conjoined with other reconfigurations, we will just fix an abstract, general architecture of the component setup as depicted in Fig. 8.13. The cloud-shaped sub-setups marked

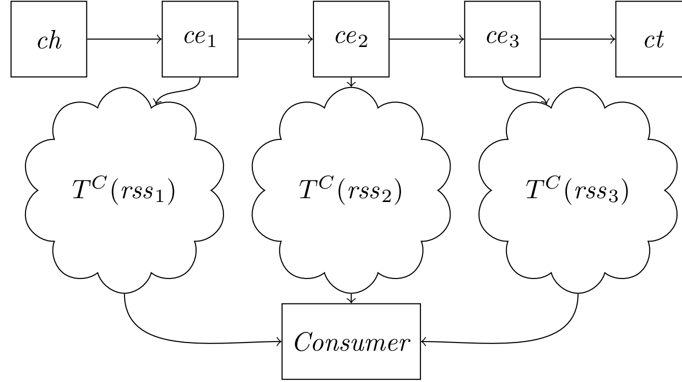


Figure 8.13: Organization of RSS reader chain

with rss_i stand for an arbitrary sub-configuration handling the RSS retrieval. Their internals are not of our concern, but we need to identify the components contained in order to remove them during reconfiguration. The sub-setups hence appear as *hierarchical components*, and we require that other reconfigurations preserve this structure of the application (this is similar to architectural styles [**CGS+02**], but on a more concrete level).

In building the plan, we need to cope with the problem of the co-domain of ρ, as discussed in Sect. 6.7.5. Since we use an external chain (i.e., dedicated chain components that link to the actual RSS readers, cf. Sect. 8.3.1), we can just reorganize the chain by building new chain components. This is not very expensive, as the chain components do not maintain data. Message retainment is unimportant, also; the chain is queried frequently, and if reconfiguration happens to interfere with spreading the query along the chain, not much will be lost. On the other hand, if we consider the chain as an abstract $1 : n$ connector, reconfiguration should not interfere with a sending operation generally (the request to retrieve RSS news is a broadcast send to all the RSS loader components), and we should never reconfigure in a situation where there are messages on the queue. We can thus use a communication monitor that counts the messages sent to the chain head and those arriving at the chain terminator, and allow reconfiguration only if those two numbers are the same; in which case the chain is in a "quiet[2]" state where reconfiguration will not loose or dislocate messages. At the same time, the monitor can delay any message sending to the queue for the duration of reconfiguration in order to avoid sending a message between the check of quietness and the detaching of the components to be removed. As stated before, this is not genuinely required for this particular application, but we have found that messages do become lost in such a reconfiguration approach in practice, and wanted to provide a generic approach to avoid this.

Let us assume that we have a set $\{rss_1, \ldots, rss_n\}$ taken from a set RSS of RSS feed names, and a component setup (C, M, e). Since an RSS feed may be handled by multiple components, we need two functions $T^C : RSS \to \wp(\mathcal{C})$ and $T^M : RSS \to (\mathcal{C} \to (\mathscr{R} \to \mathcal{C})))$. These functions calculate, for a given RSS feed name, a set of components and their connections. We require that these connections are well-connected and that only components of $T^C(R)$ are used in $T^M(R)$ for $R \in RSS$. We do not require $T^M(R)(c)$ to be completely connected; but we require that only a single role is not connected, and that this role is typed

[2] This is not to be confused with quiescence, which considers transactions – here, we need the stronger property of not conducting *any* communication until reconfiguration finishes.

with the interface provided by the *Consumer* component of Fig. 8.13. For accessing this role, we require a further function $T^r : RSS \to (\mathcal{C} \times \mathscr{R})$. Likewise, we require a function $T^i : RSS \to \mathcal{C}$ that denotes the entry point for an RSS handling sub-setup. This component needs to provide the interface that the Chain of Responsibility implementation expects.

For $T^r(R) = (c, r)$, we require $c \in T^C(R)$ and $I_R(c)(r) \in I_P(Consumer)$. Note that the functions T^C and T^M provide something reminiscent of a composite component assembler.

We further assume a partial, not necessarily injective function $C^{RSS} : \mathcal{C} \rightharpoonup RSS$ that can be used to identify the set of components used for loading an RSS feed. We need to use such a function since we do not want to fix the structure of the components connected by the chain, but at the same time we want to remove those concerned with an RSS feed no longer requested. Obviously, we require that $C^{RSS}(c) = R$ iff $c \in T^C(R)$.

Furthermore, we expect components $ch, ce_1, \ldots, ce_m, ct$ that form the Chain of Responsibility.

To accommodate the changes of the RSS feed list, we form a plan that removes those components no longer concerned with an RSS feed, adds those for a new RSS feed, and rebuilds the Chain of Responsibility. To this end, we define the set $RSS^+ = \{rss^+_1, \ldots, rss^+_{n'}\} = \{R \in \{rss_1, \ldots, rss_n\} \mid \forall c \in C \,.\, C^{RSS}(c) \neq R\}$ of RSS feeds that need to added, the set $RSS^- = \{r \in RSS \mid \exists c \in C \,.\, C^{RSS}(c) = r\} \setminus \{rss_1, \ldots, rss_n\}\}$ of RSS feeds that are no longer required, and the set $RSS^= = \{rss_1, \ldots, rss_n\} \setminus RSS^-$ of RSS feeds that are already present and are retained. We then build a set

$$RSS^C = \{rss'_1, \ldots, rss'_m\} = RSS^= \cup RSS^+.$$

We can then proceed (assuming that $RSS^C \neq \varnothing$) to build a shallow reconfiguration plan $(A, R, \alpha, \rho, \varsigma)$ for a component setup (C, M, e) by setting

- $A = \left(\bigcup_{R \in RSS^+} T^C(R)\right) \cup \{ce_{rss_i} \mid rss_i \in RSS^C\} \cup \{ct\}$,
- $R = \{c \in C \mid C^{RSS}(c) \in RSS^-\} \cup \{ce_1, \ldots, ce_m, ct\}$,
- $\alpha = \left(\bigcup_{R \in RSS^+} T^M(R) \cup \{c \mapsto \{r \mapsto Consumer\} \mid (c, r) = T^r(R)\}\right)$
$$\cup \left\{ ce_i \mapsto \left\{ next \mapsto \begin{cases} ce_{i+1}, & \text{if } i < m, \\ ct, & \text{if } i = m \end{cases} \right\} \middle| \; 1 \le i \le m \right\}$$
$$\cup \{ce_i \mapsto \{client \mapsto T^i(rss_i)\} \mid 1 \le i \le m\}$$
- $\rho = \{(ch, next) \mapsto ce_{rss_1}\}$,
- $\varsigma = \{\}$.

The plan hence removes the chain and rebuilds it again. Since we want to reach a quiet state anyway with no message in the chain prior to reconfiguration, we could also use a non-shallow plan that retains those chain elements whose RSS feed is not removed; message retainment is not required.

8.3.2.3. *Execution.* Reconfiguration may only commence if the communication monitor reports the queue being in a quiet state: No message is currently stored in the in-queue of a chain element. If we regard the Chain of Command as a realization of an abstract $1 : n$ connection, this requirement describes the necessity of not interrupting ongoing communication of a component (in this case, the abstract $1 : n$ connector) by reconfiguration. Before starting the reconfiguration, the communication monitor is also told to delay any message being sent to the chain until reconfiguration is finished. Again, this can be considered as a realization of the requirement that a component must not process a message during a reconfiguration

in which it participates. In doing so, atomicity is preserved despite a departure from the usual message retainment approaches.

8.3.2.4. *Experience and Discussion.* As can be seen from the reconfiguration plan definition, planning reconfiguration for a component setup only partially known can become quite tedious; yet, it is not difficult to do. Obviously, some management of reconfigurations is required to prevent them from overlapping – but this is easily established by maintaining a dedicated reconfiguration thread that actually implements the Active Object pattern itself (cf. Sect. 7.1).

We experienced some problems with lost communication due to the altered codomain of the ρ plan element, when we used the chain in a "big self-invocation" style; once the chain was traversed, the main component was invoked again to trigger the next round. Although there was ample delay added to the system (i.e., some time elapsed before another traversal was started), reconfiguration managed to disrupt the chain nevertheless, leading to a global deadlock. Adding communication monitors as described solved the problem; but even so we switched to the more robust design described above.

8.3.3. The Strategy Pattern. Fig. 2.3 on page 21 shows the class diagram of the Strategy pattern [**GHJV95**]. We have already argued that a reconfiguration touching the core concern of an application is closely related to the Strategy pattern; in this section, we will give an example.

Note that the Strategy pattern is closely related to the State pattern. The difference is subtle: The Strategy pattern externalizes part of the functionality of an object to the Strategy implementation, which can be chosen *from the outside*. The State pattern externalizes state-dependent functionality, but takes care of state transitions itself. This means that States implementations are aware of other State implementations, and can initiate a state transferal. In the context of reconfiguration, this is reflected nicely in the choice of the *initiator* of the reconfiguration: For the Strategy pattern, reconfiguration is triggered by an external observer, while the State pattern calls for a triggering of the reconfiguration by the components themselves.

For example, a component might wrap access to a physical device that produces data. If the device becomes unavailable, reconfiguration can be used to substitute the component by a temporary replacement that takes care of consistent production of temporary data. If this process is controlled from the outside, e.g., by an exception listener that reacts to I/O errors, then this amounts to a Strategy pattern. If the components themselves take care about the possible gain or loss of availability, then this results in a State pattern.

Obviously, the separation of concerns dictates that components should not plan and execute reconfiguration on their own, so the State pattern is not very suitable for reconfiguration. However, a hybrid approach can be beneficial: A component will usually be capable of recognizing if it is no longer suitable, and notify supervising code of that fact, e.g., by throwing an exception (cf. the discussion of error handling in Sect. 7.5.3). Thereby it can signal a request for reconfiguration, without detailing how that reconfiguration should look like. The reconfiguration plan can then be built by the supervising code. Technically, this is neither the State pattern (as components remain unaware of alternatives) nor the Strategy pattern (as components are aware of being capable of being reconfigured, and trigger the process).

In this example (illustrated in Fig. 8.14), we utilize such a hybrid approach to handle problems when fetching the RSS feeds. These problems might result from loss of network connection, or from malformed responses supplied by the RSS

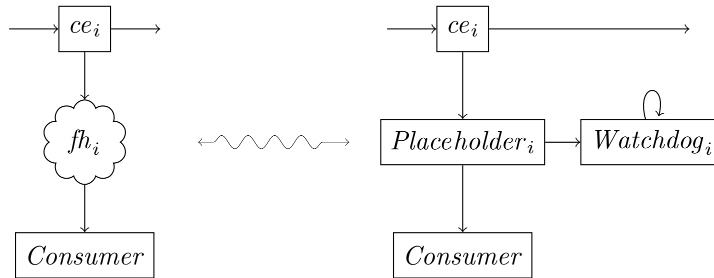


Figure 8.14: Reconfiguration of an RSS reader

provider. We assume that such problems are not permanent; instead of removing the RSS feed from the list, we add a monitoring component that frequently fetches the RSS feed and checks if the problem persists. Once normal operation resumes, the original configuration is reconstructed. The monitoring components serve two purposes: First, they interpret errors as expected, and avoid the remainder of the application to be impeded, and second, a further check is made to see if *all* RSS components are experiencing problems, which hints at a general network failure (on the retriever's side, opposed to a network problem of the RSS server). In this case, a global watchdog component is added that avoids frequent checks by the individual monitoring components.

8.3.3.1. *Monitoring and Assessment.* This kind of reconfiguration is triggered by the RSS reader components when they encounter an error. We distinguish two kinds of errors: Server errors, where the request for an RSS feed is answered, but with a malformed or error message, and network errors, where the request fails due to network problems. The former needs to be recognized by the component, which performs a successful request and receives a invalid response, e.g., a 404 HTTP return code. It then needs to raise an error in order to notify the listeners about this error. Also, it can encounter problems while parsing the response (which, according to the RSS standard [**Boa06**], is an XML document); again, an error has to be raised.

Any such error warrants immediate reconfiguration; for network errors, the connection is broken and needs to be reestablished (which we wish to do by reconfiguration and not by component-internal code). Server errors do not require immediate attention, but subsequent calls will most likely result in similar problems.

Since the errors are triggered by external entities (network or remote server), we cannot fix them by reconfiguration (although it might be possible to use an alternative RSS feed server). Instead, we reconfigure the system to cope with the error until it disappears. Technically, we replace the RSS reader by a placeholder that maintains a consistent treatment of the RSS feed request (i.e., it notifies the consumer that this RSS feed is not available at the moment). Also, we add a watchdog component that tries to reestablish the connection or get a correct server response (using a higher frequency than the original RSS polling). Once the recovery of correct operation has been detected by the watchdog component, it notifies the supervising code and another reconfiguration is used to switch back to the original RSS reading component.

8.3.3.2. *Planning.* Planning again needs to cope with a component setup only partially known in advance. The plan calls for the insertion of a watchdog component whose sole purpose is to trigger another reconfiguration once the error cause has

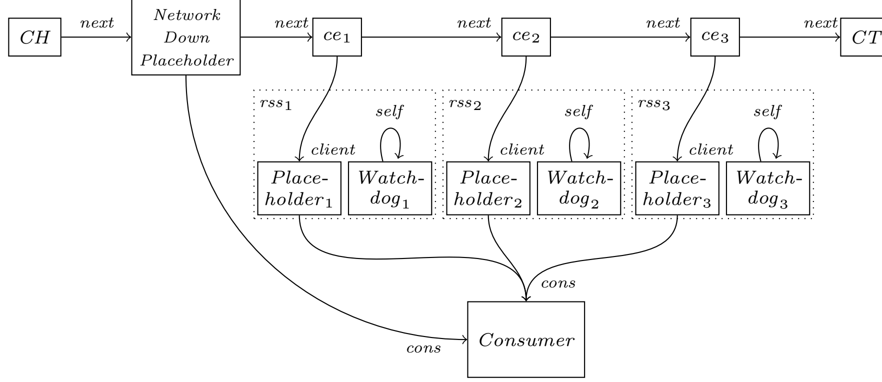


Figure 8.15: Adding a placeholder for handling network failure

disappeared. In order to keep the content generation running, a placeholder component is added to the chain that can output a short error notice if RSS content is requested for.

The removal of a placeholder and watchdog component is straightforward as well, as they are removed and replaced by a RSS reading component. These two reconfigurations are depicted in Fig. 8.14. For defining the plan for an RSS feed R within a component setup (C, M, e), we need to pick out the chain element ce_i with $C^{RSS}(M(ce_i)(client)) = R$. We also need to find the consumer component *Consumer*. We then update the RSS handler by complete replacement, obtaining a shallow plan $(A, R, \alpha, \rho, \varsigma)$ with

- A and α according to the reconfiguration, i.e., for switching to a placeholder, we have $A = \{Placeholder_r, Watchdog_r\}$ and $\alpha = \{Placeholder_r \mapsto \{cons \mapsto Consumer\}$, $Watchdog_r \mapsto \{self \mapsto Watchdog_r\}\}$, and for switching back to RSS handling, we use $A = T^C(R)$ and $\alpha = T^M(R) \cup \{c \mapsto \{r \mapsto Consumer\} \mid (c, r) = T^r(R)\}$.
- $R = \{c \in C \mid C^{RSS}(c) = R\}$ (we need to make sure that this function produces correct results also for newly added components, which is done by annotating all components of A with R such that $\forall a \in A \,.\, C^{RSS}(a) = R$),
- ρ is also defined by the reconfiguration at hand:

$$\rho(ce_i), client) = \begin{cases} Placeholder_r, & \text{if switching to a placeholder} \\ T^i(R), & \text{if switching back to normal operations,} \end{cases}$$

- $\varsigma = \{\}$.

Note that, in order for ρ to preserve the complete connectedness of components, the only components that are allowed to actually have a connection to a component c with $c \in \operatorname{ran}(T^i)$ are the chain element components. This is an example for an architectural constraint that all reconfigurations need to obey. If this constraint was to be lifted (e.g., for utilizing a second chain that connects the watchdogs to give them a globally synchronized "heartbeat"), the generation of all plans would have to be adapted.

We also take a look at a special case: If *all* the RSS reader become replaced by placeholder components, it is very much likely that not all servers have crashed, but that our own connection has. In this case, it is more sensible to no longer query a

chain of placeholder components, each reporting about an unreachable server, but have a single component state that the network seems to be down currently. We utilize a filter component plugged in before the chain (cf. Fig. 8.15) that handles requests to the chain; it just outputs a concise error message and does not relay queries to the chain. Once a watchdog reports reachability of a server again, this component is removed and normal chain operation is about to start again. The reconfiguration is a straightforward filter insertion, where we use the removal and re-insertion of the *CH* component to obtain an injective shallow reconfiguration plan. For a component setup (C, M, e), this plan is defined as follows:

- $A = \{CH', NetworkDownPlaceholder\}$,
- $R = \{CH\}$,
- $\alpha = \{CH' \mapsto \{next \mapsto NetworkDownPlaceholder\},$ $NetworkDownPlaceholder \mapsto \{next \mapsto ce_1, cons \mapsto Consumer\}\}$,
- $\rho = \{(c, r) \mapsto CH' \mid M(c)(r) = CH\}$,
- $\varsigma = \{\}$.

Similar, for the removal of the placeholder component, if network connectivity is regained:

- $A = \{CH\}$,
- $R = \{CH', NetworkDownPlaceholder\}$,
- $\alpha = \{CH \mapsto \{next \mapsto ce_1\}\}$,
- $\rho = \{(c, r) \mapsto CH \mid M(c)(r) = CH'\}$,
- $\varsigma = \{\}$.

8.3.3.3. *Execution.* Again, all state transferal is restricted to the URL the RSS handler is instantiated for, which is copied by the direct approach – i.e., by the setting of component parameters during component creation. All the plans are injective shallow reconfiguration plan, hence no further locking is required. As shown in Fig. 8.14, the watchdog components implement the self-invocation communication pattern in order to use an increased frequency of checking their target server, while remaining reconfigurable at the same time. Upon detecting a connection recovery, they will issue a "correct server response regained" message to the reconfiguration controller, which triggers their removal.

8.3.3.4. *Experience and Discussion.* Integrating with the example discussed before (Sect. 8.3.2), we integrated two reconfiguration domains in this example: The modification of the chain of RSS readers, and the modification of the RSS readers themselves. This can be perceived as a reconfiguration of hierarchical components – with the slight twist of the "network connectivity down" component placed at the head of the RSS reader chain, as shown in Fig. 8.15. We found it quite difficult to judge whether such cross-cutting reconfiguration concerns can always be understood as concerning different levels in a tree of hierarchical components; such a consideration is beyond the scope of this thesis. We found it, however, pretty straightforward to "relax" the planning a little and build the reconfiguration plan for a given component setup, picking out the appropriate components at runtime. In the actual implementation, this amounts to a number of applications of the Visitor pattern to the component graph. For this example, this is not difficult, even if it becomes a little bulky. When devising a reconfiguration, however, the planning of one reconfiguration has to consider explicitly which architectural constraints the other reconfigurations will exploit to define their plans; reconfiguration must not violate these in order to keep the other reconfigurations from coming up with malformed plans. This resembles architectural styles [**GS93**] very much, though more constraints are most likely required (e.g., the annotation of components with the

RSS URL they are working on, which is required to identify those that need to be discarded).

Building the plans requires a lot of input, however. The tuple notation applied in this example stresses the limits of what is still understandable, and the actual implementation is just about as complex. In this example, the two reconfigurations are actually independent in one important regard: Both do not change the layout of the RSS reader sub-setup. It is, however, easy to conceive that there might be examples where this does not hold: A reconfiguration that reestablishes the RSS reader needs to build it in a way that was formulated at runtime, rather than at design time, as it is now reflected in functions like T^C and C^{RSS}. There is ample opportunity to provide help for these situations, e.g., by domain-specific languages. This is a challenging field of research, and will undoubtedly be an important factor for the success of reconfiguration as an implementation of adaptivity.

8.3.3.4.1. *Multi-Layered Architectures.* Most of the research-oriented component frameworks provide hierarchical components [**Hac04, BHP06, BCL$^+$06**]. The idea is to group components together, and wrap them with some *membrane* that gives the aggregation of components a consistent external appearance, i.e., it defines the communication endpoints and how they are connected to the internal components. Usually, the membrane also contains *control components* that can manage the execution of the wrapped components, e.g., conduct their reconfiguration. This is appealing as it clearly defines where the code conducting the reconfiguration is located, and which parts of the component setup the reconfiguration can modify. Just as the concept of components itself, the idea of hierarchical software components stems from a well-established idea in the hardware industry. (Consider Fig. 2.1, which illustrates the components of the Space Shuttle stack (comprised of the tank, two booster rockets, and the orbiter), with the orbiter component refined into its sub-components below.) In the RSS reader example, hierarchical components define the scope of reconfiguration.

Neither CMC nor JCOMP provide a hierarchical approach as a part of their component model. Even worse, in JCOMP we lack an explicit consideration of *assembly code* that controls instantiation and modification of the component setup. In practice, the main thread of the application often takes this role, but this is not enforced, and often problems with reconfiguration can be traced back to parallelization issues with the control code (i.e., the monitoring capabilities described in Sect. 7.2.5) clearly state which thread will invoke the monitor, but no further support for conducting a subsequent reconfiguration in a proper own thread is given). This problem is outside the scope of this thesis, but it is a high-priority item on the future work list.

Hierarchical structures, however, are found frequently in JCOMP. There is no syntactic support provided, but on an abstract level, they can be clearly identified. Most notably, the patterns described in the previous sections are ways to implement abstract, atomic concepts with a series of components. We have briefly discussed this topic for the Chain of Responsibility pattern, which can be understood as a generic provision for the abstract idea of multi-ports. Multi-ports, again, can be understood as an implementation of the abstract idea of a client-server architecture. Fig. 8.16 illustrates this idea of abstraction levels. Note that there is a subtle difference to hierarchical components here, as the Chain of Responsibility is not a hierarchical component of fixed structure, but rather a *way to construct* a multiport connector by means of a component sub-setup. As such, the multiport connector again becomes a way to construct a client-server architecture.

Such component sub-setup construction approaches are prevalent in this thesis. For example, the way to distribute component applications by providing special

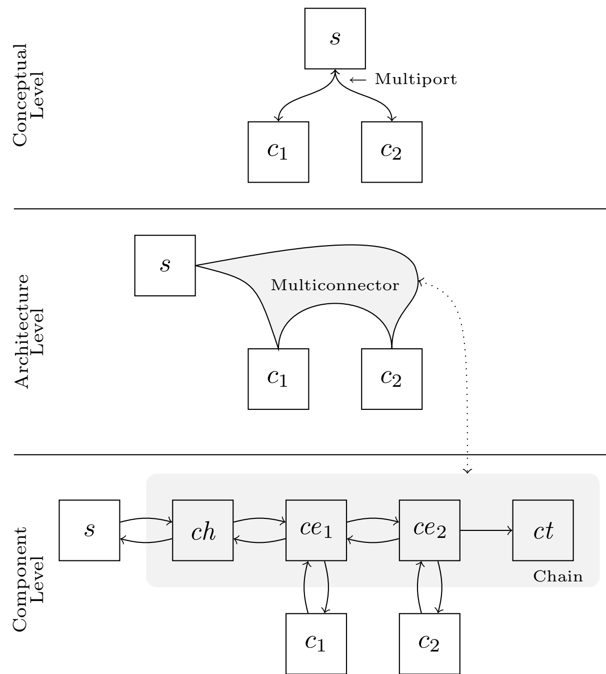


Figure 8.16: Hierarchical component abstractions

stub-components, described in Sect. 7.3, is actually a way to construct a (local part of a) distributed component setup. If JComp was to be used to develop distributed applications, a wrapper for the assembly would be provided: The user would still only use a small number of primitives as those described in Tab. 7.1 on page 149, which would be translated into a series of commands to instantiate the complete, distributed application, thus hiding the actual assembly design including the stub components and network managers from the user. Other examples are given by general-purpose filters employed in cmc (cf. Sect. 5.2.2) or by the RSS reader components in NewsCondensed. A further example will be given in Sect. 8.4.2, where filters are added to connections to monitor the communication behavior in order to take action (by starting a reconfiguration) if the queues are overflowing. For this example, the assembly generation code was wrapped in a class that provides an `instrumentAndConnect(ComponentDescriptor src, ComponentDescriptor tgt, String role)` method, which builds a filter and connects the components with the filter inserted. Again, the implementation details of the filter (and further monitoring components) are hidden from the user.

8.4. Dynamic Cross-Cutting Concerns

We discussed cross-cutting concerns in Sect. 8.1.2.1, and described the filter pattern as a way to handle such concerns that are orthogonal to the core concern of an application.

In this section, we will describe situations where cross-cutting concerns change during runtime. This is inspired by Cactus [**CHS01**] and NeCoMan [**JTSJ07**]. The latter circumscribes its purpose to be the provision of "distributed AOP", where dynamic cross-cutting concerns of the network connections of distributed applications are discussed (cf. [**JDMV04, SBG06**]). Distribution is a cross-cutting

concern, and we have discussed in Sect. 7.3 that it is favorable to handle it on the user-level, i.e., by providing appropriate filters.

NeCoMan discusses how various filters can be added to (and removed from) the network connections. Such filters might provide encryption, fault-tolerance or compression. These are examples of cross-cutting concerns in a distributed system (i.e., the system designer does not have to care about how the inter-machine connections are realized). We have argued that component-based software engineering facilitates the detection and treatment of cross-cutting concerns. Other cross-cutting concerns we have found are logging, performance optimizations, and also the fact that the system is distributed at all.

Dynamic aspect-orientation [**NA05, TT06, TJSJ08**] addresses the situation where such cross-cutting concerns change. For example, the requirement to do logging might not always be given, or the required granularity might change: If a system exhibits exceptional or problematic behavior, a finer granularity is required than during normal operation. Responding to such changed cross-cutting concerns can be handled by reconfiguration of filter components, either adding or removing them, or changing their settings (which, formally, can be described by a filter substitution).

Changing cross-cutting concerns provide good examples for reconfiguration. This is because it is more likely to see a cross-cutting concern change during the execution of a component application than a change of the core concern. This, again, is due to the problem of anticipating changes to all concerns of an application – the core concern can (and will) be taken care of, but concerns less obvious during design time can easily be forgotten, or not deemed worth the trouble to make them adaptive. As an example, it might be quite likely that an RSS reading component has provisions for handling a malformed server reply, or for handling network problems. While it might not be desirable to have adaptivity on that level since it restricts the system designer in the choice of how to recover from errors, having some fault-tolerance built into the components is not unlikely (after all, exception handling provisions of languages like Java are built to enforce an error handling as local to the source of problems as possible). It is, however, quite improbable to have adaptivity provisions for logging built into a component. Because logging is not the core concern of the component, the component implementer will not spend much effort on this concern. Actually, implementing an adaptive logging behavior into a component would violate the separation of concerns paradigm, since it fixes the method of adaptivity too early in the application development process.

In this section, we will provide examples for handling dynamic cross-cutting concerns by reconfiguration. First, we will discuss how reconfiguration can be used to dynamically modify the distribution of component-based applications. In Sect. 8.4.2, we will dynamically add filter components that take the sting from one of the lurking problems of asynchronous systems, i.e., overflowing message queues. In Sect. 8.4.3, we will attempt to generalize such an approach to generic fault-tolerance.

8.4.1. Reconfiguring the Distribution Communication. This example is inspired by the NeCoMan framework [**JDMV04**]; and it is concerned with replacing the proxy components that establish a connection over the network, as described in Sect. 7.3.4. For example, we might opt to compress or encrypt the data passed over the network (other examples, like the one provided by Cactus [**CHS01**], are too low-level for implementation in JComp, which only utilizes Java streams but does not look at any level of the TCP/IP stack beyond the application layer). Here, we will not discuss the first three stages of the MAPE loop, but consider the adaptation

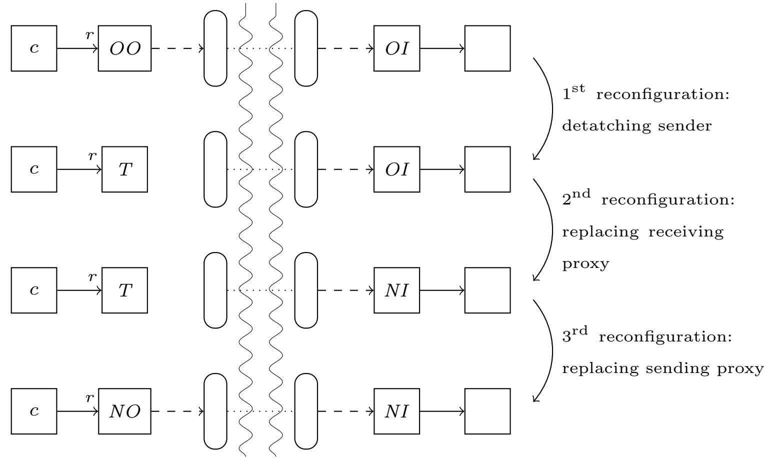


Figure 8.17: Three-stage reconfiguration of a network edge

request to be given by the user (or be produced by other means that are beyond this example).

8.4.1.1. *Execution.* The execution phase of this example, however, is quite interesting. Obviously, a simple replacement of the stub components on both sides will not suffice to guarantee correctness (in terms of an uninterrupted execution of the overall, abstract application). Instead, a coordinated sequence of reconfigurations is required to ensure that the first message sent by the new outgoing stub is already received by an incoming stub capable of processing it. This is similar to the approach exhibited by Djinn [**MNCK99**] and RDF [**ZL07**] (cf. Sect. 3.4.1), but does not require an extension of the framework that annotates messages with the configuration (in order to detect the first message that requires the new configuration of the receiver to become processed).

This can be achieved by a three-stage reconfiguration, which, in its abstract idea, is quite similar to the approach we utilize for achieving perceived atomicity. The overall reconfiguration, shown in Fig. 8.17 (see Fig. 7.6 on page 157 for a description of the components involved), requires three parameters: A component c and one of its roles r, and a description of how the new stub components should be built (used to build *NO* and *NI* in the figure). The connection of the given component and role will then be reconfigured to work with the new proxies.

In the first stage, the sending of messages over the network needs to be stopped. This is achieved by introducing a buffer component that just stores any incoming request for the duration of the replacement. This buffer component also receives any pending message of the previous outgoing stub component (using a shallow reconfiguration plan, this happens automatically due to δ^ρ). Also, necessary connection information data is retained during this reconfiguration. Let us assume that the local component setup of the node where component c is located is given by a component setup (C_1, M_1, e_1) with $c \in C_1$. We then obtain a shallow reconfiguration plan $(A_1, R_1, \alpha_1, \rho_1, \varsigma_1)$ with $A_1 = \{T\}$, $R_1 = \{M(c)(r)\}$, $\alpha_1 = \{\}$, $\rho_1 = \{(c, r) \mapsto T\}$. The buffer component T cannot give any meaningful interpretation to the messages it has received. It is therefore built to just block on them, as described in Sect. 7.5.2. ς_1 needs to copy connection-relevant data, which needs to be stored for re-utilization of the final *Outgoing Proxy NO*. In the actual implementation, all these data (like an ID used to identify the *Incoming Proxy* on the

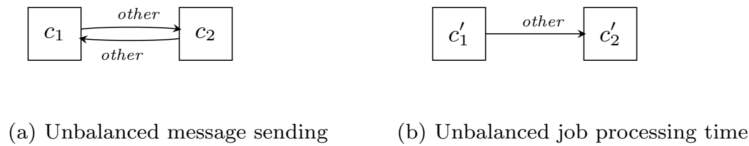
other node) are kept in component parameters, and are thus directly inserted into T at its creation time. Effectively, this presents itself as the direct state retainment approach again.

The second stage now replaces the incoming stub component with the new version supporting the desired reception capabilities. Since the incoming connection is used in a 1:1 fashion (i.e., if multiple components connect to a component over the network, each connection is realized by its own incoming stub component), the sole sending component is stopped and no messages will be received from the network in the meantime. The *Incoming Proxy* thus has reached a genuine quiescent state. However, there might be unprocessed messages pending in the internal queue from the *Network Connection Manager*. As described in Sect. 7.3.4, these queues are part of the data space of the components and not maintained by the framework. Hence, a state transferal should be done. Interestingly, for this stage, a shallow reconfiguration plan will not do: The self-invocation of the *Incoming Proxy* (cf. Fig. 7.6 on page 157) will be broken unless the corresponding self-invocation message is retained by δ. Assuming the local component setup (C_2, M_2, e_2), we use a reconfiguration plan $(A_2, R_2, \alpha_2, \rho_2, \delta_2, \varsigma_2)$ with $A_2 = \{NI\}$,$R_2 = \{OI\}$ (in practice this component is looked up from the local assembly by a component parameter that describes this proxy component to be part of the connection implementation of component c and role r), $\alpha_2 = \{NI \mapsto \{\mathit{target} \mapsto M_2(OI)(\mathit{target}), \mathit{loop} \mapsto NI\}\}$, $\rho_2 = \{\}$, $\delta_2 = \{(OI, (OI, \mathit{loop})) \mapsto NI\}$, and ς_2 implementing the retainment of the network message queue as detailed above. Additionally, the *Network Connection Manager* needs to be notified on the replacement of the stub component.

The third stage now reconfigures the sending stub component by a component supporting the new sending paradigm. Here, a shallow reconfiguration plan is sufficient; during the reconfiguration the messages pending in the queue of the temporary component T will be transported to the new proxy NO and subsequently be delivered with the new communication method. The reconfiguration plan $(A_3, R_3, \alpha_3, \rho_3, \varsigma_3)$ is now easy to build: $A_3 = \{NO\}$, $R_3 = \{T\}$, $\alpha_3 = \{\}$, $\rho_3 = \{(c, r) \mapsto NO\}$, and ς_3 again copying the communication parameters by direct state transferal as it was done in the first reconfiguration.

8.4.1.2. *Experience and Discussion.* The implementation effort of this example is divided into three parts: The implementation of the server protocol that controls the three-stage reconfiguration progression, the generation of the new proxy components and the generation of the reconfiguration plans. The protocol is just a straightforward extension of the protocol already used for establishing the local component setups. Generating the new proxy components is done by using a different template for the VELOCITY engine that generates the code for the proxies. Surprisingly, the hardest part is to build the reconfiguration plans. This is because no local information about the proxies is stored anywhere else than within the component graph maintained by the local assembly. This helps to avoid problems with redundant data, but requires a search for the correct proxy components in the component graph; in JCOMP, this is realized with a Visitor pattern implementation, which we found to be harder to write than originally anticipated (also cf. Sect. 8.3.2 and the necessity of the C^{RSS} function) – better support for finding components is definitively required.

Overall, the implementation effort is not too big, requiring approximately 200 lines of JAVA plus the protocol message classes; as the local reconfiguration provision take care about local quiescence, no waiting for a suitable point in time is required, nor is there any risk of deadlocks with pending synchronous calls – since reconfiguration will only commence if OO is not blocked, reconfiguration cannot interfere with a pending synchronous request of any of the components involved.



(a) Unbalanced message sending (b) Unbalanced job processing time

Figure 8.18: Architectural reconfiguration scenarios

Admittedly, the one problematic point is the *Outgoing Proxy*, which only receives messages from a single component, so even without the provisions for reconfiguration, a safe substitution can be implemented easily. Still, the tidiness of the components facilitates such an implementation a lot; most bugs experienced when implementing this example emerged in the assembly code that resides outside the component framework.

There is another scenario of using reconfiguration to modify the distribution of a component setup: Migrating a component from one machine to another. The abstract approach is very much alike: Placeholder components are generated and reconfigured in to cut the migration target off from the component graph. Depending on the necessity of state retainment, a temporary connection is established to the target machine, which uses a Memento component to query the target component's state (much alike to the example in Sect. 8.2.2). Then, the component is removed from its original machine, and it is instantiated anew on the target machine, possibly drawing the retained state from the Memento component. In order to preserve messages, a placeholder is reconfigured in at the source machine that takes the messages from the old component and relays them to the new component on the target machine. Finally, connections are reestablished. This procedure amounts to a lengthy sequence of reconfiguration steps, but every step relies on the existing capabilities of the component framework and the distribution framework built on top of it.

8.4.2. Reconfiguration for Resource Preservation. In this section, we discuss how reconfiguration can be used to deal with a problem that is encountered in producer-consumer examples where the producer is faster than the consumer. Observation of a running system is used to detect suboptimal behavior and mend it. Contrary to other examples, the suboptimal behavior is not the result of a badly written component, but arises from an unsuitable combination of components, made by an unwary system designer. Reconfiguration thus becomes an actual fix for a malformed architecture.

We assume that all communication is done asynchronously in this example. Synchronous calls can be incorporated as well, but they obfuscate the purpose of these reconfiguration scenario: Automatically handling problems introduced by the decoupling of components achieved by the Active Object pattern. Such problems are not altogether uncommon; actually, this reconfiguration application was inspired by problems with the web crawler example discussed in Sect. 7.4.2. Since a web page usually contains more than one outgoing links, the search front initially increases, since only few pages are revisited. Since the web crawler uses the message queues for storing pending URLs, these queues consume all the memory eventually. Here, we will consider an abstraction of this situation.

