Software engineering perspectives on physiological computing

Andreas Schroeder

Dissertation
an der Fakultät für Mathematik, Informatik und Statistik
der Ludwig–Maximilians–Universität
München

vorgelegt von
Andreas Schroeder
aus Nizza

München, den 30.11.2011

Erstgutachter: Prof. Dr. Martin Wirsing
Zweitgutachter: Prof. Dr.-Ing. Stefan Jähnichen
Tag der Mündlichen Prüfung: 22.12.2011

Abstract

Physiological computing is an interesting and promising concept to widen the communication channel between the (human) users and computers, thus allowing an increase of software systems' contextual awareness and rendering software systems smarter than they are today. Using physiological inputs in pervasive computing systems allows re-balancing the information asymmetry between the human user and the computer system: while pervasive computing systems are well able to flood the user with information and sensory input (such as sounds, lights, and visual animations), users only have a very narrow input channel to computing systems; most of the time, restricted to keyboards, mouse, touchscreens, accelerometers and GPS receivers (through smartphone usage, e.g.). Interestingly, this information asymmetry often forces the user to subdue to the quirks of the computing system to achieve his goals – for example, users may have to provide information the software system demands through a narrow, time-consuming input mode that the system could sense implicitly from the human body. Physiological computing is a way to circumvent these limitations; however, systematic means for developing and moulding physiological computing applications into software are still unknown.

This thesis proposes a methodological approach to the creation of physiological computing applications that makes use of component-based software engineering. Components help imposing a clear structure on software systems in general, and can thus be used for physiological computing systems as well. As an additional bonus, using components allow physiological computing systems to leverage reconfigurations as a means to control and adapt their own behaviours. This adaptation can be used to adjust the behaviour both to the human and to the available computing environment in terms of resources and available devices – an activity that is crucial for complex physiological computing systems. With the help of components and reconfigurations, it

is possible to structure the functionality of physiological computing applications in a way that makes them manageable and extensible, thus allowing a stepwise and systematic extension of a system's intelligence.

Using reconfigurations entails a larger issue, however. Understanding and fully capturing the behaviour of a system under reconfiguration is challenging, as the system may change its structure in ways that are difficult to fully predict. Therefore, this thesis also introduces a means for formal verification of reconfigurations based on assume-guarantee contracts. With the proposed assume-guarantee contract framework, it is possible to prove that a given system design (including component behaviours and reconfiguration specifications) is satisfying real-time properties expressed as assume-guarantee contracts using a variant of real-time linear temporal logic introduced in this thesis – metric interval temporal logic for reconfigurable systems.

Finally, this thesis embeds both the practical approach to the realisation of physiological computing systems and formal verification of reconfigurations into Scrum, a modern and agile software development methodology. The surrounding methodological approach is intended to provide a frame for the systematic development of physiological computing systems from first psychological findings to a working software system with both satisfactory functionality and software quality aspects.

By integrating practical and theoretical aspects of software engineering into a self-contained development methodology, this thesis proposes a roadmap and guidelines for the creation of new physiological computing applications.

Zusammenfassung

Physiologisches Rechnen ist ein interessantes und vielversprechendes Konzept zur Erweiterung des Kommunikationskanals zwischen (menschlichen) Nutzern und Rechnern, und dadurch die Berücksichtigung des Nutzerkontexts in Software-Systemen zu verbessern und damit Software-Systeme intelligenter zu gestalten, als sie es heute sind. Physiologische Eingangssignale in ubiquitären Rechensystemen zu verwenden, ermöglicht eine Neujustierung der Informationsasymmetrie, die heute zwischen Menschen und Rechensystemen existiert: Während ubiquitäre Rechensysteme sehr wohl in der Lage sind, den Menschen mit Informationen und sensorischen Reizen zu überfluten (z.B. durch Töne, Licht und visuelle Animationen), hat der Mensch nur sehr begrenzte Einflussmöglichkeiten zu Rechensystemen. Meistens stehen nur Tastaturen, die Maus, berührungsempfindliche Bildschirme, Beschleunigungsmesser und GPS-Empfänger (zum Beispiel durch Mobiltelefone oder digitale Assistenten) zur Verfügung. Diese Informationsasymmetrie zwingt die Benutzer zur Unterwerfung unter die Usancen der Rechensysteme, um ihre Ziele zu erreichen – zum Beispiel müssen Nutzer Daten manuell eingeben, die auch aus Sensordaten des menschlichen Körpers auf unauffällige weise erhoben werden können. Physiologisches Rechnen ist eine Möglichkeit, diese Beschränkung zu umgehen. Allerdings fehlt eine systematische Methodik für die Entwicklung physiologischer Rechensysteme bis zu fertiger Software.

Diese Dissertation präsentiert einen methodischen Ansatz zur Entwicklung physiologischer Rechenanwendungen, der auf der komponentenbasierten Softwareentwicklung aufbaut. Der komponentenbasierte Ansatz hilft im Allgemeinen dabei, eine klare Architektur des Software-Systems zu definieren, und kann deshalb auch für physiologische Rechensysteme angewendet werden. Als zusätzlichen Vorteil erlaubt die Komponentenorientierung in physiologischen Rechensystemen, Rekonfigurationen als Mittel zur Kontrolle und Anpassung des Verhaltens von physiologischen Rechensystemen zu verwen-

den. Diese Adaptionstechnik kann genutzt werden um das Verhalten von physiologischen Rechensystemen an den Benutzer anzupassen, sowie an die verfügbare Recheninfrastruktur im Sinne von Systemressourcen und Geräten – eine Maßnahme, die in komplexen physiologischen Rechensystemen entscheidend ist . Mit Hilfe der Komponentenorientierung und von Rekonfigurationen wird es möglich, die Funktionalität von physiologischen Rechensystemen so zu strukturieren, dass das System wartbar und erweiterbar bleibt. Dadurch wird eine schrittweise und systematische Erweiterung der Funktionalität des Systems möglich.

Die Verwendung von Rekonfigurationen birgt allerdings Probleme. Das Systemverhalten eines Software-Systems, das Rekonfigurationen unterworfen ist zu verstehen und vollständig einzufangen ist herausfordernd, da das System seine Struktur auf schwer vorhersehbare Weise verändern kann. Aus diesem Grund führt diese Arbeit eine Methode zur formalen Verifikation von Rekonfigurationen auf Grundlage von Annahme-Zusicherungs-Verträgen ein. Mit dem vorgeschlagenen Annahme-Zusicherungs-Vertragssystem ist es möglich zu beweisen, dass ein gegebener Systementwurf (mitsamt Komponentenverhalten und Spezifikation des Rekonfigurationsverhaltens) eine als Annahme-Zusicherungs-Vertrag spezifizierte Echtzeiteigenschaft erfüllt. Für die Spezifikation von Echtzeiteigenschaften kann eine Variante von linearer Temporallogik für Echtzeit verwendet werden, die in dieser Arbeit eingeführt wird: Die metrische Intervall-Temporallogik für rekonfigurierbare Systeme.

Schließlich wird in dieser Arbeit sowohl ein praktischer Ansatz zur Realisierung von physiologischen Rechensystemen als auch die formale Verifikation von Rekonfigurationen in Scrum eingebettet, einer modernen und agilen Softwareentwicklungsmethodik. Der methodische Ansatz bietet einen Rahmen für die systematische Entwicklung physiologischer Rechensysteme von Erkenntnissen zur menschlichen Physiologie hin zu funktionierenden physiologischen Softwaresystemen mit zufriedenstellenden funktionalen und qualitativen Eigenschaften.

Durch die Integration sowohl von praktischen wie auch theoretischen Aspekten der Softwaretechnik in eine vollständige Entwicklungsmethodik bietet diese Arbeit einen Fahrplan und Richtlinien für die Erstellung neuer physiologischer Rechenanwendungen.

Acknowledgments

To begin with, I would like to thank Martin Wirsing; not only for his support, but also for his larger efforts in creating a working environment at the PST chair that is open-minded, challenging, respectful and humane – it is with high gratitude that I will cherish my memories of the PST chair. I also thank Stefan Jähnichen for immediately and pragmatically accepting to be my second reviewer, and taking another responsibility on top of his tremendous workload.

Additionally, I would like to thank the whole PST group for their support and for the great team environment. Especially, I'd like to thank Philip Mayer. Experiencing his pragmatic zeal and his focus in discussions and pair coding sessions left a persistent mark on my career. I am also very thankful for the countless discussions Christian Kroiß provided me with, and especially for his critical eye and refreshingly dissenting view on several topics we discussed. Furthermore, I am thankful for the feedback that Sebastian Bauer provided me on the formal aspects of my thesis. I'd also like to thank Annabelle Klarl, my room mate, who provided me with valuable quick feedback on ideas.