Consider the component setup depicted in Fig. 8.18a. We assume that the roles *other* are typed with an interface $I = \{msg\}$, and that the method evaluators of the components c_1 and c_2 are given as

$$\zeta(c_1)(msg) = \text{call}(other, msg, \langle\rangle).\text{call}(other, msg, \langle\rangle).\text{success},$$
$$\zeta(c_2)(msg) = \text{call}(other, msg, \langle\rangle).\text{success}.$$

The component setup is given by the tuple

$$(\{c_1, c_2\}, \{c_1 \mapsto \{other \mapsto c_2\}, c_2 \mapsto \{other \mapsto c_1\}\}, (c_1, msg)).$$

Basically, the components c_1 (the producing component) and c_2 (the consuming component) call each other with asynchronous messages. However, for every message *msg* consumed by c_1, two messages are sent to c_2. Theoretically, this is no problem, as every message sent to c_2 will eventually be consumed. In practice, however, the queue of c_2 will eventually consume all memory. As mentioned before, such situations can be encountered in practice if a the processing of a message initiates more than one additional task to be handled.

There is a slight variant of the problem, given by the component setup depicted in Fig. 8.18b, using the role interface and the method evaluators

$$\zeta(c_1')(init) = \mu X \Rightarrow \text{call}(other, msg, \langle\rangle).X,$$
$$\zeta(c_2')(msg) = P'.\text{success}.$$

Here, we specify the component setup to be

$$(\{c_1', c_2'\}, \{c_1' \mapsto \{other \mapsto c_2'\}, c_2' \mapsto \varnothing\}, (c_1', init)).$$

P' stands for a time consuming task. Since c_1' does nothing besides sending c_2' messages over and over, and since we may assume that P' is costly enough to prohibit c_2' from processing them at the same rate as they arrive, eventually the message queue of c_2' will be overflowing.

The difference between those two setups is easy to see: $\mathcal{S}$ has a cyclic component graph, whereas $\mathcal{S}'$ has not. Given the fact that all messages are sent asynchronously, $\mathcal{S}'$ would not exhibit a problem if it were not for the costly task P' (and the much faster processing of c_1').

The problem of $\mathcal{S}'$ can be mended by an extension to the framework that allows us to impose a delay on the calling component when it sends an asynchronous message. The penalty would be chosen such that the queue does not exceed a given size. Since such a penalty – at least in a single-machine setting – corresponds to giving more processing time to c_2', we actually raise the priority of c_2''s thread. If the loop is substituted by a self-invocation, this can even be done on component level by a filter that slows the self-invocation, or by a communication monitor that stalls message delivery for some time.

Another solution is given by component layout reconfiguration, which may replace c_2' by some component where P' is solved more efficiently, or replace c_1' by a slower version. It is, however, pretty difficult to judge whether there is a scenario where such components actually exist.

Tuning the efficiency, however, will not work for $\mathcal{S}$, since no matter how the priorities are distributed, every message sent by c_2 to c_1 will trigger the sending of two messages from c_1 to c_2. The number of messages in the system is therefore monotonically increasing (actually, it can drop by one for the time during a message being read and no new message being sent, but never more than this). In our web crawler example, if there are more than one previously unvisited links on each web-page on average, the number of messages will grow.

There are basically two ways of mending this: Domain-specific solutions, and an independent solution. The former one consists of reconfiguration plans that are specific to the problem at hand, while the latter works for every component model.

For the web-crawler, a domain-specific solution could be given by a heuristic that tries to guess how many links are to be found on a given web-page, and prefer those that have few links. We may assume that a web crawler usually encounters many dead ends, while there are relatively few "hub pages" which contain many links [**BMPW98**]; by processing the dead end pages right away the number of known URLs that need to be processed can be decreased, and the problem of overflowing queues is, at least, postponed.

A generic solution is to use a filter component that is buffering the flow of messages and dumps them to disk if the buffer runs too full (motivated by the CMC's treatment of the open set, discussed in Sect. 5.3). The problem here is to recognize at what rate the messages should be sent to the consuming component by the buffer (we call this the *replay rate*). If we read messages from the disk too fast, we will run into problems with the memory consumed by the queues again; but if we read messages too slowly, the target component c_2 will eventually run out of messages to process. Thus, not only a reconfiguration step is required to introduce the filter component into a problematic connection, but also further refining steps to calibrate the flow of messages such that some bound on the number of messages stored is met while at the same time ensuring that the queue of the consuming component does not become depleted due to a sending too few messages.

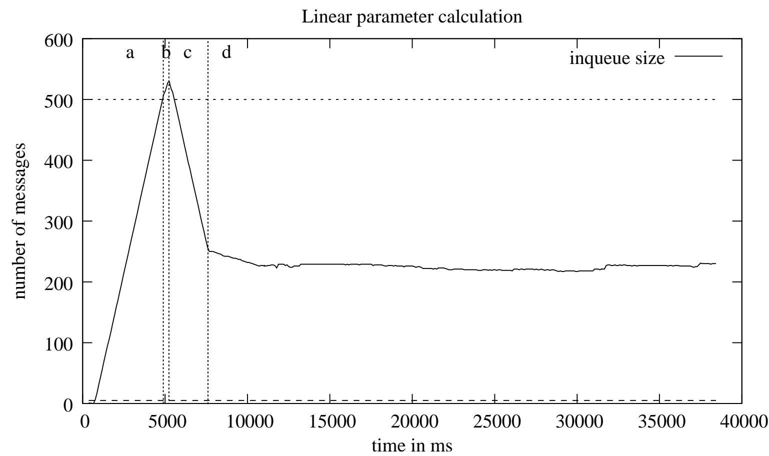
8.4.2.1. *Monitoring.* We basically only need to monitor the size of the incoming message queue of each component, which corresponds to the difference in received and processed messages. The communication monitor framework of JComp, as described in Sect. 7.2.5.1, is sufficient here: We can calculate the length of the message queue by taking the difference between the received and the processed messages.

8.4.2.2. *Assessment.* Given the current queue sizes of the monitored components, a decision about the necessity of reconfiguration needs to be made. The ultimate goal of the assessment is to intervene with the system should it detect a growing of message queues beyond allowable dimensions. There are two cases to be considered: Either no reconfiguration has been made so far, or a filter is already present but not configured well. In the latter case, reconfiguration does not have to change the architecture itself, but cope with a filter that has its replay rate set to a value that results in either a complete draining or still too much growth of the message queue of the target component; either the initial setting's choice was not good enough, or the value has deteriorated due to a changed communication or processing behavior. In both cases, assessment needs to compare the monitored queue sizes to a threshold value and trigger the reconfiguration if the queue size exceeds acceptable limits.

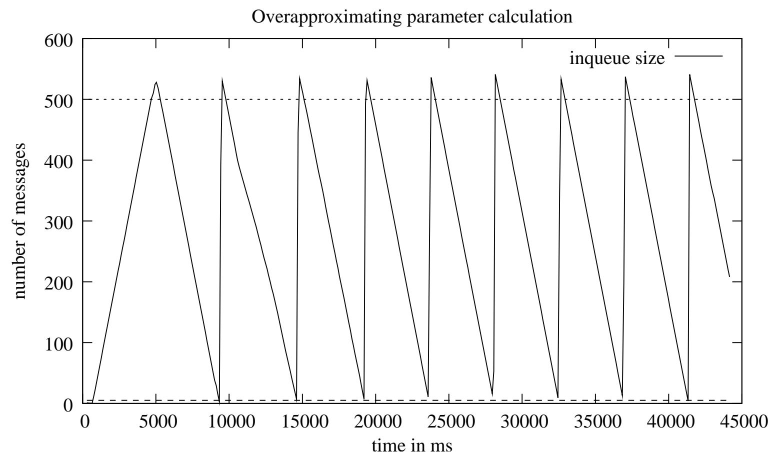
For connections that have not yet been instrumented with the buffering filter component, a simple threshold for an allowable queue size can be used. Should the number of unprocessed messages grow beyond that value, the initial reconfiguration that inserts the buffering filter is triggered.

If the buffer is already installed, a "high-/low-water-mark pair" can be used to detect derivation from the planned behavior due to a deteriorated replay rate. Fig. 8.19 shows a high-water-mark of 500 messages; if the queue grows beyond that size, further reconfiguration is required to tune the replay rate to a lower setting. Fig. 8.21 shows a low-water-mark of 0 that is eventually reached – the queue is drained, and new messages are inserted at a replay rate too low. Another reconfiguration (conducted after approximately 55 seconds) is used to re-tune the replay rate.

For the buffer already installed, planning needs to calculate a new rate of message sending depending on the results of the analysis. Alternatively (and especially



(a) Replay rate calculation



(b) Repeated reconfiguration

Figure 8.19: The "unbalanced producer/consumer" example

if calculating rates leads to oscillating behavior) the sending of messages can be stopped at reaching the high-water-mark and started at reaching the low-water-mark, calculating a rate that is conservatively too high, thus eventually reaching the high-water-mark again. This requires frequent reconfiguration, but this reconfiguration is relatively cheap to do, as no components are added; only the parameters of the buffer components are changed.

For calculating a replay rate, we need to approximate the message consumption rate of the target component. Fig. 8.20 shows a hybrid automaton (with the notation $\dot{t} = 1$ meaning that the first derivation of t over the time is 1, cf. [**Hen96**] for a definition of hybrid automata) that illustrates the different states of the analyzer for the unbalanced producer/consumer example. The variables have the following meaning: l is the message queue level of component B, r is the replay rate, i.e., the number of messages to be played to the consumer in each time step, t is a timer variable, and low, med and $high$ are fixed values with $med = \frac{high-low}{2} + low$. In the NORMAL state, no reconfiguration has been made, and r would have no influence. In the RECONF state, the reconfiguration is executed by substituting the edge

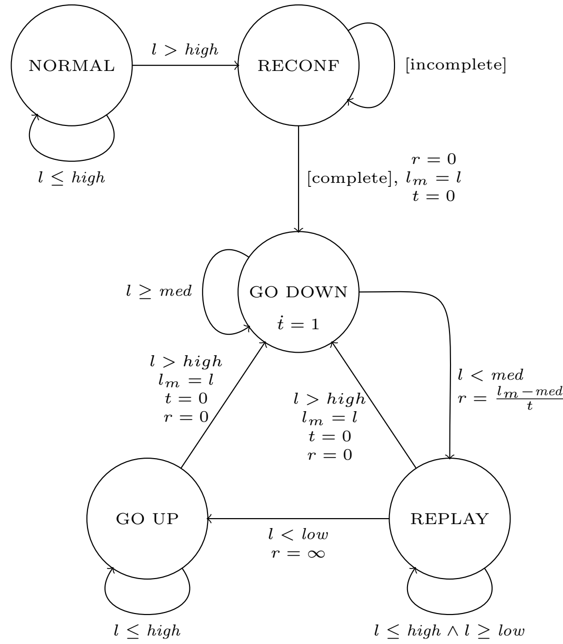


Figure 8.20: Hybrid automaton for the analyzer of the unbalanced producer/consumer example

from A to B with a filter component F and edges from A to F and from F to B. The GO DOWN state, which invariably has $r = 0$, is used to deplete the overly filled message queue of B by not adding any new messages. In the REPLAY state, a well-behaving level is maintained by using a replay rate that has been calculated to match the message consumption rate of B, as it was witnessed while in the GO DOWN state. Should the message queue level of B run out of hand, either GO UP or GO DOWN is used to bring the level to a consistent value again. GO UP will go beyond the high level, and then use GO DOWN to deplete the queue again while measuring the consumption rate of B anew. Thus, each time REPLAY is initially entered, a fresh consumption rate has been calculated, which helps to adapt to changing environments.

In Fig. 8.19a, the development of the size of the message queue of component B is shown. We can distinguish four stages:

(1) Stage **a** (state NORMAL in Fig. 8.20) is the interval where the message queue has not yet grown beyond the threshold we have set (at 500 messages, which is an arbitrary value and a notably small one). The number of messages can be seen to increase in a monotonic fashion.
(2) Stage **b** (state RECONF) is the interval where the threshold has been passed, but the reconfiguration – which also involves the generation and compilation of the filter component, as it is built dynamically to match the given interface – has not yet been completed.
(3) Stage **c** (state GO DOWN) is the interval where the buffer takes all the messages sent from component A to component B. B now consumes the messages in its message queue, and eventually reaches some fill rate we regard as sufficiently low. We measure the time that is required to reach

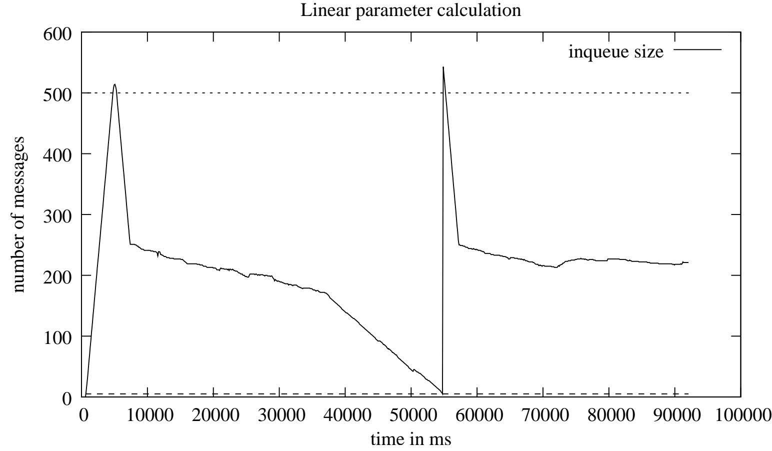


Figure 8.21: Degrading replay accuracy

that level (variable t in Fig. 8.20), and thus gain a good approximation of the message consumption rate of component B.

(4) In stage **d** (state REPLAY), we replay the messages stored in the buffer to component B at the rate we measured in phase **c**. Thus, a nearly constant message queue fill level is obtained.

In this stage, the message queue level is still monitored, since it is possible that the message consumption rate of B is changing. Fig. 8.21 shows such a system: Due to a change in the message consumption at approximately 35sec, the message queue level drops until all messages are consumed: The message queue of B is then filled with enough messages to continue with stage **c** again. Note that this also happens frequently due to the inaccuracy of system clocks.

Fig 8.19b shows a different approach: Instead of calculating some replay rate, the buffer fills the message queue of B each time it runs empty. While this requires far more reconfiguration executions (wherein the behavior of the buffer is modified to either start or stop sending messages), it requires much less overhead and is also independent of clock accuracy (which tends to be problematic with JAVA and LINUX).

8.4.2.3. *Planning and Execution.* While the assessment needs more consideration, the actual planning is fairly straightforward. It amounts to an insertion of a filter component. Since we do not want to move messages – in fact, the replay rate calculation described in the last section requires that the messages accumulated in the queue of the target component are left in place – a non-shallow reconfiguration plan is sufficient. We assume that we want to instrument the connection of c_1's role r to c_2:

- $A = \{c_f\}$ for a filter component c_f for the interface $I_R(c_1)(r)$ of the connection to be instrumented,
- $R = \{\}$,
- $\alpha = \{c_f \mapsto \{next \mapsto c_2\}\}$,
- $\rho = \{(c_1, r) \mapsto c_f\}$,
- $\delta = \{\}$, $\varsigma = \{\}$.

There is no risk of message overtaking, and messages do not get lost; any message sent to the filter will arrive at the target component c_2 at a later point in time that those messages send directly over role r.

If the filter is already present, only the replay rate needs to be reset. This rate is a component parameter; changing it actually does not require reconfiguration (thought a clean way to change it without interfering with ongoing method call processing can be achieved by reconfiguration that removes and re-adds the same component).

8.4.2.4. *Experience and Discussion.* We implemented such an "unbalanced" producer/consumer example in JComp, after the problem with overflowing queues surfaced for the web crawler. Two components communicate asynchronously in a cyclic connection. For each message component A receives, it sends two messages to B, which in turn will just send a message to A. Thus, given an even message processing rate in A and B, the message queue of B is growing, eventually exceeding available memory. This is prevented by timely reconfiguration as described.

The hardest part of this example is to come up with a good way to determine the replay rate, as the accuracy of the timers available on the Linux system used is quite limited. If the delay between message sending (i.e., the inverse of the replay rate) was determined to be 6ms, the message queue kept overflowing, and if it was determined to be 7ms, a behavior as shown in Fig. 8.21 usually emerged after a short while. However, this only increases the overhead a little, as new replay rate calculations are done; the basic functionality is not impeded.

Like most examples in this chapter, this example is quite artificial. It might also be argued (in a similar way as done in Chapter 9) that having overflowing message queues is due to a malformed system design that should have been considering the unbalanced production and consumption of jobs in the first place. On the other hand, the example outlines the applicability of reconfiguration to address cross-cutting concerns. If we consider large-scale applications, similar situations might arise; and by considering the concern independently of the application, lots of efforts might be saved.

8.4.3. Fault Tolerance. Fault tolerance, and the related problem of self-healing, is a hard problem for software that has been approached from various directions [**GSRU07**]. When NASA started using software as an indispensable component of aircraft – an endeavor risked only after computers proved reliable when landing the Lunar Excursion Module during the Apollo missions) – they immediately transferred the proven approach of using redundant hardware to computers [**Tom00**]. For the fly-by-wire project, three computers calculated the required control surface deflections, and the result was just added up; so even if one computer provided arbitrarily wrong results, the plane was still flyable. Later, this "voting" scheme was extended to majority voting, where the result produced by most machines was preferred.

Using a redundant hardware approach was a natural thing to do, since the computer hardware technology was prone to random errors. Interestingly, space flight has stayed the prime user of this approach, as random errors are still a problem for deep space probes like Voyager due to cosmic rays that can produce spontaneous memory cell flips [**Tom94**]. However, given a reasonably deterministic, earth-bound hardware, the voting technology devised for such applications can hardly be used. As a matter of fact, a common design principle used in decentralized computing (e.g., peer-to-peer-systems) is to have multiple participants calculate some result, for example who is to become the leader. The participants then use this result

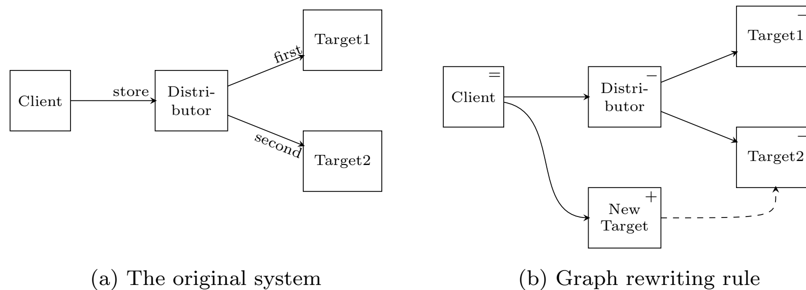
without communicating it to the other participants, as, given the same input, it can be trusted that they reached the same conclusion.

In taking the idea of n-participant voting to software engineering, the approach of "n-version programming" [**Avi85**] was devised. Here, multiple programmer teams were expected to work on the same problem, producing hopefully independent results that could be run in parallel, providing redundancy should one of the implementations fail. For example, the Space Shuttle has a backup flight computer with a minimalistic software that is supposed to take over if the primary computers (a redundant array of four computers, running the same software) encounter the same software bug [**HP83**]. This idea of n-version programming has been subjected to much criticism [**KL86**]. Nevertheless, it is still believed that fault tolerant software is necessary in high-risk environments, and cannot be replaced [**Wil00**].

In the following example, we will utilize a moderate assumption of n-version programming, assuming the existence of a non-functionally well-performing, but functionally untrustworthy component as well as a functionally reliable, but non-functionally inferior replacement. Efficiency concerns stipulate the use of the former component, with the latter one as a backup implementation. In the case of a functional failure, which we expect to be recognized immediately, a switchover by reconfiguration is to take place, including a state transferal. The problem here is that the state of the old component might have become corrupted due to the failure; either due to an incomplete processing of a method or due to a bug that mangled the data. Here, we propose an architecture that can help to cope with such a situation.

8.4.3.1. *Maintaining a Sane State.* Consider the untrustworthy component to be one that stores data, maybe an accounting system. We know that this component fails sometimes, and if it does, we need to assume that its state is corrupted (i.e., we cannot make any safe assumption about it). We assume, however, that the component is deterministic (i.e., the failure is determined only by the communication sequence, not by some nondeterministic internal choice), and that an error is communicated back to the callee immediately (e.g., in an actual implementation, by throwing an exception).

The basic idea (see Fig. 8.22a) is to maintain two copies of the unreliable component (Target1 and Target2), and to issue calls (from Client) to both of them (by Distributor); the second copy only receives the call after the call to the first copy has been completed successfully. Hence, any call issued to the second copy will successfully complete, so the state remains intact. Upon detecting a failure of the first copy, reconfiguration is employed to substitute the storage subsystem by another component (NewTarget, see Fig. 8.22b), which might be a different, maybe less



(a) The original system (b) Graph rewriting rule

Figure 8.22: Fault tolerance components

method $\mathit{Distributor}::\mathit{sync}(\mathit{prm})$:
  $\mathit{ret} \leftarrow \mathrm{call}(\mathit{first}, \mathit{sync}, \mathit{prm})$.
  if $(\mathit{ret} = \mathit{err})$ **then** fail
  else $\mathrm{call}(\mathit{second}, \mathit{sync}, \mathit{prm})$.
    $\mathrm{return}(\mathit{ret})$.success **fi**

method $\mathit{Distributor}::\mathit{async}(\mathit{prm})$:
  $\mathit{ret} \leftarrow \mathrm{call}(\mathit{first}, \mathit{async}, \mathit{prm})$.
  if $(\mathit{ret} = \mathit{err})$ **then** fail
  else $\mathrm{call}(\mathit{second}, \mathit{async}, \mathit{prm})$.
    success **fi**

method $\mathit{Unsafe}::\mathit{sync}(\mathit{prm})$:
  if $(\mathit{prm} \in E_{\mathrm{sync}})$
  then $\mathit{state} \leftarrow \mathit{err}.\mathrm{return}(\mathit{err})$
  else $\mathit{state} \leftarrow \mathit{su}^{\mathrm{sync}}(\mathit{state}, \mathit{prm})$.
    $\mathrm{return}(\mathit{sr}(\mathit{state}, \mathit{prm}))$ **fi**.
  success

method $\mathit{Unsafe}::\mathit{async}(\mathit{prm})$:
  if $(\mathit{prm} \in E_{\mathrm{async}})$
  then $\mathit{state} \leftarrow \mathit{err}.\mathrm{return}(\mathit{err})$
  else $\mathit{state} \leftarrow \mathit{su}^{\mathrm{async}}(\mathit{state}, \mathit{prm})$.
    $\mathrm{return}(*)$ **fi**.
  success

method $\mathit{Safe}::\mathit{sync}(\mathit{prm})$:
  $\mathit{state} \leftarrow \mathit{su}^{\mathrm{sync}}(\mathit{state}, \mathit{prm})$.
  $\mathrm{return}(\mathit{sr}(\mathit{state}, \mathit{prm}))$.success

method $\mathit{Safe}::\mathit{async}(\mathit{prm})$:
  $\mathit{state} \leftarrow \mathit{su}^{\mathrm{async}}(\mathit{state}, \mathit{prm})$.
  success

where, for $m \in \{\mathit{sync}, \mathit{async}\}$, $E_m \subseteq \mathscr{V}$ are sets of error-inducing parameter values and $\mathit{su}^m : \mathscr{S} \times \mathscr{V} \to \mathscr{S}$ are functions that produce the updated state of a component as the effect of method m, and $\mathit{sr} : \mathscr{S} \times \mathscr{V} \to \mathscr{V}$ is a function yielding the result of *sync*.

Figure 8.23: Specification of the fault tolerance example

efficient implementation, or some stub that takes care of a graceful and recoverable system shutdown. This is similar to check-pointing, where the state is exported frequently in order to be able to recover a computation; a technique well established for providing fault tolerance to systems with mobility, e.g., agents [**PY04**] or peer-to-peer systems [**Spr06**].

Let the interface I provided by the unreliable component consist of the methods sync for synchronous and async for asynchronous invocations; and let the four components involved in the initial component graph (see Fig. 8.22a) be given by

- $(\mathrm{Client}, \{\}, \{\mathrm{store} \mapsto \mathrm{I}\}, \zeta_{\mathrm{Client}}, \iota_{\mathrm{Client}})$,
- $(\mathrm{Distributor}, \{\mathrm{I}\}, \{\mathrm{first} \mapsto \mathrm{I}, \mathrm{second} \mapsto \mathrm{I}\}, \zeta_{\mathrm{Distributor}}, \iota_{\mathrm{Distributor}})$,
- $(\mathrm{Target1}, \{\mathrm{I}\}, \{\}, \zeta_{\mathrm{Unsafe}}, \iota_{\mathrm{Target}})$ and $(\mathrm{Target2}, \{\mathrm{I}\}, \{\}, \zeta_{\mathrm{Unsafe}}, \iota_{\mathrm{Target}})$.

with arbitrary initial states ι_{Client}, $\iota_{\mathrm{Distributor}}$, and ι_{Target}. For the method environment μ_{Client} we only assume that it keeps sending an arbitrary stream of sync and async messages to Distributor. The method environments $\zeta_{\mathrm{Distributor}}$ and ζ_{Unsafe} are specified in the upper part of Fig. 8.23, in a pseudo-code notation: Distributor relays synchronous and asynchronous messages from Client to the store components Target1 and Target2 via the roles first and second, but always uses synchronous calls such that the end of message processing becomes known. Target1 and Target2 are modeled as "unsafe": A method $m \in \mathrm{I}$ called with a parameter value from a fixed set E_m will result in an error, which is communicated back by returning *err*, after destroying the state by setting it to *err*. Fixing E_m makes failure deterministic, such that Target2 cannot fail if Target1 just succeeded. When *err* is handed back to Distributor, the component issues fail, which triggers the reconfiguration.

8.4.3.2. *Monitoring and Assessment.* When the component Distributor has moved to fail, a reconfiguration has to be launched. This is an example of error-triggered reconfiguration (cf. Sect. 7.5.3), and we utilize an exception monitor to have the assembly code informed of the error's occurrence. Since we already operate a component setup with special provisions for fault tolerance, it is sensible to assume

that we also have a plan prepared that can be instantiated as soon as the error is detected.

8.4.3.3. *Planning.* The plan to be employed is illustrated in Fig. 8.22b using a single push-out graph transformation rule as described in Sect. 8.1.4.1; the dashed line is an "update edge" indicating state retrieval. In our usual notation, this shallow plan $(A, R, \alpha, \rho, \varsigma)$ is represented by

$$A = \{(\text{NewTarget}, \{\text{I}\}, \{\}, \zeta_{\text{Safe}}, \iota_{\text{Target}})\} \ ,$$
$$R = \{\text{Distributor}, \text{Target1}, \text{Target2}\} \ , \quad \alpha = \{\} \ ,$$
$$\rho = \{\text{Client}, \text{store} \mapsto \text{NewTarget}\} \ ,$$
$$\varsigma = \{(r, \text{NewTarget}) \mapsto r(\text{Target2})\} \ .$$

where the method environment ζ_{Safe} is specified in pseudo-code in the lower part of Fig. 8.23. During the possibly interleaved plan execution according to Sect. 6.7.2, not only the untainted state of Target2 is copied to NewTarget, but also those messages that have not yet been processed by Distributor are moved to NewTarget (due to using a shallow plan, they follow the rewiring). This also contains the problematic message that triggered the error in Target1; the application of RCSTOPF re-enqueues this message such that it is not lost. Due to Prop. 6.1 reconfiguration is thus entirely transparent to Client; if the new component behaves like Target1, the client component cannot directly observe whether the reconfiguration was conducted.

8.4.3.4. *Execution.* In this example, execution needs closer inspection: In order not to corrupt the state of Target2, it has to be triggered exactly after Distributor learned about the failure of Target1. In this situation, Target1 and Distributor have moved to fail and are hence blocked until they get removed by the reconfiguration. Both components still keep a reference about the method that caused all the trouble, which is important, as it must not get lost, but has not yet been applied to the state that is about to be copied from Target2 to NewTarget. Using the rule RCSTOPF, the message is re-enqueued prior to shutting down the component, and hence δ will move the method m that was executed by Distributor when the error was detected to NewTarget, taking the first place in the new component's queue. By copying the state from Target2, exactly the state effected by the messages sent before m is retained. Hence, no gap in message processing is observed.

8.4.3.5. *Experience and Discussion.* Like many examples in this chapter, this example needs to make a lot of assumptions: Determinism (which is quite credible) and the existence of a suitable, non-failing substitute (which is maybe not so credible, although component-based software engineering encourages keeping early, unoptimized versions of components for comparison and fallback strategies; cf. Sect. 5.5 for a related experience with CMC. Also, the backup flight computer on the Space Shuttle is actually just such a component with severely degraded, but sufficient capabilities.). Further, we require state accessor methods for the Target2 component (depending on the source of the components, not too likely) and the immediacy of error detection (quite unlikely). Also, maintaining an exact duplicate of a data-intensive component for fault tolerance might not be allowable.

Depending on the system at hand, however, such schemes might be employable. Instead of using an exact duplicate, a check-pointing component might be used that dumps the contents of the target component at frequent intervals and keeps a record of messages sent after the last checkpoint was stored, thus making the system capable of tedious, yet effective reproduction of the state the failed Target1 component had. The replacing component NewTarget might be an old version that was replaced in a hot code update before, as the SIMPLEX framework [**SRG96**]

proposes. Or it might be a component that just dumps the state and signals the error in a way that allows for debugging.

Obviously, there are too many restrictions and special cases involved to declare this example as generic; yet it might have shown a situation where available components – even legacy components – can be assembled in a fault-tolerant setup without needing too much glue code or component modification.

8.4.4. Software Phases – Cross-cutting Performance Tuning. The overflowing message queue example of Sect. 8.4.2 requires reconfiguration to mend a problem of the system that will lead to erroneous behavior (i.e., an out-of-memory error). Now, we will look at a different class of reconfiguration scenarios, where the systems behavior is to be improved rather than fixed. The difference between these two scenarios is not well defined, as, for example, using a memory-preserving component might become both beneficial in one situation and crucial in another, depending on the memory available.

In this example, we will consider an application that will run unperturbed without any reconfiguration, but that will benefit performance-wise from replacing certain components by reconfiguration. We assume that multiple interchangeable implementations of a common interface exist, and use reconfiguration to employ the implementation most suitable for the current usage scenario, similar to the example reported in [**Yel03**]. As a concrete example, we will use various implementations of `java.util.Collection`, wrap them in components, and choose the most suitable implementation for a certain usage profile.

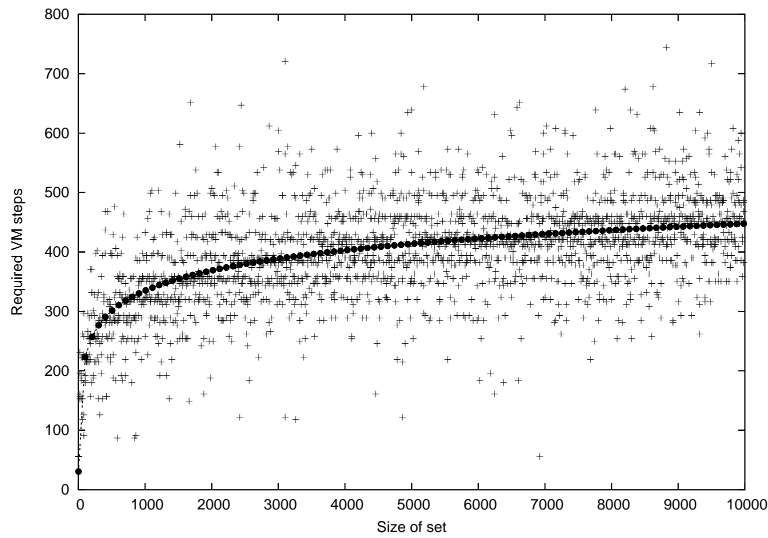
Let $\mathcal{R}$ be a finite (and usually quite small) set of *resources*. A *component method resource cost function* is a function $\mathit{cost} : \mathcal{C} \times \mathcal{M} \rightharpoonup (\mathcal{R} \to (\mathcal{S} \to \mathbb{R}_{\geq 0}))$, which is defined for all elements $\{(c, m) \in \mathcal{C} \times \mathcal{M} \mid m \in \bigcup_{i \in I_P(c)} i\}$. Such a cost function assigns to each method implemented in a component a measurement for a resource and a component state.

For example, the set $\mathcal{R}$ might consist of the sole value *time*, and a cost function assigns to each method provided by each component the time that is required to execute it, depending on the component's state. Fig. 8.24 shows the measurements of the number of virtual machine steps required by two different kinds of search operations conducted by a `java.util.TreeMap` instance: The first graphic depicts the lookup of a randomly chosen element, while the second graphic depicts the lookup of the smallest element. The first graphic shows a lot of scattering, while the second graphic shows deterministic behavior. Both exhibit a logarithmic progression, as depicted by the dotted line, which is a logarithmic curve fitted by the Ehrenberg-Marquette algorithm [**Lev44, Mar63**]. This is of course not coincidental, as the `TreeMap` employs an algorithm that exhibits a (worst-case and average) runtime complexity of $O(\log(n))$. The exact measurements of VM steps taken yields the cost function f which is partially defined as

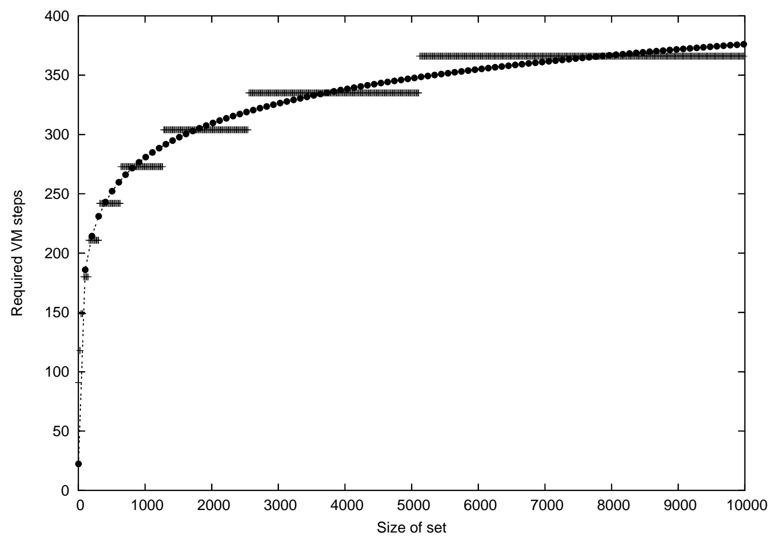
$$\begin{aligned} & f(\texttt{java.lang.TreeSet}, \texttt{contains}) = \\ & \quad \mathit{time} \mapsto (48.9862 \cdot \ln(0.91984 \cdot \texttt{size}) + 0.615969). \end{aligned}$$

This gives an average-case cost function based on the observations made. We utilize VM steps here, because it is very difficult to do measurements at the granularity involved in this example. Available timers have a granularity more coarse than a single operation on an efficient data structure (e.g., JAVA's `System.currentTimeMillis()` only provides a granularity of 1 millisecond on LINUX, and other methods are subjectible to large amplitudes due to process switching, e.g., when counting the CPU cycles. We hence implemented our own virtual machine (in JAVA, using the existing memory management[3]) which is then capable

[3]The implementation can be found at http://www.pst.ifi.lmu.de/~hammer/jvvm/.



(a) Seeking random elements



(b) Seeking the smallest element

Figure 8.24: Time measurements of `java.util.TreeSet`

of doing exact counting of bytecode instructions executed. There are a number of peculiarities, like native calls (most notably, `System.arrayCopy()`), but since wall-clock-time measurements are infeasible, this seems to be the best approach. Obviously, it is not possible to just add an element a million times and measure the mean time required, as the data structure's size changes due to the insertion. This is why the *cost* function includes the component's state as a parameter.

8.4.4.1. *Monitoring and Assessment.* Following the idea of concept drift described in Sect. 8.1.3.2, we can evaluate if the component is a good choice for its *recent usage* by using a *limited history monitor*. By evaluating the different costs for

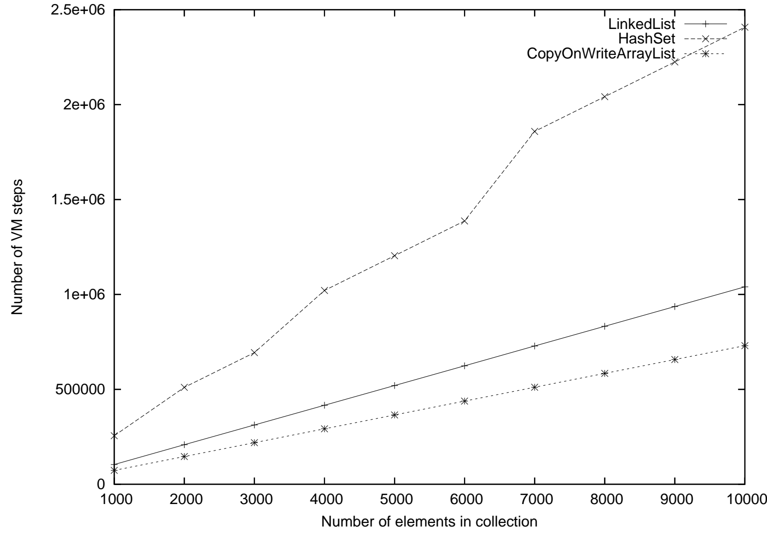


Figure 8.25: Cost of reconfiguring `java.util.TreeSet`

the algorithms provided by various possible substitutes for the component under observation, we can maybe find a component that will perform better.

Of course, it is imperative that such a reconfiguration retains the data stored in the collection in order to be entirely opaque to an external observer. Fig. 8.25 shows the price that has to be paid in order to reconfigure a `java.util.TreeSet` (that is, a component encapsulating such a data structure) into other subclasses of `java.util.Collection`. This price does not yet include the overhead of reconfiguring the components, but this can be regarded to be constant. If it can be assessed that the recent use warrants the cost of reconfiguration to a component that is more suitable, we can do reconfiguration.

8.4.4.2. *Planning and Execution.* The situation we are looking at is given by a component that wraps a data structure like a `java.util.Collection` implementation. It is used by other components, but does not require connections on its own. Defining a shallow reconfiguration plan for substituting a component c by c' in a component setup (C, M, e) is therefore straightforward:

- $R = \{c\}$, $A = \{c'\}$,
- $\rho = \{(c'', r) \mapsto c' \mid c'' \in C \wedge r \in \mathscr{R} \wedge M(c'')(r) = c\}$, $\alpha = \{\}$,
- $\varsigma = \{(f, c') \mapsto f(c)\}$.

Obviously, implementing ς can be a little troublesome, but, for this example, this is easily managed by using the `toArray` method of `java.util.Collection`.

If there is a sole user of the `Collection`-wrapping component, the reconfiguration plan is injective as well, providing a clean reconfiguration that can be triggered anytime. Otherwise, message order needs to be established by timestamps, as described in Sect. 6.7.4.

8.4.4.3. *Experience.* Fig. 8.27 shows a very artificial example: A collection representation (initially, a `java.util.LinkedList`) is supplied with a number of elements. Afterwards, the list elements are iterated a few times. Using a linked list is a good choice for inserting elements, since it requires constant time only. The bulk of maintaining the list structure does not make it ideal for iteration; here, a `java.util.CopyOnWriteArrayList` excels. This collection implementation just

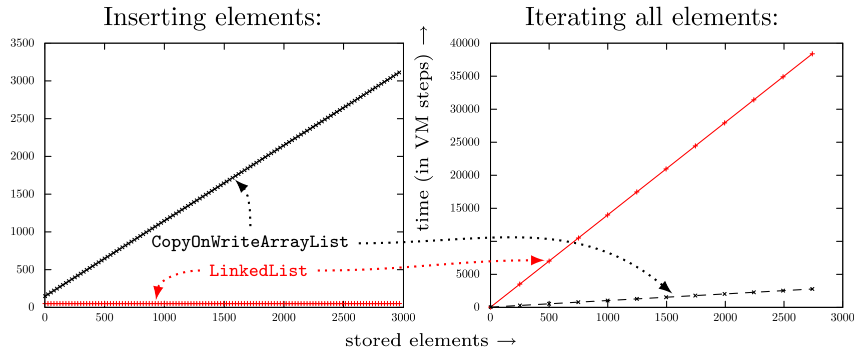


Figure 8.26: Comparing insertion and iteration efficiency for `java.util.LinkedList` and `java.util.CopyOnWriteArrayList`

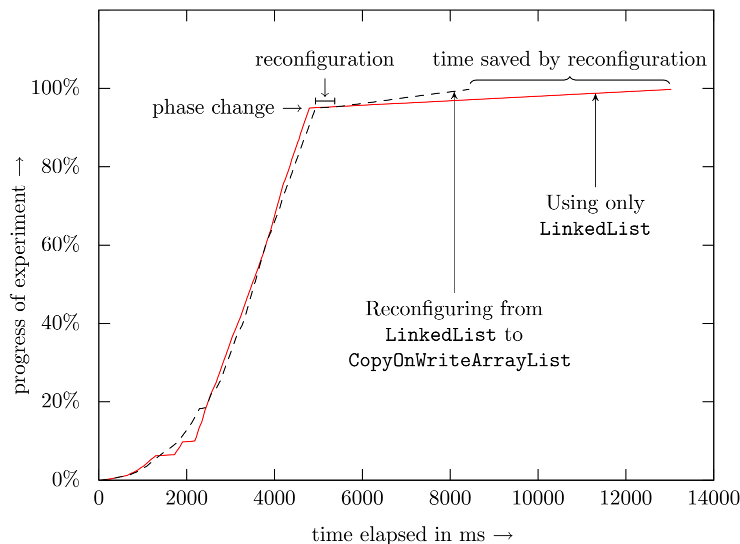


Figure 8.27: Comparing two experiments to illustrate the speedup obtained by reconfiguring to a more suitable data structure

wraps an array; each element insertion results in a complete copying of the data structure. Hence, an iterator can be made very fast, since it can omit many checks. On the other hand, using such a `CopyOnWriteArrayList` is prohibitively slow for adding elements. The performances for adding and iterating elements is compared in Fig. 8.26.

The best strategy, given these two data structures and the application behavior detailed above, is to use a `LinkedList` for element insertion, and reconfigure it to become a `CopyOnWriteArrayList` once the iterations start. Fig. 8.27 illustrates that doing so outperforms the sole use of a `LinkedList`, although monitoring and reconfiguration overhead is required. Obviously, such an example is very much unlikely to be ever found in practice. There are, however, examples where different uses of data structures require their (usually, manual) reconfiguration: For example, if large bulk inserts are to be made to a data base, it is advisable to turn off indexing and other provisions for providing fast read access before, as they interfere with the

bulk writing (cf. the discussion in Sect. 8.1.3.2). Reconfiguration can be employed to automatically choose a data structure that exhibits best runtime behavior for the current situation.

8.5. Reconfiguration for CMC

In this section, we will discuss the applicability of reconfiguration to the CMC model checker. Ever since the first version of CMC was finished, we have pondered on ways to utilize reconfiguration for CMC. However, this never came to pass; and this section will describe why.

The CMC model checker, as described in Chapter 5, seems to provide ample opportunity to do reconfiguration. After all, a lot of configuration is done at startup time, which includes choosing the size of the caches and buffers, the state removal strategy (i.e., the actual algorithm that decides which state should be moved from the lossy hash table to the disk), and, most importantly, the overall architecture, i.e., if the disk should be used, if states should be compressed after creation[4], if the single-successor-strategy should be employed, and much more. Most of these choices are dependent on the characteristics (most notably, the size) of the model at hand, and since most of these characteristics only become known during the model checking run, an adaptation to the newly learned facts might pay off handsomely.

The choice of a good configuration can greatly improve the "throughput" of the model checker, which is ultimately measured in the time to finish the search; or, more discriminating, in the ability to exhaustively search a model's state space without running out of memory. It is not uncommon to halve the required runtime just by adjusting the sizes of the different buffers and caches – and this relates to a situation where experience was used to start with already well-chosen parameter values.

Despite this necessity to consider the *configuration* of CMC, there is no genuine need for runtime *reconfiguration* in order to increase the throughput:

- Many of the relevant parameters of the model – e.g., state size, state connectivity, compressibility of states – can be found out in just a few seconds or minutes of running the model checker. In preparing a run that will take days to complete, this price can be paid easily.
- Large model runs spend most of their time in what would be the final configuration: A diminished portion of states residing in the lossy hash table, with much time spent on the disk look-ups. Switching to disk-based model checking after depleting the memory, reorganizing it to make room for the Bloom filter and the disk buffers just postpones the start of this lengthy final phase, and the overall speedup is very little, even if we neglect the price of the reconfiguration itself. For example, the LUNAR example 5, for which the exhaustive state space exploration took about 1.5 days to complete, started swapping states to disk after 3:45 minutes. Approx. 4GB of the 16GB available were allocated to the lossy hashing, so even if we neglect reconfiguration cost and the size of other indispensable data structures (e.g., the open set), a dynamic approach could only postpone the use of the disk by 11 minutes.

[4]This is usually a bad choice – it consumes computational resources, and it usually just postpones the write-out to disk, where the states are compressed anyway; and since they are compressed in blocks, a better compression can be achieved. State compression might pay off if it can compress the states sufficiently to avoid using the disk at all. This is one example for a dedicated architecture that works well for a class of models, but just slows down the model checking process for others.

At the same time, the CMC component model is not very suitable for reconfiguration – due to possible recursive calls, replacing a component might lead to having code executed in the old component *after* code has been executed in the new, leading to an awkward situation. Hence, reconfiguration needs to wait until the stack of the old component is empty; and this might require the component implementer to implement the component in an altogether different way. Hence, runtime reconfiguration for CMC is not very feasible.

But reconfiguration does not necessarily have to be done at runtime. The component setup can also be rearranged between two distinct runs; the progress made in exploring the state space is lost, but this does not derogate much, as argued above. In areas not concerned with software components, like programmable chip configuration, reconfiguration is divided into runtime reconfiguration and compile-time reconfiguration [**ALF00**]. For CMC, we can use compile-time reconfiguration to build a model checker that is suitable for the task at hand. This involves three steps: Finding out the characteristics of the model that is to be checked, choosing an appropriate architecture, and running it with the model. Actually, this amounts to an *open* control loop, as the effect of the architecture choice will not be investigated (and hence will not lead to further reconfiguration)[5]. Such a tailoring of a system's configuration to a special machine or problem has seen some success with highly optimized algorithms like SAT solvers [**HBHH07**] or linear algebra software [**WD98**]; yet these approaches do not consider the data at hand, but are tuned to representative problems [**HBHH07**] or the computer hardware the software is compiled on [**WD98**]. For CMC, we tried to come up with an algorithm producing an architecture suitable for one single model.

For CMC, the most important property is the size of the state space of the model. Of course, this property is also one of the hardest to obtain – its exact value can only be found out by doing an exhaustive search in the first place. While this voids the benefit of a reconfiguration in most cases, there are a number of ways how this number can be obtained, without doing a dedicated, exhaustive search:

(1) Maybe the size of the state space is known already from previous runs. Often, the model is only slightly tuned to handle a single message in a different way, or to use an atomic block (a sequence of steps that is not interleaved with other processes' steps); in both cases, the number of states might stay roughly the same. However, it requires expert knowledge to judge this, as even small changes can blow up (or cut down) the state space by orders of magnitude.

(2) The state space might be approximated by estimating the number of combined states. If there are two processes, each with 10 states, and full interleaving is possible, we can estimate the state space to contain 100 states. We have tried this approach in the context of [**Now07**], but the estimates usually over-approximate the actual size (as they tend to ignore restrictions of the state space due to inter-process dependencies), again by orders of magnitude. Also, this requires expert knowledge.

    For parameterized models (which instantiate a process a selectable number of times, e.g., the dining philosophers protocol, where the number of philosophers can be chosen freely), the state space can be approximated by extrapolating a parameterized model (e.g., if the state space grows by a factor m for each addition of a further process instance, we can extrapolate how big a model for a given number of process instances will be).

[5] This is but a preliminary design. Adding monitoring and assessment to CMC is not hard, and the performance of the application can be compared to expected values, possibly triggering another restart of the run if the architecture proves to be insufficient.

(3) The state space might be obtainable by heuristic search, e.g., by utilization of a Bloom filter (as the sole means of checking state revisits). Bloom filters, being a set over-approximation, cut off too many branches, and hence provide a state space under-approximation. Their good performance, however, will most likely produce a good state space size approximation. Of course, almost the entire state space needs to be explicitly generated, which can also take a good measure of time.

(4) We have discussed an approach at guessing the state space size based on the characteristical shape of the open-set size development in Sect. 8.1.3.1. This seems promising, although costly, and the general applicability remains questionable.

(5) There are approaches for estimating the size of a graph based on random samples. For CMC, we used random walks to come up with some statistics on the model. Following the idea of "Monte-Carlo Model Checking" [**GS05**] and utilizing components that have been developed for CMC (most importantly, the state generator, the hash functions and a closed hash table), we built a small random walking tool that walks the graph, choosing random successor nodes, until a loop is found. Due to the reusability of components, this was done in a few hours and approximately 100 additional lines of code. Since, even for vast models, a random walk usually does not require more than a few hundred steps until a loop is encountered (i.e., a state is visited that is part of the walk path already), only very moderate memory requirements are given. Theoretically, it should be possible to derive the graph's size from the out-degree of nodes encountered, though, for unbiased estimation, the in-degree is required also [**MS89**].

    Using a biased evaluator considering the outgoing edges only, we obtained values too high by orders of magnitude. Obviously, this is due to the neglect of merging graphs (the biased estimation calculates the size of a tree, whereas unbiased estimation would need to consider the search graph as a DAG). We hence used small breadth-first searches to judge the rate of merging paths, and found that the growth of the breadth-first-search front is between 1.05 and 1.5 for various models, whereas the average outgoing degree is between 1.5 and 5. The resulting graph size estimations are better, but still they are very volatile due to the fact that we calculate the size as

$$size(G) = avg(BFSgrowth(G))^{avg(RandomWalkLength(G))},$$

    and even very modest modifications to the BFSGROWTH function result in large changes to the estimated size $size(G)$.

If the state space size is not known in advance, it is difficult to come up with a good initial configuration; and restarts with a changed configuration become more interesting. One way to do this would be the following:

(1) First, a model checker is started that attempts an exhaustive search with the available main memory. This model checker might solve the model within relatively short time, or run out of memory.

(2) If the memory is depleted, we need to start another run with a changed configuration that utilizes the disk. We have learned some important features of the model at hand: The state vector size (or its distribution), the utility of the auto-atomic filter. Also, a rough estimate on the final model size can be done using the peaking of the open set; if it has peaked

already, we can estimate that we have seen 10% to 50% of the state space so far, otherwise, we might encounter a state space much larger.

(3) Based on these information and guesses, we can start another, tuned run. We can also monitor if our guesses appear to be good; otherwise, another restart might be expedient.

While this seems straightforward, one should not underestimate the difficulties at producing a configuration automatically. The available memory needs to be split on at least four components (open set, lossy hash table, Bloom filter, candidate set); and how this should be done again depends on the model at hand (cf. the difference between the two graphs in Fig. 5.10 on page 100) and the state space size. Ultimately, more experience needs to be gained with CMC to provide rules for calculating optimal memory distribution. In this light, automated restarts with reconfiguration might prove vastly helpful for CMC (if only to lessen the need for user-supplied configuration), but they require more research.

8.6. Discussion

Coming up with the examples of this chapter has been quite a challenge. As we have seen in Chapter 3, few reconfiguration frameworks provide any examples at all. This is most likely not because no examples exist – but because it is hard to find concise, yet meaningful and credible examples. Small examples often appear artificial or too cumbersome; we have discussed some reasons in the context of the CMC model checker in the last section. The examples in this chapter are chosen to be representative for specific areas of reconfiguration use. Approaching the problem of defining examples from this direction proved helpful, and we hope that the examples can illustrate the general utility of reconfiguration.

Reconfiguration can only be motivated by the separation of roles, and neither of these examples does have a credible separation of persons associated – they are all devised, written and assembled by the same person. If we go one step further and claim that reconfiguration excels at handling changes of cross-cutting concerns, this is even more problematic: The examples are geared at showing the reconfiguration, so their true core concern is that of illustrating reconfiguration. Obviously, a solution would have been to implement a large-scale example using JComp, as CMC was implemented for its associated component framework; but given the experience with CMC which does not accommodate reconfiguration well planning such a large-scale example is difficult. We hence cannot give an example where a true cross-cutting concern is involved, but only provide small-scale scenarios which illustrate a situation that can be expected to be found in large-scale applications and requires reconfiguration.

The example closest to being a real example is that of resource preservation, described in Sect. 8.4.2, as it was motivated by an unforeseen problem with the web crawler of Sect. 7.4.2. But this example also shows the problem with reconfiguration: It is hard to implement a working mechanism to cache the queues, and the solution is not very elegant, as the disk is limited as well. Utilizing external storage capacities for swapping parts of the in-queue will only postpone the point of failure. Also, first-class connectors might offer a swapping capability without reconfiguration; for JComp, we might substitute this by instrumenting each connection with an appropriate filter component at configuration time. Such considerations always need to be made when judging the benefits of reconfiguration, and they need to consider the application domain. The web crawler might not be a suitable scenario, since the problem of overflowing queues is not truly a cross-cutting concern, but relates to the core algorithm directly. For an application that is subjected to phases as discussed in Sect. 8.4.4, adding a temporary remedy for overflowing message

queues by reconfiguration might pay off due to the decrease in complexity for the initial configuration and the improved performance.

CHAPTER 9

Component Correctness and Correct Reconfiguration

With this division of responsibility established,
matters went downhill quickly.
— T. A. Heppenheimer, Development of the Space Shuttle

Specification of components, their communication and their assembly is an area that has seen much research (e.g., [**HTC98**, **PV02**, **HJK08**]). The explicitness of communication is attractive, since it defines the set of different events that need to be considered. At the same time, even very basic asynchronously communicating systems are Turing complete [**Rus08**], making even the most fundamental specification language impossible to be verified for vital properties.

It is well beyond the scope of this thesis to provide satisfying (in theory and practice) means of component specification. We will, however, discuss how components can be specified for the JComp component model; which, being asynchronous, requires a consideration slightly different from many prevalent ones (e.g., [**dAH01**]).

This is useful for two things: First, it sheds a different light on the separation of roles. Especially, the difficult question of concurrent pre-/post-condition can be answered in the context of component-based software engineering in an elegant way that underlines the benefits obtained from splitting the responsibilities between different roles. Second, we can use this kind of specification to probe into the problems of reconfiguration, as we will do in the second part of this chapter, starting with Sect. 9.4.

9.1. Concurrent Contracts

There is some debate on when to evaluate a method's precondition in a system that supports message queues, and whether contracts are beneficial for such systems at all. This discussion stems from the fact that, in general, a precondition can only be met if the caller gathers some information about the target component's state before. Here, we will use the example of a bank component that offers a `balance` and a `debit` method, with the following specification:

$Bank$ ::`balance`() :	$Bank$ ::`debit`($amount$) :
pre: **true**	**pre**: $amount > 0 \wedge amount \leq balance$
post: $result = balance$	**post**: $balance = balance@pre - amount$

First, observe that the precondition consists of two clauses: The *independent* clause (or *correctness condition* [**AENV06**]) $amount > 0$ that places a constraint on the *account* parameter and is independent of the target component or any other data structure that is changed over time. Obviously, such a constraint can be verified at any point in time between issuing and processing the method call, and no difference to a non-concurrent environment are to be considered. The *dependent* clause $amount \leq balance$, however, restricts the set of states in which the method may be invoked to the set of all states that have a sufficiently large balance.

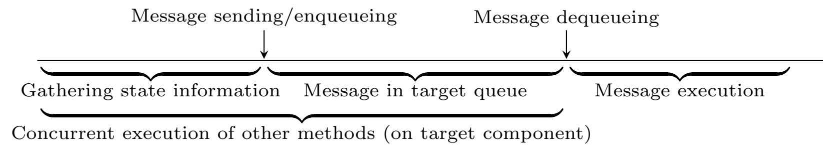


Figure 9.1: Phases of message execution

The problem with the dependent precondition clause is that any client that calls `debit` needs to obtain the current *balance* value first by calling `balance`. In the time between the return of this query and the execution of `debit`, other clients might have also had `debit` messages processed, which lower the *balance* so that the precondition of `debit` is no longer met. Fig. 9.1 illustrates the abstract problem.

Of course, this problem is well-known. Betrand Meyer, the advocate of contract-based development, names it as the reason why contracts are "inapplicable" for concurrent environments [**NAM03**]. His remedy is to redefine the semantics of the precondition to be no longer just a check, but a wait statement instead; the message execution is postponed until the precondition is satisfied again [**Mey97, NAM03**].

We refrain from such a solution, because such an interpretation of preconditions vastly interferes with the established communication paradigm, like non-overtaking of messages. Scoop [**AENV06**], a framework for concurrent programs that supports contracts, first introduced the requirement for eventual precondition satisfaction for the client [**NM06**] (i.e., the client is responsible for ensuring that the precondition eventually holds), and later departed from a strict waiting solution by allowing for user-level exception handling [**BP07**], thus allowing the user to provide an own interpretation of how to handle violated preconditions. We assume that this departure from a predefined strategy for handling violations reflects the experience that waiting is not always applicable.

Depending on the implementation, either problems with deadlocks will emerge (if the caller is blocked) or message ordering is greatly complicated (if the call is postponed). After all, the component developer makes assumptions about the effect sent messages have on their target component, and if their execution order is modified by the framework, only very defensive assumptions can be made. Also, the solution is very technical, and does not really cope with the underlying problem: If some money is withdrawn from a bank, this is done for using the money elsewhere, and if the withdrawal is postponed, so is its use. In a practical example, if we withdraw money to pay for a bill, and the call is postponed until enough money is available (which might take a complete month), we get into legal problems. Of course, the situation is difficult anyway, with there not being enough money to pay for said bill, but this has to be made known to the user, and not just be postponed until better days.

9.1.1. Interpreting Contracts in Concurrent Environments. Let us assume that no remedy is enforced, and that the bank indeed allows concurrent debits while disallowing overcharging. Let us consider the canonical problem scenario: Customer C_1 retrieves the balance b, customer C_2 retrieves the same balance b, customer C_1 issues a debit of $0 < m_1 \leq b$ money, then customer C_2 issues a debit of $b - m_1 < m_2 \leq b$ money. After the debit of the m_1 amount is executed, the precondition for the debit of the m_2 amount no longer holds. When should this be declared an error?

Let us recall that the responsibility of satisfying the precondition is placed on the caller. This makes it appear reasonable to have the precondition checked at the time of issuing the call, as this is the last point in time where the caller has any control of the method call. Of course, this raises the immediate question on what to do if the precondition is violated at the time of method execution begin. Also, preconditions can be violated because of outdated information (in the example above, if the debiting of m_1 money is executed after C_2 retrieved the balance b, but before the debiting of m_2 gets issued).

Our approach to this problem is to assume a third responsibility: The *system designer*. We understand component-based systems as composed from different components, each with a limited local view, but being assembled by a single person (or a closely collaborating team), which is well-aware of the way the components interact (we stressed this point as the major difference to services in Sect. 2.2.2). A single component cannot guarantee that it respects the preconditions of the methods it calls on other components without some *inter-component protocol*. The localized view of a component, i.e., the inability to see anything but the provided interfaces of the components it is connected to, put it beyond the capability of a single component (or, more practically, the implementer concerned with a single component only) to follow such a protocol. It is the task of the system designer to choose appropriate components such that the global protocol is followed.

By assuming such a role, we can determine the time of precondition checking where it makes most sense: At dequeueing the message. While the caller usually does not have the means to ensure that the precondition is met at message execution, a system designer has. The time of dequeueing is also the most important one for message execution; here the state in which the message execution is begun has been reached (and will not become modified by a method other than the dequeued one until method execution is finished, as guaranteed by the mono-threadedness of the components). Hence, validating the precondition at message enqueueing is not sufficient to ensure proper execution.

Another possible interpretation of contracts might be the requirement to have a valid precondition at both message enqueueing and dequeueing. This can even be extended to the requirement of having a valid precondition at any point in time between (and including) those two events. We do not use this approach here; it rules out perfectly legal sequences of messages, e.g., for two asynchronous messages A and B of a component C with the contracts

spec C::A() :	and	**spec** C::B() :
pre: state=1		**pre**: state=2
post: state=2		**post**: state=3

the sequence A, B may lead to precondition violations (if A is not executed fast enough and before B arrives); but if A and B are interpreted as, say, initialization methods, there is nothing wrong with invoking them in this fashion.

So, we decide to verify a message's precondition at the point where the message's processing begins. It is then placed on the system designer to make sure that no violations occur. This approach is motivated by the separation of roles paradigm and the specific roles involved in writing a component system – and their knowledge and *assumptions*.

9.1.2. Assumptions and Guarantees. In an ideal world (at least, ideal in the sense as envisioned by McIlroy), the component implementer is unaware about how the component that gets written is to be used. Being not his concern, he is not obliged to anticipate the exact communication pattern the component will be subjected to. Obviously, such a deployment of a component in a totally unspecified environment is only possible for very simplistic components (although, in practice,

most components tend to be very simplistic in this regard, e.g., a hash table or a RSS feed loader). For components conducting a more elaborate communication, the environments they can be utilized in are more restricted, as they need to provide this communication behavior. Here, we will call such a communication restriction an "assumption", being made by a component about its environment.

Assumptions can take different forms: In its easiest form, an assumption is a static restriction of the method parameters. For example, a component might provide a method `sqrt(int n)` that implements the function $\sqrt{n} : \mathbb{N} \to \mathbb{N}$. With the parameter being an integer, it has to be stated explicitly that negative values for n are not allowed. Another example is the requirement $amount > 0$ in the specification BI.`debit` at the beginning of this chapter.

The restriction of method parameters may also depend on the component's state. An example is given by the requirement $amount \leq balance$ in the bank example given above; here, it is required that the parameter *amount* does not exceed the state variable *balance*.

A further restriction might be given on the state in which a message can be accepted at all. For example, many components require an explicit call to a method `initialize` before they can process other requests. This can be reflected in the state by maintaining a flag `isInitialized`, which is set by the `initialize` method and required to be set by any other method. Such state restrictions can also be short-termed, e.g., a component requires its client to send a `login` message before any request, and requires a `logout` message before allowing the client to sign in again. If such a message sequence has a defined start and end point, we speak about a *transaction*.

In making these assumptions about the environment explicit, the component implementer restricts the component setups the component can be utilized in to those that provide suitable communication. It then becomes the responsibility of the system designer to assemble compatible components. In order to do so, however, some knowledge about the component's own communication behavior needs to be made explicit; as we do not want to burden the system designer with reading the components implementation, we can make the communication behavior explicit as a *guaranteed* behavior, just as we do with the *assumptions* about the environment.

9.1.2.1. *Formal Definition.* For specifying assumptions and guarantees, we use a "pre-post-comm-eve"-specification following the OCL specification style [**OMG06b**, **HBKW01**]. It is comprised of four elements:

spec $C{::}M$:

 pre: *Pre*
 post: *Post*
 comm: *Comm*
 eve: *Eve*

We define these four elements as predicates:

- the *precondition* $Pre \subseteq \mathcal{S} \times \mathcal{V}$,
- the *postcondition* $Post \subseteq (\mathcal{S} \times \mathcal{V}) \times (\mathcal{S} \times \mathcal{V})$,
- the *(outgoing) communication specification* $Comm \subseteq \mathcal{S} \times (\mathscr{R} \times \mathscr{M} \times \mathcal{V})^*$,
- and the *future method requirement* $Eve \subseteq \mathcal{S} \times (\mathscr{M} \times \mathcal{V}) \cup \{\tau\}$.

Informally, *Pre* relates the called component's state and the method parameter. As observed in [**AENV06**], this can be split into two concerns: Restricting the method parameters and defining admissible component states. *Post* relates the state at method dequeueing, the state at method processing termination and the parameters. For synchronous calls, we need to relate the return value also. *Comm* relates the state at method dequeueing and sequences of method calls conducted

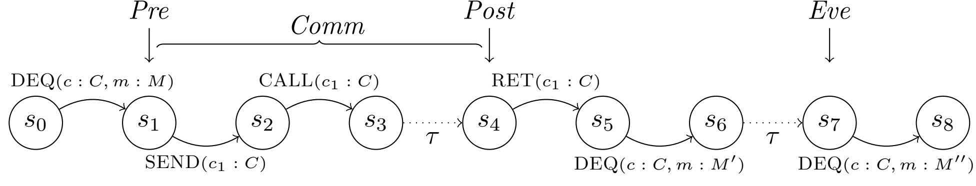


Figure 9.2: Evaluation of a method specification $C{:}M$

during the processing of the method. For including method return values, we can extend the sequence of method calls to $\mathcal{S} \times (\mathscr{R} \times \mathscr{M} \times \mathcal{V} \times \mathcal{V})^*$; but we do not do this here in order to restrict the complexity of the approach. *Eve* relates the state at method dequeueing and a method that needs to be received in the future. We will usually write the sets in a symbolic way, i.e., write $u \in \mathit{users}$ for the set $\{(s, v) \in \mathcal{S} \times \mathcal{V} \mid v(u) \in s(\mathit{users})\}$ using the "programming language" notation introduced in Sect. 4.6.

Formally, we define that the conditions of a specification $C{::}M$ hold as follows:

- The precondition *Pre* is evaluated against a pair (l, s) with l being a rule instantiation of the form $l \in \textsc{deq}(c : C, m : M, \overline{v : V})$ and s a component configuration with $c^r, \langle\sigma\rangle$. *Pre* holds in (l, s) if $(\sigma, V) \in \mathit{Pre}$.
- The postcondition *Post* is evaluated against a tuple (l, s, l', s') with l being a rule instantiation of the form $l \in \textsc{deq}(c : C, m : M, \overline{v : V})$ and l' being a rule instantiation of the form $l' \in \textsc{ret}(c_1 : C, \overline{v : V'}) \cup \textsc{deq}(c : C)$ and s a component configuration with $c^r, \langle\sigma\rangle$ and s' a component configuration with $c^r, \langle\sigma'\rangle$. *Post* holds in (l, s, l', s') if $((\sigma, V), (\sigma', \mathit{res})) \in \mathit{Post}$, with $\mathit{res} = V'$ if $l' \in \textsc{ret}()$ and $\mathit{res} = \perp$ otherwise.
- The communication *Comm* is evaluated against a run of the form $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_n} s_n$ with $l_i \in \textsc{send}(c : C, \overline{r : R_i}, \overline{m : M_i}, \overline{v : V_i}) \cup \textsc{call}(c : C, \overline{r : R_i}, \overline{m : M_i}, \overline{v : V_i})$ and states $s_i = c^r, \langle\sigma_i\rangle$. *Comm* holds in this run if $(\sigma_0, ((R_0, M_0, V_0), \ldots, (R_n, M_n, V_n))) \in \mathit{Comm}$.
- The eventuality requirement *Eve* is evaluated against a pair (s, l) with l being a rule instantiation of the form $l \in \textsc{deq}(c : C, \overline{m : M'}, \overline{v : V'})$ and s a component configuration with $c^r, \langle\sigma\rangle$. *Eve* holds in (s, l) if $\mathit{Eve} = \tau$ or $(\sigma, M', V') \in \mathit{Eve}$.

A specification $C{:}M$ holds for a run $s_0 \xrightarrow{l_0} s_1 \xrightarrow{l_1} \ldots \xrightarrow{l_{n-1}} s_n$ if for every i with $l_i \in \textsc{deq}(c : C, m : M)$

- *Pre* holds in (l_i, s_{i+1}),
- *Post* holds in (l_i, s_{i+1}, l_j, s_j) for the smallest $j > i$ with $l_j \in \textsc{deq}(c : C) \cup \textsc{ret}(c_1 : C)$ (we assume that such a j exists, otherwise, i.e., if the run ends in the mid of an execution of M, we only consider the precondition *Pre*),
- *Comm* holds in $\left(s_i \xrightarrow{l_i} \ldots \xrightarrow{l_{j-1}} s_j\right)\Big|_p$ for $p \equiv \textsc{send}(c : C) \cup \textsc{call}(c : C)$ for the smallest $j > i$ with $l_j \in \textsc{deq}(c : C) \cup \textsc{ret}(c_1 : C)$,
- *Eve* holds in (s_{i+1}, l_j) for some $j > i$ with $l_j \in \textsc{deq}(c : C)$.

Fig. 9.2 shows the investigation of a run for its conformance to a specification. A component setup conforms to a set of specifications if all specifications hold for each run $r \in \mathit{Runs}(\mathcal{L})$ for its LTSi $\mathcal{L}$.

For the (important) class of deterministic component systems (where all method evaluations of all components are deterministic component process terms) we can relate the satisfaction of a specification to the received communication, expressed in the communication traces.

LEMMA 9.1. *Let $\mathscr{T} = (S, L, T, i)$ be the LTSi of a deterministic component system, and let r_1, r_2 be initial runs with $\mathit{Traces}^{comm}(r_1) = \mathit{Traces}^{comm}(r_2)$. Then a specification of a component $c \in \mathrm{dom}(s)$ for $s \in S$ holds for r_1 iff it holds for r_2.*

PROOF. The specification of a component c only investigates the state elements of c. Hence, it suffices to show that $r_1|_{L_c} = r_2|_{L_c}$ with $L_c = \textsc{call}(c_1 : c) \cup \ldots \cup \textsc{loop}(c : c)$. Let $r_1|_{L_c} = r_1^1 \xrightarrow{l_1^1} r_1^2 \xrightarrow{l_1^2} \ldots$ and $r_2|_{L_c} = r_2^1 \xrightarrow{l_2^1} r_2^2 \xrightarrow{l_2^2} \ldots$. By induction on the length i of the run:

- $r_1^1 = r_2^1$ since the runs r_1 and r_2 are initial.
- $r_1^i = r_2^i \rightarrow l_1^i = l_2^i$: If $l_1^i \in \textsc{deq}()$, then $l_1^i = l_2^i$ since the communication traces of r_1 and r_2 are the same. Otherwise, since $r_1^i = r_2^i$, $P^{r_1^i}(c) = P^{r_2^i}(c)$, and as the component system is deterministic, l_1^i is uniquely determined, as is l_2^i. Hence, we have $r_1^{i+1} = r_2^{i+1}$. □

9.1.2.2. *The Separation of Roles.* The four elements of this specification illustrate the separation of roles quite well: it is the responsibility of the component implementer to ensure that the postcondition *Post* holds at method processing termination, and that the communication specification *Comm* is respected by the calls the component makes. As the component designer is also the author of the specification (at least in the "highly separated" scenario of McIlroy), the postcondition and communication specification are what is *guaranteed* by the component implementer. The precondition *Pre* and the future method requirement *Eve* apply to states that are not within the control of the component, hence they are *assumed* by the component (this also applies to possible return values of synchronous calls in *Comm*, which we neglect here). It is hence required from the system designer to assemble a system that guarantees that *Pre* and *Eve* will hold at the required points in time. The system designers task is facilitated by the postcondition and communication specification, which declare how the method will be executed, and what connected components can expect.

This can be illustrated with the self-invocation pattern, as described in Fig. 7.10. The specification of a method that performs self-invocation reads (with a relaxed notation which translates in a straightforward manner to the set-oriented representation of the definition):

spec $C{:}m()$:
  pre: φ
  post: ψ
  comm: $\{(\mathbf{true}, \mathit{self}.m())\}$
  eve: $\{(\mathbf{true}, m())\}$

Without considering the pre- and postcondition φ respectively ψ, we can see that C guarantees that, once m is processed, it will invoke m on its role *self* once (and then remain silent until the method processing terminates). It also requires the future reception of method m. It is then straightforward for the system designer to use the guaranteed behavior to satisfy the assumptions the component makes about the environment, and connect the role *self* to C itself. By no means, however, is there any obligation to do it exactly like this; any source of messages m, sent repeatedly, will suffice to satisfy C's assumptions. This freedom is given to the system designer as it is beyond the concern of the component where it receives the message m from.

9.2. Patterns for Precondition-Preserving Concurrency

You can't handle the truth!
— Aaron Sorkin, A Few Good Men

The problem of concurrent precondition violation can be phrased as the problem of making assumptions about that target component's state that become obsolete by concurrent access. The prevention of such a situation is comparable to the search for *serializability* [**Pap79**] in the context of databases. Serializability states that the effect of concurrent database reads and writes must not be distinguishable from non-parallelized, i.e., serial, execution of these operations.

The following patterns establish the preservation of an assumption about another component's state under concurrent modification by various approaches. We will refer to the assumption as a predicate, because it actually describes a subset of the component's state that we assume the component to be in. We will use the bank example of Sect. 9.1 to illustrate the different approaches.

9.2.1. Architecture. Of course, the easiest way to avoid concurrent falsification of assumptions is to inhibit concurrency. If there is no other client attached to the bank, there will be no problem with concurrent debits.

Such a pattern is not suitable to fix a problematic situation, of course. We mention it here because most of the time, there will be no problem with concurrent preconditions due to an architecture that avoids problematic concurrent access.

9.2.2. Quotas. If concurrency is desired, we can try to inhibit that the assumption can be violated concurrently. One way to do this is to make sure that the various clients only know about, and only modify, a distinct section of the state space. In the problematic example, the assumption predicate formed about the state space of the target component reflects the total balance of the account. By having the predicate be about an *allocated* amount, i.e., a quota, we can avoid interfering debits to invalidate the assumption.

In the "real world", quotas and sub-accounts may serve many purposes, but they also provide a means to avoid concurrency problems. The idea is to split the commonly requested quantity (here, the account's balance) and allocate an amount to each user. Thus, the concurrency problem is mitigated, at the risk that misallocation might keep users from debiting money that exceeds their slot, while a sufficient amount is still given in the other slots. This then makes complicated (and, again, problematic with respect to concurrent access) re-booking necessary.

The benefit of quotas, in our setting, is their complete transparency. They can be retrofitted to an existing system, possibly even by reconfiguration, as neither the client nor the bank need to be adapted. A proxy component serves as a quota manager by keeping a balance of its own. This balance needs to be assigned initially by a quota manager, which can later also take care of reassigning quotas for new incomes, or handle the re-booking, should it become necessary. To obtain the initial balance, the manager itself needs to be connected to the bank. Fig. 9.3 illustrates this setup.

9.2.3. Protocols. Another problem of the debiting example is that the specification of the bank's provided interface is oblivious to the fact that there might be concurrent access. The problem is that the bank does not relate the result of the `balance` method to subsequent effects of `debit`. If the bank was designed with concurrency in mind, a protocol might have been employed to avoid the problem of "concurrent invalidation of assumptions" – e.g., that a client gets an answer that

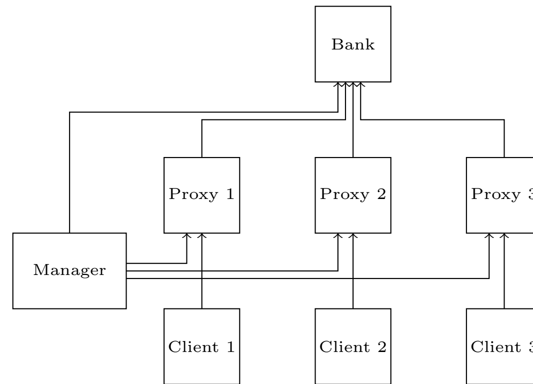


Figure 9.3: Bank example with quotas

she uses to calculate a seemingly valid request, only to find that concurrent access has made that answer obsolete. For example, instead of querying the balance, the client might issue its intent to debit a certain amount of money. The bank then allocates that amount for subsequent debits, or neglects the request if the balance is too low. A debit following a successful allocation will then always succeed. Of course, if the client decides to abort the debit, the amount might be locked forever, unless quantitative measures are used (e.g., the amount stays assigned for a limited time period. This is also prone to concurrency problems, if the deadline is tight and concurrent requests delay the processing of the `debit` message – so if such an approach is attempted, quality of service parameters need to be provided also. This is well beyond the scope of this thesis.) The assumption predicate, however, will again be concerned only with a per-client portion of the state space, and, like previously achieved with quotas, be uninfluenced by the operations of other clients.