A considerable part of this work was created during the REFLECT project, and the results of my work would not have been the same without the fruitful collaboration I experienced. From the project team, I would like to highlight Alessandro Ragnoni and Amadeo Visconti, whose dedication, persistence and humanity I highly admire.

Finally, I'd like to thank my family and friends for their support and patience during the writing of my thesis. Especially, my thanks go to Ylva. I cannot express how thankful I am for the love, confidence and vitality she brought into my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Today's computing devices and infrastructure fail more and more to satisfy the increasing expectations and needs of everyday users. Today's computing infrastructure is accompanying us throughout our everyday life in form of smartphones, tablets, mobile computers, and desktop computers that allow us to access data and processing power almost everywhere. Interestingly however, our companions today are largely oblivious of their user, and may, as a consequence, nag us inappropriately, or even put us in embarrassing situations. This behaviour is a symptom of an underlying problem: the communication bandwidth from the user to the computer is fairly limited today [89]. The available input methods are restricted to keyboard, mouse, touch screens, GPS and accelerometers (e.g. through worn gadgets such as smartphones or tablets). However, none of these input method allow computer to understand their user's emotions, cognitive engagement, or physical comfort. Physiological computing [51] is a promising concept to widen the communication channel between the (human) users and computers, thus allowing an increase of software systems' contextual awareness of their user and rendering software systems smarter than they are today. Using physiological inputs in pervasive computing systems allows re-balancing the information asymmetry between the human user and the computer system, for the benefit of the user. Physiological computing (using physiological data as input for on-line computing) is different from pervasive computing in both the data it processes, and in the inferred concepts (e.g. mood, cognitive load, comfort) that influence computing processes.

Software frameworks and methodologies for the creation of pervasive systems and intelligent systems exist, but none of the existing framework tackle

the specific requirements that emerge from physiological computing, i.e. the specifics of the input data, and the requirements of the entailed behaviour adaptation of the software process. Systematic means for developing and moulding physiological computing applications from an initial idea or concept into a working software implementation are still lacking.

In this thesis, we therefore present a methodological approach to the creation of physiological computing systems based on the principle of component-based software engineering. Here, a component is a software entity that declares all services it requires and provides, so that the dependencies and provisions can be managed through a framework. Software components allow for coarse-grained management of software systems, and help to provide a clear picture of a software's architecture. This applies not only to physiological computing systems, but to software systems in general as well. However, physiological computing systems benefit specifically from a component-based approach, as it allows to leverage reconfigurations [85] as means to control and adapt behaviour. Reconfigurations are run-time changes in a software system configurations, and include adding and removing components, adding and removing links between component provisions and dependencies, as well as changing exposed component parameters. Through reconfigurations, a physiological computing system can adjust its behaviour both to the human and to the available computing environment in terms of resources and available devices – an activity that is crucial for complex physiological computing systems. With the help of components and reconfigurations, it is possible to structure the functionality of physiological computing applications in a way that makes them manageable, extensible, and separates meta-reasoning logic from normal operation logic, thus allowing a stepwise and systematic extension of a system's intelligence.

While reconfigurations are a useful tool in the creation of adaptive behaviour, they also entail problems. Fully understanding and capturing the behaviour of a system that reconfigures itself is difficult, as reconfigurations may alter the structure of a system in ways that are challenging to predict. Starting from an initial configuration, it may be easy to predict how the system may be altered by a single reconfiguration. It becomes harder to pinpoint the set of potential system configurations as the system endures more and more reconfiguration steps.

In this thesis, we propose an approach to the software engineering of physiological computing system that is based on components and reconfigurations, and allows to verify the correct behaviour of reconfigurations through

a formal assume-guarantee contract framework. Furthermore, we present a component-based software framework for implementing physiological computing systems, as well as case studies that give evidence on the applicability of the component frameworks – both formal and software-based. The proposed software engineering approach is comprehensive, and covers issues from the formal verification of reconfigurations as well as their implementation without losing track of in the intended domain of application: physiological computing.

An introduction to the software engineering approach to physiological computing is given in Section 1.1, before our core contributions are highlighted in Section 1.2. Our approach has been developed within the EU-Project REFLECT, and the major case study of the project was realised using the methodology presented in this thesis. We therefore present an overview of the REFLECT project in Section 1.3. Finally, Section 1.4 gives an outline of the structure of this thesis.

1.1 Approach

This thesis presents a comprehensive approach to the software engineering of physiological computing systems that is intended to guide the development from a first idea of a physiological computing system to an implementation, and help in overcoming some of the core challenges in this process. Our approach allows to establish a formal proof of correctness for reconfigurable systems, and hence allows physiological computing systems to leverage the benefits of reconfigurations without creating unmanageable system behaviour.

Figure 1.1 shows an overview of our approach to the software engineering of physiological computing system. The basic course of action is to create a system design and desired system behaviours from an initial idea (a process that is not covered in this thesis). Using the formal verification framework proposed, a proof of refinement between the desired system behaviours and the system design can be established. Furthermore, the system design can be transferred with relative ease to an implementation using the concepts and features our component-based software framework offers. This basic process is not progressing independently, but is operating in the context of Scrum [107], an agile software development methodology. Physiological computing is another a background theme of this process that makes use of

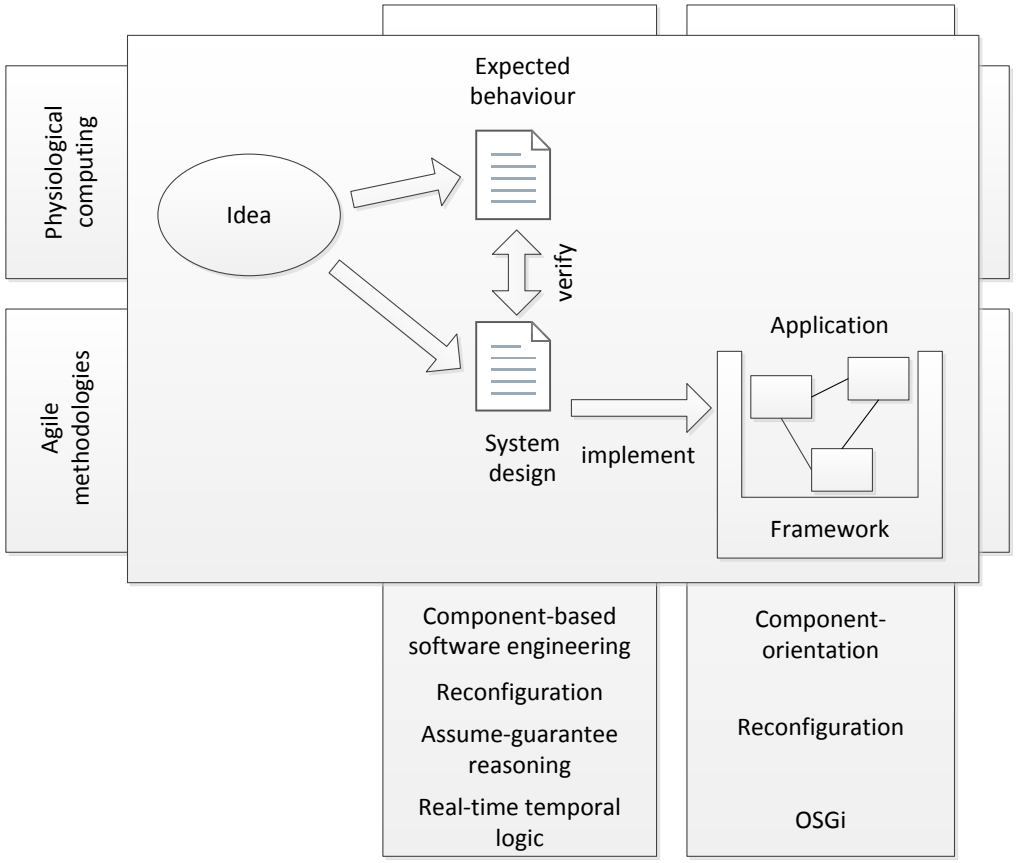


Figure 1.1: Software engineering for physiological computing

several formal techniques (assume-guarantee reasoning, real-time temporal logic, component-based software engineering, reconfigurations), and implementation techniques (component-orientation, reconfigurations, and OSGi, as depicted in Figure 1.1).