We can even retrofit this approach to an existing legacy system, by adding a proxy to the existing bank that provides such a protocol, as shown in Fig. 9.4. The client, however, needs to follow the protocol and declare her intent of debiting a certain amount before issuing the actual request. But even if the client just uses the `debit` message, proxies can establish the aforementioned protocol between the two legacy components to allow for a better recoverability in the case of an error, which we will discuss in Sect. 9.2.6.

9.2.4. Transactions. Transactions are well-researched concepts for database systems. From the ACID principle, we primarily use the isolation aspect which asserts that a client sees a "logical one-user system", i.e., she cannot find out if another user is concurrently using the system. Considering the assumption predicate again, this isolation guarantees that the part of the state space that the predicate is about is not concurrently modified. In a sense, protocols as discussed above already provide a transaction, but here we will use the more technical approach of transactions.

Transactions can be enforced by either pessimistic or optimistic approaches, which both have advantages and problems [**SLS92**]. The pessimistic approach, which avoids any error before it can come to pass, is done by locking (usually employing a two-phase-locking (2PL) protocol [**EGLT76**]), whereas optimistic synchronization is just watching for problems and mending them, should they occur. For database systems, this is usually done by a *rollback*, which requires the ability to undo changes made to the database in the course of an uncompleted transaction. As we have no provisions to do a rollback in our component model, we

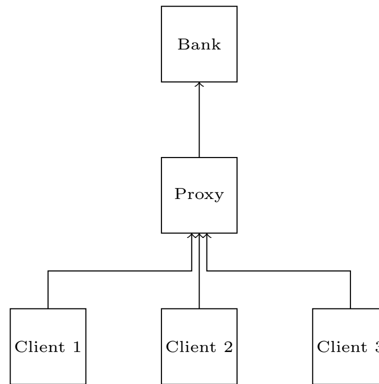


Figure 9.4: Bank example with a proxy

will just consider locking here (and consider variants of the optimistic approach in Sect. 9.2.6).

For ensuring transaction isolation, it has to be known when a transaction starts and when it ends. This can either be made explicit by the client (cf. Sect. 9.6.1 for a way to derive transactions from communication specifications), or by knowledge about the clients behavior, i.e., that any `balance` request is followed by a `debit` message. Since the `balance` message is a synchronous call, it is easy to delay answers to further clients until the current transaction is finished using postponing (see Sect. 7.5.1). Of course, such a blocking is very prone to deadlock errors, and needs extra caution.

Given knowledge about the client's behavior, this approach can be retrofitted onto legacy systems by installing a transaction manager proxy, as illustrated in Fig. 9.4. Postponing is required, since synchronous requests need to be delayed. It is also useful to have information about the calling component to further check consistency (i.e., if a `debit` is received it has been sent by the client for whom the current transaction was begun).

9.2.5. Postponing. Meyer's approach of reinterpreting the precondition as a wait-for statement [**Mey97**] can also be used to deal with the problem of concurrent precondition access. The idea here is to avoid problems with concurrent modification of the assumption predicate by waiting until the predicate is valid again. Like for transactions, the basic approach is to postpone `debit` messages until the precondition holds again, which can also be handled by a proxy and hence be retrofitted to any existing system.

9.2.6. Fault Tolerance. So far, all approaches aimed at avoiding any falsification of the assumption predicate. Any such approach restricts the system in a way and introduces other problems, like the problem of a complicated communication extension for postponing or protocols, or the problem of misallocation for quotas. Fault tolerance, however, tries a more optimistic approach by letting the system run unchanged, and coping with errors if they occur [**Her90**]. Depending on the system at hand, errors might be extremely unlikely (e.g., if the client in the example only withdraws minor amounts, and always leaves a certain margin to the balance reported). As put in [**Her90**], it might then be "easier to apologize than to ask permission".

Again, we can distinguish two kinds of approaches here: The components can be made fault-tolerant, or, if that is not possible or feasible, the system can be made fault-tolerant by reconfiguration.

For making components fault-tolerant, there are two ways in the bank example: Either we relax the precondition of the `debit` method, so that overcharging is allowed; after all, there can be no problem with preconditions if they read **true**. This is, however, not a solution we are interested in here, as we are trying to find ways to avoid concurrent reconfiguration problems, and certainly not every precondition can be made true. Or, we can make the client component aware of the chance that the precondition fails and make it cope with the error. Again, the worthiness of such an approach is questionable, as it imposes a meaning on preconditions that considers failing as "not too problematic". Instead, we want preconditions to hold for every method call, and if an error is acceptable and should be mediated between components, this should be done in the user space – which amounts to relaxing the precondition of the `debit` method again, and adding an error processing to its postcondition. But here we do not look for application-specific solutions, but for more general approaches that are enabled by the component framework.

9.2.6.1. *Fault Tolerance by Reconfiguration with Other Strategies.* Instead, we try to make the system fault-tolerant by a dedicated architecture. By monitoring the communication between the bank and its clients, we can detect problematic withdrawals, even if they do not exceed the balance yet (i.e., the *communication pattern* can be detected). Anyway, if we detect that a precondition is about to fail – or, depending on the error model, that it has just failed – reconfiguration can be employed to redesign the system with one of the aforementioned alternatives.

This is, of course, not always easy or even possible. Great care has to be taken that the state is preserved; especially the validity of the assumption predicate the clients have. The main problem is that the balance has just been exceeded, and the last call (as well as subsequent debiting requests) can only be successful if some credit is obtained. Let us assume that we can briefly obtain such a credit to serve problematic requests. Another problem emerges from the set of clients that issued a `balance` request, but have not yet issued a `debit` message. We call this set the *in-transaction set*, and it has the form $\{(c, n) \in C \times \mathbb{N} \mid \text{last response to } \texttt{balance} \text{ called by } c \text{ was } n\}$.

We briefly review the various methods and discuss the feasibility of reconfiguring the system such that they become effective in order to cope with the error and prevent further errors:

- Quotas – The in-transaction set needs to be known. Then, a temporary credit can be used to fill the quotas to sufficient levels.
- Protocols – not possible if the clients must not be changed. If it is feasible to reconfigure the clients also, this approach is still hard as the clients of the in-transaction set need to pick up the protocol in its middle; this makes their initial configuration as well as the configuration of the bank's proxy difficult, yet not impossible: A large-enough temporary credit can be used to assign over-approximated allocations to the clients.
- Transactions – transactions can only be introduced over time, as the clients of the in-transaction set need to finish their "unguarded" transaction first, which will most likely lead to new precondition violations unless a credit large enough is provided, which again can be calculated from the in-transaction set.
- Postponing – this is the only approach that can be used without considering the in-transaction set, and without a temporary credit. A proxy is introduced that delays `debit` messages until the balance suffices, including

the first problematic message. Of course, this introduces all the problems already discussed for this approach. It might, however, be combined with transactions or quotas to subsequently introduce this technique, while waiting for sufficient money to arrive.

9.2.6.2. *Fault Tolerance Preserving Optimism.* The aforementioned reconfiguration approaches switch from an optimistic to a pessimistic approach. Such an approach, which is similar to the example provided in Sect. 8.4.2, is useful to instrument parts of a complex system with necessary precautions. This is useful if the error is generally unlikely to occur (and thus not justifying initial configuration with, say, transactions), but can be expected to happen again if encountered once. If, on the other hand, the single occurrence of an error does not warrant the assumption that further errors will emerge, the fallback to pessimism is not justified. For example, the clients might be well aware of the problem of concurrent access, and leave ample margin to the balance reported, such that only rarely encountered big debits can really cause problems. For such situations, and these might be expected to be the more common ones, an *intermediate* reconfiguration will be better: Like the fallback strategies, a pessimistic solution is introduced that sees to it that the precondition is not violated. But as soon as possible, this solution is removed again, in order to continue as before.

Since we do not want to wait for the precondition to become true, and since an error is imminent, we need to take a very problem-specific action; in this case get a credit. This sort of breaks the idea of a providing a pattern, but generally speaking, the external fixing of a precondition that is already violated – as opposed to protecting it from becoming falsified by concurrent access – requires semantic insight into the precondition's purpose.

For the bank example, we introduce a component that grants a loan to the problematic client[1]. Technically, it acts as a proxy for the bank. It serves the problematic request by issuing the money, and then proceeds to debit the bank for the granted amount.

9.3. Verification of Specifications

In this section we will briefly outline the way how specifications can be checked. We introduce a number of automata that can be used to capture the specification as well as the behavior of a component system. We then discuss how the various infinite sets can be made finite by abstraction, in order to allow for model checking. Regarding the separation of roles as discussed in Sect. 9.1.2.2, this aims at assisting the system designer in assembling components in a way such that their communication expectations are satisfied.

9.3.1. State Automata. The precondition restricts the set of messages a component can process in a given state, and the postcondition advances this state. Consider the following specification of a component C:

spec C:$login(u)$:
  pre: $u \notin users$
  post: $users = users$@pre $\cup \{u\}$
  eve: $\{(\mathbf{true}, logout(u))\}$

spec C:$logout(u)$:
  pre: $u \in users$
  post: $users = users$@pre $\setminus \{u\}$
  eve: τ

spec C:$secret(u)$:
  pre: $u \in users$
  post:$users = users$@pre
  eve: τ

[1]This example was conceived in spring 2008, when this was still a pretty normal thing to do.

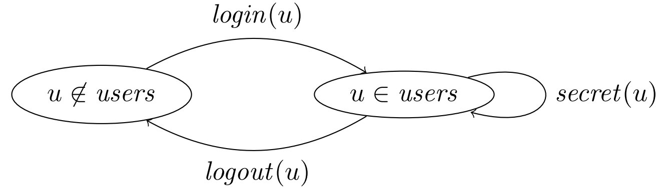


Figure 9.5: State automaton example

A *login*(u) message can only be accepted if its parameter u (to be interpreted as a user's ID) has not been parameter to a preceding call, or if a *logout*(u) message was received after the most recent *login*(u) method reception. Invocation of *secret*(u) can only be accepted if a preceding call of *login*(u) was received that has not been succeeded by a *logout*(u) call yet. This behavior can be expressed in an automaton, as shown in Fig. 9.5 (note that the automaton is about a fixed user u only; the automaton for all users would yield $2^{|U|}$ states for a finite set U of users).

This automaton describes a *protocol* that constrains the order in which methods can be invoked. If we add the outgoing communication as specified by *Comm* to this automaton, we obtain an automaton that is quite similar to the *interface automata* [**dAH01**] or *component-interaction automata* [**BvVZ05**]. The difference, which also applies to calculi like the π-calculus [**Mil93, Mil99**] and protocol specifications like [**AP03**], is the asynchronous nature of calls in our component model, which requires a more elaborate automaton to allow for composition of specifications. It is always possible to express asynchronous communication with synchronous means by implementing a queue in the data space, but it voids the finiteness of the automaton in general, and adds a complexity that should remain hidden. The most notable difference is due to the fact that components can never reject a message in the JComp component model; they are *input-enabled* and thus quite similar to I/O-automata [**LT89**]. An even more exact characterization that makes the message queues part of the automata is given by queue automata [**BZ83**], which we will later use to capture the behavior of a component setup in Sect. 9.3.2.

9.3.1.1. *Büchi Automata for Protocol Violations.* We now build an automaton that represents the assumptions a component makes. A Büchi automaton is a nondeterministic automaton that accepts infinite words by requiring that its acceptance states are visited infinitely often. Here, we will give the construction of a Büchi automaton for the *negated* specification – i.e., an automaton that accepts precisely the runs that *violate* the communication specification. We will later use this to find out whether a given component setup violates the specification of one of its components by producing a run that is accepted by the negated specification automaton – and thus a counter-example to the claim that no such run exists. Since complementing Büchi automata, while possible, produces an exponential blowup [**Var05**], producing a negated automaton directly is more feasible.

Definition 9.1 (Büchi automaton). *A Büchi automaton is a tuple* $\mathcal{A} = (\Sigma, P, p_0, \delta, F)$ *with* Σ *a (finite) alphabet,* P *a (finite) set of locations,* $p_0 \in P$ *an initial location,* $F \subseteq P$ *a set of acceptance locations, and* $\delta \subseteq P \times \Sigma \times P$ *a transition relation. We write* $p \xrightarrow{l} p' \in \delta$ *for* $(p, l, p') \in \delta$*.*

An infinite word $w_0 w_1 \ldots \in \Sigma^\omega$ *is accepted by* $\mathcal{A}$ *if there is a sequence* $p_0 p_1 \ldots \in P^\omega$ *such that*

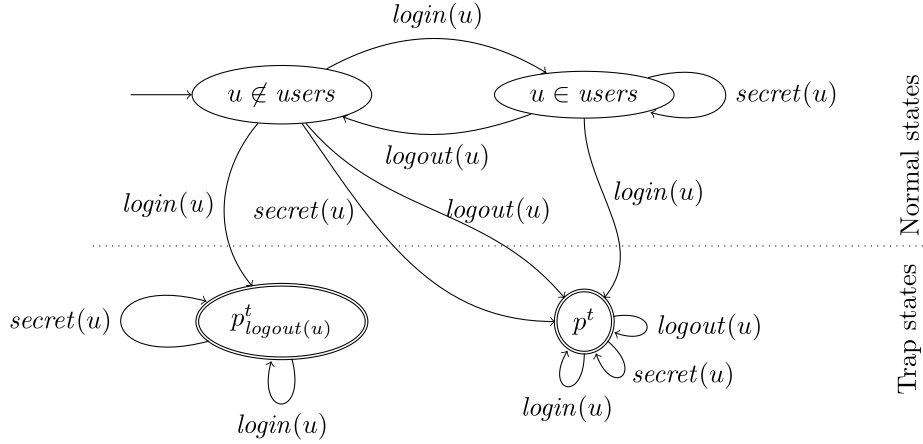


Figure 9.6: Specification Büchi automaton example

- $(p_i \xrightarrow{w_i} p_{i+1}) \in \delta$ *for all* $i \in \mathbb{N}$,
- *for every* $i \in \mathbb{N}$ *there is a* $j \geq i$ *such that* $p_j \in F$.

The traditional definition of Büchi automata requires finite alphabets and finite states, but here we will allow infinite sets also, and later revert to finite sets again.

The basic idea of the translation is to consider how a specification can be violated, and two causes can be distinguished:

(1) A message is received in a state that violates the method specification's precondition, or
(2) a message that is required by the *Eve*-part of a specification is never received.

If the former condition is ever satisfied, we can immediately deduce that the run we are currently investigating does not satisfy the specification; whereas for the second condition, we need to ensure that the run will *never* contain a message reception that would satisfy the specification; this is why we require a Büchi acceptance condition. The idea here is to nondeterministically go to a state that claims that the required message will indeed never be received – in Fig. 9.6 (note that this automaton is a simplified version, lacking many states and transitions), the left trap state is entered in an attempt to prove that *logout* is never received. The right trap state is reached upon a precondition violation, e.g., an attempt to send *secret* before *login* has completed successfully. Note that for checking such Büchi automata against a system, a *fairness constraint* [**KPRS06**] needs to be imposed: If two components keep sending each other a message over and over again, a third component might never advance to sending a desired message that would satisfy an *Eve* condition. Since a component that can dequeue a message (which is what we are interested in) can not be deprived of the chance to do so, weak fairness is sufficient to verify *Eve* specifications. Otherwise, the specification automaton is similar to the state automaton described informally above.

We hence build, for a component c out of a set of components C, a Büchi automaton $(\Sigma, P, p_0, \delta, F)$ with $\Sigma = C \times \mathscr{M} \times \mathcal{V}$, $P = P^s \cup P^t$ with $P^s = \{p_\sigma \mid \sigma \in \mathcal{S}\}$ and $P^t = \{p^t_{m(v)} \mid m \in \mathscr{M}, v \in \mathcal{V}\} \cup \{p^t\}$, $p_0 = \iota(c)$ and $F = P^t$.

As for δ, let us assume that we have a specification

spec $c : m()$:
  pre: pre_m

post: $post_m$
comm: $comm_m$
eve: eve_m

for each $m \in \bigcup_{I \in I_P(c)} I$. Then δ is the smallest set such that the following conditions hold:

- $(p \xrightarrow{(c,m,v)} p') \in \delta$ for $p, p' \in P^s$ if the method $m(v)$ is receivable in state p, i.e., $(p, v) \in pre_m$, and p' is an effect due to the postcondition, i.e., $((p, v), (p', v')) \in post_m$ for at least one $v' \in \mathcal{V}$,
- $(p \xrightarrow{(c,m,v)} p^t) \in \delta$ for $p \in P^s$, if $m(v)$ is not receivable in state p, i.e., $(p, v) \notin pre_m$,
- $(p \xrightarrow{(c,m,v)} p^t_{m'(v')}) \in \delta$ for $p \in P^s$, if $m'(v)$ is required by eve_m, i.e., $(p, (m, v')) \in eve_m$,
- $(p^t \xrightarrow{M} p^t) \in \delta$ for all $M \in \Sigma$,
- $(p^t_m \xrightarrow{(c,m',v)} p^t_m) \in \delta$ for all $c \in C, m' \in \mathscr{M} \setminus \{m\}, v \in \mathcal{V}$,
- $(p \xrightarrow{(c',m,v)} p) \in \delta$ for all $p \in P$ and $c' \in C \setminus \{c\}$.

By taking the union of such Büchi automata, a specification automaton for an entire component (and, subsequently, for an entire component setup) can be built:

Definition 9.2 (Union of Büchi automata). *The union of two Büchi automata* $(\Sigma, P^1, p^1_0, \delta^1, F^1)$ *and* $(\Sigma, P^2, p^2_0, \delta^2, F^2)$ *is the automaton* $(\Sigma, P^1 \times P^2, (p^1_0, p^2_0), \delta, F^1 \times P^2 \cup P^1 \times F^2)$ *with* δ *such that* $((p^1, p^2), s, (p'^1.p'^2)) \in \delta$ *if* $(p^1, s, p'^1) \in \delta^1$ *and* $(p^2, s, p'^2) \in \delta^2$.

The last statement about δ (i.e., informally $p \xrightarrow{c'.m(v)} p$ for any c' not identical to the c under consideration) guarantees that any union of Büchi automata built for the specification of a set of components will change only one of the elements of the tuples that form the state space P. This is also why the acting component (i.e., the receiving component) is made part of the alphabet Σ.

9.3.2. Queue Automata. Queue automata [**BZ83**, **Rus08**] are similar to nondeterministic finite automata, but also provide a number of queues. Each transaction either receives from or sends to a queue.

Definition 9.3 (Queue automaton). *A (finite)* queue automaton *is a tuple* $\mathcal{A} = (Q, \Sigma, P, l_0, \delta, F)$ *where*

- *Q is a finite set of queues,*
- *Σ is a finite set of messages,*
- *P is a finite set of locations (also called* control states*),*
- *$p_0 \in P$ is the initial location,*
- *$\delta \subseteq P \times Q \times \Sigma \times \{?, !\} \times P$ is the transition relation,*
- *and $F \subseteq P$ is the set of final locations.*

A state *of $\mathcal{A}$ is given by a pair $s = (p, q)$ with $p \in P$ and $q : Q \to \Sigma^*$. Let $S_\mathcal{A}$ denote the set of all states of $\mathcal{A}$. A state $(p, q) \in S_\mathcal{A}$ is called* initial *if $p = p_0$ and $q(q_i) = \varepsilon$ for every $q_i \in Q$. A state is called* accepting, *if $p \in F$. We write $p \xrightarrow{q?s} p'$ for $(p, q, s, ?, p') \in \delta$ and $p \xrightarrow{q!s} p'$ for $(p, q, s, !, p') \in \delta$.*

An infinite sequence $(p_0, q_0)(p_1, q_1) \ldots \in S^\omega_\mathcal{A}$ is accepted *by $\mathcal{A}$ if*

- *for all $i \in \mathbb{N}$ there exists $q \in Q, m \in \Sigma, f \in \{?, !\}$ such that*
  - $(p_i, q, m, f, p_{i+1}) \in \delta$,
  - $f =? \to q_i(q) = m :: q_{i+1}(q)$,
  - $f =! \to q_{i+1}(q) = q_i(q) :: m$,

– $\forall q' \in Q \,.\, q' \neq q \rightarrow q_{i+1}(q) = q_i(q)$.

- (p_0, q_0) *is initial.*
- *For all* $i \in \mathbb{N}$ *there exists a* $j > i$ *such that* (p_j, q_j) *is accepting.*

The language $\mathcal{L}(\mathcal{A})$ *is the set of all accepted sequences of* $S^{\omega}_{\mathcal{A}}$.

Queue automata hence do not accept words, but runs that represent the progression of states and queues of a system. It is usually not important how such a run can be assembled; we are only interested in whether any accepting run exists.

Queue automata are Turing-complete, as the queue can be used to simulate a tape. The basic proof idea is that both a left shift and a right shift can be simulated on the queue: One direction via reading and re-enqueueing an element, the other direction via adding a special element to the queue, then polling and re-enqueueing elements until the special element is read again; the last element read can be stored in the control state [**Rus08**]. Hence, the class of queue automata with as little as one queue and a three-letter Σ is already Turing-complete.

There are two differences between queue automata and our approach towards specifying systems: For the specification as given above, no finiteness of states is required, and communication is not directed to queues, but roles. But if we ignore the infiniteness of $\mathcal{S}$ for now, and redirect the communication to the target components, a direct translation is possible. We first build a queue automaton for a single component c. It is convenient to extend δ to be a subset of $P \times ((Q \times \Sigma \times \{?, !\}) \cup \{\tau\}) \times P$ and thus allow the automaton to do silent moves that reflect advancement of components without communication. The τ-labelled transitions can later be removed again.

The set P of control states is given by the set $\mathcal{S} \times \mathscr{P} \times (C \cup \{\bot\}) \times \{r, b\}$, the alphabet Σ by $C \times \mathscr{M} \times \mathcal{V}$ and the queues Q by $\{q^r_{c'} \mid c' \in C\} \cup \{q^s_{c'} \mid c' \in C\}$. For the initial state, we have $p_0 = (\iota(c), \text{success}, \bot, r)$. δ can be defined by the runs restricted to the actions performed by c, so let $L_c = \textsc{call}(c_1 : c) \cup \ldots \cup \textsc{loop}(c : c)$.

Let $\mathscr{T}$ be the LTS of the component setup containing c. δ is defined as the smallest set such that, for each run $r \in \mathit{Runs}(\mathscr{T})$ and all $i \in \mathbb{N}$, we have, if $s_0 \xrightarrow{l_0} \ldots = r|_{L_c}$:

- $(\sigma, P, c', r) \xrightarrow{q^s_c?(c'', m, v)} (\sigma', P', c'', r)$ if $l_i \in \textsc{deq}(c : c, \overline{c' : c''}, \overline{m : m}, \overline{v : v})$ and $\sigma(s_i(c)) = \sigma$ and $\sigma(s_{i+1}(c)) = \sigma'$ and $P(s_i(c)) = P$ and $P(s_{i+1}(c)) = P'$,
- $(\sigma, P, c', r) \xrightarrow{q^s_{c''}!(c, m, v)} (\sigma', P', c', b)$ if $l_i \in \textsc{call}(c_1 : c, \overline{c_2 : c''}, \overline{m : m}, \overline{v : v})$ and $\sigma(s_i(c)) = \sigma$ and $\sigma(s_{i+1}(c)) = \sigma'$ and $P(s_i(c)) = P$ and $P(s_{i+1}(c)) = P'$,
- $(\sigma, P, c', r) \xrightarrow{q^s_{c''}!(c, m, v)} (\sigma', P', c', r)$ if $l_i \in \textsc{send}(c_1 : c, \overline{c_2 : c''}, \overline{m : m}, \overline{v : v})$ and $\sigma(s_i(c)) = \sigma$ and $\sigma(s_{i+1}(c)) = \sigma'$ and $P(s_i(c)) = P$ and $P(s_{i+1}(c)) = P'$,
- $(\sigma, P, c', r) \xrightarrow{q^r_{c'}!(c, m, v)} (\sigma', P', c', r)$ if $l_i \in \textsc{ret}(c_1 : c, \overline{c_2 : c'}, \overline{m : m}, \overline{v : v})$ and $\sigma(s_i(c)) = \sigma$ and $\sigma(s_{i+1}(c)) = \sigma'$ and $P(s_i(c)) = P$ and $P(s_{i+1}(c)) = P'$,
- $(\sigma, P, c', r) \xrightarrow{\tau} (\sigma', P', c', r)$ if $l_i \notin \textsc{deq}(c : c) \cup \textsc{call}(c : c) \cup \textsc{send}(c : c)\textsc{ret}(c : c)$ and $\sigma(s_i(c)) = \sigma$ and $\sigma(s_{i+1}(c)) = \sigma'$ and $P(s_i(c)) = P$ and $P(s_{i+1}(c)) = P'$.

Additionally, we have $(\sigma, P, c', b) \xrightarrow{q^r_c?(c'', m, v)} (\sigma, P, c', r)$ for all $P \in \mathscr{P}, c' \in C \setminus \{c\}, \sigma \in \Sigma$. We then set $F = P$, since any execution that goes on forever is acceptable for us here. Note that such an automaton will most likely not accept anything, as most of the queues are not used elsewhere, and, especially, no communication is transmitted.

For assembling systems of multiple components, queue automata can be parallelized; they do not synchronize, but identify the name-identical queues. So, we can define the union of two queue automata as

DEFINITION 9.4. *Let* $(Q, \Sigma, P^1, p^1_0, \delta^1, F^1)$ *and* $(Q, \Sigma, P^2, p^2_0, \delta^2, F^2)$ *be queue automata. Their union is defined as the queue automaton* $(Q, \Sigma, P^1 \times P^2, (p^1_0, p^2_0), \delta, (F^1 \times P^2) \cup (P^1 \times F^2))$ *with* $((p^1, p^2), q, m, f, (p'^1, p'^2)) \in \delta$ *if* $p^2 = p'^2$ *and* $(p^1, q, m, f, p'^1) \in \delta^1$*, or* $p^1 = p'^1$ *and* $(p^2, q, m, f, p'^2) \in \delta^2$*.*

In this way, a queue automaton can be built for a component setup.

Note that even as we define the queue automata by means of runs, which in turn require a ready-made LTS, it is straightforward to define them on basis of the method evaluators; it is merely for the additional technical complexity that we prefer the run-derived approach.

9.3.3. Abstracting the State. The original definition of queue automata only had a finite set of control states, and a finite set of messages. Since $\mathcal{S}$ and $\mathcal{V}$ are infinite, we get into a problem here, which can be solved by abstraction. Technically, an abstraction is a function, in this case $\mathcal{S} \to S$ and $\mathcal{V} \to V$, and, in this context, we require S and V to be finite. For example, we might try and use a set $S = \{\mathit{empty}, \mathit{nonempty}\}$ and $V = \{\mathit{user}\}$ in the login example given above in Sect. 9.3.1, with functions

$$a_s(s) = \begin{cases} \mathit{empty}, & \text{if } C.\mathit{users} = \varnothing \\ \mathit{nonempty}, & \text{otherwise} \end{cases}$$

and $a_v(v) = \mathit{user}$. Such an abstraction is mostly worthless, however: When evaluating the specification, we need to be *conservative*, meaning that even if the states of C is *nonempty*, we always have to assume that $u \notin \mathit{users}$ at the precondition of $C.\mathit{secret}(u)$.

An abstraction more suitable is what we call a "one singled out"-abstraction where we abstract the parameter values by $V = \{\mathit{theuser}, \mathit{otheruser}\}$ meaning that either the one user we care about is mentioned, or another user we do not further consider. The states are abstracted by $S = \{\mathit{theoneLoggedIn}, \mathit{theoneNotLoggedIn}\}$, with the obvious abstraction functions. This abstraction is sufficient to exclude a protocol failure: If *theoneLoggedIn* is the state of the component C upon dequeueing of method *secret*, we can rule out that an illegal access is made. A similar abstraction can be given for hash tables, where a single element can be explicitly investigated, and the others can be identified. Of course, such an abstraction might still be insufficient if elements are to be compared according to some means, but in our experience, the most prevalent utilization of specification is sufficiently abstracted by this approach.

9.3.4. Model Checking Component Setups. If the state is abstracted such that the domains of the component state abstractions and method parameter abstractions are finite, model checking can be used to check the conformance of component specifications within a component setup, but for one problem: The queues, which are unbounded (and hence make the LTS of a component setup infinite, even if the queue automata states are finite). In essence, we wish to show that any run of the component setup is admissible by the specification.

Let $\mathscr{T}_c$ be the LTS of the component setup, and $\mathscr{T}_s$ be the LTS of the specification; we then wish to show $\mathit{Runs}(\mathscr{T}_c) \subseteq \mathit{Runs}(\mathscr{T}_s)$. Constraining ourselves to asynchronous messages for brevity, we can show this by the usual automata-based

model checking approach:

$$\begin{aligned} & Runs(\mathscr{T}_c) \subseteq Runs(\mathscr{T}_s) \\ \text{iff} \quad & \mathcal{L}(\mathcal{A}_c) \subseteq \mathcal{L}(\mathcal{A}_s) \\ \text{iff} \quad & \mathcal{L}(\mathcal{A}_c) \cap \overline{\mathcal{L}(\mathcal{A}_s)} = \varnothing \\ \text{iff} \quad & \mathcal{L}(\mathcal{A}_c) \cap \mathcal{L}(\mathcal{A}_{\neg s}) = \varnothing \\ \text{iff} \quad & \mathcal{L}(\mathcal{A}_c \times \mathcal{A}_{\neg s}) = \varnothing. \end{aligned}$$

Solving the question of the last line is known as the *emptiness problem*, and it is well known how to do this for Büchi automata [**CGP00**]. Given an interpretation for the conjunction $\times$ of a queue automaton $\mathcal{A}_c$ and a Büchi automaton $\mathcal{A}_{\neg s}$ as defined in Sect. 9.3.1.1, we can use this approach to model check a system for conformance to a specification. This definition is fairly straightforward:

DEFINITION 9.5 (Conjunction of a queue and a Büchi automaton). *Let* $\mathcal{A}_c = (Q, \Sigma, L, l_0, \delta, F)$ *be a queue automaton and* $\mathcal{A}_s = (\Sigma, P', p'_0, \delta', F')$ *be a Büchi automaton. The conjunction* $\mathcal{A}_c \times \mathcal{A}_s$ *for a mapping* $msg : Q \times \Sigma \to \Sigma'$ *is a queue automaton* $(Q, \Sigma, L'', l''_0, \delta'', F'')$ *with*

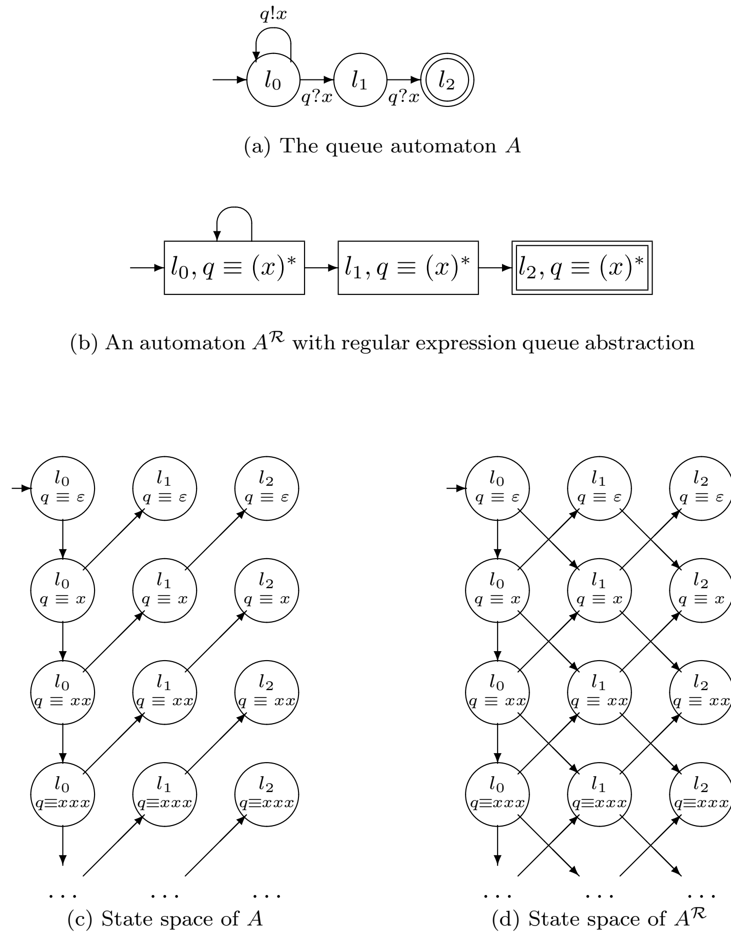
- $L'' = L \times P'$,
- $l''_0 = (l_0, p'_0)$,
- δ'' *defined as the smallest set such that*
  - *if* $l \xrightarrow{\tau} l' \in \delta$ *then* $(l, p) \xrightarrow{\tau} (l', p) \in \delta''$ *for all* $p \in P'$,
  - *if* $l \xrightarrow{q!m} l' \in \delta$ *then* $(l, p) \xrightarrow{\tau} (l', p) \in \delta''$ *for all* $p \in P'$,
  - *if* $l \xrightarrow{q?m}, l' \in \delta$ *and* $p \xrightarrow{s} p' \in \delta'$ *and* $msg(q, m) = s$, *then* $(l, p) \xrightarrow{q?m} (l', p') \in \delta''$.
- $F'' = F \times F'$.

For component specification, we define the mapping *msg* as $msg(q^s_{c'}, (c'', m, v)) = (c', m, v)$ (i.e., we map the message reception event of message $m(v)$ on queue $q^s_{c'}$ to the Büchi automaton label (c', m, v)).

The queues are the sole difference between Büchi automata and queue automata, but as queue automata are Turing-complete, most properties about them are undecidable, including their emptiness. Luckily, components will usually not try to simulate a Turing machine tape with their communication, and resort to rather limited patterns of communication. If these patterns can be described by regular languages, a technique called *regular model checking* [**BG99, BJNT00**] can be employed. As mentioned in Chapter 5, model checkers require a finite graph to operate on, and regular model checking makes queue automata finite by describing an infinite set of queue contents by a regular expression, hoping that only a finite number of different regular expressions are required to do so. For example, if a component c can send an unlimited number of messages m to another component c' (and there are no other connections to c'), we can abstract the queue of c' as m^*, meaning that it can contain any number of m messages, including none at all. These scenarios are not too unlikely, with an example given in Sect. 8.4.2.

Fig. 9.7 shows how a queue automaton can be abstracted. For the actual techniques to obtain such an abstraction, the reader is referred to [**BG99, BJNT00, Nil05**]. Obviously, not every queue automaton can be abstracted like this, with a counter-example given in Fig. 9.8.

While techniques exist to approach such more complicated automata [**FP01, BH99**], we can expect components to be more benign. Either the components are involved in tightly-knit transactions, or they send an unbounded number of messages without waiting for response, which is perfect for regular model checking. The author has the hunch, however, that these cases can be proven to be checkable



(a) The queue automaton A

(b) An automaton $A^{\mathcal{R}}$ with regular expression queue abstraction

(c) State space of A

(d) State space of $A^{\mathcal{R}}$

Figure 9.7: A queue automaton and its abstraction

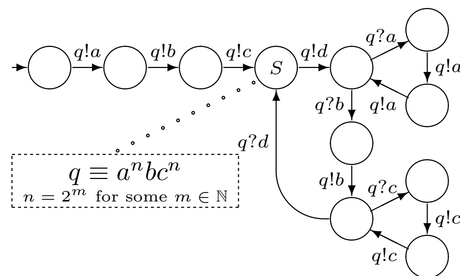


Figure 9.8: A queue automaton that cannot be finitely abstracted by regular model checking

by normal model checking with bounded queues all the same, though a proof cannot yet be given[2].

[2]The basic idea is similar to the Myhill-Nerode theorem for regular languages: As only a finite number of control states are available for a queue automaton, only finitely many queue content

Note that it is, in general, insufficient to check just directly connected components, as their communication behavior can depend on other components. For example, the semantical behavior of a filter F for the interface I is specified as follows: For each (asynchronous) method $m \in I$, we use a specification

spec F::$m()$:
  pre: **true**
  post: **true**
  comm: $\{next.M()\}$
  eve: τ

which guarantees that the reception of method M results in a sending of M over the role *next*, but does not constrain the reception of messages in any way. If the component that is targeted by role *next* uses a more restricted protocol, a mere consideration of the filter and its target component will appear as an inconsistent system, neglecting that the component that sends messages to the filter might respect the protocol of the filter's target. We believe that a subsequent addition of further components to rule out errors originating from considering too few components (in a style similar to CEGAR model checking [**CCG+03**, **HJMS03**]) could offer modular model checking in such a situation, but this exceeds the scope of this thesis.

Supporting model checking of component setups provides a tool to the system designer that can be used for detecting inconsistencies in the communication behavior and the expectations of the individual components. We do not provide a similar approach for the component designer: Such a tool would be useful for comparing the component specification to the actual component implementation, guaranteeing that the specified communication behavior is refined by the actual component implementation. Doing such checks might utilize well-developed tools like JAVA PATHFINDER [**VHB+03**] as demonstrated in [**PPK06**]. The necessity to provide a finite-state abstraction of a component's states, however, requires a mapping of actual component data states to the abstraction. This is an interesting topic, not only because of model checking techniques, but also for automated test generation.

9.4. Evaluating Correctness of Reconfiguration

The failed transmitter has a backup aboard, but settings on five other instruments in the telescope must be changed to use the device. The instruments have not been activated in their backup mode since the early 1990s or late 1980s, the space agency said Monday.

— Houston Chronicle, "NASA pushes Hubble's makeover back to February", Sept. 29, 2008

When describing the basic idea of reconfiguration in Chapter 2.4.5, we stressed the importance of the separation of roles. The author of a component should not be required to take every possible reconfiguration scenario into account, and ideally not be concerned with reconfiguration at all: If the component is to be employed in a scenario with reconfiguration, its regular interfaces should be sufficient to allow for complete reconfiguration.

---

equivalence classes can be distinguished. As an example, if the queue content is abstracted by m^* and we read m two times, it would be sufficient to bound the queue with a length of 2 to assert that this reading is possible. The true problem is how to find out the bound, without doing the regular model checking process before, and then analyzing the read operations of the state space obtained.

The downside to such a separation is the loss of information. Implementation details are hidden behind the interfaces; and of these interfaces we usually just know the syntactical structure. If the internal behavior of components is not known, a reconfiguration designer will have a hard time to figure out whether a reconfiguration plan will work as intended. Component specifications provide remedy to this problem. The system designer again takes the responsibility to build the reconfiguration in such a way as to preserve the system's function. But the challenge is a little more subtle in the context of reconfiguration: not only does the final configuration have to satisfy the assumptions of the individual components, but the reconfiguration also needs to be conducted in a way such that, at no point in time, components experience a failure.

Both aspects are discussed in the literature. The aspect of reconfiguring the setup in a way such that the assumptions of components are preserved is known as the concept of "mutual consistent states"[3]. The consideration of maintaining consistency in the presence of the reconfiguration process has seen much research, although often only some aspect is covered, e.g., the absence of concurrent communication with the reconfiguration process in [**AP05**], or the often-cited criteria of quiescence [**KM90**] and tranquility [**VEBD07**] which avoid the disruption of ongoing transactions by reconfiguration. In this chapter, we will relate our approach of defining components and conducting reconfiguration with this research.

9.4.1. How Can a Reconfiguration Fail? Basically, reconfiguration fails if the new system violates the assumptions of one of its components, thus violating the mutual consistent state requirement. In this section, we will use the example of a component c_r that expects to receive the messages a, b and c, over and over again, in exactly this order. Let us assume that the sender c_s of those messages is replaced by c'_s. Now, the assumptions of c_r can be violated in three ways:

- First, c'_s might be a badly chosen component. It might send cba and thus violate the assumptions of c_r. Obviously, the reconfiguration designer failed to adhere to c_r's requirements and devised an invalid reconfiguration plan. In this case, it would be nice to detect this problem before reconfiguration and prevent the problematic replacement from ever happening.
- Second, c'_s might be a component that just keeps sending abc as desired, but c_s might have just sent ab when the reconfiguration happens. After reconfiguration, c'_s starts to send abc, thus producing a sequence not expected by c_r. Here, the reconfiguration plan itself would allow for a consistent reconfiguration, but the point in time is chosen badly.
- Third, c'_s might send abc as required and pick up the message sequence right where c_s stopped it. Still, the reconfiguration process itself might mix up the messages and thus produce an invalid sequence. We have seen two examples of such problems in Sect. 6.7.4 and Sect. 6.7.5. If those problems are known, it is easy to use message sequence IDs or extended component locking to avoid the resulting assumption violation (as it is, in a slightly different context, suggested in [**AP05**]).

We can consider the latter problem accounted for, and focus on the two former problems.

[3] As mentioned before, this term was introduced by Kaveh Moazami-Goudarzi, but as this work is not available, we omit the reference here. The term is also mentioned in [**Weg03**].

9.5. Mutual Consistent States

When a component A is connected to a component B via a role r, then A needs to make some assumptions about B, as we have discussed in Chapter 9.1. If B is a hash table, the assumption might be that adding an object x by calling r.add(x) will put B into a state that stores the fact that x is stored. But actually, A will not reason about the actual component B, since that would violate the separation of roles: How B is chosen is not the concern of A. Instead, it will reason about the assumptions it makes to the component that is linked to its role r.

This is actually the major benefit of using a component-based architecture as a basis for reconfiguration: If we substitute B by C and take the necessary precautions, A will not be aware of the reconfiguration. It will continue communicating over r; and it will continue to make its assumptions about r. Reconfiguration is consistent if these assumptions are met before and after the reconfiguration (and, of course, at any time during the reconfiguration, an issue addressed in Sect. 6.4). This is the concept of *mutually consistent states*: Reconfiguration is consistent if the communication assumptions of those components not directly involved in the reconfiguration (i.e., those in the R element of the plan that get removed) are not violated during or after the reconfiguration.

9.5.1. Choosing a Suitable Component. Theorem 6.2 covers an artificial case of component replacement by substituting a component with an identical version. The communication traces of a system configured this way remain the same as for the unreconfigured system; and since the assumptions of components are only about received communication, as illustrated by Lemma 9.1, all components have their assumptions met.

Such an "invisible" replacement is obviously entirely useless, and the example was provided as a mere indicator for the ability of reconfiguration to remain invisible. There are domains, like hot code update or nonfunctional replacements as discussed in Sect. 8.4.4, where it might be feasible to replace a component with a functionally equivalent component that satisfies the same specification. In other domains, the modification of communication is a vital property of the reconfiguration. However, taking a closer look at the examples of Chapter 8, only few examples actually change the communication conducted with components which are truly "outside" of the extent of the reconfiguration plan; the example that comes closest is that of adapting to external change in Sect. 8.3. Here, substituting an RSS reader with a placeholder component during network problems does change the messages received by the consuming component, but neither changes the fact that they are sent (this is what is actually established by reconfiguration) nor their conformance to a relaxed specification.

This "conformance to a relaxed specification" is the basic idea of reconfiguration: By allowing a broad range of communication (e.g., for the RSS example, any meaningful report on the outcome of a RSS fetch attempt by the consumer component) reconfigured components can alter the communication without violating assumptions.

9.5.2. Refinement. Let us investigate the case where a component is replaced by another component that has the same communication expectations and behavior. Obviously, if the former system was correct with respect to these assumptions, then so is the reconfigured one. Note that, however, a system that does not fully respect the specifications might have been working well before the reconfiguration, and produce errors afterwards, as the implementation of the components might produce behavior that now illustrates the missing adherence to the specification.

Refinement is a generalization of the concept of replacing components with an identical specification. A component method specification $S' \equiv (\mathit{Pre}', \mathit{Post}', \mathit{Comm}', \mathit{Eve}')$ refines another component specification $S \equiv (\mathit{Pre}, \mathit{Post}, \mathit{Comm}, \mathit{Eve})$ if it works in the same setups as S did, and possibly more (obviously, this holds if $S' = S$). The idea is to have S' be defined in a way such that it accepts each run S accepts (possibly more) and produces only the runs that S produces (possibly less). For the four specification elements, this means:

- $\mathit{Pre}' \supseteq \mathit{Pre}$ (i.e., the admissible set of state and value combinations at the time of method dequeueing has grown, now accepting more situations),
- $\mathit{Post}' \subseteq \mathit{Post}$ (i.e., the admissible states at method execution have been reduced, allowing less nondeterminism during method execution),
- $\mathit{Comm}' \subseteq \mathit{Comm}$ (i.e., as for Post', less communication sequences are now allowed, making the method's behavior less nondeterministic),
- $\mathit{Eve}' \supseteq \mathit{Eve}$ (i.e., more variants of satisfying future method invocation are given).

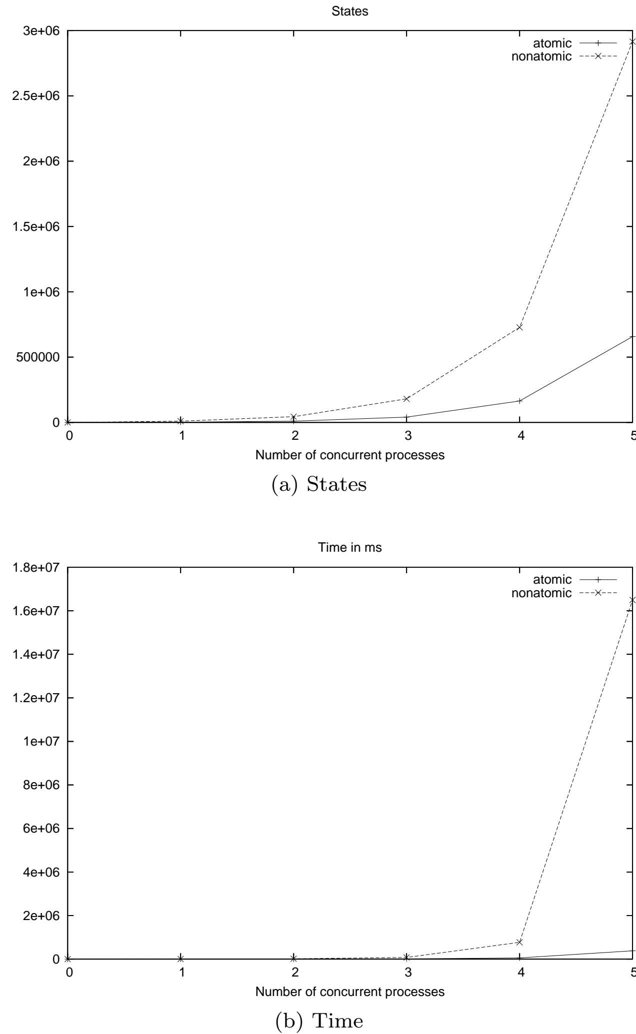
These relations reflect an advancement in the design of the component, which consists of more dedicated choices (i.e., a more dedicated communication sequence and postcondition, obtained by further investigating implementation possibilities of the component) and more flexibility (i.e., a weakened precondition and less incoming communication expectations, obtained by an improved handling of previously incompatible environment behavior).

A component whose method specifications all are refinements of the method specifications of a second component can be used as a substitute: With the environment staying the same, all runs experienced are already compatible to the preconditions and communication expectations, and since those got less strict, they remain compatible. At the same time, since the component behaves in a way that was already admissible for the previous version, the environment's assumptions are satisfied.

For the substitution of multiple components, they often can be considered as a combined, hierarchical component, with a behavior derived from their open ports.

But sometimes, a component will have to be replaced by a component that exhibits an altogether different communication behavior, especially with respect to the *Comm* element – e.g., the placeholder component of the example in Sect. 8.3.3, which does not relay the communication to the chain of readers, but directly to the target, something that is most likely not covered by the original specification of the chain head. Likewise, if a component is genuinely added, a substitutability criterion will not do. There might be further investigations of port protocols, but here, we suggest the direct checking of the new configuration.

9.5.3. Checking Generic Reconfiguration. Since Theorem 6.1 asserts us that reconfiguration using injective shallow reconfiguration plans is perceived to be atomic, we do not need to consider intermediate states between the two configurations. If we attempt model checking for the system by making its state finite, we can model the reconfiguration as an atomic step, which dramatically cuts down the state space – as shown in Fig. 9.9. For this example, we implemented the component model in the MAUDE programming language [CDE+07]. We then implemented the fault tolerance example described in Sect. 8.4.3. Building on Corollary 6.1 we introduced a global blocking flag that prohibits advancement of any component during the execution of the reconfiguration, thus obtaining atomicity. We then added a variable number of processes that do nothing besides doing some action that interleaves with the reconfiguration if the global blocking flag is not set.



(a) States

(b) Time

Figure 9.9: Atomic and interleaved reconfiguration checking with MAUDE

Prohibiting interleaving cuts down the state space, and, in our experience even more drastically, the time required. Ironically, our interest here is exactly opposite to the original interest in perceived atomicity: While we do not want to block components in the component model in order to minimize the impact of reconfiguration, for model checking we want to spend as little steps as possible on the reconfiguration, again to minimize the impact; but this time, on the extent of the state space.

We can also use this for verifying properties of the component state. Let $\varphi = (c, \Sigma) \in \mathcal{C} \times \wp(\mathcal{S})$ be a pair consisting of a component and a set of data states. For an LTS $\mathscr{T} = (S, L, T)$, we write $\mathscr{T} \models \varphi$ if, in each reachable state $s \in S$ with $c \in \mathrm{dom}(s)$, $\sigma(s(c)) \in \Sigma$.

COROLLARY 9.1. *Let $\mathscr{T}_a$ be the LTS of a component system that can get reconfigured by atomic execution of an injective shallow plan Δ_s, and let $\mathscr{T}_i$ be the*

transition system of the same component system that can get reconfigured by interleaved execution of Δ_s*. Then* $\mathscr{T}_a \models \varphi$ *for a state property* $\varphi = (c, \Sigma)$ *iff* $\mathscr{T}_i \models \varphi$*.*

PROOF. This follows almost directly from Corollary 6.1. We know that $\mathit{Traces}^{comm}(\mathscr{T}_a) = \mathit{Traces}^{comm}(\mathscr{T}_i)$. Assume $\mathscr{T}_a \not\models \varphi$. Then there is a run $r = s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} s_n \in \mathit{Runs}(\mathscr{T}_a)$ with $s_n \notin \Sigma$ and $\mathit{Traces}^{comm}(r) \in \mathit{Traces}^{comm}(\mathscr{T}_a)$. Assume $\mathscr{T}_i \models \varphi$, hence there is no run $r' = \ldots \xrightarrow{l'_{n-1}} s_n \in \mathit{Runs}(\mathscr{T}_i)$. But since $\mathit{Traces}^{comm}(\mathscr{T}_a) = \mathit{Traces}^{comm}(\mathscr{T}_i)$, there is a run r' with $\mathit{Traces}^{comm}(r') = \mathit{Traces}^{comm}(r)$; hence the last message received in r and r' by component c is the same. Since the method evaluator would allow reaching s_n in $\mathscr{T}_i$, the state of c at this message reception needs to differ. By repeating this argument inductively, we finally conclude that there need to be different initial states, which is a contradiction to the assumption that both LTS are about the same component system. The other direction ($\mathscr{T}_i \not\models \varphi \rightarrow \mathscr{T}_a \not\models \varphi$) is shown in the same way. □

Besides the improved performance of model checking with atomic instead of interleaved reconfiguration, Fig. 9.9 also illustrates the basic problem of model checking asynchronous systems: Even if we neglect the problem of unbounded queues (in our example, we took care of not allowing repeated asynchronous sending of messages, so that the queues never hold more than one message), every additional component multiplies the state space, leading to exponential growth. In the MAUDE implementation, we also ruled out another source of exponential growth: A nondeterministically chosen point in time where the reconfiguration is started. These three elements – unbounded queues, concurrent processes, and nondeterministic reconfiguration – each are a threat to the feasibility of model checking, and together are beyond tractability for anything exceeding very high-level abstractions in special environments [**GH06**]. Thus, a deduction-based approach seems more promising [**BBGH08**].

9.6. Transaction-Preserving State Transferal

The second problem discussed in Sect. 9.4.1 is the problem of a badly chosen point in time to do the reconfiguration, as it interferes with an ongoing transaction. In Sect. 8.1.4.2 we have introduced the notions of quiescence and tranquility. We have seen that tranquility is a concept that can be used to refrain from state transferal, or, at least, from a state transferal of transaction-local data. Hence, the problem of interrupting partially finished transactions may be considered to be an otherwise sound reconfiguration that is executed at an unsuitable point in time, or as a reconfiguration that omits the transferal of important "transaction progression" data that keeps track on how far transactions have progressed. After all, if c_s can become replaced after having sent ab only, then c_s maintains internal information that tells it that it has to send c next – as reconfiguration cannot interrupt a method execution in our model. This information needs to be carried to c'_s. A failure to do so again amounts to a malformed reconfiguration plan; and since quiescence and tranquility are well-researched, we might find it fashionable to take that position.

There are two different aspects to state transferal with respect to reconfiguration correctness: First, transaction-local data needs to be retained, in the given example the information that ab got sent and c is to be sent next, and second, global data that is independent of ongoing transactions, but influences further communication nevertheless. The distinction, however, is vague: a global data state might be considered to be the transaction-local state of an everlasting transaction in which all communication partners participate.

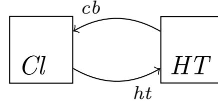


Figure 9.10: Client/Hash table example

For example, a hash table might require a transaction protocol that mandates that an element is searched before it is inserted (a requirement that might originate from the component using the query to cache the insertion point; while this rationale is unknown to the component user, the protocol can be enforced by the communication specification), and at the same time guarantees that any further query for an element is answered correctly, here by an asynchronous callback[4]:

spec $HT::query(e)$:
- **pre**: **true**
- **post**: $setelement = e \wedge elements = elements@\mathrm{pre}$
- **comm**: $(e \in elements, cb.elementFound()) \cup$ $(e \notin elements, cb.elementNotFound())$
- **eve**: τ

spec $HT::add(e)$:
- **pre**: $setelement = e$
- **post**: $elements = elements@\mathrm{pre} \cup \{e\}$
- **comm**: $(\mathbf{true}, \varepsilon)$
- **eve**: τ

While common sense tells us that the *setelement* state variable is transaction-local, while the *elements* state variable is global state, this cannot be deduced from the specification, nor is the fact that a transaction is required visible.

Let us consider the following specification of a client component as illustrated in Fig. 9.10:

spec $Cl::handle(e)$:
- **pre**: $element = \bot$
- **post**: $stored = stored@\mathrm{pre} \wedge element = e$
- **comm**: $(\mathbf{true}, ht.query(e))$
- **eve**: $(e \in stored, elementFound()) \cup (e \notin stored, elementNotFound())$

spec $Cl::elementFound()$:
- **pre**: $element \neq \bot$
- **post**: $stored = stored@\mathrm{pre} \wedge element = \bot$
- **comm**: $(\mathbf{true}, \varepsilon)$
- **eve**: τ

spec $Cl::elementNotFound()$:
- **pre**: $element \neq \bot$
- **post**: $stored = stored@\mathrm{pre} \cup \{element\} \wedge element = \bot$
- **comm**: $(\mathbf{true}, ht.add(element))$
- **eve**: τ

Transactions become visible in the *Eve*-clause of the client; and generally, *Eve*-clauses hint at pending transactions. Still, the transaction-local data is not visible

[4] This example is actually used in the CMC model checker in a closed hash table implementation that uses rehashing to find a free element position. Obviously, it is easy to relax the example such that a $setelement \neq e$ leads to a recalculation of the insertion position, but for the sake of the example we leave it like this.

(or, at least, not without a careful analysis of the use of the variables). In many programming languages, the programmer is required to explicitly distinguish between transaction-local and global data (e.g., in JAVA with the seldom used `transient` keyword [OK99]), thus explicitly declaring data to be transaction-local.

9.6.1. Formalizing Tranquility. Let us assume a set $\mathscr{T}$ of transaction identifiers. Components can initiate transactions, they can terminate transactions they have begun (meaning that the protocol that is encapsulated within a transaction has finished), and they can participate in other components' transactions. In the above example, the client component *Cl* initiates a transaction in its *handle* method and terminates it in either *elementFound* or *elementNotFound*; while the hash table component *HT* participates in the transaction without starting its own.

We hence define methods $tc_s, tc_t, tc_r : C \times \mathscr{M} \times \mathcal{V} \times \mathcal{S} \to \wp(\mathscr{T})$, that, given a component, a method that is executed by this component, the values of the parameters for the method invocation and the component's state tell us which transactions are begun (tc_s), terminated (tc_t) or required to be active (tc_r). For a finite run $r = s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{n-1}} s_n$, we can calculate the set of active transactions $trans_a(c, r) \subseteq \mathscr{T}$ of a component c by defining inductively:

- $trans_a(c, s_0) = \varnothing$,
- $trans_a(c, s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{i-1}} s_i \xrightarrow{l_i} s_{i+1}) = trans_a(c, s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{i-1}} s_i) \cup A \setminus R$ with $A = R = \varnothing$ if $l_i \notin \text{DEQ}(c : c)$ and, if $l_i \in \text{DEQ}(c : c, \overline{m : m}, \overline{v : v})$, $A = tc_s(c, m, v, \sigma(s_{i+1}(c)))$ and $R = tc_t(c, m, v, \sigma(s_{i+1}(c)))$.

The set $trans_p(c, r)$ of *participated transactions* is calculated in a similar fashion:

- $trans_p(c, s_0) = \varnothing$,
- $trans_p(c, s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{i-1}} s_i \xrightarrow{l_i} s_{i+1}) = trans_p(c, s_0 \xrightarrow{l_0} \ldots \xrightarrow{l_{i-1}} s_i) \cup A \setminus R$ with
  - $A = tc_s(c', m, v, \sigma(s_{i+1}(c'))) \cup tc_r(c', m, v, \sigma(s_{i+1}(c')))$ if $l_i \in \text{CALL}(c_1 : c, \overline{c_2 : c'}, \overline{m : m}, \overline{v : v}) \cup \text{SEND}(c_1 : c, \overline{c_2 : c'}, \overline{m : m}, \overline{v : v})$, and $R = \varnothing$,
  - $A = \varnothing$ and $R = tc_t(c', m, v, \sigma(s_{i+1}(c')))$, if $l_i \in \text{DEQ}(c : c', c, \overline{m : m}, \overline{v : v})$, and
  - $A = R = \varnothing$, otherwise.

$trans_a(c, r)$ tells us which transactions a component is involved in; these transactions are started and ended by dequeueing an appropriate message. $trans_p(c, r)$ tells us which transactions a component participates in, which are those that are active (or about to become active) in some component, and to which a message has participated (possibly starting the transaction).

We have discussed in Sect. 8.1.4.2.2 that two of the conditions of tranquility of a component c in a state s are already ensured by our approach towards reconfiguration:

(2) It [component c] will not initiate new transactions, and
(3) it is not actively processing a request.

Both are guaranteed due to reconfiguration only being possible in states where a component is not actively executing a method, i.e., $P(\tilde{c}) \in \{\text{success}, \text{fail}\}$, and reconfiguration will take care of no further message processing on behalf of c. Hence, tranquility needs to be checked for the remaining two properties:

(1) It is not currently engaged in a transaction that it initiated, and
(4) none of its adjacent nodes are engaged in a transaction in which this node has already participated and might still participate in the future.

We can now formalize the tranquility requirements: A component c is in a tranquil state, if for the current run $r = s_0 \xrightarrow{l_0} \dots \xrightarrow{l_{n-1}} s_n$ of the system holds:

(1) $trans_a(c, r) = \varnothing$,
(2,3) RECONF is applicable,
(4) $trans_p(c, r) = \varnothing$.

Note that the property (4) is not an exact translation: We omit the possibility of removing those transactions that are active and have been contributed to, but will not be contributed to any further; our framework does not provide a means to find out if a method will no longer be called in general (though static analysis might be used to obtain an under-approximation).

In the hash table example, $query(e)$ starts a transaction t_e for the element e and also ends all transactions $t_{e'}$ for elements $e' \neq e$. $add(e)$ merely requires the transaction t_e to be active. Consequently (and in the absence of other methods not specified here), the set $trans_a(HT, r)$ contains at most one element. The component Cl participates in this transaction by the method $handle(e)$; note that while it is not active in this transaction, it maintains a transaction-local state in the *element* variable.

Another example is given in Sect. 9.3.1 on page 239; here a transaction t_u is begun by a call to $login(u)$, required to be active by $secret(u)$ and terminated by $logout(u)$. Again, any client that participates in this transaction will either produce all these calls in the course of a single method's evaluation, or maintain an internal, transaction-local state that tells it that it is currently logged in.

9.6.2. Tranquility and ς. As mentioned before, tranquility and state transferal are related: If a component is in a state that is not tranquil with respect to a transaction, an update preserving other components' assumptions is still possible if the transaction-local state is retained. Put the other way round, retaining transaction-local state during reconfiguration frees us from considering tranquility for that transaction.

We have spoken of transaction-global and transaction-local state; but this is an unnecessary differentiation: The persistent, global state can also be considered as part of a large transaction that never terminates. Let us assume that we are somehow (e.g., by annotations) supplied with a function $ts : C \times \mathscr{T} \times \mathcal{S} \to \wp(\mathcal{S})$; this function tells us, for a component, a transaction and the component's state which part of the state is relevant to it (by giving the set of all states that share this relevant state part). In the example above, we have two transactions: A global transaction t_g that is initiated by HT at the first method reception and never terminates, and the insertion transaction t_a that is begun by Cl:*handle* and terminated by either *elementFound* or *elementNotFound*; in this transaction HT:*query* and HT:*add* both participate.

For the former transaction, the part of the state space stored in variable *elements* of HT is relevant, thus

$$ts(HT, t_g, \sigma) = \{\sigma' \mid \sigma'(elements) = \sigma(elements)\}.$$

Cl.stored is not relevant, as it is only a "virtual" variable used to specify the expected communication, but is not made part of the actual state of Cl. For the latter transaction, *Cl.element* and *HT.setelement* are relevant.

During a component update (happening at the end of a finite run r), i.e., when replacing a component c by a structurally identical component c', possibly with changed method evaluators, but with the same specification, the ς-element of a plan needs to transport the part of the state space that is relevant to ongoing

transactions, i.e., if the component c has a state $\tilde{c}$, reconfiguration needs to provide:

$$\forall t \in \mathit{trans}_a(c, r) \cup \mathit{trans}_p(c, r) \,.\, \mathit{ts}(c', t, \varsigma(\{c \mapsto \sigma(\tilde{c}), c'\}) = \mathit{ts}(c, t, \sigma(\tilde{c})).$$

We thus require that the transaction-local state is retained by ς. Considering the state automata presented in Sect. 9.3.1, this is the requirement that the automaton state is preserved: If ς omitted the transaction-local state, the state automaton would be reset to its initial location, and subsequent communication might violate the protocol. By retaining the transaction-local state, this is avoided. Tranquility is just another way of avoiding such problems by enforcing that the state automaton has moved back to its initial state before reconfiguration commences. In this sense, state transferal is the more versatile approach, and can be used to handle long or even forever-running transactions. Tranquility, on the other hand, only requires read access to the transaction-local data (or, from an abstract perspective, to the current state of the state automaton).

CHAPTER 10

Conclusion

The largest of these will be the range of our projectiles – and even here we need consider only the artillery – which, however great, will never exceed four of those miles of which as many thousand separate us from the center of the earth.

— Galileo Galilei, Dialogues Concerning Two New Sciences

In this thesis, we have discussed runtime reconfiguration of component systems, and viewed the problem from two directions: by defining an algorithm with a formal backing, and by investigating application domains where reconfiguration can be utilized. These views are complementary: the formally defined algorithm resulted in a framework that allows for reconfiguration in a controlled setting, and the examples helped to understand the important aspects of reconfiguration. One of these examples is the CMC model checker, which, although not being reconfigurable, helped to understand the necessities for a reconfigurable system, and led to the definition of the JCOMP component model, which became the test-bed for reconfiguration.

JCOMP has a unique property among the reconfiguration frameworks: The provability of perceived atomicity in the context of reconfiguration, while being directly implementable in a regular programming language. We used JCOMP to investigate a number of examples, which we tried to categorize in order to work out areas where reconfiguration can be beneficial. This helped to develop a distinct view on reconfiguration that is different from many views expressed in the literature. We presented the examples in the context of a four-stage control loop, called the MAPE loop, which we used to accommodate the steps that precede a reconfiguration execution: the detecting of the need to conduct reconfiguration, and the planning of how to do it. The experience of this thesis suggests that reconfiguration is not a well-suited means for providing general adaptivity (as adaptivity is easier to obtain with case distinction), but rather for obtaining adaptivity in the context of differing (cross-cutting or separated) concerns and roles. We finally discussed the verification of component systems, relating a novel view on contract-based specification for static component setups with correctness requirements in the context of reconfiguration; this enables us to relate our approach to other approaches that do have a formal backing by considering quiescence and tranquility.

10.1. Discussion

We have investigated reconfiguration of component systems, exemplified with a component framework that implements a particular component model and places emphasis on certain aspects, while neglecting others. This influenced the view we took on reconfiguration, accentuating the need for continuous communication, while ignoring real-time properties. Also, the primary interest of this thesis is the software engineering aspect, both of writing a framework capable of reconfiguration, and performing reconfiguration in the context of software applications. This results

in a concentration on the last stage of the MAPE loop, as well as a strong interest in meaningful application areas for reconfiguration.

In this section, we will discuss the various design decisions we have made, comparing them to related work and pointing out strengths and weaknesses.

10.1.1. Using Reconfiguration for Adaptivity. Many of the work surveyed in Chapter 3 stress the importance of reconfiguration for modern software systems, e.g.

> *The ability to dynamically reconfigure the applications enhances the flexibility and adaptability of distributed systems.* [**CS02**]

Usually, this is taken for granted; and a distributed system surely needs to be adaptive and flexible. However, we feel that an important point is missing in this kind of reasoning: Reconfiguration is just one technique for providing flexibility, with certain strengths and weaknesses. The separation of roles actually represents both: It is the strength of reconfiguration to operate on a coarse level where entire components are replaced; this coarseness alleviates the problem of retaining the correctness of the complete system during a series of adaptations. On the other hand, a pure application of the separation of roles paradigm requires that adaptation is not made a concern of the individual components – which is often impossible. For example, an algorithm for scientific computing (e.g., a distributed model checker) has to be devised explicitly to be distributed. If the distribution architecture changes due to some external event (like the imminent departure of a participating computer), the algorithm itself has to be adapted – which is too fine-grained for reconfiguration, or at least requires that the algorithm is divided into components which already consider the reconfiguration.

Actually, reconfiguration of components is an attractive realization of adaptivity because it is *less* capable than customized approaches. By lifting the level of consideration to the scope of whole components, it can be hoped that devising reconfiguration on such a coarse level is easier and more tractable than operating on a finer level (e.g., self-modifying code). Much potential is sacrificed, however; using reconfiguration for adaptivity restricts the ways adaptivity can be obtained. Reconfiguration hence offers a compromise between capability and tractability.

It is most likely no coincidence that the applications of reconfiguration we found to be most convincing cover runtime modification of aspects [**JDMV04, CHS01, GBS03**]. Here, the cross-cutting concerns are integrated at a high level of coarseness, making full use of the capabilities of component-based reconfiguration. At the same time, having a cross-cutting concern change is more likely as having an unanticipated necessity for a change to the core concern of an application. We have addressed dynamic cross-cutting concerns by enabling filter component generation in JComp.

We may conclude that the use of reconfiguration of component systems is given by its ability to *combine* concerns of different collaborators. Whether they are cross-cutting or separated by granularity, reconfiguration provides means to handle changes of these concerns. Reconfiguration is a software engineering approach: It provides a means to devise adaptive systems in a well-described manner, offering assistance in an area that is inherently complex.

It is important to remember that most of the word *reconfiguration* reads *configuration* – if we cannot find a configuration for a system in a flexible manner, we will have a hard time to change this system by (generic) reconfiguration. This again indicates the success of dynamic cross-cutting concerns as a provider of examples for reconfiguration: It is as easy to provide provisions for adding an aspect during configuration as it is to reconfigure such an aspect at runtime, should the

cross-cutting concern change. It also indicates why providing reconfiguration provisions alone is not sufficient for self-$\star$ properties: If we cannot plan how a system providing some functionality can be configured, we will also not be able to provide a plan for realizing self-healing or self-optimization by reconfiguration.

Providing means to *execute* plans for self-healing often ignores that those plans are hard to come by (and hence, credible examples are missing). The MAPE loop discussed in Chapter 8.1 remains to be necessary for reconfiguration – and deferring some of its stages to the future neglects that there are ample situations where it can already be fully established. This situation is similar to what is described in [**BD06**] – we can employ reconfiguration, although the application domains are different from those originally anticipated. We hope that this thesis provides some indications in this regard, but we are well aware of the fact that many of the topics required for successful application of reconfiguration in our sense are still missing.

10.1.2. The Formal Approach. One of the goals of this thesis is to provide a formal component model and implement it as precisely as possible. We take pride in the fact that this succeeded to an extent that allows for a precise description of ways to circumvent the restrictions (e.g., use of `static` variables in JComp). Compared to other works, this property is unique; while the established component frameworks like SOFA and Fractal/ProActive do have formal frameworks, the abstraction level is usually higher. For our component models, we can claim that the component process terms are low-level enough to offer a direct connection to actual code. Also, we are not aware of a reconfiguration implementation that is formally described on a level as fine as we provide it.

We have shown some interesting and important results, albeit only for a severely restricted class of plans. This restriction is quite limiting, especially since we see the addition and removal of filter components as a primary application of reconfiguration (since changes to cross-cutting concerns are responded to like this). While theoretically not difficult to circumvent, in practice we often use the full reconfiguration plans and revert to problem-specific reasoning for providing correctness arguments. Usually, this is not very hard, since the interruption of communication by removal or addition of a filter is quite controllable. Our experience indicates, however, that these correctness considerations are indispensable, since concurrent reconfiguration is prone to getting stuck because of lost messages. We would like to provide further assistance within the component model for such situations. For example, when investigating non-injective shallow reconfiguration plans, we had to craft an example that violates communication expectations carefully. For most examples, the nondeterminism introduced by regrouping messages is no problem, since the messages might have just arrived in that order anyway. We feel that further research might be beneficial in this area, maybe offering a hierarchy of properties that can be guaranteed for various plans. Possibly, communication protocols need to be considered as well; while we formulate such protocols in Chapter 9 we have not linked them to correctness properties of reconfiguration.

Providing a formal description of reconfiguration plans and their execution is quite beneficial. Early versions of JComp used different reconfiguration plans which were more general (e.g., allowing for explicit removal and addition of edges). It was easy to build plans that would not work at all, or disrupt the communication of components in an unexpected way. After switching over to plans as described in this thesis, the effort required to come up with consistent plans was greatly decreased – and the framework got considerably less complicated at the same time. Likewise, the use of δ^ρ made writing plans easier as well: For the situations where message retainment is useful, the messages usually follow the rewiring. Admittedly, often message retainment is not required at all.

When developing a first idea about the features reconfiguration should possess, CMC was very influential. Originally, we wanted to substitute a genuine hash table by the caching/disk structure, retaining the state space explored so far. While this never came to pass for reasons explained in Sect. 8.5, it sparked great interest in state transfer. We then realized that in applying these ideas to the asynchronous model of JComp, messages are as important. Even more, for proving properties like perceived atomicity, message retainment is much more crucial than data state retainment. This is reflected in the difference of abstraction between the δ reconfiguration plan element, which is not very general, and the abstract definition of the ς element. Only at a later point in time we realized that in going a step further and considering a transaction-preserving reconfiguration (as an alternative to tranquility considerations) ς becomes as important as δ. Again, we lack sufficiently complicated examples – transaction retainment is now done on the user level and has not posed any problem yet. At the same time, we have not come across examples where δ was insufficient because it lacks provisions for transforming the messages that are retained (e.g., discarding messages meeting some semantical criteria, or modifying their parameters). Still, we are fully aware of the fact that our choice of features offers a certain view on reconfiguration that does not make communication protocols a concern of the reconfiguration algorithm. In different and more complicated settings, this might become indispensable to retain the property we value most about our reconfiguration approach: For a class of reconfiguration plans, some properties are guaranteed by design.

As for state transferal, we have struggled much with the choice of using the direct or indirect approach, which either violate the encapsulation of components or the separation of roles paradigm. We have not found an answer that is satisfying, but feel that the hybrid state transferal described in Sect. 6.8.2 is a good compromise. State transferal is a difficult problem, and a general solution is unlikely to exist (cf. the honest statement in [**Van07**, pp. 173]). We feel that we have introduced a good inclusion of the problem of state transfer to the general software engineering approach of reconfiguration, but also consider this to be a problem that needs to be addressed for particular problem domains.

10.1.3. The JComp Framework. JComp was devised to be a minimal framework that offers just the features really required, and put as much responsibility as possible – like distribution or tranquility – on the user. This helps to keep things understandable, and the framework remains lean. On the other hand, the support for reconfiguration is deliberately limited to some aspects. Actually, the JComp framework implementation closely follows the formal description: It provides provisions for building component setups and reconfiguration plans and a way to execute the plan. It does not provide a dedicated thread for reconfiguration execution, leaving the choice to the user (subsequently allowing for deadlocks if reconfiguration is started from the thread of a component that is contained in the R plan element).

The most important thing about an actual implementation of reconfiguration is to be meticulously aware of who does what. If an exception is to be dealt with by reconfiguration, a great deal of caution needs to be exercised to delegate the execution of the reconfiguration to a worker thread (as the thread that actually reports the reconfiguration to the monitor is that of the component that just failed, and hence the thread that is required for the subsequent state transferal). Clearly, JComp lacks support here. This is because reconfiguration resides on the assembly code level, which is not supported explicitly by JComp. For supporting reconfiguration as an "every day concept", a more elaborate steering mechanism is required (although the literature suggests that few frameworks genuinely support this, cf. Chapter 3).

The JComp model fixes the communication means to message passing. It supports asynchronous calls, and – as an exception to the claim of minimalism – synchronous calls as well. We included the latter because they add an interesting dimension to the problem of reconfiguration, as well as the possibility for deadlocks that needs to be addressed. Also, we considered synchronous calls as necessary when devising the JComp model, a view that has changed ever since. Ultimately, the use of indirect state transfer protocols required the retainment of synchronous calls, as we did not want to allow a component stopped for reconfiguration to issue calls; this would have been required for conducting state transfer with a subprotocol and asynchronous calls only. For proving properties, the restriction of communication is desirable, since we require a precise semantics of component process terms. On the other hand, many interesting communication means (e.g., multimedia data streams) are not accessible in JComp, and it is possible that our search for examples was limited by this design decision.