Details and benefits of each element of our approach is given in the following.

1.1.1 Methodology

Physiological computing is an application domain that still features heavy research activities. Therefore, software developers operating in this domain

require not only an appropriate introduction in the domain of physiological computing, but also a guiding methodology providing a reference frame for the realisation of physiological computing applications. In addition to a solid methodological shell for finer-grained tools and processes, software developers also need examples and patterns to showcase the use of tools and methods.

Our methodology provides both a shell for the formal verification framework and the transition to implementation activities, as well as patterns for the use of reconfigurations in physiological computing applications. The presented methodology is based on Scrum [107], an agile software development methodology that is particularly suitable for new product development and research-focused projects. Especially, we show how agile software development can be combined with formal verification activities, as formal verification of systems under reconfigurations is a crucial part of our approach. Furthermore, we introduce two application patterns for reconfiguration in physiological computing that showcase both the formal verification process and the transition to implementation on the component-based software framework.

With the help of our methodology, we are able to combine an agile process with formal verification activities, and provide helpful guidance for software developers through a minimal set of rules. Furthermore, reconfiguration patterns provide insights on the impact and consequences of the application of reconfigurations as well as application examples for our formal verification framework and the component-based software framework.

1.1.2 Implementation

Implementing physiological computing applications is an endeavour that can benefit significantly from a software framework. Data handling, concurrent process organisation and communication, distribution, and reconfiguration (requiring software monitoring and software modification) are complex aspects that require solid consideration and implementation.

Our component-based software framework, the REFLECT framework, provides a simple and powerful component model that focuses on developer support, natural embedding in its host language Java, and favourable scaling behaviour from restricted resource environments to mutlicore systems.

The REFLECT framework provides functionality for distribution, handling of physiological data, and concurrency, which allows the software developer to concentrate on creating the physiological computing application business logic, to experiment with it, and to build up an understanding on

the peculiarities of his application along the way.

1.1.3 Formal verification

Reconfigurations are a useful tool in the creation of complex physiological computing applications. As detailed before, however, it is difficult to keep track of all possible system modifications, and the entailed system behaviour. This fact makes software engineers reluctant towards using reconfigurations for adapting the behaviour of a system, even though it would allow to elegantly separate meta-reasoning on the system behaviour from the system behaviour itself. In the context of physiological computing, system behaviour (and transitively, reconfigurations as well) are additionally dependent on the continuous, real-time behaviour of the environment, making the application of canonical temporal formalisms harder to apply.

In this thesis, we introduce an assume-guarantee-based contract framework and an extension of metric interval temporal logic to compositions that allows specifying the real-time behaviour of components, environments, and reconfiguration. With its theory of composition and refinement that covers structural modifications, the framework allows to establish refinement proofs of systems under reconfigurations, and hence to prove that a physiological computing system using reconfigurations behaves as expected.

1.2 Contributions

This thesis presents a comprehensive approach to the software engineering of physiological computing application. In particular, it includes a component-based software framework, a formal contract-based framework for the verification of systems under reconfigurations, and a software development approach that combines the two elements in an agile software development methodology.

- The contribution in the *implementation* domain lies in the introduction of a component-based software framework for physiological computing systems. The REFLECT framework provides a carefully chosen set of features that integrate well with its host language, Java. The REFLECT framework builds on OSGi as implementation platform, and port-based components as the underlying implementation concept. The

features offered by the REFLECT framework simplify coping with repeatedly surfacing challenges and tasks by providing ready-to-use facilities and services.

- The contributions with respect to *formal verification* lies in the presentation of a assume-guarantee framework with dense real-time semantics for the verification of systems under reconfigurations. The framework is general in the sense that it is not restricted to physiological computing applications, but can be applied to any system undergoing reconfigurations and featuring real-time behaviour. Given the real-time character of physiological computing applications and the utility of reconfigurations for those applications, our formal verification framework constitutes a perfectly matching tool for the early analysis of physiological computing system designs. The formal verification framework offers both a contract-based dense real-time semantics framework that allows for composition and refinement of components, as well as a syntax for the specification of contracts, and a set of rules for the manual elaboration of refinement proofs.

- Our contribution to software development *methodology* consists of a proposal for combining development activities resulting from the application of our formal verification framework, and software implementation tasks that emerge from the use of the REFLECT framework. We propose a scheme on how both activities can be successfully combined and managed in Scrum, an agile software development methodology. Furthermore, we provide two application patterns for reconfigurations that both showcases the use of reconfigurations in physiological computing, and demonstrate the synergies of combining our formal verification framework and the REFLECT framework.

Finally, we provide two extensive case studies that show the applicability and utility of both our formal verification framework for systems under reconfigurations, and our component-based software framework. The first of the case studies presented it the automotive demonstrator of the REFLECT project that showcases the use of the REFLECT framework for the development of physiological computing applications. The second case study presented demonstrates the use of our contract-based verification framework, and is based on the adaptive advertising scenario – an advertisement that features awareness of its audience, and reacts and responds to its viewers.

1.3 Reflect

The presented approach to software engineering of physiological computing was developed by the author during the REFLECT project, a "Specific Targeted Research Project" (STREP) that started January 2008 and ended March 2011, with a project funding of 2.6 million Euro. The goal of the REFLECT project was to research new concepts and means in software and psychology for the creation of adaptive systems that sense the user's presence, mood, and intentions. The envisioned systems are intended to take into account at least three aspects defining the human condition: emotions (e.g. annoyance or anger), cognitive engagement (e.g. boredom or overload), and physical conditions and actions (e.g. comfort and physical workload).

The goal of the REFLECT project has been to push the frontier in physiological computing along two major lines:

1. support for software engineering activities involved in the creation of pervasive physiological computing systems through frameworks and tools.

2. researching in the concepts for creation of a user's state from physiological data, and in means for influencing a user's state and adapt the environment to the user's current context.

The contributions of this thesis stem from work revolving around the first major line of research: creating tools and frameworks for pervasive physiological computing applications. The software framework presented in this thesis, the REFLECT framework, constituted the core framework on which the software development activities of the REFLECT project built upon. Additionally, the contract-based verification framework we present in this thesis is one of the main contributions of the REFLECT project in the domain of software engineering.

1.4 Structure

This thesis covers all contributed topics listed in Section 1.2. First however, this thesis introduces the background for each contribution in Chapter 2. This chapter presents more details about physiological computing, component-based software, assume-guarantee contract-based verification, and

about software development methodologies. Chapters 3, 4, and 5 detail our main contributions: first, Chapter 3 presents the REFLECT component framework, starting from initial requirements of the physiological computing domain, progressing by introducing its underlying concepts, and finally presenting details of its implementation, use, and more detailed rationale and insights of each framework feature. Next, Chapter 4 tackles the issue of formally verifying component-based systems under reconfigurations by introducing a formal assume-guarantee contract framework that is able to handle reconfigurations natively. Finally, Chapter 5 shows how both the REFLECT framework and our contract-based verification framework can be combined, and how both contributions can be applied to two recurring usage patterns of reconfiguration.

After presenting our our main contributions, we continue with introducing two larger case studies in Chapter 6 that demonstrate the applicability of the REFLECT framework and our formal verification framework to larger systems.

This thesis concludes with Chapter 7 that provides a summary of our contributions, a review of their accomplishments, and presents possible lines of research that emerge from our work.

Chapter 2

Background

Before presenting the main work of this thesis, it is necessary to set the stage and present the foundations on which this thesis builds on. Since we present a component-based software engineering approach to the development of physiological computing applications, this chapter starts with an introduction to physiological computing in Section 2.1. Since we present an component-based software framework for the implementation of physiological computing applications as component-based software systems, we present the underlying concepts and terms of component-orientation in Section 2.2, as well as the platform on which our framework implementation relies on (OSGi) in Section 2.3. A theoretical framework for contract-based verification of real-time systems is presented next in Section 2.4; it sets the stage for our contract-based framework for systems under reconfiguration introduced in Chapter 4. As this thesis aims at presenting a coherent approach to the development of physiological computing systems, we also look at software development methodologies to find a frame in which all parts – both the theoretical verification approach, and the implementation approach – integrate (Section 2.5). Finally, this chapter concludes in Section 2.6.