This sparked interest in introducing a new layer into the system and in placing more emphasis on *connectors*, i.e., objects that handle the communication between components. Connectors are first-class entities in SOFA 2.0 [**BHP06**] and recommended by various researchers [**LEW05, BMH08**]. By utilizing different types of connectors, reconfiguration can be handled by the connector [**BHP07**], which can be aware of how pending communication needs to be retained. We have proposed this as the reconfiguration means for the REFLECT project [**SvdZH08**], where both control messages and multimedia data need to be communicated between components, and where a genuine need for reconfiguration is given.

Overall, JComp should not be mistaken for a production-grade component framework in the league of Fractal/ProActive or Corba. Instead, it is a framework dedicated for supporting our particular view on reconfiguration, useful for investigating various application domains of reconfiguration and learning about their particular problems and requirements. In the experience obtained with this thesis, formulating reconfiguration in the form of plans and relying on their atomic execution eases the development of reconfiguration examples considerably, especially since much of the risk of running into deadlocks is diverted to the framework.

10.1.3.1. *Flexibility and Clarity.* Having first-class connectors provides much flexibility, and allows for a wider range of reconfiguration – maybe also reconfiguration that is less complicated than the approach presented in this thesis (because it might decide to not consider message retainment, or state transfer). At the same time, having such flexibility voids any chance to get any guarantees about the reconfiguration from the framework: Since reconfiguration is actually implemented on the user level, the framework can offer less provisions for handling the pending problems.

As mentioned before, the difference between the concreteness of the δ and ς reconfiguration plan elements is motivated by the necessity to have a close control on the message retainment in order to guarantee a minimal impact of reconfiguration. On the other hand, the experience obtained when distributing JComp (cf. Sect. 7.3) indicates that supporting too much within the framework can lead to inflexible designs while bloating the framework. We can phrase this as the choice between clarity and flexibility. Obviously, the results obtained for the JComp model carry over to a connector that supports the same communication paradigms, but then the guarantees need to be checked for each situation.

Any reconfiguration-enabling framework needs to provide a mixture of flexibility and restriction in order to provide clarity. The literature reviewed in Chapter 3 suggests that usually (sometimes with the exception of quiescence and tranquility, cf. Tab. 3.1) not much restriction is imposed, if it is not already imposed by the

framework, or mandated by the problem domain (e.g., the necessity to retain the component's signature during hot code updates). This thesis aims to be more on the restrictive side.

In the experience obtained during writing this thesis, reconfiguration proved to be extraordinary hard to debug. It needs to cope with so many problems that often (e.g., with the CMC model checker, cf. Sect. 8.5) its gain is outweighed by the effort required. Hence, any provision taken to reduce the effort seems to be a step in the right direction, unless (and this happens with examples like the one presented in Sect. 8.3.2) the restrictions need copious workarounds for problems that are not natively supported. The calculation of δ^ρ for shallow reconfiguration plans is one attempt at reducing the effort required for devising and debugging reconfiguration, by making it more robust against interleaving with other components' communication. For practical use of reconfiguration, more effort is required to avoid the resulting workarounds; we now feel that the focus should migrate more to flexibility considerations, and we regard first-class connectors as a promising step towards this direction.

10.1.3.2. *Benefits of a Hierarchical Component Model.* We have mentioned it a few times in the examples, but let us point out this once more: Hierarchical components are necessary for a reconfiguration-enabled framework. JComp does not have provisions for hierarchical components, which was motivated by the Erlang approach, which allows its processes to be connected in an arbitrary graph structure, but suggests that a hierarchical, tree-like structure is used [**AVWW96**, pp. 67]. In a similar fashion, for JComp we wanted to desist from making a component hierarchy first class, and rather encourage understanding components as grouped on a conceptual level that is reflected only by a naming convention or component annotation (cf. the example in Sect. 8.3, which maintains user-defined groups of components responsible for loading an RSS thread). It may be discussed whether keeping the component model simplistic is not offset by the complications of defining reconfiguration plans in such a setting. The support of JComp for generating plans, however, is not very sophisticated, and needs further research.

What *is* missing is a well-defined provision for *controlling* reconfiguration. The code that implements the MAPE loop is not part of a component implementation; it is commonly referred to as "assembly code". Likewise, the thread that performs the stages of the MAPE loop is not clearly defined – it might be a component's thread, obtained by the monitoring provisions, or a special, user-handled thread that is started alongside the assembly. The component models of Fractal [**BCL+06**] and SOFA [**BHP06**] provide provisions here by means of *membranes* and *controllers* respectively. A corresponding entity is required for JComp in order to avoid ad-hoc solutions for every reconfiguration scenario.

10.1.4. On MAPE Loops. Like most other component frameworks supporting reconfiguration (a notable exception being [**KM07**]), JComp focuses on the execution stage of the MAPE loop. From a software engineering perspective, this is the stage most interesting when building a framework capable of reconfiguration. We support the monitoring phase by provisions for communication and exception monitors. The support is limited, however; as we have discussed above JComp only provides hooks for attaching listeners, but does not directly support a reconfiguration thread. Using hierarchical components with a dedicated controller would provide such a thread automatically; right now, we require the user to maintain a dedicated switch and data structures for passing information.

The assessment and planning phase are currently not directly supported by JComp, although we provide some tools that might be useful. We have implemented the graph transformation algorithm described in Sect. 8.1.4.1 to integrate with the Groove graph transformation rule editor [**Ren03**]. This offers a convenient tool, but we have never used the full power of graph transformations – we have not come across an example where the set R was not uniquely defined. Graph transformation would offer just that: Matching to different locations in the component graph, thus yielding different reconfiguration plans. For filter insertion, this might be useful, but filter insertion can also be planned by a single traversal of the component graph.

For the assessment phase, we support the decision-making by providing access to the JVMTI API to obtain memory and time consumption of components. This helps to detect resource-intensive components. We also implemented provisions for real-time temporal logic monitoring, but then found this to be an approach that does not integrate well with the JComp framework, which is largely ignorant of real-time requirements. For example, when imposing a tight allowable delay on message execution times, eventually an adverse interleaving (possibly combined with garbage collection) will violate such a constraint.

For our examples, the planning phase is not very prominent. We have pointed out some related work in Sect. 8.1.4, and feel that this is a challenging area of research. We believe that planning reconfiguration should be done in the context of an application area – for example, the planning of the example in Sect. 8.4.4, where we chose a suitable implementation of a set representation, can be completely automated; but the communication constraints imposed by this particular setup are modest. We do not believe that generic planning is possible with the technology available today, but that there are interesting fields where planning can be employed successfully.

10.1.5. Applicability of Reconfiguration. Based on the software engineering background of this thesis, we tried to evaluate the capabilities of JComp against small-scale, yet real-world examples. As discussed in Sect. 3.5, good examples are scarce for general-purpose frameworks with reconfiguration support. Often, an "inverted invest-gain-schema" is the problem: While a lot of problems can be addressed by reconfiguration, the effort to maintain a dedicated component framework and handle all the related issues do not seem worth the effort. For example, a component that reads the contents of a file might be reconfigured if the user selects a different source file – but such a setup can most likely be solved easier by just providing an interface for switching to a new file name.

Still, there are application domains where reconfiguration can be put to good use. Good examples are found in work that addresses less generic application domains. This is obvious for hot code updates and frameworks like Simplex [**SRG96**], Hercules [**CD99**] and Draco/Fresco [**VB05**]. In Sect. 8.4, we have argued that the dynamic response to changing cross-cutting concerns is more likely to provide truly useful applications for reconfiguration, as cross-cutting concerns are more likely to change than the core concern is.

Also, the consideration of the separation of roles paradigm leads to good examples. The problem here is that obtaining a true separation of roles is often not possible in research projects, unless they have a certain size (e.g., [**KM07**]). We will try to explicitly use this in the REFLECT project [**SvdZH08**], where, ultimately, the reconfiguration plans are to be written by human-computer-interaction experts, who are not necessarily computer scientists (but psychologists). At projects of this size and structure (i.e., collaborations of different disciplines to build adaptive software), reconfiguration might prove to be just the granularity required.

Applicability of reconfiguration is closely tied to the separation of roles. We have become convinced that only if such a separation is given, reconfiguration can be truly gainful. Otherwise, a case distinction within the components involved will usually be much less troublesome to implement. But if such a case distinction is not feasible, because the components are not available in source code or because the application developer does not want to be concerned with modifying components, reconfiguration is required to obtain adaptivity. It is our hope that the larger the applications become, the more importance is placed on the clean separation between the concern of the component developer and the application designer.

10.1.6. Formal Methods and Reconfiguration. Reconfiguration is difficult because of multiple sources of nondeterminism: The interleaving with the system not stopped during reconfiguration, the point in time when reconfiguration commences, but also the exact shape of the component setup when reconfiguration is triggered, if multiple reconfigurations are to be combined. Obviously, nondeterminism is a source of errors, since it is so easy to miss a problematic sequence of events. In fact, the association of reconfiguration to components is most likely due to the fact that components provide *some* control over nondeterminism (by decoupling the concurrent components by explicit communication). There are few attempts to use self-modifying code for adaptation (although this is a topic of research, cf. [**TY05**]); the "live" introduction of code changes in an otherwise unknown system is way too dangerous, and hot code update is usually done relying on a coarse modularization of the software, e.g., in processes (as in ERLANG [**Arm07**]) or components [**VB02**].

Considering a coarse-level reconfiguration aims at reducing the (relevant) nondeterminism such that reconfiguration remains tractable. Still, not all nondeterminism can be accounted for this way, most notable the point in time when reconfiguration starts. Quiescence and tranquility considerations try to reduce the risk imposed by this kind of nondeterminism, but this may not always be feasible to do. Formal methods can then be used to handle the remaining uncertainty and provide formal proof that all possible traces of the system are indeed admissible.

Formal methods do not come for free. They require careful annotation of the components, or the use of a restricted programming language. Usually, only a single problem is addressed, e.g., communication protocols [**PV02, Kof07**]. The guarantees offered are then limited to this problem; in this light formal methods are useful as a developing support and for clarifying certain aspects of the component application. Communication protocols are interesting for building component applications with stateful components – we did not consider them for CMC, but actually most of the troublesome memory leaks originate in faulty assumptions on the communication behavior. For example, if we freed a state and subsequently ran into a segmentation fault, we falsely believed that recent communication suggested that the state was no longer in use elsewhere. We did not check communication protocols for CMC, but attaining this view might have been beneficial.

The separation of roles is again the key to profitable use of formal methods, and, as stated before, during this thesis there was no practical chance to work at large-scale projects with a true separation of roles. It is pretty much conceivable that, if components need to be combined that have been developed independently, contracts become more important. This directly carries over to reconfiguration: Specification could support some means of defining substitutability, allowing for an automated plan generation. We are not sure if this is a feasible way to go. Detecting suitable replacements for a malfunctioning component is hard, and we are unsure if the assumption that a component repository can become large enough such that replacability can no longer be explicitly defined using much less effort is justified. For some time we considered this to be the weakest aspect of the MAPE loop, and

subsequently searched for examples where planning was expressible by a simple algorithm. The work done on automated planning is encouraging, though. At least for special application domains (similar to the set representation replacement example of Sect. 8.4.4) planning might be able to do more than a mere instantiation of a predefined plan.

We still consider the most prevalent benefit of formal methods the ability to judge the correctness of a component setup against the expectations of the individual components. This is also quite useful when building reconfiguration plans, since the resulting configurations might be checked for communication consistency. For large-scale applications, such an approach might prove indispensable in order to avoid non-reproducible errors after repeated reconfiguration.

10.2. Future Work

Lots of aspects of reconfiguration are not yet fully understood. Maybe the most difficult problem is the need of semantical information about the components. The more generic the assessment and planning phases of the MAPE loop are required to be, the more semantic information is required. Semantic web services illustrate how difficult finding a component with a required functionality can be, and services can be expected to be much less demanding than components which not only provide, but also require functionality.

10.2.1. The JComp Framework. We have discussed the most pressing extension to the JComp framework a number of times: The provision of hierarchical components. Hierarchical components are not really required, but they offer a way to specify exactly where the assembly code that realizes the initial configuration as well as subsequent reconfiguration should be placed. This is indispensable for realizing complex reconfiguration scenarios.

However, we feel that simply organizing components in a hierarchical manner might be insufficient for realizing architectures like a Chain of Responsibility pattern. We have argued in Sect. 8.3.1 that such a chain substitutes a multi-port. As such, it can be considered to be a hierarchical component – but the number of roles visible from the outside depends on its internal architecture. Even more, if the cardinality of the multi-port changes, its internal structure needs to be reconfigured as described in Sect. 8.3.2. Such a change will be the result of a reconfiguration of the application, in which the chain is a single component. Hence, the reconfiguration of the application also triggers a reconfiguration in one of its (composite) components. Addressing such scenarios requires careful planning and poses an interesting challenge, both for describing and executing the reconfiguration process.

JComp, however, is way too limited for practical use. Most notably, the fixing of the message passing, no-shared-memory communication paradigm is too limiting for practical use. We feel that JComp should be integrated into a more capable component model which supports more communication paradigms and component concepts, and preferably offers generic connectors [**LEW05**, **BMH08**] and custom controllers [**BHP07**]. The algorithm for reconfiguration might then be included by a dedicated connector and controller, allowing the user to leverage the guarantees provided if this is desired, without giving up the flexibility of the component model, which is required for addressing a range of application scenarios. We plan to do this in the context of the Reflect project, as discussed in Sect. 10.2.4.

10.2.2. MAPE Loops. Triggering and planning the reconfiguration is as important as its execution. Using closed-control loops is the most prevalent means in the literature to link the preparation and the execution of reconfiguration. Of the four stages of the MAPE loop presented in this thesis, the planning stage demands

most improvement. We have argued that this needs to be done in a domain-specific way, but there are also ample opportunities to provide generic support for planning, without offering concrete solutions.

For example, we found it quite tedious to come up with plans in concrete implementations, because they require multiple traversals of the component graph while picking out the actual components to be reconfigured and their connections to components that need to be retained. In JComp, this is done by implementing the Visitor pattern in Java. Offering a domain specific language for these traversals might ease the task of generating plans. Also, the various means of matching and modifying graphs might be adapted, although we are sure that the matching criteria are usually more complex with regard to the actual components to be matched than they are with respect to the layout of the component subgraph (e.g., we might want to identify all components that declare themselves responsible for handling a given RSS feed, remove them, and redirect all incoming connections to a placeholder component).

Another issue that needs to be considered is the necessity to change the planning over time. We have not provided an example in this thesis, but there might be situations where two different MAPE loops address the same sub-setup of a component application. For example, we might combine the aforementioned substitution of RSS processing components with a reconfiguration that alters the layout of the sub-setup (or hierarchical component) used for reading an RSS feed. If the former reconfiguration switches from a placeholder component to an RSS processing sub-setup, it needs to plan this sub-setup to take the form mandated by the latter reconfiguration. We think that hierarchical MAPE loops might be useful for such an operation. Here, a higher-level MAPE loop not only reconfigures parts of the component setup, but also other, lower-level MAPE loops. This integrates well with hierarchical components, but we are currently not sure if a strict hierarchy can be applied in all situations – reconfigurations might even be mutually interacting.

Another interesting aspect of planning a reconfiguration is the description of the state transfer. Again, promising approaches are provided for special domains [**VB05**], whereas a general solution usually conflicts with traditional component-based software development techniques. However, state transferal might be improved by having components describe parts of their internal state (as well as the methods used for accessing it). Also, given that the separation of roles cannot be completely maintained if state transferal is desired, an investigation of the changed software engineering process would be interesting – if components are to be written with reconfiguration scenarios in mind, then some techniques and patterns might emerge that allow for flexibility while retaining a clean component structure.

Finally, we would like to stress the importance of planning entire configurations (as opposed to planning the change of a given architecture, to be conducted by reconfiguration). The cmc example suggests that there is much to be gained from automated, problem-specific reasoning. There is some research being conducted, e.g., for automated performance tuning [**WD98, HBHH07**] or in the context of "software product lines" [**PBL05**]. The Amphion project even configures entire applications from atomic code pieces [**LPPU94, PL96**]. We have, however, not yet seen an approach that configures an application for a special task that it needs to handle. We feel that not only applications like cmc might benefit from automated configuration, but that the insights obtained also carry over to reconfiguration planning.

10.2.3. Reconfiguration for Large-Scale Systems. A related problem is given by the necessity of multiple reconfigurations conducted sequentially. If a system is to be truly adaptive, it has to support repeated reconfiguration, possibly going

back to configurations previously encountered, possibly iterating the effects of multiple reconfigurations addressing various cross-cutting concerns over time. We have investigated such a problem in Sect. 8.3, but have to admit that this is actually just a hierarchical component setup with reconfiguration operating on different levels of granularity.

The reconfiguration examples presented in this thesis assume complete knowledge about the situation for which a reconfiguration plan gets formulated. For a truly adaptive system with a sufficiently strong planning approach, such an assumption can hardly be maintained. Instead, a loosely evolving system needs to be handled, with plan generation operating on an abstracted design of the system. There is some research that might support such a direction, e.g., the research on architectural styles [**BLM08**], but we feel that the purpose that is most evident in the research presented in this thesis is not emphasized much.

10.2.4. Special-Purpose Reconfiguration: REFLECT. We believe that the basic approaches towards reconfiguration are understood well enough to utilize reconfiguration for special domains in practice. An improvement of the software engineering process can be expected, if reconfiguration is to replace component-internal adaptivity solutions.

In the REFLECT project, we try to use reconfiguration of multimedia systems (comprised of components and connectors that transport messages as well as data streams) in order to allow HCI experts to design adaptive software [**SvdZH08**]. In such a setting, some problems emerge that are usually not considered: The requirement for defining control loops in a domain-specific language, and the ability to remain metaphorical – the "plug-ability" of components is required to understand how the system works, and how it can be reconfigured. We believe that the medium-term success of reconfiguration as a tool of software engineering will be dependent on finding such problem domains which are succinctly describable and give rise to special assessment and planning algorithms. In such settings, reconfiguration planning can be formulated by a domain expert, requiring only modest amounts of automated reasoning. In such situations, where a genuine separation of roles is provided by the application domain, reconfiguration might prove a valuable tool for attaining flexibility while retaining an overall clarity and tractability of the application. Although REFLECT will use first-class connectors, this thesis provides the formal basis for the approach to reconfiguration, and many results can be used directly for message-oriented connectors.

Bibliography

[AAA+06] Alexandre Alves, Assaf Arkin, Sid Askary, Ben Bloch, Francisco Curbera, Yaron Goland, Neelakantan Kartha, Sterling, Dieter König, Vinkesh Mehta, Satish Thatte, Danny van der Rijn, Prasad Yendluri, and Alex Yiu. Web services business process execution language version 2.0. OASIS Committee Draft, 2006.

[ÅBD+07] Karl-Erik Årzén, Antonio Bicchi, Gianluca Dini, Sephen Hailes, Karl H. Johansson, John Lygeros, and Anthony Tzes. A component-based approach to the design of networked control systems. *European Journal of Control*, 13(2–3), 2007.

[ACN02a] Jonathan Aldrich, Craig Chambers, and David Notkin. Architectural reasoning in ArchJava. In Boris Magnusson, editor, *16th European Conference on Object-Oriented Programming (ECOOP '02)*, volume 2374, pages 334–367. Springer, 2002.

[ACN02b] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *24th International Conference on Software Engineering (ICSE '02)*. ACM Press, 2002.

[ADG98] Robert Allen, Remi Douence, and David Garlan. Specifying and analyzing dynamic software architectures. In *Conference on Fundamental Approaches to Software Engineering (FASE'98)*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 1998.

[AEHS06] Mourad Alia, Frank Eliassen, Svein Hallsteinsen, and Erlend Stav. MADAM: towards a flexible planning-based middleware. In *International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS '06)*, pages 96–96. ACM Press, 2006.

[AENV06] Volkan Arslan, Patrick Eugster, Piotr Nienaltowski, and Sebastien Vaucouleur. SCOOP - concurrency made easy. In Jürg Kohlas, Bertrand Meyer, and André Schiper, editors, *Research Results of the DICS Program*, volume 4028 of *Lecture Notes in Computer Science*, pages 82–102. Springer, 2006.

[AFZ05] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A calculus for dynamic reconfiguration with low priority linking. *Electronic Notes in Theoretical Computer Science*, 138(2):3–35, 2005.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[AGD97] Robert Allen, David Garlan, and Remi Douence. Specifying dynamism in software architectures. In *Workshop on Foundations of Component-Based Software Engineering*, 1997.

[AH03] Wil M. P. van der Aalst and Arthur H. M. ter Hofstede. YAWL: Yet Another Workflow Language (revised version). Technical Report FIT-TR-2003-04, Queensland University of Technology, 2003.

[AH05] Naveed Arshad and Dennis Heimbigner. A comparison of planning based models for component reconfiguration. Technical Report CU-CS-995-05, Department of Computer Science, University of Colorado, 2005.

[AHKB03] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[AHP94] Brent Agnew, Christine R. Hofmeister, and James M. Purtilo. Planning for change: a reconfiguration language for distributed systems. In *2nd International Workshop on Configurable Distributed Systems (IWCDS '94)*, volume 5, pages 313–322. IEEE Computer Society Press, 1994.

[AHW03] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. In *15th IEEE International Conference on Tools with Artificial Intelligence (ICTAI '03)*, pages 39–46. IEEE Computer Society Press, 2003.

[Ald03] Jonathan Aldrich. *Using Types to Enforce Architectural Structure*. PhD thesis, University of Washington, 2003.

[Ald08] Jonathan Aldrich. Using types to enforce architectural structure. In *7th Working IEEE/IFIP Conference on Software Architecture (WICSA '08)*, pages 211–220. IEEE Computer Society Press, 2008.

[ALF00] Lörinc Antoni, Régis Leveugle, and Béla Fehér. Using run-time reconfiguration for fault injection in hardware prototypes. In *15th IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems (DFT '00)*, pages 405–413. IEEE Computer Society Press, 2000.

[All97] Robert Allen. *A Formal Approach to Software Architecture*. PhD thesis, Carnegie Mellon, School of Computer Science, 1997.

[And06] Neculai Andrei. Modern control theory – a historical perspective. *Studies in Informatics and Control*, 15:51–62, 2006.

[Ant07] Richard Anthony. Existing technologies. Technical Report D1.1A, Dynamically Self-Configuring Automotive Systems (DySCAS) – Project no. FP6-IST-2006-034904, 2007.

[AP03] Jirí Adámek and František Plášil. Behavior protocols capturing errors and updates. In *2nd International Workshop on Unanticipated Software Evolution (USE '03)*, pages 17–25. University of Warshaw, 2003.

[AP05] Jirí Adámek and František Plášil. Component composition errors and update atomicity: static analysis. *Journal of Software Maintenance*, 17(5):363–377, 2005.

[Apa08] The Apache Software Foundation. The Apache Velocity project. http://velocity.apache.org/, 2008.

[App08] Austin Appleby. MurmurHash 2.0. http://murmurhash.googlepages.com, 2008.

[AR00] Jesper Andersson and Tobias Ritzau. Dynamic code update in JDrums. In *1st Workshop on Software Engineering for Wearable and Pervasive Computing (SEWPC '00)*, 2000.

[Ara06] Gustavo Quirós Araya. Static byte-code analysis for state space reduction. Master's thesis, Faculty of Mathematics, Computer Science and Natural Sciences of RWTH University, Aachen, 2006.

[Arm07] Joe Armstrong. A history of Erlang. In *3rd ACM/SIGPLAN Conference on History of Programming Languages (HOPL '07)*, pages 1–26. ACM Press, 2007.

[Arn99] Ken Arnold. The Jini architecture: dynamic services in a flexible network. In *36th ACM/IEEE Conference on Design Automation (DAC '99)*, pages 157–162. ACM Press, 1999.

[Ars03] Naveed Arshad. Dynamic reconfiguration of software systems using temporal planning. Master's thesis, Department of Computer Science, University of Colorado, 2003.

[ASCN03] Jonathan Aldrich, Vibha Sazawal, Craig Chambers, and David Notkin. Language support for connector abstractions. In Luca Cardelli, editor, *17th European Conference on Object-Oriented Programming (ECOOP '03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 74–102. Springer, 2003.

[ASS07] Marcio Augusto, Sekeff Sallem, and Francisco Jose da Silva e Silva. The Adapta framework for building self-adaptive distributed applications. *icas*, 0, 2007.

[Avi85] Algirdas Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[AVWW96] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice Hall, second edition edition, 1996.

[AWSN01] João Paulo A. Almeida, Maarten Wegdam, Marten van Sinderen, and Lambert J. M. Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *3rd International Symposium on Distributed Objects and Applications (DOA '01)*, pages 197–207. IEEE Computer Society Press, 2001.

[BA01] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[BABR96] Luc Bellissard, Slim Ben Atallah, Fabienne Boyer, and Michel Riveill. Distributed application configuration. In *16th International Conference on Distributed Computing Systems*, pages 579–585. IEEE Computer Society Press, 1996.

[Bar96] Ingo Barth. Configuring distributed multimedia applications using CINEMA. In *International Workshop on Multimedia Software Development (MMSD '96)*, page 69. IEEE Computer Society Press, 1996.

[Bau05] Andreas Bauer. Simplifying diagnosis using LSAT: a propositional approach to reasoning from first principles. In Roman Barták and Michela Milano, editors, *International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR '05)*, volume 3524 of *Lecture Notes in Computer Science*, pages 49–63. Springer, 2005.

[BBC+06] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Software Environments and Tools*, chapter Programming, Deploying, Composing, for the Grid. Springer, 2006.

[BBC+08] Lubomír Bulej, Tomas Bures, Thierry Coupaye, Martin Decký, Pavel Jezek, Pavel Parizek, Frantisek Plasil, Tomás Poch, Nicolas Rivierre, Ondrej Sery, and Petr Tuma. CoCoME in Fractal. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 357–387. Springer, 2008.

[BBGH08] Alessandro Basso, Alexander Bolotov, Vladimir Getov, and Ludovic Henrio. Dynamic reconfiguration of GCM components. Technical Report TR-0173, CoreGRID - Network of Excellence, 2008.

[BCDW04] Jeremy S. Bradbury, James R. Cordy, Juergen Dingel, and Michel Wermelinger. A survey of self-management in dynamic software architecture specifications. In *1st ACM SIGSOFT Workshop on Self-managed Systems (WOSS '04)*, pages 28–33. ACM Press, 2004.

[BCG+05] Boualem Benatallah, Fabio Casati, Daniela Grigori, Hamid R. Motahari Nezhad, and Farouk Toumani. Developing adapters for web services integration. In Oscar Pastor, Jo ao, Oscar Pastor, and Jo ao, editors, *17th International Conference on Advanced Information Systems Engineering (CAiSE '05)*, volume 3520 of *Lecture Notes in Computer Science*, pages 415–429. Springer, 2005.

[BCL+04] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. An open component model and its support in java. In Ivica Crnkovic, Judith A. Stafford, Heinz W. Schmidt, and Kurt C. Wallnau, editors, *7th International Symposium on Component-Based Software Engineering (CBSE '04)*, volume 3054 of *Lecture Notes in Computer Science*, pages 7–22. Springer, 2004.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The FRACTAL component model and its support in Java. *Software—Practice & Experience*, 36(11-12):1257–1284, 2006.

[BD06] Genevieve Bell and Paul Dourish. Yesterday's tomorrows: Notes on ubiquitous computing's dominant vision. *Personal and Ubiquitous Computing*, 2006.

[BDH+98] Manfred Broy, Anton Deimel, Juergen Henn, Kai Koskimies, František Plášil, Gustav Pomberger, Wolfgang Pree, Michael Stal, and Clemens A. Szyperski. What characterizes a (software) component? *Software - Concepts and Tools*, 19(1):49–56, 1998.

[BDH+08] Tomáš Bureš, Martin Děcký, Petr Hnětynka, Jan Kofroň, Pavel Parízek, František Plášil, Tomáš Poch, Ondřej Šerý, and Petr Tůma. CoCoME in SOFA. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 388–417. Springer, 2008.

[BDWW89] Mario R. Barbacci, Dennis L. Doubleday, Charles B. Weinstock, and Jeannette M. Wing. Developing applications for heterogeneous machine networks: The Durra environment. *Computing Systems*, 2:7–35, 1989.

[Bec00] Kent Beck. *eXtreme Programming explained.* Addison-Wesley, 2000.

[BFG+01] Stefan Blom, Wan Fokkink, Jan Friso Groote, Izak van Langevelde, Bert Lisser, and Jaco van de Pol. μCRL: A toolset for analysing algebraic specifications. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *13th International Conference on Computer Aided Verification (CAV '01)*, volume 2102 of *Lecture Notes in Computer Science*, pages 250–254. Springer, 2001.

[BG99] Bernard Boigelot and Patrice Godefroid. Symbolic verification of communication protocols with infinite state spaces using qdds. *Formal Methods in System Design*, 14(3):237–255, 1999.

[BGHS04] Howard Barringer, Allen Goldberg, Klaus Havelund, and Koushik Sen. Program monitoring with LTL in EAGLE. In *18th International Parallel and Distributed Processing Symposium (IPDPS '04)*, volume 17. IEEE Computer Society Press, 2004.

[BH77] Henry C. Baker and Carl Hewitt. The incremental garbage collection of processes. *ACM SIGPLAN Notices*, 12(8):55–59, 1977.

[BH99] Ahmed Bouajjani and Peter Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1-2):211–250, 1999.

[BHH+06] Hubert Baumeister, Florian Hacklinger, Rolf Hennicker, Alexander Knapp, and Martin Wirsing. A component model for architectural programming. *Electronic Notes in Theoretical Computer Science*, 160:75–96, 2006.

[BHP06] Tomás Bureš, Petr Hnětynka, and František Plášil. SOFA 2.0: balancing advanced features in a hierarchical component model. In *4h International Conference on Software Engineering Research, Management and Applications (SERA '06)*, pages 40–48. IEEE Computer Society Press, 2006.

[BHP07] Tomáš Bureš, Petr Hnětynka, and Frantisek Plášil. Runtime concepts of hierarchical software components. *International Journal of Computer & Information Science*, 8:454–463, 2007.

[BHSS03] Gavin Bierman, Michael Hicks, Peter Sewell, and Gareth Stoyle. Formalizing dynamic software updating. In *2nd International Workshop on Unanticipated Software Evolution (USE '03)*, 2003.

[BISZ98] Christophe Bidan, Valerie Issarny, Titos Saridakis, and Apostolos Zarras. A dynamic reconfiguration service for CORBA. In *International Conference on Configurable Distributed Systems (CDS '98)*, page 35. IEEE Computer Society Press, 1998.

[BJ05] Tonglaga Bao and Michael Jones. Time-efficient model checking with magnetic disk. In Nicolas Halbwachs and Lenore D. Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 526–540. Springer, 2005.

[BJC05] Thaís Vasconcelos Batista, Ackbar Joolia, and Geoff Coulson. Managing dynamic reconfiguration in component-based systems. In Ronald Morrison and Flávio Oquendo, editors, *2nd European Workshop on Software Architecture (EWSA '05)*, volume 3527 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2005.

[BJNT00] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. Regular model checking. In *12th International Conference on Computer Aided Verification (CAV'00)*, volume 1855 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[BK04] Mihir Bellare and Tadayoshi Kohno. Hash function balance and its impact on birthday attacks. In *EUROCRYPT*, volume 3027 of *Lecture Notes in Computer Science*, pages 401 – 418. Springer, 2004.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[BLM08] Roberto Bruni, Alberto Lluch Lafuente, and Ugo Montanari. Hierarchical design rewriting with Maude. In *7th International Workshop on Rewriting Logic and its Applications (WRLA '08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[BLMS01] Luís Baptista, Inês Lynce, and Joao P. Marques-Silva. Complete search restart strategies for satisfiability. In *IJCAI Workshop on Stochastic Search Algorithms (IJCAI-SSA '01)*, 2001.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[Blo83] Toby Bloom. Dynamic module replacement in a distributed programming system. Technical Report MIT/LCS/TR-303, Massachusetts Institute of Technology, 1983.

[BLP03] Gerd Behrmann, Kim Guldstrand Larsen, and Radek Pelánek. To store or not to store. In *15th International Conference on Computer Aided Verification (CAV '03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 433–445. Springer, 2003.

[BLS06] Andreas Bauer, Martin Leucker, and Christian Schallhart. Monitoring of real-time properties. In *26th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS '06)*, volume 4337 of *Lecture Notes in Computer Science*. Springer, 2006.

[BLW02] Benedikt Bollig, Martin Leucker, and Michael Weber. Local parallel model checking for the alternation-free mu-calculus. In *9th International SPIN Workshop on Model Checking of Software (SPIN '02)*, Lecture Notes in Computer Science. Springer, 2002.

[BLWG99] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: Issues and directions. *IEEE Software*, 16(5):103–111, 1999.

[BMH08] Tomáš Bureš, Michal Malohlava, and Petr Hnětynka. Using DSL for automatic generation of software connectors. In *7th International Conference on Composition-Based Software Systems (ICCBSS '08)*, pages 138–147. IEEE Computer Society Press, 2008.

[BMMC00] Philippe Boinot, Renaud Marlet, Gilles Muller, and Charles Consel. A declarative approach for designing and developing adaptive components. In *15th IEEE International Conference on Automated Software Engineering (ASE '00)*, pages 111–127. IEEE Computer Society Press, 2000.

[BMPW98] Sergey Brin, Rajeev Motwani, Lawrence Page, and Terry Winograd. What can you do with a web in your pocket. *Data Engineering Bulletin*, 21:37–47, 1998.

[BNS+05] Jaiganesh Balasubramanian, Balachandran Natarajan, Douglas C. Schmidt, Aniruddha S. Gokhale, Jeff Parsons, and Gan Deng. Middleware support for dynamic component updating. In *On the Move to Meaningful Internet Systems '05*, volume 3761 of *Lecture Notes in Computer Science*, pages 978–996. Springer, 2005.

[Boa06] RSS Advisory Board. RSS 2.0 Specification. `http://www.rssboard.org/rss-specification`, 2006.

[Box98] Don Box. *Essential COM*. Addison-Wesley, 1998.

[BP07] Phillip J. Brooke and Richard F. Paige. Exceptions in Concurrent Eiffel. *Journal of Object Technology*, 6(10):111–126, 2007.

[BR00] Thaís Vasconcelos Batista and Noemi Rodriguez. Dynamic reconfiguration of component-based applications. In *International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE '00)*, page 32. IEEE Computer Society Press, 2000.

[Bra04] Jeremy S. Bradbury. Organizing definitions and formalisms of dynamic software architectures. Technical Report 2004-477, Queen's University, 2004.

[Bro85] Rodney A. Brooks. A robust layered control system for a mobile robot. Technical report, Massachusetts Institute of Technology, 1985.

[BU97] Michael Bursell and Takanori Ugai. Comparison of autonomous mobile agent technologies. Technical report, ANSA Project Document, 1997.

[Bui08] Hai-Lam Bui. Temporale Logiken für endliche Abläufe. Fortgeschrittenenpraktikum, LMU München, 2008.

[BvVZ05] Luboš Brim, Ivana Černá, Pavlína Vařeková, and Barbora Zimmerova. Component-interaction automata as a verification-oriented component-based system specification. In *Conference on Specification and Verification of Component-based Systems (SAVCBS '05)*, volume 31 of *ACM SIGSOFT Software Engineering Notes*. ACM Press, 2005.

[BW97] Martin Büchi and Wolfgang Weck. A plea for grey-box components. Technical report, Turku Centre for Computer Science, 1997.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2):323–342, 1983.

[Car07] Denis Caromel. From theory to practice in distributed component systems. *Electronic Notes in Theoretical Computer Science*, 182:33–38, 2007.

[CB01] Geoff Coulson and Shakuntala Baichoo. Implementing the CORBA GIOP in a high-performance object request broker environment. *Distributed Computing*, 14(2):113–126, 2001.

[CBCP01] Michael Clarke, Gordon S. Blair, Geoff Coulson, and Nikos Parlavantzas. An efficient component model for the construction of adaptive middleware. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware '01)*, volume 2218 of *Lecture Notes in Computer Science*, pages 160–178. Springer-Verlag, 2001.

[CBG+04] Geoff Coulson, Gordon S. Blair, Paul Grace, Ackbar Joolia, Kevin Lee, and Jo Ueyama. A component model for building systems software. In *IASTED Software Engineering and Applications (SEA '04)*, 2004.

[CBMT96] Bernadette Charron-Bost, Friedemann Mattern, and Gerard Tel. Synchronous, asynchronous, and causally ordered communication. *Distributed Computing*, 9(4):173–191, 1996.

[CBZ+02] Fei Cao, Barrett R. Bryant, Wei Zhao, Carol C. Burt, Rajeev R. Raje, Mikhail Auguston, and Andrew M. Olson. A translation approach to component specification. In *OOPSLA '02: Companion of the 17th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 54–55. ACM Press, 2002.

[CCG+03] Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in C. In *25th International Conference on Software Engineering (ICSE)*, pages 385–395, 2003.

[CCGR99] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. NUSMV: A new symbolic model verifier. In *11th International Conference on Computer Aided Verification (CAV '99)*, pages 495–499. Springer, 1999.

[CD99] Jonathan E. Cook and Jeffrey A. Dage. Highly reliable upgrading of components. In *21st international conference on Software engineering (ICSE '99)*, pages 203–212. IEEE Computer Society Press, 1999.

[CDE+07] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[CEM03] Licia Capra, Wolfgang Emmerich, and Cecilia Mascolo. CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. *IEEE Transactions on Software Engineering*, 29:929–945, 2003.

[CES04] David Culler, Deborah Estrin, and Mani Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37(8):41–49, 2004.

[CGMP01] Manuel Aguilar Cornejo, Hubert Garavel, Radu Mateescu, and Noel De Palma. Specification and verification of a dynamic reconfiguration protocol for agent-based applications. In *3rd International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 229–244. Kluwer Academic Publishers, 2001.

[CGP00] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 2000.

[CGS+02] Shang-Wen Cheng, David Garlan, Bradley R. Schmerl, Joao Pedro Sousa, Bridget Spitnagel, and Peter Steenkiste. Using architectural style as a basis for system self-repair. In *17th IFIP World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture (WICSA '02)*, pages 45–59. Kluwer Academic Publishers, 2002.

[CH01] William T. Councill and George T. Heineman. Definition of a software component and its elements. In George T. Heineman and William T. Councill, editors, *Component-based software engineering: putting the pieces together*, pages 5–19. Addison-Wesley, 2001.

[CH03] Humberto Cervantes and Richard S. Hall. Automating service dependency management in a service-oriented component model. In *6th Workshop of the European Software Engineering Conference on Foundations of Software Engineering*, pages 379–382, 2003.

[CH04] Humberto Cervantes and Richard S. Hall. Autonomous adaptation to dynamic availability using a service-oriented component model. In *26th International Conference on Software Engineering (ICSE '04)*, pages 614–623. IEEE Computer Society Press, 2004.

[CH05] Denis Caromel and Ludovic Henrio. *A Theory of Distributed Objects*. Springer, 2005.

[Che02] Xuejun Chen. Dynamic reconfiguration of a component-based software system for vehicles. In *Proceeding of the 15th IFAC World Congress on Automatic Control*. Elsevier, 2002.

[CHM08] Antonio Cansado, Ludovic Henrio, and Eric Madelaine. Transparent first-class futures and distributed components. In *5th International Workshop on Formal Aspects of Component Software (FACS'08)*, 2008.

[CHS01] Wen-Ke Chen, Matti A. Hiltunen, and Richard D. Schlichting. Constructing adaptive software in distributed systems. In *21st International Conference on Distributed Computing Systems*, pages 635–643, 2001.

[CHY+98] P. Emerald Chung, Yennun Huang, Shalini Yajnik, Deron Liang, Joanne C. Shih, Chung-Yih Wang, , and Yi-Min Wang. DCOM and CORBA side by side, step by step, and layer by layer. *C++ Report*, 1998.

[CLG+06] Geoff Coulson, Manish Lad, Richard Gold, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis. Dynamic reconfiguration in the RUNES middleware. In *International Conference on Mobile Ad Hoc and Sensor Systems (MASS06)*. IEEE Computer Society Press, 2006.

[CM84] K. Mani Chandy and Jayadev Misra. The drinking philosophers problem. *ACM Transactions on Programming Languages and Systems*, 6(4):632–646, 1984.

[CMZ02] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. *ACM SIGPLAN Notices*, 37(11):246–261, 2002.

[Col06] Adrian Colyer. Simplifying enterprise applications with Spring 2.0 and AspectJ. *InfoQ.com*, 2006.

[Cox90] Brad J. Cox. Planning the software industrial revolution. *IEEE Software*, 7(6):25–33, 1990.

[CS02] Xuejun Chen and Martin Simons. A component framework for dynamic reconfiguration of distributed systems. In *IFIP/ACM Working Conference on Component Deployment (CD '02)*, pages 82–96. Springer, 2002.

[CS06] Thierry Coupaye and Jean-Bernard Stefani. Fractal component-based software engineering. In Mario Südholt and Charles Consel, editors, *Workshop on Object-Oriented Technology (ECOOP '06 Workshops)*, volume 4379 of *Lecture Notes in Computer Science*, pages 117–129. Springer, 2006.

[CSKO02] Angelo Corsaro, Douglas C. Schmidt, Raymond Klefstad, and Carlos O'Ryan. Virtual component: a design pattern for memory-constrained embedded applications. In *9th Annual Conference on the Pattern Languages of Programs*, 2002.

[CSPZ04] Saehoon Cheon, Chungman Seo, Sunwoo Park, and Bernard P. Zeigler. Design and implementation of distributed DEVS simulation in a Peer to Peer network system. In *ASTC'04*, pages 18–22. Society for Modeling and Simulation International, 2004.

[CTTV04] Edmund M. Clarke, Muralidhar Talupur, Tayssir Touili, and Helmut Veith. Verification by network decomposition. In *15th International Conference on Concurrency Theory (CONCUR '04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 276–291. Springer, 2004.

[dAH01] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *8th European software engineering conference (ESEC '01)*, pages 109–120. ACM Press, 2001.

[dB00] Hans de Bruin. A grey-box approach to component composition. In *1st International Symposium on Generative and Component-Based Software Engineering (GCSE '99)*, pages 195–209. Springer, 2000.

[DD98] Molisa Dianne Derk and Linda DeBrunner. Reconfiguration for fault tolerance using graph grammars. *ACM Transactions on Computer Systems*, 16(1):41–54, 1998.

[DDF+06] Simon Dobson, Spyros Denazis, Antonio Fernández, Dominique Gaïti, Erol Gelenbe, Fabio Massacci, Paddy Nixon, Fabrice Saffre, Nikita Schmidt, and Franco Zambonelli. A survey of autonomic communications. *ACM Transactions on Autonomous Adaptive Systems*, 1(2):223–259, 2006.

[DDHY92] David L. Dill, Andreas J. Drexler, Alan J. Hu, and C. Han Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design on VLSI in Computer & Processors (ICCD '92)*, pages 522–525. IEEE Computer Society Press, 1992.

[Dij82] Edsger W. Dijkstra. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*, pages 60–66. Springer, 1982.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 243–320. Elsevier, 1990.

[DK75] Frank DeRemer and Hans Kron. Programming-in-the large versus programming-in-the-small. In *International Conference on Reliable Software*, pages 114–121. ACM Press, 1975.

[DL06] Philippe David and Thomas Ledoux. Safe dynamic reconfigurations of Fractal architectures with FScript. In *5th Fractal CBSE Workshop*, 2006.

[DM04] Peter C. Dillinger and Panagiotis Manolios. Fast and accurate bitstate verification for SPIN. In *11th International SPIN Workshop on Model Checking of Software (SPIN '04)*, volume 2989 of *Lecture Notes in Computer Science*. Springer, 2004.

[DM07] Alastair F. Donaldson and Alice Miller. Symmetry reduction techniques for explicit-state model checking. In *1st International Symmetry Conference*, pages 41–45, 2007.

[DMM04] Ada Diaconescu, Adrian Mos, and John Murphy. Automatic performance management in component based software systems. In *1st International Conference on Autonomic Computing (ICAC'04)*, pages 214–221. IEEE Computer Society Press, 2004.

[DP07] Nurzhan Duzbayev and Iman Poernomo. Pre-emptive adaptation through classical control theory. In *3rd International Conference on Quality of Software Architectures (QoSA '07)*, volume 4880 of *Lecture Notes in Computer Science*, pages 164–181. Springer, 2007.

[EEPT06] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[EGLT76] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[EJ01] Donald E. Eastlake and Paul E. Jones. US secure hash algorithm 1 (SHA1). Internet informational RFC 3174, 2001.

[EMS+08] Xabier Elkorobarrutia, Mikel Muxika, Goiuri Sagardui, Frank Barbier, and Xabier Aretxandieta. A framework for statechart based component reconfiguration. *Fifth IEEE Workshop on Engineering of Autonomic and Autonomous Systems (EASE '08)*, 0:37–45, 2008.

[ESE06] Stephan Eichler, Christoph Schroth, and Jörg Eberspächer. Car-to-car communication. In *VDE-Kongress – Innovations for Europe*. VDE Verlag, 2006.

[Eur06] European Association of Software Science and Technology. The EASST newsletter. Nr. 14, http://www.easst.org/newsletter/EASSTNLDec2006.pdf, 2006.

[EVDB05] Peter Ebraert, Yves Vandewoude, Theo D'Hondt, and Yolande Berbers. Pitfalls in unanticipated dynamic software evolution. In *Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE '05)*, pages 41–49, 2005.

[EW07] Ahmed Elkhodary and Jon Whittle. A survey of approaches to adaptive application security. In *International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS '07)*, page 16. IEEE Computer Society Press, 2007.

[Fav91] John Favaro. What price reusability?: a case study. *ACM SIGAda Ada Letters archive Volume XI*, XI(3):115–124, 1991.

[Fer02] Giovanna Ferrari. Applying feedback control to QoS management. In *7th CaberNet Radicals Workshop*, 2002.

[Fon93] Leonard N. Foner. What's an agent anyway? - a sociological case study. FTP Report - MIT Media Lab, 1993.

[Fos95] Ian Foster. *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. In *21th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS '01)*, volume 2245 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2001.

[FZ06] Sonia Fagorzi and Elena Zucca. A calculus for reconfiguration. In *First International Workshop on Developments in Computational Models (DCM 2005)*, volume 135 of *Electronic Notes in Theoretical Computer Science*, pages 49–59, 2006.

[FZ07] Sonia Fagorzi and Elena Zucca. A calculus of components with dynamic typechecking. *Electronic Notes in Theoretical Computer Science*, 182:73–90, 2007.

[GA95] Jean–loup Gailly and Mark Adler. zlib data compression library. http://www.zlib.net, 1995.

[GA04] Miguel A. Goulão and Fernando M. Brito Abreu. Software components evaluation: an overview. In *5ª Conferência da APSI (CAPSI '04)*. Associação Portuguesa de Sistemas de Informação, 2004.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch or why it's hard to build systems out of existing parts. In *17th International Conference on Software Engineering (ICSE '95)*, pages 179–185, 1995.

[Gat98] Erann Gat. On three-layer architectures. In R. Peter Bonnasso and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots*, pages 195–210. AAAI Press, 1998.

[GBS03] Paul Grace, Gordon S. Blair, and Sam Samuel. ReMMoC: A reflective middleware to support mobile client interoperability. In *International Symposium of Distributed Objects and Applications (DOA '03)*, 2003.

[GCB+06] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, and Danny Hughes. Dynamic reconfiguration in sensor middleware. In *International Workshop on Middleware for Sensor Networks (MidSens '06)*, pages 1–6. ACM Press, 2006.

[GCH+04] David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self adaptation with reusable infrastructure. *IEEE Computer*, 37(10), 2004.

[GG03] Nasib S. Gill and P. S. Grover. Component-based measurement: few useful guidelines. *ACM SIGSOFT Software Engineering Notes*, 28(6):4–4, 2003.

[GH06] Holger Giese and Martin Hirsch. Modular verification of safe online-reconfiguration for proactive components in Mechatronic UML. In *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 67–78. Springer, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GJ93] Deepak Gupta and Pankaj Jalote. On-line software version change using state transfer between processes. *Software - Practice and Experience*, 23(9):949–964, 1993.

[GKC99] Dimitra Giannakopoulou, Jeff Kramer, and Shing Chi Cheung. Behaviour analysis of distributed systems using the Tracta approach. *Automated Software Engineering*, 6(1):7–35, 1999.

[GMK02] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *First workshop on Self-healing Systems (WOSS '02)*, pages 33–38. ACM Press, 2002.

[GMW00] David Garlan, Robert T. Monroe, and David Wile. Acme: Architectural description of component-based systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[Gon01] Li Gong. JXTA: A network programming environment. *IEEE Internet Computing*, 5(3):88–95, 2001.

[GP95] Jan Friso Groote and Alban Ponse. The syntax and semantics of μCRL. In *Algebra of Communicating Processes '94*, Workshops in Computing Series, pages 26–62. Springer, 1995.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–39. World Scientific Publishing Company, 1993.

[GS02] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *1st Workshop on Self-healing Systems (WOSS '02)*, pages 27–32. ACM Press, 2002.

[GS05] Radu Grosu and Scott A. Smolka. Monte carlo model checking. In *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2005.

[GSPW98] Ashvin Goel, David Steere, Calton Pu, and Jonathan Walpole. SWiFT: A feedback control and dynamic reconfiguration toolkit. Technical report, 2nd USENIX Windows NT Symposium, 1998.

[GSRU07] Debanjan Ghosh, Raj Sharman, H. Raghav Rao, and Shambhu Upadhyaya. Self-healing systems – survey and synthesis. *Decision Support Systems*, 42(4):2164–2185, 2007. Decision Support Systems in Emerging Economies.

[Hac04] Florian Hacklinger. Java/A - taking components into Java. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE '04)*, pages 163–168. ISCA, 2004.

[Ham04] Moritz Hammer. Linear weak alternating automata and the model checking problem. Diplomarbeit, LMU München, 2004.

[Han01] Hans Hansson. SAVE – component based design of safety critical vehicular systems. Proposal submitted to SSF "Ramanslag för forskning inom informationsteknik 2001", 2001. shortened version at http://www.artes.uu.se/++/SAVE/SAVE2-final-brief.pdf.

[HB07] Gerard J. Holzmann and Dragan Bosnacki. The design of a multicore extension of the SPIN model checker. *IEEE Transactions on Software Engineering*, 33(10):659–674, 2007.

[HBHH07] Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Formal Methods in Computer Aided Design (FMCAD '07)*, pages 27–34. IEEE Computer Society Press, 2007.

[HBKW01] Rolf Hennicker, Hubert Baumeister, Alexander Knapp, and Martin Wirsing. Specifying component invariants with OCL. In *GI Jahrestagung (1)*, pages 600–607, 2001.

[HBSBA03] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-based scheduling to improve web performance. *ACM Transactions on Computer Systems*, 21(2):207–233, 2003.

[HC03] Richard S. Hall and Humberto Cervantes. Gravity: supporting dynamically available services in client-side applications. *ACM SIGSOFT Software Engineering Notes*, 28(5):379–382, 2003.

[Hei99] Ernst Heinz. *Scalable Search in Computer Chess*. PhD thesis, IPD Tichy, University of Karlsruhe, Germany, 1999.

[Hel94] Tobias Helbig. Development and control of distributed multimedia applications. In Paul Kühn, editor, *4th Open Workshop on High-Speed Networks*, pages 208–213, 1994.

[Hen96] Thomas A. Henzinger. The theory of hybrid automata. In *11th Annual Symposium on Logic in Computer Science (LICS '96)*, pages 278–292. IEEE Computer Society Press, 1996.

[Hep02] Thomas A. Heppenheimer. *Development of the Space Shuttle 1972 – 1981*, volume 2 of *History of the Space Shuttle*. Smithsonian Institution Press, 2002.

[Her90] Maurice Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Transactions on Database Systems*, 15(1):96–124, 1990.

[HFS05] Svein Hallsteinsen, Jacqueline Floch, and Erlend Stav. A middleware centric approach to building self-adapting systems. In *4th International Workshop on Software Engineering and Middleware (SEM '04)*, volume 3437 of *Lecture Notes in Computer Science*. Springer, 2005.

[HG98] Gísli Hjálmtýsson and Robert Gray. Dynamic C++ classes: a lightweight mechanism to update code in a running program. In *USENIX Annual Technical Conference (ATEC '98)*, pages 6–6. USENIX Association, 1998.

[HGP+06] Danny Hughes, Phil Greenwood, Barry Porter, Paul Grace, Geoff Coulson, Gordon Blair, Francois Taiani, Florian Pappenberger, Paul Smith, and Keith Beven. Using grid technologies to optimise a wireless sensor network for flood management. In *4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 389–390. ACM Press, 2006.

[HJK08] Rolf Hennicker, Stephan Janisch, and Alexander Knapp. On the observable behaviour of composite components. In Carlos Canal and Corina Pasareanu, editors, *5th International Workshop on Formal Aspects of Component Software (FACS '08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with Blast. In *10th International Workshop on Model Checking of Software (SPIN '03)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[HK08] Moritz Hammer and Alexander Knapp. Correct execution of reconfiguration for stateful components. In *5th International Workshop on Formal Aspects of Component Software (FACS'08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[HKM05] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '05)*, volume 3440 of *Lecture Notes in Computer Science*, pages 191–205. Springer, 2005.

[HKMR08] Moritz Hammer, Bernhard Kempter, Florian Mangold, and Harald Roelle. Skalierbare Performanzanalyse durch Prolongation. In Korbinian Herrmann and Bernd Bruegge, editors, *Software Engineering (SE 2008)*, volume 121 of *Lecture Notes in Informatics*, pages 127–139. Gesellschaft für Informatik, 2008.

[HKW+08] Sebastian Herold, Holger Klus, Yannick Welsch, Constanze Deiters, Andreas Rausch, Ralf Reussner, Klaus Krogmann, Heiko Koziolek, Raffaela Mirandola, Benjamin Hummel, Michael Meisinger, and Christian Pfaller. CoCoME - The Common Component Modeling Example. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 16–53. Springer, 2008.

[HL82] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM ACM Transactions on Programming Languages and Systems*, 4(4):527–551, 1982.

[HM08] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Computing Surveys*, 40(3):1–28, 2008.

[HMW08] Matthias Hölzl, Max Meier, and Martin Wirsing. Which soft constraints do you prefer? In Grigore Rosu, editor, *7th International Workshop on Rewriting Logic and its Applications (WRLA '08)*, Electronic Notes in Theoretical Computer Science. Elsevier, 2008.

[Hoa83] Charles A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 26(1):100–106, 1983.

[Hof93] Christine R. Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, Computer Science Department, University of Maryland, 1993.

[Hol97] Gerard J. Holzmann. State compression in SPIN: Recursive indexing and compression training runs. In *3th International SPIN Workshop*, 1997.

[Hol98] Gerard J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3):289–307, 1998.

[Hol03] Gerard J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[Hop00] Jon Hopkins. Component primer. *Communications of the ACM*, 43(10):27–30, 2000.

[HP83] Terry D. Humphrey and Charles R. Price. Flight software fault tolerance via the backup flight system. In *Space Shuttle Technical Conference*, volume 2342 of *NASA Conference Publication*, pages 35–37. National Aeronautics and Space Administration, 1983.

[HP93] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report UMIACS-TR-93-78, Univ. of Maryland Advanced Institute for Computer Studies, 1993.

[HP94] Gerard .J. Holzmann and Doron Peled. An improvement in formal verification. In *7th International Conference on Formal Description Techniques (FORTE '94)*, pages 197–211. Chapman & Hall, 1994.

[HP99] Gerard J. Holzmann and Anuj Puri. A minimized automaton representation of reachable states. *Software Tools for Technology Transfer*, 2(3):270–278, 1999.

[HP06] Petr Hnětynka and František Plášil. Dynamic reconfiguration and access to services in hierarchical component models. In Ian Gorton, George T. Heineman, Ivica Crnkovic, Heinz W. Schmidt, Judith A. Stafford, Clemens A. Szyperski, and Kurt C. Wallnau, editors, *Component-Based Software Engineering, 9th International Symposium, CBSE 2006, Västerås*, volume 4063 of *Lecture Notes in Computer Science*, pages 352–359. Springer, 2006.

[HSB98] Yanbo Han, Amit Sheth, and Christoph Bussler. A taxonomy of adaptive workflow management. In *Workshop Towards Adaptive Workflow Systems (CSCW '98)*, 1998.

[HTC98] Mats P.E. Heimdahl, Jeffrey M. Thompson, and Barbara J. Czerny. Specification and analysis of intercomponent communication. *Computer*, 31(4):47–54, 1998.

[HW04] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *26th International Conference on Software Engineering (ICSE '04)*, pages 594–603. IEEE Computer Society Press, 2004.

[HW06] Moritz Hammer and Michael Weber. "To store or not to store" reloaded: Reclaiming memory on demand. In Lubos Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Application and Technology (FMICS'2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2006.

[HZ04] Xiaolin Hu and Bernard P. Zeigler. Model continuity to support software development for distributed robotic systems: a team formation example. *Journal of Intelligent & Robotic Systems, Theory & Application*, 39(1):71–87, 2004.

[HZM05] Xiaolin Hu, Bernard P. Zeigler, and Saurabh Mittal. Variable structure in DEVS component-based modeling and simulation. *Simulation*, 81(2):91–102, 2005.

[IFMW08] Florian Irmert, Thomas Fischer, and Klaus Meyer-Wegener. Runtime adaptation in a service-oriented component model. In Marin Litoiu and Holger Giese, editors, *SEAMS 2008: Software Engineering for Adaptive and Self-Managing Systems*, pages 97–104. ACM Press, 2008.

[IKKW01] Torsten Illmann, Tilman Krueger, Frank Kargl, and Michael Weber. Transparent migration of mobile agents using the Java platform debugger architecture. In Gian Pietro Picco, editor, *Mobile Agents*, volume 2240 of *Lecture Notes in Computer Science*, pages 198–212. Springer, 2001.

[IKWK00] Torsten Illmann, Frank Kargl, Michael Weber, and Tilman Krüger. Migration of mobile agents in Java: Problems, classification and solutions. In *International ICSC Symposium on Multi-Agents and Mobile Agents in Virtual Organizationa and E-Commerce (MAMA '00)*. Springer, 2000.

[IW95] Paola Inverardi and Alexander L. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, 1995.

[Jac02] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[JAF+06] Shawn R. Jeffery, Gustavo Alonso, Michael J. Franklin, Wei Hong, and Jennifer Widom. Declarative support for sensor data cleaning. In Kenneth P. Fishkin, Bernt Schiele, Paddy Nixon, and Aaron J. Quigley, editors, *Pervasive*, volume 3968 of *Lecture Notes in Computer Science*, pages 83–100. Springer, 2006.

[Jah07] Isabell Jahnich. Evaluation of existing technologies. Technical Report D1.1B, Dynamically Self-Configuring Automotive Systems (DySCAS) – Project no. FP6-IST-2006-034904, 2007.

[JDMV04] Nico Janssens, Lieven Desmet, Sam Michiels, and Pierre Verbaeten. NeCoMan: middleware for safe distributed service deployment in programmable networks. In *3rd*

Workshop on Adaptive and Reflective Middleware (ARM '04), pages 256–261. ACM Press, 2004.

[Jen97] Bob Jenkins. Hash functions. "Algorithm Alley", Dr. Dobb's Journal, 1997.

[JTSJ07] Nico Janssens, Eddy Truyen, Frans Sanen, and Wouter Joosen. Adding dynamic reconfiguration support to JBoss AOP. In *1st Workshop on Middleware-Application Interaction (MAI '07)*, pages 1–8. ACM Press, 2007.

[JV00] Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '00)*, pages 14–25. ACM Press, 2000.

[KB04] Abdelmadjid Ketfi and Noureddine Belkhatir. Open framework for the dynamic reconfiguration of component-based sotware. In Hamid R. Arabnia and Hassan Reza, editors, *International Conference on Software Engineering Research and Practice (SERP '04)*, pages 948–951. CSREA Press, 2004.

[KBC02] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre-Yves Cunin. Adapting applications on the fly. In *17th IEEE International Conference on Automated Software Engineering (ASE '02)*, page 313. IEEE Computer Society Press, 2002.

[KBHR08] Heiko Koziolek, Steffen Becker, Jens Happe, and Ralf Reussner. Life-cycle aware modelling of software components. In *11th International Symposium on Component-Based Software Engineering (CBSE '08)*, volume 5282 of *Lecture Notes in Computer Science*, pages 278–285. Springer, 2008.

[KByC02] Abdelmadjid Ketfi, Noureddine Belkhatir, and Pierre yves Cunin. Dynamic updating of component-based applications. In *International Conference on Software Engineering Research and Practice (SERP'02)*. IEEE Computer Society Press, 2002.

[KC03] Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[KD99] Justin Kruger and David Dunning. Unskilled and unaware of it: How difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of Personality and Social Psychology*, 77(6):1121–1134, 1999.

[KGK+02] Gail Kaiser, Phil Gross, Gaurav Kc, Janak Parekh, and Giuseppe Valetto. An approach to autonomizing legacy systems. In *Workshop on Self-Healing, Adaptive and Self-Managed Systems (SHAMAN '02)*, 2002.

[KHP+03] Bithika Khargharia, Salim Hariri, Manish Parashar, Lewis Ntaimo, and Byoung Uk Kim. vGrid: A framework for building autonomic applications. In *International Workshop on Challenges for Large Applications in Distributed Environments (CLADE 2003*. IEEE Computer Society Press, 2003.

[KHW+01a] John C. Knight, Dennis Heimbigner, Alexander L. Wolf, Antonio Carzaniga, Jonathan Hill, Premkumar Devanbu, and Michael Gertz. The Willow architecture: Comprehensive survivability for large-scale distributed applications. Technical Report CU-CS-926-01, Department of Computer Science, University of Colorado, 2001.

[KHW+01b] John C. Knight, Dennis Heimbigner, Alexander L. Wolf, Antonio Carzaniga, and Jonathan C. Hill. The Willow survivability architecture (isw '01). In *4th Information Survivability Workshop*. IEEE Computer Society Press, 2001.

[KJH+08] Alexander Knapp, Stephan Janisch, Rolf Hennicker, Allan Clark, Stephen Gilmore, Florian Hacklinger, Hubert Baumeister, and Martin Wirsing. Modelling the CoCoME with the Java/A component model. In Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors, *The Common Component Modeling Example: Comparing Software Component Models*, volume 5153 of *Lecture Notes in Computer Science*, pages 207–237. Springer, 2008.

[KK98] George Karypis and Vipin Kumar. Multilevel algorithms for multi-constraint graph partitioning. In *ACM/IEEE Conference on Supercomputing (Supercomputing '98)*, pages 1–13. IEEE Computer Society Press, 1998.

[KL86] John C. Knight and Nancy G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, 1986.

[KL00] Moataz Kamel and Stefan Leue. Formalization and validation of the General Inter-ORB Protocol (GIOP) using PROMELA and SPIN. *STTT*, 2(4):394–409, 2000.

[KLL+02] Gregor J. Kiczales, John O. Lamping, Cristina V. Lopes, James J. Hugunin, Erik A. Hilsdale, and Chandrasekhar Boyapati. Aspect-oriented programming. United States Patent 6,467,086, 2002.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet

Akşit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[KM90] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[KM91] Hirofumi Katsuno and Alberto Mendelzon. On the difference between updating a knowledge base and revising it. In James F. Allen, Richard Fikes, and Erik Sandewall, editors, *KR'91: Principles of Knowledge Representation and Reasoning*, pages 387–394. Morgan Kaufmann, 1991.

[KM02] Alexander Knapp and Stephan Merz. Model checking and code generation for UML state machines and collaborations. In G. Schellhorn and W. Reif, editors, *5th Workshop on Tools for System Design and Verification (FM-TOOLS '02)*, Report 2002-11. Institut für Informatik, Universität Augsburg, 2002.

[KM07] Jeff Kramer and Jeff Magee. Self-managed systems: an architectural challenge. In *Future of Software Engineering (FOSE '07)*, pages 259–268. IEEE Computer Society Press, 2007.

[KM08a] Jeff Kramer and Jeff Magee. Towards robust self-managed systems. *Progress in Informatics*, 5:1–4, 2008.

[KM08b] Fred Kröger and Stephan Merz. *Temporal Logic and State Systems*. Springer, 2008.

[KMW03] Richard Krutisch, Philipp Meier, and Martin Wirsing. The AgentComponent approach, combining agents, and components. In Michael Schillo, Matthias Klusch, Jörg P. Müller, and Huaglory Tianfield, editors, *1st German Conference on Multiagent System Technologies (MATES '03)*, volume 2831 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2003.

[Kni98] Günter Kniesel. Type-safe delegation for dynamic component adaptation. In *Object-Oriented Technology: ECOOP '98 Workshop Reader*, number 1543 in Lecture Notes in Computer Science, pages 136–137. Springer, 1998.

[Kof07] Jan Kofroň. Checking software component behavior using behavior protocols and Spin. In Yookun Cho, Roger L. Wainwright, Hisham Haddad, Sung Y. Shin, and Yong Wan Koo, editors, *ACM Symposium on Applied Computing (SAC '07)*, pages 1513–1517. ACM Press, 2007.

[Kol07] Jan Kolfroň. *Behavior Protocols Extensions*. PhD thesis, Department of Software Engineering, Faculty of Mathematics and Physics, Charles University in Prague, 2007.

[KPRS06] Yonit Kesten, Amir Pnueli, Li-on Raviv, and Elad Shahar. Model checking with strong fairness. *Formal Methods in System Design*, 28(1):57–84, 2006.

[KRL+00] Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in Lecture Notes in Computer Science, pages 121–143. Springer, 2000.

[KS99] Reino Kurki-Suonio. Component and interface refinement in closed-system specifications. In *Wold Congress on Formal Methods in the Development of Computing Systems (FM '99)*, pages 134–154. Springer, 1999.

[Lad00] Robert Laddaga. Active software. In Paul Robertson, Howard E. Shrobe, and Robert Laddaga, editors, *1st International Workshop on Self-Adaptive Software (IWSAS '00)*, volume 1936 of *Lecture Notes in Computer Science*, pages 11–26. Springer, 2000.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[Lam04] Butler Lampson. Software components: Only the giants survive. In *Computer Systems : Theory, Technology, and Applications*, chapter 20, pages 137–146. Springer, 2004.

[Lau06] Kung-Kiu Lau. Software component models. In *28th International Conference on Software Engineering (ICSE '06)*, pages 1081–1082. ACM Press, 2006.

[LB03] Johnny Li-Chang Lo and Judith Bishop. Component-based interchangeable cryptographic architecture for securing wireless connectivity in Java™ applications. In *Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT '03)*, pages 301–307. South African Institute for Computer Scientists and Information Technologists, 2003.

[Lev44] Kenneth Levenberg. A method for the solution of certain problems in least squares. *The Quarterly of Applied Mathematics*, 2:164–168, 1944.

[LEW05] Kung-Kiu Lau, Perla Velasco Elizondo, and Zheng Wang. Exogenous connectors for software components. In *8th International SIGSOFT Symposium on Component-based Software Engineering*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106. Springer, 2005.

[Lim93] Alvin Sek See Lim. *A state machine approach to reliable and dynamically reconfigurable distributed systems.* PhD thesis, University of Wisconsin at Madison, 1993.

[Lim96] Alvin Sek See Lim. Abstraction and composition techniques for reconfiguration of large-scale complex applications. In *3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, page 186. IEEE Computer Society Press, 1996.

[Lin01] Jürgen Lind. Relating agent technology and component models, 2001.

[Lit00] Radu Litiu. *Providing Flexibility in Distributed Applications Using a Mobile Component Framework.* PhD thesis, University of Michigan, Electrical Engineering and Computer Science, 2000.

[LLC07] Marc Léger, Thomas Ledoux, and Thierry Coupaye. Reliable dynamic reconfigurations in the Fractal component model. In *6th International Workshop on Adaptive and Reflective Middleware (ARM '07)*, pages 1–6. ACM Press, 2007.

[LMH07] Marc Lohmann, Leonardo Mariani, and Reiko Heckel. A model-driven approach to discovery, testing, and monitoring of web services. In Luciano Baresi and Elisabetta Di Nitto, editors, *Test and Analysis of Web Services*, pages 173 – 204. Springer, 2007.

[LP00] Radu Litiu and Atul Prakash. Developing adaptive groupware applications using a mobile component framework. In *ACM Conference on Computer Supported Cooperative Work (CSCW 2000)*. ACM Press, 2000.

[LP06] Hua Liu and Manish Parashar. Accord: A programming framework for autonomic applications. *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, 36(3):341–352, 2006.

[LPPU94] Michael R. Lowry, Andrew Philpot, Thomas Pressburger, and Ian Underwood. AMPHION: Automatic programming for scientific subroutine libraries. In *8th International Symposium on Methodologies for Intelligent Systems (ISMIS '94)*, pages 326–335. Springer, 1994.

[LS96] R. Greg Lavender and Douglas C. Schmidt. *Active object: an object behavioral pattern for concurrent programming*, volume 2, chapter Pattern Languages of Program Design, pages 483–499. Addison-Wesley, 1996.

[LS05] Rafael Fernandes Lopes and Francisco José Da Silva E Silva. Migration transparency in a mobile agent based computational grid. In *5th WSEAS International Conference on Simulation, Modeling and Optimization*, pages 31–36. World Scientific Publishing Company, 2005.

[LSW03] Martin Leucker, Rafal Somla, and Michael Weber. Parallel model checking for LTL, CTL* and L^2_μ. In Lubos Brim and Orna Grumberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LVB04] Mihai Lazarescu, Svetha Venkatesh, and Hung Hai Bui. Using multiple windows to track concept drift. *Intelligent Data Analysis*, 8(1):29–59, 2004.

[LW05a] Kung-Kiu Lau and Zheng Wang. A survey of software component models. Technical Report CSPP-30, School of Computer Science, University of Manchester, 2005.

[LW05b] Kung-Kiu Lau and Zheng Wang. A taxonomy of software component models. In *31st EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA '05)*, pages 88–95. IEEE Computer Society Press, 2005.

[LW06] Kung-Kiu Lau and Zheng Wang. A survey of software component models (second edition). Technical Report CSPP-38, School of Computer Science, The University of Manchester, 2006.

[LW07] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[Mar63] Donald Marquardt. An algorithm for least-squares estimation of nonlinear parameters. *SIAM Journal of Applied Mathematics*, 11(2):431–441, 1963.

[Mar94] Assaf Marron. Method of operating a data processing system having a dynamic software update facility. United States Patent 5359730, 1994.

[MBC03] Rui Moreira, Gordon Blair, and Eurico Carrapatoso. FORMAware: Framework of reflective components for managing architecture adaptation. In *Software Engineering and Middleware*, volume 2596 of *Lecture Notes in Computer Science*, pages 115–129. Springer, 2003.

[MBC04] Rui S. Moreira, Gordon S. Blair, and Eurico Carrapatoso. Supporting adaptable distributed systems with FORMAware. In *24th International Conference on Distributed Computing Systems Workshops (ICDCSW'04)*, pages 320–325. IEEE Computer Society Press, 2004.

[McI69] Douglas McIlroy. Mass produced software components. In *Report of a Conference Sponsored by the NATO Science Committee*, pages 88–98. NATO Science Commitee, 1969.

[MDK93] Jeff Magee, Naranker Dulay, and Jeffrey Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73–82, 1993.

[MDK94] Jeff Magee, Naranker Dulay, and Jeff Kramer. Regis: A constructive development environment for distributed programs. *IOP/IEE/BCS Distributed Systems Engineering*, 1(5):304–312, 1994.

[Med96] Nenad Medvidovic. ADLs and dynamic architecture changes. In *2nd International Software Architecture Workshop (ISAW '96)*, pages 24–27. ACM Press, 1996.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Professional Technical Reference. Prentice Hall, 1997.

[Mey00] Bertrand Meyer. What to compose. *Software Development*, 8(3):59–75, 2000.

[MG04] Arun Mukhija and Martin Glinz. A framework for dynamically adaptive applications in a self-organized mobile network environment. In *24th International Conference on Distributed Computing Systems Workshops (ICDCSW '04)*, pages 368–374. IEEE Computer Society Press, 2004.

[MG05a] Arun Mukhija and Martin Glinz. The CASA approach to autonomic applications. In *5th IEEE Workshop on Applications and Services in Wireless Networks (ASWN '05)*, pages 173–182. IEEE Computer Society Press, 2005.

[MG05b] Arun Mukhija and Martin Glinz. Runtime adaptation of applications through dynamic recomposition of components. In Michael Beigl and Paul Lukowicz, editors, *ARCS*, volume 3432 of *Lecture Notes in Computer Science*, pages 124–138. Springer, 2005.

[MGK96] Kaveh Moazami-Goudarzi and Jeff Kramer. Maintaining node consistency in the face of dynamic change. In *3rd International Conference on Configurable Distributed Systems (ICCDS '96)*, page 62. IEEE Computer Society Press, 1996.

[MHW04] Bruce MacDonald, Barry Po-Sheng Hsieh, and Ian Warren. Design for dynamic reconfiguration of robot software. In *2nd International Conference on Autonomous Robots and Agents (ICARA '04)*, pages 19–24. IEEE Computer Society Press, 2004.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[Mil93] Robin Milner. The polyadic pi-calculus: A tutorial. In F. L. Hamer, W. Brauer, and H. Schwichtenberg, editors, *Logic and Algebra of Specification*, pages 203–246. Springer, 1993.

[Mil99] Robin Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.

[MKSD90] Jeff Magee, Jeff Kramer, Morris Sloman, and Naranker Dulay. An overview of the REX software architecture. In *2nd IEEE Computer Society Workshop on Future Trends of Distributed Computing Systems*. IEEE Computer Society Press, 1990.

[MKSD92] Jeff Magee, Jeff Kramer, Morris Sloman, and Naranker Dulay. Configuring object-based distributed programs in REX. *Software Engineering Journal*, 7(2):139–149, 1992.

[MMST02] Louise E. Moser, P. M. Melliar-Smith, and L. A. Tewksbury. Online upgrades become standard. *compsac*, 00:982, 2002.

[MNCK98] Scott Mitchell, Hani Naguib, George Coulouris, and Tim Kindberg. Dynamically reconfiguring multimedia components: a model-based approach. In *8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, pages 40–47. ACM Press, 1998.

[MNCK99] Scott Mitchell, Hani Naguib, George Coulouris, and Tim Kindberg. A QoS support framework for dynamically reconfigurable multimedia applications. In *International Working Conference on Distributed Applications and Interoperable Systems*, pages 17–30. Kluwer Academic Publishers, 1999.

[Mon01] Robert T. Monroe. Capturing software architecture design expertise with Armani. Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University, 2001.

[MS89] Alberto Marchetti-Spaccamela. On the estimate of the size of a directed graph. In *Graph-Theoretic Concepts in Computer Science*, volume 344 of *Lecture Notes in Computer Science*. Springer, 1989.

[MSBC00] Scott Mitchell, Mark D. Spiteri, John Bates, and George Coulouris. Context-aware multimedia computing in the intelligent hospital. In *9th ACM SIGOPS European Workshop*, pages 13–18. ACM Press, 2000.