2.1 Physiological computing

Today's computing systems and infrastructures are constantly failing to satisfy the increasing expectations of everyday users. This inability has several causes, one of which is the asymmetry and shortcomings of current communication channels between computer systems and users [89]. While the

output of computer systems features a high bandwidth (visuals, audio, even tactile vibrations are used as output channels), the input to computers is still fairly limited. Even though touch screens are beginning to enter everyday use through smartphones and tablets, communication paths that are used in human-to-human interactions through speech, facial expression and gesture are not available in human-computer interaction. Instead, computers mostly offer keyboard and mouse as input devices. Together with touch screens and general forms of button-pushing, they constitute virtually the complete set of input possibilities that modern computer systems offer. It is a very limited input channel, still, and entails that computer systems remain unaware of the environment, the state of the user and his goals. This unawareness is often perceived as stubbornness and dullness. As users learn to adapt to this behaviour of their computer, an odd phenomenon can be observed: the command chain between computer and user is virtually inverted [89, 51]. The flexible and smart user subdues himself to the obstinate and dense computer in order to achieve his own goal with the help of the limited machine available. Giving the computer awareness of the context and user will alleviate the imbalance of communication between user and machine and the entailed inversion of command. Computers that are aware of the user and the environment can be made able to detect their own wrong-doings easier and gain the possibility to compensate their actions. Extending the communication bandwidth from the user to the computer also promises to offer means for capturing motivations and goals of the user even on a subconscious level. This would allow to foresee a user's decision and conflicts with the current system operation and to adjust the system operation smoothly, making interaction with the computing facilities that become available in the everyday environment of the user more enjoyable.

In the following, we first present a definition of physiological computing, its adjacent research areas and their possible synergies (Section 2.1.1). Next, we take a closer look at the research domains that physiological computing encompasses, and which application scenarios are motivating each research domain in Section 2.1.2. Section 2.1.3 sheds more light on software design concepts for physiological computing, before Section 2.1.4 motivates the need for a more systematic overall software engineering approach to physiological computing. As becomes apparent through the definition of physiological computing (Section 2.1.1), it is unwise to discuss physiological computing applications without highlighting ethical implications physiological computing entails. Section 2.1.5 details on the ethical implications before Section 2.1.6

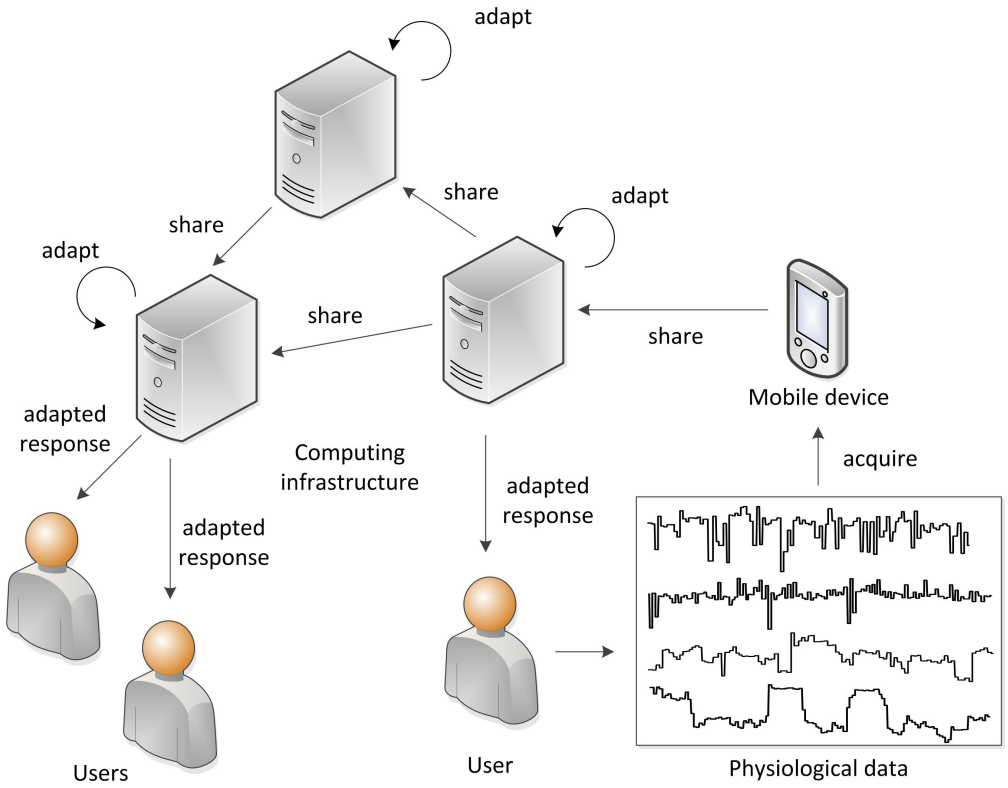


Figure 2.1: Physiological computing – basic scheme

provides a brief summary.

2.1.1 Definition and origin

Physiological computing [51] is a relatively new area of research that tries to provide more input to computing systems – namely, physiological input – in order to alleviate the communication asymmetry that state-of-the-art computing systems feature. It finds itself at the intersection between artificial intelligence, human-computer interaction and applied psychology. The term of physiological computing itself is integrative since it denotes all research about computing systems that measure physiological signals from the human body in with the intent of processing it in real-time and leverage it in on-line operation; physiological computing is defined by on-line processing

of physiological data. Figure 2.1 shows the basic scheme that a physiological computing system implements: typically using a mobile device in the proximity of the user, the system captures physiological data for processing it either locally, in its computing infrastructure, or both. Results of physiological data processing can influence the behaviour of the system towards the source user itself, or other users as well.

While the definition of physiological computing does not specify any kind of computing infrastructure it should be running on, a convenient infrastructure for physiological computing are not desktop computers, but computing infrastructures that permeate the environment and are capable of controlling and altering parameters of the environment such as illumination, temperature, and background music, as well as providing contextual information directly to the user. Those infrastructures also have the benefit of allowing access to physiological parameters, as part of the infrastructure may become wearable as gadgets or jewels, integrated in clothes and even in the human body [15]. At the same time, a computing infrastructure that offers software controls for a significant part of the environment promise to provide meaningful means for improving the experience of the user.

Such computing infrastructures have partially become reality already and will continue to invade our environment due to the continued improvements in computer hardware performance and in device miniaturisation. Systems that provide computing power in our everyday environment – in a similar fashion as electric power is provided virtually everywhere today – were coined ubiquitous computing systems about a decade ago [120, 103, 82].

A further step in ubiquitous computing research (that is also coined pervasive computing) is pervasive adaptation [96], which is a research branch that was created based on the insight that ubiquitous computing faces similar challenges as artificial intelligence research faced years ago [82]. Hence, bringing expertise from artificial intelligence into the ubiquitous computing research community seems to be a promising approach for accelerating research progress. In the research domain of pervasive adaptation, system adaptation to the user's needs, habits, and emotions constitutes a key research focus [96]. The convergence of pervasive adaptation and physiological computing would allow to sense the user's emotions and current cognitive engagement on a more informed basis, and to adapt the environment in a satisfactory way. The overall goal of applications that leverage both pervasive adaptive and physiological computing concepts would be to control and adapt the environment of a user, and by this adaptation to influence and im-

prove the well-being of its user. This is achieved by changing the parameters of his environment where influencing these parameters is possible.

By the fact alone that they use physiological data, physiological computing systems share a set of major challenges. First and most obviously, there is the on-line processing challenge that physiological input bears. The raw physiological input data is contains noise and perturbations (also known as artefacts) due to interfering signal sources, the user's movement, temperature changes, or generally, influences in the environment that are unrelated with the quantity to be measured. Also, there is no simple one-to-one mapping between physiological data and psychological state: a single change in physiological data can be caused by several changes in physiological or psychological state [51]. For example, increase in heart rate can be either explained by increased physical activity, or increased cognitive effort performed by the measured person. Physiological data constitutes personalised data of singular quality; processing it in computing infrastructures entails a plethora of critical ethical issues, as discussed in Section 2.1.5. Finally, developing physiological computing applications from a first idea to a working implementation is hard – due to the above-mentioned issues, but also because the software design for physiological computing applications poses its own challenges: how can a software system be structured in a way that it exhibits a satisfactory behaviour towards the user is a question that is discussed in Section 2.1.4.

2.1.2 Research domains and applications

The term physiological computing is integrative in its nature; it includes efforts from several streams of research such as brain-computer-interface (BCI) [118] and affective computing [97, 98], as both research streams use physiological data from the human body and uses the data in computing responses, or modifying computation processes. BCI research aims at using information from the human brain to provide applications with more contextual information. Contextual information, to be more precise, encompasses user's activities and goals, where the user goals may also be commands directed towards the BCI-enhanced application as in the case control of commands towards a prosthetic hand. BCI applications are not only limited to prosthetics, however: BCI-provided contextual information can be used to create or enhance assisting, recreational, and diagnostics applications.

Physiological computing additionally encompasses research performed in

affective computing [97, 98], which is aimed at making systems and applications aware of human emotions and competent about them. Affective computing systems are envisioned not only to be aware of the user's emotions and desired emotional states, but as well to be well informed about means and strategies to move the user from one state to another in a sensible and acceptable way. In that sense, affective computing aims at making computer systems understand emotional states of the user, and provide a role model for emotional competences such as understanding emotions, mitigating emotions as well as dealing with emotions.

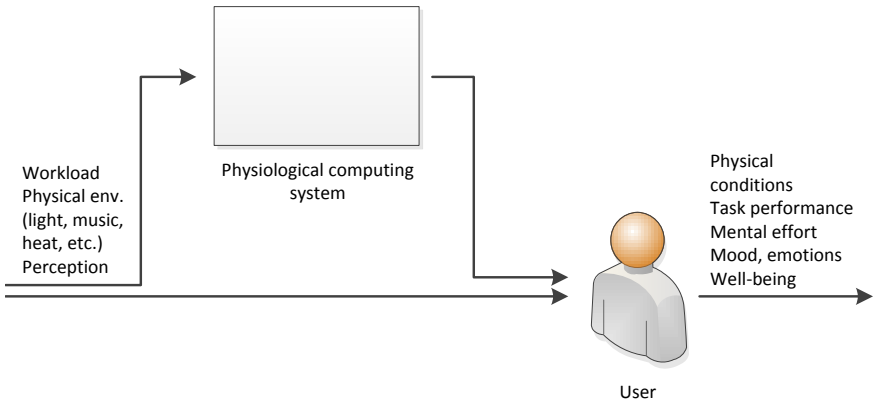
How comes, however, that a whole research area was coined affective computing? There is a very cogent reason for this, which is also exposed in an own book by Damasio: Descartes' Error [43] shows how logical reasoning and emotion is deeply linked in the human's mind; not only are decision biased by emotions – emotions are necessary for rational thinking. Indeed, patients that are lacking the ability of emotional reflection due to irreparable brain injury (as the famous case of Elliot [43] shows, for example), make wrong decisions. They continue to trust untrustworthy people, or repeatedly lose money in financial investments in which they already lost money. In a way, people without emotions are lacking crucial guiding heuristics helping them to make correct rational decisions. Given that emotions have such a deep linking with rational decision making and general importance for the single individual, it is therefore not surprising that affective computing emerged as an own research stream. As does BCI, affective computing promises to enhance existing applications and to enable the creation of new applications [97]: Applications that mirror emotions in order to provide biofeedback to the user and allow to train emotional competences are one kind of applications that can be created once emotion recognition becomes reliable technology. Video conferencing may be enhanced with emotional channels in order to let communication partners convey emotional cues in addition to what is said. Self-teaching or e-learning systems may benefit greatly from affective information. Keeping track of the emotional state of the learner, distinguishing disengaged from engaged learning situations and providing appropriate responses is a key factor for effective learning, and a skill good teachers (both human and artificial) have to master. Computer games may also benefit from affective computing to create new entertaining and challenging puzzles. In current computer games, it is of lesser importance how an action is performed, as long as a certain outcome is achieved. With affective computing (or, more generally, with physiological computing as well), the

physiology of the player can be taken into account, and actions performed may get different impact on the game world based on the emotional state the user is currently in. An action performed in a meditative or calm state may become more precise, or the effect of an action performed in an excited state may have greater impact on the game world.

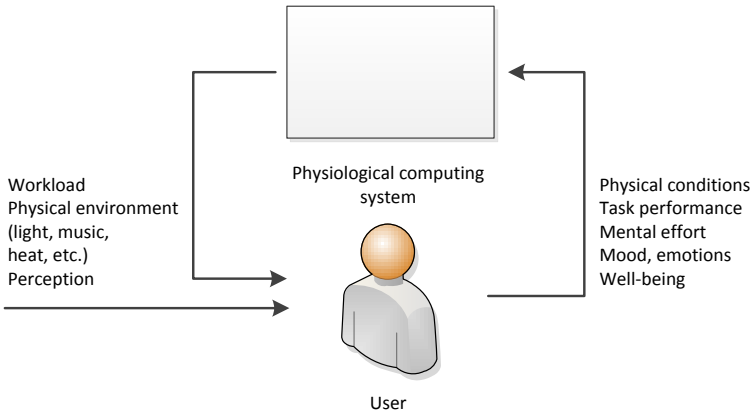
Not only healthy or average people can benefit from affective computing applications, however. Text to speech applications for handicapped, for example, may be improved to create a more natural speech melody by using affective information from the writer – either at the time of the writing, or at the time of the speech generation. Autistic people can also greatly benefit from affective computing. Autism is characterised by difficulties in emotional understanding; autistic people often have reduced empathy and find it hard to produce appropriate emotional responses in social situations. As autistic people are (as all human) avid learners, it is possible to teach them empathy and correct emotional responses [97]. However, their lack of basic emotional skills and capabilities to abstract from single examples of emotional reactions to general patterns makes teaching autistic people endlessly repetitive, as they need to be trained and guided through uncountable instances of example situations and correct reactions discussed; this is were affective computing approaches may help. Using computers as learning environments that are able to assess the emotional response of the user would feature both endless patience and competence to provide the user with a correct assessment of his emotional response.

2.1.3 Feedback design

In theory, it would be possible to realise physiological computing systems in two ways: as a feed-forward system (see Figure 2.2a), or as feedback system (see Figure 2.2b, also called closed control loop). The key distinctive factor between the two approaches are the conditions under which they can operate. A feed forward system uses information from the environment, but no information from the system that should be controlled; the effects of the environment on the system are computed using a-priori knowledge about the impact of the environment on the system, and corrective actions are made. The effects of the environment and the corrective actions on the system under control are never measured, as all effects are known and designed into the feed forward system. An example for a feed forward system would be an indoor temperature control device that doesn't measure the indoor temperature to



(a) Feed forward loop



(b) Feedback loop

Figure 2.2: Physiological computing systems and the user

determine the level of heating it needs to provide, but deduces it from the external temperature and solar radiation intensity. Such a system can operate correctly only if the effect of external temperature and solar radiation on the internal temperature are known, and no other major factors influencing the room temperature exist. For example, opening doors or windows would not alter the behaviour of such a feed forward heating control system; the room temperature would be greatly altered, but the system wouldn't respond.

A feedback control system on the other hand does not try to observe and measure the environment, but instead observes the controlled system, and feeds back the distance of the current state it finds it from the target state (i.e. the error) into the control of the system through a means that alters the state of the system. Going back to the heating example, a feedback loop equates to a thermostat control that opens its valve if the room temperature drops below the desired temperature, and closes the valve if the room temperature reaches (or surpasses) the desired level.

Looking back from the simple example to physiological computing, it is clear that it is almost impossible to influence the well-being of a user using a feed forward control scheme (Figure 2.2a) due to the fact that very few is known about the influences of the environment on a user. Events, interactions, anticipated future events may all affect the well-being of the user, but may remain invisible or unknown to the physiological computing system. These circumstances mandate the use of a feedback control scheme (Figure 2.2b), as a feedback control system is able to cope with disturbances from unknown sources and can try to make corrective actions towards the desired state. In order to use feedback control in physiological computing, it is necessary to infer corrective actions from measures of the human body. In fact, physiological computing research focuses heavily on the feedback paradigm, even to the point of not even considering and ruling out feed forward control [51]. Physiological computing applications are thought of as bio-cybernetic (feedback) loops [52, 111] that sense the human's physiology, analyses it, and reacts accordingly.

While a feed-forward physiological computing loop would need to know how the environment affects the user, a feedback loop needs to know how physiological input maps to the user's state in terms of cognitive, emotional or physical (e.g. physical exhaustion or comfort) conditions. However, the mappings from physiology to psychological constructs that are usable in autonomous systems are still not settled. Physiological computing research is hence also preoccupied with solving issues of finding and validating mappings

that can be operationalised in software [51].