[MSKC04a] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. Composing adaptive software. *Computer*, 37(7):56–64, 2004.

[MSKC04b] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, 2004.

[MSTS99] Tomi Männistö, Timo Soininen, Juha Tiihonen, and Reijo Sulonen. Framework and conceptual model for reconfiguration. Technical report, AAAI Workshop on Configuration, AAAI Press, 1999.

[MT98] Lynette I. Millett and Tim Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *4th International SPIN Workshop (SPIN '98)*, pages 75–83, 1998.

[MT00] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000.

[Mue97] Max Muehlhaeuser. Special issues in object-oriented programming. In *Workshop Reader of the 10th European Conference on Object-Oriented Programming ECOOP'96, Linz*. dpunkt.verlag, 1997.

[MZP+05] Cecilia Mascolo, Stefanos Zachariadis, Gian Pietro Picco, Paolo Costa, Gordon Blair, Nelly Bencomo, Geoff Coulson, Paul Okanda, and Thirunavukkarasu Sivaharan. Runes middleware architecture. Technical report, Deliverable D5.2.1 of FP6 IP "RUNES", 2005.

[NA05] Angela Nicoara and Gustavo Alonso. Dynamic AOP with PROSE. In Jaelson Castro and Ernest Teniente, editors, *17th International Conference on Advanced Information Systems Engineering (CAiSE '05)*, pages 125–138. FEUP Edições, 2005.

[NAM03] Piotr Nienaltowski, Volkan Arslan, and Bertrand Meyer. Concurrent object-oriented programming on .NET. *IEE Proceedings – Software*, 150(5):308–314, 2003.

[NFH+03] Mikael Nolin, Johan Fredriksson, Jerker Hammarberg, Joel Huselius, John Håkansson, Annika Karlsson, Ola Larses, , Goran Mustapic, Anders Möller, Thomas Nolte, Jonas Norberg, Dag Nyström, Aleksandra Tesanovic, and Mikael Åkerholm. Component based software engineering for embedded systems – a literature survey. Technical Report ISSN 1404-3041 ISRN MDH-MRTC-102/2003-1-SE, Mälardalen University, 2003.

[NFM07] Marlon Núñez, Raúl Fidalgo, and Rafael Morales. Learning in environments with unknown dynamics: Towards more robust concept learners. *The Journal of Machine Learning Research*, 8:2595–2628, 2007.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-oriented software development. *Communications of the ACM*, 35(9):160–165, 1992.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *ACM Conference on Programming Language Design and Implementation (PLDI '06)*, pages 72–83. ACM Press, 2006.

[Nil05] Marcus Nilsson. *Regular Model Checking*. PhD thesis, Department of Information Technology, Uppsala University, 2005.

[NK08] Joe Niski and Kirk Knoernschild. Spring: A container-agnostic Java platform. Application Platform Strategies, In-Depth Research Overview. burton Group, 2008.

[NM06] Piotr Nienaltowski and Bertrand Meyer. Contracts for concurrency. In *First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, 2006.

[Now07] Cornelius Nowald. Model checking of UML state machines – a case study of the TWIN® elevator system. Bachelor thesis, Universität Augsburg, 2007.

[NS00] Badri Nath and Pradeep Sudame. Gathercast: The design and implementation of a programmable aggregation mechanism for the internet. In *IEEE International Conference on Computer Communications and Networks (ICCCN '00)*, pages 206–213. IEEE Computer Society Press, 2000.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In Jeanne Ferrante and Kathryn S. McKinley, editors, *PLDI*, pages 89–100. ACM Press, 2007.

[OK99] Marcus Oestreicher and Ksheerabdhi Krishna. Object lifetimes in Java card. In *USENIX Workshop on Smartcard Technology (WOST '99)*, pages 15–25. USENIX Association, 1999.

[OLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala component model for consumer electronics software. *Computer*, 33(3):78–85, 2000.

[OMG06a] Object Management Group. CORBA component model specification, v4.0. Online version at http://www.omg.org/docs/formal/06-04-01.pdf, 2006.

[OMG06b] Object Management Group. Object constraint language specification, version 2.0. http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf, 2006.

[OMG08] Object Management Group. Common Object Request Broker Architecture (CORBA/IIOP), version 3.1. Online version at http://www.omg.org/spec/CORBA/3.1/, 2008.

[Omm98] Rob C. van Ommering. Koala, a component model for consumer electronics product software. In *2nd International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, pages 76–86. Springer, 1998.

[OV06] Walter Oberschelp and Gottfried Vossen. *Rechneraufbau und Rechnerstrukturen*. R. Oldenbourg Verlag, 10. Auflage 2006.

[Pan68] Richard J. Pankhurst. Operating systems: Program overlay techniques. *Communication of the ACM*, 11(2):119–125, 1968.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[Par72] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par07] David Lorge Parnas. Stop the numbers game. *Communications of the ACM*, 50(11):19–21, 2007.

[PBJ97] František Plášil, Dušan Bálek, and Radovan Janeček. DCUP: Dynamic component updating in Java/CORBA environment. Technical Report 97/10, Department of SW Engineering, Charles University, Prague, 1997.

[PBJ98] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: Architecture for component trading and dynamic updating. In *International Conference on Configurable Distributed Systems (ICCDS '98)*, pages 43–52. IEEE Computer Society Press, 1998.

[PBL05] Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering. Foundations, Principles, and Techniques*. Springer, 2005.

[PCDJ08] Gilles Perrouin, Franck Chauvel, Julien DeAntoni, and Jean-Marc Jézéquel. Modeling the variability space of self-adaptive applications. In *2nd Dynamic Software Product Lines Workshop (DSPL '08)*. IEEE Computer Society Press, 2008.

[Pet81] Gary L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3):115–116, 1981.

[PITZ02] Giuseppe Della Penna, Benedetto Intrigila, Enrico Tronci, and Marisa Venturini Zilli. Exploiting transition locality in the disk based Murφ verifier. In *4th International Conference on Formal Methods in Computer-Aided Design (FMCAD '02)*, volume 2517 of *Lecture Notes in Computer Science*, pages 202–219. Springer, 2002.

[PL96] Thomas Pressburger and Michael Lowry. Automating software reuse with Amphion. In *NASA Workshop on Software Reuse*, 1996.

[PLB01] Noel De Palma, Philippe Laumay, and Luc Bellissard. Ensuring dynamic reconfiguration consistency. In *6th International Workshop on Component-Oriented Programming (WCOP '01)*, volume 2323 of *Lecture Notes in Computer Science*. Springer, 2001.

[PLL+05] Manish Parashar, Zhen Li, Hua Liu, Vincent Matossian, and Cristina Schmidt. Enabling autonomic grid applications: Requirements, models and infrastructure. In *Self-star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 273–290. Springer, 2005.

[PMSD07] Juraj Polakovic, Sebastien Mazare, Jean-Bernard Stefani, and Pierre-Charles David. Experience with safe dynamic reconfigurations in component-based embedded systems. In *Component-Based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 242–257. Springer, 2007.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *18th IEEE Symposium Foundations of Computer Science (FOCS '77)*, pages 46–57. IEEE Computer Society Press, 1977.

[PPK06] Pavel Parízek, František Plášil, and Jan Kofroň. Model checking of software components: Combining Java PathFinder and behavior protocol model checker. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 '06)*, pages 133–141. IEEE Computer Society Press, 2006.

[Pra07] Philipp Pracht. Multi-Core Model Checking mit Komponententechnologie. Diplomarbeit, LMU München, 2007.

[Pre07] Susanne Cech Previtali. Dynamic updates: another middleware service? In *1st Workshop on Middleware-Application Interaction (MAI '07)*, pages 49–54. ACM Press, 2007.

[PRRL04] C.K. Prasad, Rajesh Ramchandani, Gopinath Rao, and Kim Levesque. Creating a debugging and profiling agent with JVMTI. *Sun Developer Network*, 2004. http://java.sun.com/developer/technicalArticles/Programming/jvmti/.

[PS07] Vassilis Prevelakis and Diomidis Spinellis. The Athens affair. *IEEE Spectrum*, 44(7):26–33, 2007.

[PSJT08] Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems: an experience report. In *1st International Workshop on Multicore Software Engineering (IWMSE '08)*, pages 53–60. ACM Press, 2008.

[PST06] Jérémy Philippe, Sylvain Sicard, and Christophe Taton. Component-based autonomic management system. In *5th Fractal Workshop*, 2006.

[PT08] Victor Pankratius and Walter F. Tichy. Die Multicore-Revolution und ihre Bedeutung für die Softwareentwicklung. *Objektspektrum*, 4:30–32, 2008.

[Pud08] Arno Puder. CORBA product profiles. http://www.puder.org/corba/matrix/, 2008. retrieved 11/20/2008.

[Pur94] James M. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, 1994.

[PV02] František Plášil and Stanislav Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[PY04] Taesoon Park and Jaehwan Youn. The k-fault-tolerant checkpointing scheme for the reliable mobile agent system. In Kim-Meow Liew, Hong Shen, Simon See, Wentong Cai, Pingzhi Fan, and Susumu Horiguchi, editors, *5th International Conference on Parallel and Distributed Computing: Applications and Technologies (PDCAT '04)*, volume 3320 of *Lecture Notes in Computer Science*, pages 577–581. Springer, 2004.

[RA03] Mikaël Rémond and Joe Amstrong. *Erlang programmation*. Eyrolles, 2003.

[RAC+02] Matthew J. Rutherford, Kenneth M. Anderson, Antonio Carzaniga, Dennis Heimbigner, and Alexander L. Wolf. Reconfiguration in the Enterprise Java Bean component model. In *IFIP/ACM Working Conference on Component Deployment (CD '02)*, pages 67–81. Springer, 2002.

[RBH94] Kurt Rothermel, Ingo Barth, and Tobias Helbig. CINEMA – an architecture for distributed multimedia applications. In *1st International Workshop on Architecture and Protocols for High-Speed Networks*, pages 253–271. Kluwer Academic Publishers, 1994.

[REF+08] Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein O. Hallsteinsen, and Erlend Stav. Composing components and services using a planning-based adaptation middleware. In *Software Composition, 7th International Symposium*, volume 4954 of *Lecture Notes in Computer Science*, pages 52–67. Springer, 2008.

[Rei07] Sebastian Reichelt. Open-source component models: Bonobo and KParts. In Steffen Becker, Jens Happe, Heiko Koziolek, Klaus Krogmann, Michael Kuperberg, and Ralf Reussner, editors, *Software-Komponentenmodelle*. Universität Karlsruhe, 2007. Interner Bericht 2007-10.

[Ren03] Arend Rensink. GROOVE: A graph transformation tool set for the simulation and analysis of graph grammars. Available at http://www.cs.utwente.nl/~groove, 2003.

[Ren04] Arend Rensink. The GROOVE simulator: A tool for state space generation. In J. Pfalz, M. Nagl, and B. Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume 3062 of *Lecture Notes in Computer Science*, pages 479–485. Springer, 2004.

[RLS01] Paul Robertson, Robert Laddaga, and Howie Shrobe. Introduction: The first international workshop on self-adaptive software. In *Self-Adaptive Software*, volume 1936 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2001.

[Röl02] Harald Rölle. A hot-failover state machine for gateway services and its application to a Linux firewall. In *13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM '02)*, volume 2506 of *Lecture Notes in Computer Science*, pages 181–194. Springer, 2002.

[RP08] Andreas Rasche and Andreas Polze. ReDAC dynamic reconfiguration of distributed component-based applications with cyclic dependencies. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008)Lim*, pages 322–330. IEEE Computer Society Press, 2008.

[RRMP08] Andreas Rausch, Ralf Reussner, Raffaela Mirandola, and František Plášil, editors. *The Common Component Modeling Example: Comparing Software Component Models [result from the Dagstuhl research seminar for CoCoME, August 1-3, 2007]*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.

[RS07a] Andreas Rasche and Wolfgang Schult. Dynamic updates of graphical components in the .NET framework. In *Workshop on Selbstorganisierende, Adaptive, Kontextsensitive verteilte Systeme (SAKS '07)*. VDE Verlag, 2007.

[RS07b] Andreas Rogge-Solti. Verteilung eines Komponentenframeworks. Fortgeschrittenenpraktikum, LMU München, 2007.

[Rus08] Teodor Rus. Variants of Turing machines. Lecture Notes on Course 22C:135 – Theory of Computation, University of Iowa, 2008.

[SAW94] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE Computer Society Press, 1994.

[SBG06] Saša Subotić, Judith Bishop, and Stefan Gruner. Aspect-oriented programming for a distributed framework. *South African Computer Journal (Suid Afrikaanse Rekenaar Tydskrif)*, 37:81–89, 2006.

[SC00] João Costa Seco and Luís Caires. A basic model of typed components. In Elisa Bertino, editor, *14th European Conference on Object-Oriented Programming (ECOOP '00)*, volume 1850 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2000.

[SC02] João Costa Seco and Luís Caires. ComponentJ: The reference manual. Technical Report UNL-DI-6-2002, Departamento de Informática, Universidade Nova de Lisboa, 2002.

[SC06] João Costa Seco and Luís Caires. Types for dynamic reconfiguration. In Peter Sestoft, editor, *ESOP*, volume 3924 of *Lecture Notes in Computer Science*, pages 214–229. Springer, 2006.

[Sca00] Todd Scallan. Monitoring and diagnostics of Corba systems. *Java Developer's Journal*, 5(6):138–144, 2000.

[Sch05] Christian Schmid. Course on dynamics of multidisplicinary and controlled systems. http://www.esr.ruhr-uni-bochum.de/rt1/syscontrol, 2005.

[SD94] Ian Sommerville and Graham Dean. PCL – a language for modelling system architecture. *IEE Colloquium on Reverse Engineering for Software Based Systems*, pages 3/1–3/3, 1994.

[SD97] Ulrich Stern and David L. Dill. Parallelizing the Murφ verifier. In Orna Grumberg, editor, *Computer-Aided Verification, 9th International Conference*, volume 1254 of *Lecture Notes in Computer Science*, pages 256–267. Springer, 1997. Haifa, Israel, June 22-25.

[SD98] Ulrich Stern and David L. Dill. Using magnatic disk instead of main memory in the Murφ verifier. In *10th International Conference on Computer Aided Verification (CAV '98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 1998.

[SE05] Stefan Schwoon and Javier Esparza. A note on on-the-fly verification algorithms. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 174–190. Springer, 2005.

[Sec06] João Costa Seco. *Languages and Types for Component-Based Programming*. PhD thesis, Universidade Nova de Lisboa, Faculdade de Ciencias e Tecnologia, Departamento de Informatica, 2006.

[SG86] Jeffrey C. Schlimmer and Richard H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.

[SGM02] Clemens Szyperski, Domiinik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, second edition edition, 2002.

[Sha00] David C. Sharp. Reducing avionics software cost through component based product line development. In Patrick Donohoe, editor, *17th AIAA/IEEE/SAE Digital Avionics Systems Conference (DASC '00)*. Kluwer Academic Publishers, 2000.

[SKB98] Janos Sztipanovits, Gabor Karsai, and Ted Bapty. Self-adaptive software for signal processing. *Communications of the ACM*, 41(5):66–73, 1998.

[SLS92] Sang Hyuk Son, Juhnyoung Lee, and Savita Shamsunder. Real-time transaction processing: Pessimistic, optimistic, and hybrid approaches. In *2nd International Workshop on Transaction and Query Processing (RIDE-TQP '92)*, pages 222–. IEEE Computer Society Press, 1992.

[SM03] S. Masoud Sadjadi and Philip K. McKinley. A survey of adaptive middleware. Technical Report MSU-CSE-03-35, Computer Science and Engineering, Michigan State University, 2003.

[SM04] Seyed Masoud Sadjadi and Philip K. McKinley. ACT: An adaptive CORBA template to support unanticipated adaptation. In *24th IEEE International Conference on Distributed Computing Systems (ICDCS '04)*, pages 74–83. IEEE Computer Society Press, 2004.

[Spr06] Felix Sprick. Rapid service migration in peer-to-peer networks. Diplomarbeit, LMU München, 2006.

[SPW03] Nigamanth Sridhar, Scott M. Pike, and Bruce W. Weide. Dynamic module replacement in distributed protocols. In *23rd International Conference on Distributed Computing Systems (ICDCS '03)*, page 620. IEEE Computer Society Press, 2003.

[SRG96] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. Evolving dependable real-time systems. In *IEEE Aerospace Applications Conference*, pages 335–46. IEEE Computer Society Press, 1996.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Sun97] Sun Microsystems. JavaBeans™. Specification Version 1.01-A, 1997.

[SWL+94] Mark E. Stickel, Richard J. Waldinger, Michael R. Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In *12th International Conference on Automated Deduction (CADE '94)*, pages 341–355. Springer, 1994.

[SZH08] Andreas Schroeder, Marjolein van der Zwaag, and Moritz Hammer. A middleware architecture for human-centred pervasive adaptive applications. In *2nd International Conference on Self-Adaptive and Self-Organizing Systems (PerAda '08)*, pages 138–143. IEEE Computer Society Press, 2008.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, 1998.

[Tar72] Robert E. Tarjan. Depth-first search and linear graph algorithms. In *SIAM Journal of Computing*, volume 1, pages 146–160. Society for Industrial and Applied Mathematics, 1972.

[TBB+05] Christophe Taton, Sara Bouchenak, Fabienne Boyer, Noël de Palma, Daniel Hagimont, and Adrian Mos. Self-manageable replicated servers. In *VLDB Workshop on Design, Implementation, and Deployment of Database Replication (WDIDDR '05)*. ACM Press, 2005.

[TJSJ08] Eddy Truyen, Nico Janssens, Frans Sanen, and Wouter Joosen. Support for distributed adaptations in aspect-oriented middleware. In *7th International Conference on Aspect-Oriented Software Development (AOSD '08)*, pages 120–131. ACM Press, 2008.

[TMMS01] L. A. Tewksbury, Louise E. Moser, and P. M. Melliar-Smith. Live upgrade techniques for CORBA applications. In *3rd International Working Conference on New Developments in Distributed Applications and Interoperable Systems*, pages 257–272. Kluwer Academic Publishers, 2001.

[Tom94] James E. Tomayko. *Computers in Space: Journeys with NASA*. Alpha Books, 1994.

[Tom00] James E. Tomayko. *Computers Take Flight: A History of NASA's Pioneering Digital Fly-by-Wire Project*. United States Government Printing, 2000.

[Tra88] Will Tracz. Software reuse myths. *ACM SIGSOFT Software Engineering Notes*, 13(1):17–21, 1988.

[Tra94] Will Tracz. Software reuse myths revisited. In *16th International Conference on Software Engineering (ICSE '94)*, pages 271–272. IEEE Computer Society Press, 1994.

[TT06] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In Frank Eliassen and Alberto Montresor, editors, *6th International Conference on Distributed Applications and Interoperable Systems*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331. Springer, 2006.

[TY05] Christian F. Tschudin and Lidia Yamamoto. Harnessing self-modifying code for resilient software. In *2nd International Workshop on Radical Agent Concepts (WRAC '05)*, volume 3825 of *Lecture Notes in Computer Science*, pages 197–204. Springer, 2005.

[USAF05] United States Air Force. Combat Support. Air Force Doctrine Document 2-4, 2005.

[Van07] Yves Vandewoude. *Dynamically updating component-oriented systems*. PhD thesis, Katholieke Universiteit Leuven, 2007.

[Var05] Moshe Y. Vardi. Büchi complementation: A forty-year saga. In *5th symposium on Atomic Level Characterizations (ALC '05)*, 2005.

[VB02] Yves Vandewoude and Yolande Berbers. An overview and assessment of dynamic update methods for component-oriented embedded systems. In Hamid R. Arabnia and Youngsong Mun, editors, *International Conference on Software Engineering Research and Practice (SERP '02)*, pages 521–527. CSREA Press, 2002.

[VB05] Yves Vandewoude and Yolande Berbers. Fresco: Flexible and reliable evolution system for components. *Electronic Notes in Theoretical Computer Science*, 127(3):197–205, 2005.

[VDBM97] Jean-Yves Vion-Dury, Luc Bellissard, and Vladimir Marangozov. A component calculus for modeling the Olan configuration language. In David Garlan and Daniel Le Métayer, editors, *2nd International Conference on Coordination Languages and Models*, volume 1282 of *Lecture Notes in Computer Science*, pages 392–409. Springer, 1997.

[VEBD07] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquillity: A low disruptive alternative to Quiescence for ensuring safe dynamic updating. *IEEE Transactions on Software Engineering*, 33(12), 2007.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[VK02] Giuseppe Valetto and Gail Kaiser. A case study in software adaptation. In *1st Workshop on Self-Healing Systems (WOSS '02)*, pages 73–78. ACM Press, 2002.

[Völ03] Markus Völter. A taxonomy of components. *Journal of Object Technology*, 2(4):119–125, 2003.

[VZL+98] Rodrigo Vanegas, John A. Zinky, Joseph P. Loyall, David Karr, Richard E. Schantz, and David E. Bakken. QuO's runtime support for quality of service in distributed objects. In *International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 15–18. Springer, 1998.

[WB05] Craig Walls and Ryan Breidenbach. *Spring in Action*. Manning, 2005.

[WCL+01] Dan Werthimer, Jeff Cobb, Matt Lebofsky, David Anderson, and Eric Korpela. SETI@HOME—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, 2001.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *ACM/IEEE conference on Supercomputing (Supercomputing '98)*, pages 1–27. IEEE Computer Society Press, 1998.

[WDT+07] Martin Wirsing, Grit Denker, Carolyn Talcott, Andy Poggio, and Linda Briesemeister. A rewriting logic framework for soft constraints. *Electronic Notes in Theoretical Computer Science*, 176(4):181–197, 2007.

[Web06] Michael Weber. *Parallel Algorithms for Verification of Large Systems*. PhD thesis, RWTH Aachen University, 2006.

[Web07] Michael Weber. An embeddable virtual machine for state space generation. In Dragan Bosnacki and Stefan Edelkamp, editors, *14th International SPIN Workshop on Model Checking Software (SPIN '07)*, volume 4595 of *Lecture Notes in Computer Science*, pages 168–185. Springer, 2007.

[Weg03] Marteen Wegdam. *Dynamic Reconfiguration and Load Distribution in Component Middleware*. PhD thesis, University of Twente, Enschede, The Netherlands, 2003.

[Wei89] Charles B. Weinstock. Performance and reliability enhancement of the Durra runtime environment. Technical Report CMU/SEI-89-TR-008, Carnegie Mellon University, 1989.

[Wei91] Mark Weiser. The computer for the 21st century. *Scientific American*, 9, 1991.

[Wer98] Michel Wermelinger. Towards a chemical model for software architecture reconfiguration. In *International Conference on Configurable Distributed Systems (CDS '98)*, pages 111–126. IEEE Computer Society Press, 1998.

[WF99] Michel Wermelinger and José Luiz Fiadeiro. Algebraic software architecture reconfiguration. *ACM SIGSOFT Software Engineering Notes*, 24(6):393–409, 1999.

[WH04] Ian Warren and Jamie Hillman. Quantitative analysis of dynamic reconfiguration algorithms. In *International Conference on Design, Analysis and Simulation of Distributed Systems (DASD '04)*, pages 18–22. The Society for Modeling and Simulation International, 2004.

[Wil00] Torres Wilfredo. Software fault tolerance: A tutorial. Technical Report TM-2000-210616, NASA Langley Technical Report Server, 2000.

[WK92] Gerhard Widmer and Miroslav Kubat. Learning flexible concepts from streams of examples: FLORA 2. In *10th European Conference on Artificial Intelligence (ECAI '92)*, pages 463–467. John Wiley & Sons, 1992.

[WK96] Gerhard Widmer and Miroslav Kubat. Learning in the presence of concept drift and hidden contexts. *Machine Learning*, 23(1):69–101, 1996.

[WL93] Pierre Wolper and Denis Leroy. Reliable hashing without collosion detection. In *5th International Conference on Computer Aided Verification (CAV '93)*, pages 59–70. Springer, 1993.

[WL98] Yi-Min Wang and Woei-Jyh Lee. COMERA: COM extensible remoting architecture. In *4th USENIX Conference on Object-Oriented Technologies and Systems (COOTS '98)*, pages 6–17. USENIX Association, 1998.

[WLF01] Michel Wermelinger, Antónia Lopes, and José Luiz Fiadeiro. A graph based architectural (re)configuration language. *ACM SIGSOFT Software Engineering Notes*, 26(5):21–32, 2001.

[WPP04] Oskar Wibling, Joachim Parrow, and Arnold Pears. Automatized verification of ad hoc routing protocols. In *24th International Conference on Formal Techniques for Networked and Distributed Systems (FORTE '04)*, volume 3235 of *Lecture Notes in Computer Science*, pages 343–358. Springer, 2004.

[WS85] Colin Whitby-Strevens. The transputer. In *12th Annual International Symposium on Computer Architecture (ISCA '85)*, pages 292–300. IEEE Computer Society Press, 1985.

[WS96] Ian Warren and Ian Sommerville. A model for dynamic configuration which preserves application integrity. *iccds*, 00:81, 1996.

[WSG+03] Nanbor Wang, Douglas C. Schmidt, Aniruddha Gokhale, Christopher D. Gill, Balachandran Natarajan, Craig Rodrigues, Joseph P. Loyall, and Richard E. Schantz. Total quality of service provisioning in middleware and applications. *Microprocessors and Microsystems*, 27(2):45–54, 2003.

[WSKW06] Ian Warren, Jing Sun, Sanjev Krishnamohan, and Thiranjith Weerasinghe. An automated formal approach to managing dynamic reconfiguration. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE '06)*, pages 37–46. IEEE Computer Society Press, 2006.

[WSO01] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. *Overview of the CORBA component model*, chapter 38, pages 557–571. Addison-Wesley, 2001.

[Ye01] Yunwen Ye. *Supporting component-based software development with active component repository systems.* PhD thesis, University of Colorado, 2001.

[Yel03] Daniel M. Yellin. Competitive algorithms for the dynamic selection of component implementations. *IBM Systems Journal*, 42(1):85–97, 2003.

[ZBBF07] Gianluigi Zavattaro, Michele Boreale, Mario Bravetti, and GianLuigi Ferrari. D2.a: Mechanisms for Service Composition, Query and Discovery. Technical report, Sensoria WP2 Core Deliverable, 2007.

[ZBS97] John A. Zinky, David E. Bakken, and Richard E. Schantz. Architectural support for quality of service for CORBA objects. *Theory and Practice of Object Systems*, 3(1), 1997.

[ZBZ07] Ming Zhang, Azzedine Boukerche, and Bernard P. Zeigler. Exploiting the concept of activity for dynamic reconfiguration of distributed simulation. In *11th IEEE International Symposium on Distributed Simulation and Real-Time Applications (DS-RT '07)*, pages 87–94. IEEE Computer Society Press, 2007.

[ZL07] Zhikun Zhao and Wei Li. Influence control for dynamic reconfiguration. In *Australian Software Engineering Conference (ASWEC '07)*, pages 59–70. IEEE Computer Society Press, 2007.

[ZLAS02] Ronghua Zhang, Chenyang Lu, Tarek F. Abdelzaher, and John A. Stankovic. Controlware: A middleware architecture for feedback control of software performance. In *22nd International Conference on Distributed Computing Systems (ICDCS '02)*. IEEE Computer Society Press, 2002.

Index

ERLANG, 166, 184, 189, 262, 264
JAVA, 3, 8, 11, 12, 14, 31, 33, 34, 79, 87, 145, 147, 164, 208, 221, 254
  annotation, 147
  security model, 147
  static variable, 147

abstraction, 97, 103, 121, 124, 205, 244, 245
ACID, 26, 32, 236
activity flow, 36, 87, 90
adaptivity, 1–3
agent, 2, 13, 37, 218
architectural erosion, 18
architecture
  multi-layered, 205
aspect-orientation, 177
assembly, 14, 17, 148, 205
assessment, 172, 179
assumption, 231, 233, 234
automaton
  Büchi, **240**, 245
  cellular, 63
  component-interaction, 240
  hybrid, 213
  I/O, 240
  interface, 240
  pushdown, 71
  queue, 242
  state, 239, 256
autonomy, 13

bean, 12, 31
bisimulation, 75, 130
black box, 9, 13, 14, 176
blocking, 163
Bloom filter, 83, 95, 99, 106, 226, 227
business process, 17

callback, 36, 87, 114, 115, 253
client, 10, 11
client-server, 196
co-future, 70
communication, 3, 12, 66
  broadcast, 196
  gather-cast, 34, 69, 198
  paradigm, 147
  postpone, 162
  self-invocation, 162, 204, 234
  specification, 232
  trace, 116, 129, 233
completely connected, **72**, 90, 120, 149
component, 3, **67**, 8–67
  application, 16, 66
  behavior, 16
  configuration, 115
  COTS, 15
  data, 12
  definition, 8–15
  designer, 10, 18
  framework, 1, 14, 66, 79
  graph, 14, 148, 149, 153, 155, 184, 198, 210, 211
  hierarchical, 199
  identifier, 66, 67
  initial, 149, 157, 159
  locator, 154
  model, 10, 14, 66, **66**, 70, **78**, 109
    hierarchical, 33
  mono-threaded, 114, 178, 231
  parameter, 32, 142, 149
  process term, 79
  proxy, 157
  repository, 18, 33
  setup, 14, **71**, 78, 148, 155, 181, 190, 205, 229
  signature, 67
  software component, 8
  stateful, 3, 14, 66
  type, 14, 15
component-based software engineering, 5, 9, 13, 14, 16, 100, 229
concept drift, 158, 179
concurrency, 113
configuration, 3, 115
configuration transition rule, 115
connection, 14
connector, 14, 31, 33, 35, 154, 164, 198, 205, 261, 267
control loop, 172
control theory, 172
correctness, 16
counter-example, 82
covert channel, 145, 147
cross-cutting concern, 98, 177

deadlock, 2, 114, 116, 117, 121, 124, 125, 143, 164, 176, 209, 230, 237, 248
deep copy, 145

distribution, 153
  node, 153
diversity, 32
domain decomposition, 97, 161

environment, 2
execution, 74, 172

fault tolerance, 1, 153, 175, 183, 216, 237–239
feedback, 173, 175
feedback loop, 172
fluctuation, 173
functional decomposition, 97, 161

granularity, 8, 10, 11, 20, 22, 117, 176, 184, 190, 207, 220, 263, 267
graph, 72
  component, 148
  directed labelled, 72
  partitioning, 155
  subgraph, 72
  weighted, 155
gray box, 14
guarantee, 234

hash table, 2, 11, 14, 66, 83, 95, 103, 105, 226, 232, 244, 252, 254
hierarchy, 205
host language, 11, 12, 26, 32, 34, 79, 86
hot code update, 1, 19, 24, 110, 171, 175, 176, 188, 190, 191

input-enabled, 240
interface, 3, 13, 14, 66
  provided, 3, **67**
  required, 3, 66, **67**
interface provider, 154, 197
interleaving, *see* race condition

JVMTI, 151, 155, 176

knowledge, 175

labelled transition system, 73, 79, 82
  bisimilarity, 76
  disjoint union, 75
latency, 24
liveness property, 183
lossy hash table, 83, 89, 95, 99, 104, 224
LTL, 182

MAPE, 29, 172
membrane, 205
message, 12, 13
  future, 114
  synchronous, 114
message queue, 161
method, 3, 66
method evaluator, 67, 244
method invocation, 70
method request, 143, 148
migration, 13
model checking, 82, 87, 244, 250
  regular, 245

module, 14, 17
monitor, 17
  exception, 218
monitoring, 151, 172, 176
  communication, 151
  component graph, 153
  exception, 152
multiport, 69, 196

nondeterminism, 2, 71, 91, 95, 122, 127, 179, 217, 240–242, 250, 252, 264

object, 11
object-orientation, 12
OCL, 8, 14
optimization, 173

parallel programs, 97
pattern
  Active Object, 34, 143, 144, 201, 210
  Adapter, 178
  Chain of Responsibility, 69, 83, 134, 195, 196, 198
  Composite, 12
  Double-Dispatch, 143
  Factory, 31
  Filter, 133, 156, 177, 184, 190, 204, 206, 211, 212, 214, 215
  Memento, 135, 210
  Proxy, 143, 154
  Singleton, 158, 198
  State, 201
  Strategy, 21, 201
  Utility Interface, 147, 158, 198
  Visitor, 204, 266
  Workflow, 17
phase, 180
planning, 2, 172
port, 14, 68
postcondition, 232
precondition, 232
process term, 67, **70**, 77
  query, 71
  subterm, 71
  variable, 70
programming-in-the-large, 17
protocol, 240
proxy, 157

quality of service, 179
query, 71
queue, 114
quiescence, 23, 121, 176, 187, 209, 252

race condition, 98, 114, 119
reconfiguration, 3, 14, 17
  benefit, 182
  coarse-grained, 3
  cost, 182
refinement, 250
REFLECT, 154, 164, 261, 263, 267
robotics, 175
role, 18, 66
rule, 77, 78, 90

instantiation, 78
run, **74**, 116

safety property, 182
self-⋆, 1, 20, 51, 174, 259
self-healing, 216
self-invocation, 91
semantics, 70
separation of concerns, 3, 17, 18, 62, 176, 177, 201
separation of roles, 18, 22, 23, 94, 116, 136, 140, 197, 227, 229, 231, 234, 239, 248, 258
serializability, 235
service, 11, 12
shared memory, 113, 145
simulation, 75
software crisis, 8
software engineering, 1, 9, 11, 15, 217
software reuse, 15
space-time diagram, 121
specification, 10
stability, 182
state, 3, 9, 14, 66, 73, 75, 77
data, 15, 24, 66, 71, 73, 79, 118
retainment, 24, 119, 120, 135
state rewriting system, 76
state transfer, 3, 26, 53, 135, 187, 191, 201, 209, 252, 255
direct, 53, 136, 142
hybrid, 140, 192, 194
indirect, 53, 136
static analysis, 145, 189, 255
system designer, **18**, 72, 140, 159, 161, 175, 207, 231, 232, 234

terminator, 196
three-valued, 83, 183
trace, 74, **75**, 116, 129
abstraction, 75, 116
communication, 116, 131
tranquility, 23, 187, **252**
transaction, 187, 232, 252
Turing-completeness, 71, 229, 243, 245

Ubiquitous computing, 1

value, 66
variable, 79
verification, 229

weak bisimulation, 75
well-connected, **72**, 90, 120, 126, 137, 149
window size, 181

Lebenslauf

Persönliche Daten

Moritz Hammer
Waldfriedhofstrasse 1
81377 München

Tel.: (089) 48 95 42 38
E-Mail: hammer@pst.ifi.lmu.de

Geb. am 10. 01. 1978 in München
Ledig, deutsch

Schulbildung

08/1985–06/1989	Grundschule Großenheidorn
08/1989–06/1991	Orientierungsstufe Steinhude
08/1991–05/1997	Hölty-Gymnasium Wunstorf (Leistungskurse Deutsch und Physik)

Zivildienst

07/1997–08/1998	Altenheim am Bürgerpark, Wunstorf

Studium

10/1998–10/1999	Studium der Theaterwissenschaft/Dramaturgie in Mainz und an der LMU München
10/1999–11/2001	Grundstudium der Informatik (mit Nebenfach Kommunikationswissenschaft) an der LMU München, Vordiplomnote: Sehr gut
11/2001–08/2004	Hauptstudium der Informatik an der LMU München. Diplomnote: Mit Auszeichnung
seit 09/2004	Promotionsstudium am Institut für Informatik, LMU München

Berufserfahrung

07/1999–06/2001	Programmierertätigkeit für das Campus LMU-Projekt
11/2001–06/2003	MGM-EDV-Beratung, Java-Programmierer für B2B-Webanwendungen
seit 04/2006	Wissenschaftlicher Assistent am Lehrstuhl ≪Programmierung und Softwaretechnik≫, Institut für Informatik, LMU München

Stipendien

03/2002–03/2003	eFellows.net Stipendium
08/2004–03/2006	Stipendium des Graduiertenkollegs ≪Logik in der Informatik≫

Publikationen

[HKM05] Moritz Hammer, Alexander Knapp, and Stephan Merz. Truly on-the-fly LTL model checking. In Nicolas Halbwachs and Lenore Zuck, editors, *11th Intl. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)*, Lecture Notes in Computer Science. Springer, 2005.

[HW06a] Moritz Hammer and Michael Weber. "To store or not to store" reloaded: Reclaiming memory on demand. In Lubos Brim, Boudewijn Haverkort, Martin Leucker, and Jaco van de Pol, editors, *Formal Methods: Application and Technology (FMICS'2006)*, volume 4346 of *Lecture Notes in Computer Science*, pages 51–66. Springer, 2006. Won the EASST Best Paper Award of FMICS '2006.

[HW06b] Moritz Hammer and Michael Weber. "To store or not to store" reloaded: Reclaiming memory on demand. *The EASST Newsletter Volume*, 14:5–11, 2006. Shortened version of [**HW06a**].

[HKMR08] Moritz Hammer, Bernhard Kempter, Florian Mangold, and Harald Roelle. Skalierbare Performanzanalyse durch Prolongation. In Korbinian Herrmann and Bernd Bruegge, editors, *Software Engineering (SE 2008)*, volume 121 of *Lecture Notes in Informatics*, pages 127–139. Gesellschaft für Informatik, 2008.

[HK08] Moritz Hammer and Alexander Knapp. Correct execution of reconfiguration for stateful components. In *5th International Workshop on Formal Aspects of Component Software (FACS'08)*, 2008. to appear.

[SvdZH08] Andreas Schroeder, Marjolein van der Zwaag, and Moritz Hammer. A middleware architecture for human-centred pervasive adaptive applications. *2nd International Conference on Self-Adaptive and Self-Organizing Systems (PerAda '08)*, pages 138–143. IEEE Computer Society Press, 2008.