The use of the terms feed-forward and feedback in physiological computing is inspired by control theory, but is has no similar mathematical foundation; the terms are used in a very loose, hand-waving fashion. In physiological computing, it is sufficient to have means that potentially influence the observed parameters into desired directions to create a feedback loop; for example, systems providing visualisations to the user about their physiological conditions are already coined biofeedback systems. Such systems find application clinical treatments against migraines, and chronic pain, as well as stress management therapy [109].

2.1.4 Systematic software engineering

The underlying vision of a successful physiological computing system needs to be more than a simple feedback loop that tries to minimise an error value. Such feedback loops are often perceived as simplistic, as humans – being avid learners – rapidly understand the operational scheme of the feedback loop and adapt to it consciously and subconsciously. Applications that are intended to successfully monitor and adjust to humans need to operate like a good DJ or film; they need to create an (acceptable) suspense and narrative flow. Therefore, a physiological computing feedback loop needs to alter its control regime and target values continuously; a second level of control and adaptation beyond the basic feedback loop is needed [51]. This is one aspect of physiological computing that suggests a systematic approach to the design and development of software for physiological computing systems. For this second level of adaptation, a control concept that operates on the level of software is required. For the systematic development of this second level of control, existing concepts of component-based reconfiguration can be leveraged. Components allow to package behaviour and functionality into interacting modules of code, while reconfigurations allow to adjust and change the behaviour at run-time.

Another aspect that argues for a systematic approach is the need for distribution that most physiological computing systems have. In order to be usable and embedded in the everyday environment, the analysis, computation, and actuation logic will have to be distributed over several devices. Actuators may also be distributed in the environment, and only accessible remotely. At the same time, an application may need to aggregate data from multiple distributed sources and create a more complete picture of the

system's physical and physiological context. From the perspective of communication, distribution, and ubiquity, physiological computing can be seen as a special variant of ubiquitous computing or pervasive computing.

Besides the algorithmic and distribution challenges, organisational challenges also arise when building physiological computing applications. Frameworks for such applications must support different activities, namely 1) exploring and experimenting with new algorithms and new techniques, 2) testing and validation of the system and its parts, and 3) rapid creation of software artefacts.

Exploration and experimentation and *testing and validation* must be supported since the algorithms that will actually allow to extract knowledge about the emotional, cognitive, and physical state of a human from basic sensors and features are, to a large extent, still fields of active research [51]. A software design or software framework must therefore support algorithm exchange and experimentation at low cost. *Rapid creation of software artefacts* is crucial while still being in the process of understanding the application domain and identifying the hard problems within it. Even later, while creating a product, being able to swiftly create the software artefacts may turn out as crucial for reducing the product's time to market.

2.1.5 Ethical implications

Physiological computing has potential for a deep transformational impact on society, and hence require ethical considerations. Unfortunately, conclusive and satisfactory answers to ethical questions raised through physiological computing were not yet found. While Norman [89] urges researchers and designers to focus on the empowerment of the user when creating and designing things and systems, the real implications of the continued exposition to physiological computing systems are unknown for now [51]. Nevertheless, the vectors of endangerment that physiological computing systems feature can be clearly identified:

- **Privacy.** If physiological computing becomes as non-intrusive as envisioned, and physiological data acquisition finds adoption in environments that are not under the control of the user (such as public and work environments), severe privacy issues entail. Physiological computing allows to access data with a completely new quality, as the accessible data stems from the human body itself. Additionally, phys-

iological data is hard to constantly manipulate,[1] and will be used for detailed and precise analysis of the user's condition if the vision of physiological computing becomes a reality. Although having a computer constantly monitor the human body is at least questionable from a privacy point of view, humans can accept and feel comfortable with such situations as long as they are in full control of the system [99], or experience a symmetry of communication, power and accountability – a concept that is called reciprocal accountability [34].

- **Autonomy.** Not only privacy is at risk when introducing physiological computing systems, but also individual autonomy [51]. The issue is not that physiological computing systems aim to manipulate the user's state; a plethora of techniques and means are willfully used by people to manipulate their emotional and cognitive states, e.g. drinking coffee, listening to music, reading, watching movies, hiking and sports in general. The danger to autonomy is given by the potential transfer of control that computing system allow: the controller of the physiological computing system can be separated from the individual that is experiencing the manipulation. Keeping the user of a physiological computing system in control of the system he exposes himself is therefore a norm that must be established in physiological computing [89].

- **Integrity of self.** Many physiological computing designs use cues that the human body itself uses for self-assessment of its condition [51]. Individuals perceive feelings and emotions through aggregation and evaluation of somatic markers and cues [44]. Providing another source of self-assessment through a physiological computing system may create a conflict with the natural self-assessment through different interpretations of available information – or, even simpler, through plain wrong interpretation of available data. It has been argued that sustained exposure to contradicting and interfering sources of self-assessment may cause a fracture of the unitary experience of self [66], and may thereby cause mental illnesses. However, the long term effects of constant exposure to physiological computing applications are not well understood;

[1] Short-term manipulation of physiological data is feasible. Willfully thinking of certain events, experiencing pain, or performing physical exercises can be used to "jam" physiological sensors. However, none of the available jamming options are activities that can be performed for a prolonged duration.

potential impact of physiological computing on mental health is an uncharted area of research as for now [51].

2.1.6 Summary

Physiological computing is an interesting topic for research, not only from a psychological perspective, but also from a computer science perspective. From the practical point of view of ubiquitous computing, physiological computing offers new sources of data, poses new questions and generates new challenges, especially in the domain of privacy. From a methodological point of view, physiological computing offers a lot of challenges as well. Especially, the question arises how such systems can be designed and developed systematically. In the following, this thesis tries to address the issue of systematic development by proposing a component-based approach (consisting both of a software implementation framework and a formal verification framework) embedded in an agile process to form a consistent methodology.

2.2 Component-based software

Successful physiological computing systems need to be entertaining and versatile; they need to guide the user through several states, and must have a convincing and entertaining devolution. To achieve this goal, a physiological computing application needs to adapt its behaviour on a strategic level. This strategic change in application behaviour can be achieved by encapsulating behaviours into components and switching between them, as well as by changing well-defined parameters of behavioural components. This approach follows the idea of component-based reconfiguration, for which a formal theory is also presented in this thesis. Here, we present the existing terminology in the domain of component-based software engineering.

Component-based software engineering is an approach to the design and implementation of software systems that revolves around the notion of components. Both verification and realisation benefit from the designing and implementing of software systems on the basis of components:

1. **Verification.** A component-based system design allows for a divide-and-conquer approach to the formal verification of systems [25], as it allows to define specifications of system parts (i.e. components) as

intermediate verification goals. In a second step, the component specifications can be used to prove a global system specification, thereby establishing the correctness of the global system.

2. **Realisation.** A component-based approach can be used when realising a software system. Realisation of systems often needs distribution of work amongst several developers. Assigning responsibilities for the realisation of system parts (i.e. components) to developer teams is one divide-and-conquer approach to work distribution. This approach however requires that the responsibilities of components and the communication between components are well understood. Separating a system into components with well defined responsibilities and communication can also make modifying the system easier. It requires however the correct design of communication and encapsulation of system internals. A system is not automatically ready for change through splitting it up into components; it requires additional careful design and correct anticipation of points of variability.

It is important to note that the requirements for formal verification and practical realisation to a component model diverge. While of course a component model is deemed necessary for encapsulating behaviour in both usage scenarios, several requirements are contradictory. For example, practical realisation requires a feature-rich component model that is difficult to handle by mathematical models. On the other side, while divide-and-conquer approaches to formal verification benefit greatly from hierarchical composition of component systems [94], such hierarchical composition is often unnecessary and constitutes a painfully noticeable overhead in most system implementations.

Component-based software engineering is a software engineering concept that recently has been used as the foundation for several widespread open-source software development frameworks. In these frameworks that build on widespread programming languages such as Java, components are native software entities and are at the foundation of the framework's programming model. The main motivations of component-based software frameworks are twofold:

1. Providing a simple programming model that supports the software engineer in defining clear responsibilities, interfaces and in enforcing separation of concerns.

2. Preventing architecture erosion when transitioning from a software design to a software implementation.

Before delving into the details of each framework and concepts, the terminology that is used in the communities of component-base software frameworks needs to be presented and explained. The terminology that is presented in the following is based on the terminology introduced by Szyperski in [117]. Not all concepts explained here are found in Szyperski's book, however, and sometimes, the terminology was altered to provide a more natural description for the component frameworks presented is this thesis.

Framework A framework defines an architectural style for object-oriented systems by providing a set of classes constituting both a programming interface and a programming model [117]. Some of the provided classes are open for extension through the programmer; the extensions of the programmer are however plugged into the general architecture that the framework provides, and by this enforce a specific programming style – often through the restricted means of interaction it offers to the developer. Frameworks are ubiquitous in web application development, where they define a structure for the processing of http requests and the creation of dynamic html pages through access to a persistent storage like a SQL database. In the context of component-based software, the framework concept is often used to offer a component-based programming model and means for encapsulating code and state into components.

Provision A provision, also called service or incoming interface [117], is a service an object offers to its environment, i.e. to other objects. A provision in Java often consists of an interface declaring the functionality provided, and an implementation of that interface that is made accessible to others.

Dependency If an object requires a service, this requirement is a dependency to another service (also called outgoing interface [117]). A dependency can be satisfied through an appropriate provision from another object. In the Java context, a dependency is often defined by the need of an interface implementation. A dependency may be declared through a class field that needs to be assigned through calling a setter before the class can operate as expected. In the context of web development for example, it often arises that an http request processing class has a dependency to the database interface.

This dependency needs to be satisfied before the request processing class can process any http requests.

Component A (software) component [117] is a code entity that strictly defines dependencies to its environments and follows all contractual rules specified by the component framework it adheres to. Additionally, components may specify services it provides to its environment – mostly for use by other components. In the Java context, a component is often declared by a class. This component class either extends a specific base class or is made to a component through specific declarations in configuration files or code.

Connection, binding A connection is a link from a component dependency to the provision of another component [117]. The connection declares how the dependency of one component is satisfied through the provision of another component. The terms connection and binding are used interchangeably in the following depending on whether the link itself is highlighted (connection) or the provision and dependency (binding).

Component framework A component framework defines a programming model revolving around the notion of encapsulation [117]. Component frameworks typically define means of interaction between components and means for controlling parallelism. Some component frameworks allow for hierarchical composition of components, in which a component can be created from interconnected components. In these composite components, both the internal components and the internal connectors adhere to the rules of the component framework. In such hierarchical component frameworks, the resulting composite component can be treated as component again, making it possible to create arbitrary deep component nestings.

Inversion of control Inversion of control [55] is a communication pattern used in frameworks where framework client code provides control flow entry points (such as start, stop methods) to the framework that the framework may call. The client code itself is intended to be reactive to the framework instead of proactive; the client code may not initiate communication with the framework, and must instead wait to be called by the framework. This communication pattern is also called the “Hollywood principle” based on the

well known phrase “don’t call us, we’ll call you” that precisely describes the idea of inversion of control.

Dependency injection Dependency injection [55] describes a means of satisfying dependencies in which the code stating a dependency does not try to actively resolve the dependency by e.g. looking up a registry. Instead, the dependency is resolved by an external entity that injects a provision into a dependency. In object oriented languages, this injection process can be performed in three different ways. **Constructor injection** denotes injection into the constructor. In constructor injection, the dependency resolving entity is also responsible for creating instances. **Method injection** or **setter injection** denotes injection through methods or setters. It requires either external declaration of injection methods, annotations of injection methods or a convention on the signature and naming of injection methods. The third injection method, **field injection** assigns values to fields directly in order to inject provisions.

Configuration The configuration of a component-based software system [117] consists of three parts: the definition of component types, component instances and bindings. Component types definition are straightforward: a component type definition declares an instantiable component type. A component instance definition advises the component framework to create an instance of a component type with a specific set of parameters. The definition of bindings declares how dependencies of component instances are to be satisfied by provisions of other component instances. Note that instance and binding definitions can be performed in very different ways: definitions can be pre-defined by the framework, may be defined explicitly, e.g. through listing them in a configuration file, or implicitly, i.e. by defining rules that are applied by a rule-based engine. To discuss the different approaches to instantiation and binding, we speak of **instantiation regimes** and **binding regimes**.

Reconfiguration Reconfiguration [85] denotes the act of changing the configuration of a component-based system at runtime. The configuration of a component-based system is thereby understood as consisting of the currently existing components, primitive configuration parameters that components expose, and the connectors between components. The reconfiguration of a

component-based system may therefore create or remove components from the system, change configuration parameters of components, or change connectors between components by either adding or removing connectors, or changing source and targets of single connectors.

POJO a Plain Old Java Object [53] is a Java object (more precisely, a Java class) that has no dependency to the framework on which it is deployed on. A framework supporting POJOs oftentimes uses configuration files, conventions and reflection to instantiate and invoke methods of POJOs accordingly. The goal of POJOs is to reduce dependencies of code to frameworks, and by this to make code reusable in different contexts.

Java Annotations [63, pp. 281] is a Java programming language feature that allows to annotate code with meta-information. Annotations can be added to classes, interfaces, methods, fields, method parameters, and variable declarations. Using annotations, it is possible to have meta-information and code residing in one location, significantly reducing maintenance efforts and increasing refactoring robustness of systems.

Domain Specific Language (DSL) A domain specific language [57] is a custom programming language that is purposefully restricted to a specific application domain. This restriction allows to enrich the semantics of the language in a way that is meaningful for the application domain, since its semantics need not be usable for generic programming. Domain specific languages come in two styles: **internal** and **external**. External DSL resemble classical programming language in their tool support and usage: they come with an own syntax and semantics, compiler or interpreter; some may feature editors with syntax highlighting and programming support. They live independently from other programming languages. Internal DSLs on the other hand resemble programming libraries: they reside within a host programming language, and are defined by means of library code of a specific flavour. An internal DSL defines a programming interface in terms of methods or procedures, but takes extra efforts in providing a simple and natural interface that allows to construct meaningful method chains that read just like (domain specific) programming statements [57]. For example, an internal domain specific language for state machines may allow to write a sentence like `from(lightOff).transition("push", call("lighton")).to(lighOn).`

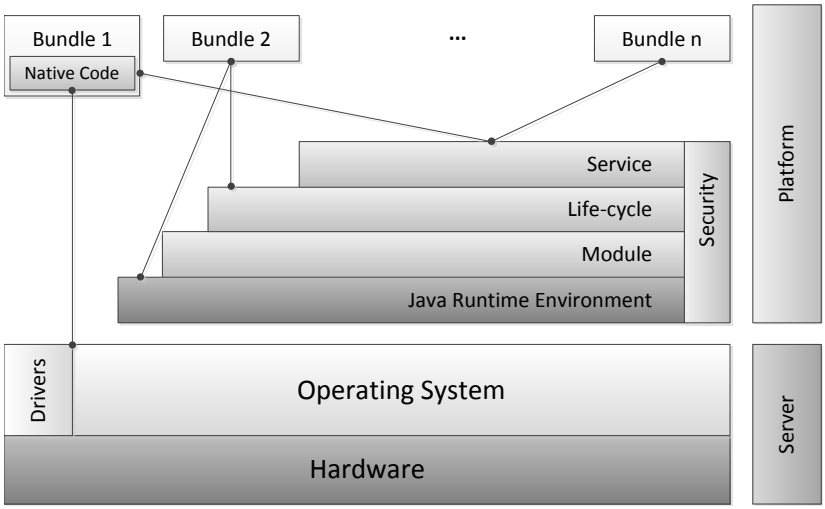


Figure 2.3: OSGi layers

The terminology of component-based software frameworks presented here introduced both abstract concepts as well as techniques and technologies that are found in existing Java component frameworks. In Chapter 3, we use this terminology to present our own component framework and compare it with others. The next section, similar to this section, introduces the terminology and background needed for the theoretical part of this work in a more rigorous fashion.

2.3 OSGi

The software framework that is presented in this thesis relies on the OSGi platform. This section introduces the OSGi platform, a service-oriented module system for the Java programming language [63]. The frameworks that implement the platform defined in the OSGi specification [92] must provide an environment for modularisation, service discovery, and service provisioning. The OSGi platform manages *bundles* as basic entities that provide *services*.

The basic module system that OSGi provides is a solid foundation for a component-based software development framework. As Section 3.4 discusses, several component frameworks rely on OSGi to provide (at least) type-level

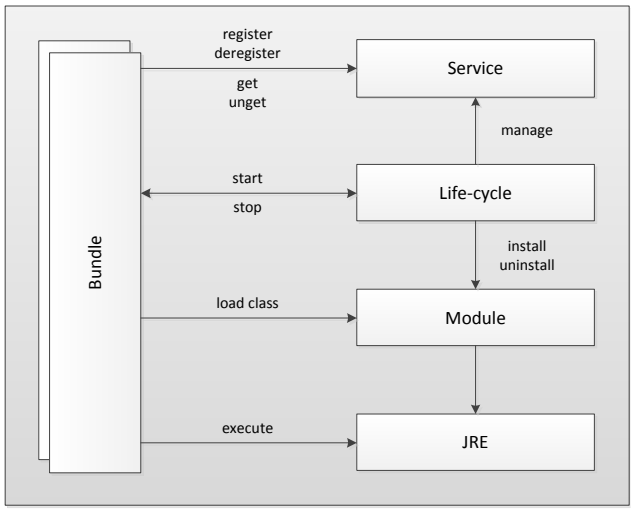


Figure 2.4: OSGi layer interactions

modularity. Type-level modularity thereby denotes the possibility of hiding, exporting, and importing of types – in Java, types correspond to Java classes – as well as means for removing and adding types.

The OSGi platform is organised in layers of functionality (cf. Figure 2.3), each layer adding a distinct set of features on top of the existing ones. The operating environment for the OSGi framework consists of the underlying hardware, the operating system with the drivers installed in it, as well as the Java runtime environment (JRE) that is installed in the operating system. The JRE provides the Java virtual machine (JVM) on top of which the OSGi platform resides.

The OSGi platform layers provide the following functionality.

- The *security layer* is an exception to the layering rule in the sense that (a) it cross-cuts other layers and (b) it constitutes an optional layer. The security layer adds fine-grained permission control over applications and allows for signing and verification of bundles.

- The lower-most layer, the *module layer* provides a module system for the Java programming language that defines the deployment structure and strict rules for inter-module class visibility. The module system uses the term *bundle* to denote the modules managed by the OSGi platform.

- The *life-cycle layer* provides a life-cycle interface for bundles. It defines bundle states, allowed state transitions and preconditions for transitions. Additionally, it defines an eventing system that allows to attain notifications of bundle state changes.

- The *service layer* provides simple and consistent means to bundles for registering services (i.e. implementations of java interfaces), as well as for discovering and using services provided by other bundles. The service layer also provides functionality allowing to react to service registrations (in order to e.g. register own services as other, needed services are registered).

Figure 2.4 shows the interactions between the OSGi layers. Client code in bundles use the service layer to register and deregister services to the OSGi service registry. After discovering required services, bundles use the service layer to retrieve and release the services needed (in OSGi terminology, to *get* and *unget* services).

A bundle consists of a tightly integrated set of classes, resources and native code with a specially defined entry point, called the *bundle activator*. The bundle activator allows client code to interact with the OSGi layers in general, and with the life-cycle layer specifically. The life-cycle layer invokes the bundle activator's start and stop callback methods, while passing a bundle context parameter allowing access to the OSGi infrastructure. In particular, the bundle context allows to access the OSGi service registry for service lookup and service registration.

The life-cycle layer starts and stops bundles by invoking the bundle activator start and stop callbacks. At the same time, bundles can use the life cycle layer to request to start, stop, or change the state of other bundles.

The life-cycle layer itself interacts with the service layer by revoking service registrations when a bundle is stopped. At the same time, it commands the module layer to install and uninstall bundles, making classes available and revoking their availability to other bundles, respectively. The module layer in turn is used by the bundle class loaders to load classes.

Finally, the bundle itself as well as the module layer (and transitively, also all other layers) use the execution environment (JRE) to run their code.

In order to illustrate the workings and usage of OSGi, the following text makes use of an example from the domain of physiological computing: a set of analysis components are existing in a system, and a single monitoring service

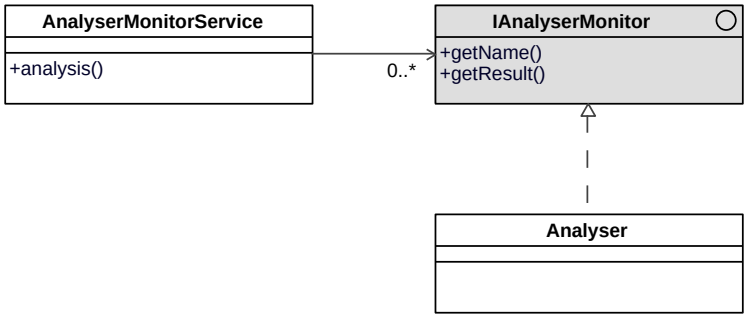


Figure 2.5: Analyser example plug-in architecture

allows to inspect the current analysis results of each analysis components. The monitoring service hence aggregates the monitoring interfaces of several analysis components. This example makes use of an OSGi console interface to display the current analysis results. Basically, this design makes use a plug-in architecture [54].

Figure 2.5 gives an overview of the plug-in architecture of the example. The aggregating class is the `AnalyserMonitorService` (see listing 2.1). Note how the implementation is thread-safe, as the console thread invoking the `_analysis()` method is a different one from the OSGi service listening thread that is used to add and remove plug-ins. The sample analyser is realised in the `Analyser` class, and is shown in listing 2.3. It uses a dummy data generator that creates pseudorandom data in an own thread. This generated data is used as analysis output. The common plug-in interface, `IAnalyserMonitor`, is shown in listing 2.2.

Listing 2.1: Aggregating service

```
public class AnalyserMonitorService implements CommandProvider
    {

  private final List<IAnalyserMonitor> fAnalysers;
  private final NumberFormat fFormat;

  public AnalyserMonitorService() {
    fAnalysers = new CopyOnWriteArrayList<IAnalyserMonitor>();
    fFormat = NumberFormat.getNumberInstance();
  }
```

```
  public void _analysis(CommandInterpreter ci) {
    ci.println("Current snapshot of analysis results");
    for (IAnalyserMonitor analyser : fAnalysers) {
      double value = analyser.getResult();
      String name = analyser.getName();
      ci.println(name + ": " + fFormat.format(value));
    }
  }

  public String getHelp() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("---Analysis Monitor Help---\n");
    buffer.append("\tanalysis - lists the current analysis
       results\n");
    return buffer.toString();
  }

  public void addAnalyser(IAnalyserMonitor d) {
    fAnalysers.add(d);
  }

  public void removeAnalyser(IAnalyserMonitor d) {
    fAnalysers.remove(d);
  }
}
```

Listing 2.2: **IAnalyserMonitor** plug-in interface

```
public interface IAnalyserMonitor {

  /**
   * Returns the current analysis result.
   *
   * @return the current result of the analysis process.
   */
  public double getResult();

  /**
   * Returns the name of the analyser.
   *
   * @return the name of the analyser.
   */
  public String getName();
}
```

Listing 2.3: Mock analyser class

```
public class Analyser implements IAnalyserMonitor {

  private final DataGenerator fGenerator;

  private final String fName;

  private double fResult;

  public Analyser(String name) {
    fName = name;
    fGenerator = new DataGenerator(new IDoubleTarget() {
      @Override
      public void push(double value) throws
          InterruptedException {
        synchronized(Analyser.this) {
          fResult = value;
        }
      }
    });
  }

  public void start() {
    fGenerator.start();
  }

  public void stop() {
    fGenerator.stop();
  }

  @Override
  public synchronized double getResult() {
    return fResult;
  }

  @Override
  public String getName() {
    return fName;
  }

  public String toString() {
    return "Analyser[" + fName + "]";
  }
}
```

Listing 2.4 shows a typical OSGi bundle activator. The source listing shows how the bundle activator registers services (for the interfaces `IAnalyserMonitor` and `CommandProvider`, where the command provider interface is an OSGi interface for interacting with its runtime console), and registers a service listener (implemented by itself) to react to appearing and disappearing services by injecting them into and removing them from the `AnalyserMonitorService`, respectively.

Listing 2.4: OSGi activator for the analyser example

```
public class Activator implements BundleActivator,
    ServiceListener {

  private BundleContext fContext;
  private AnalyserMonitorService fService;
  private ServiceRegistration fRegistration;

  @Override
  public void start(BundleContext context) throws Exception {
    fContext = context;
    fService = new AnalyserMonitorService();

    // listen to newly registered dictionaries.
    fContext.addServiceListener(this, "(objectclass="
        + IAnalyserMonitor.class.getName() + ")");

    // register a new analyser.
    Analyser analyser = new Analyser("Mood Valence");
    analyser.start();
    fContext.registerService(IAnalyserMonitor.class.getName(),
         analyser,
        null);

    // register the console service.
    fRegistration = fContext.registerService(
        CommandProvider.class.getName(), fService, null);
  }

  @Override
  public void stop(BundleContext context) throws Exception {
    fRegistration.unregister();
    fContext = null;
    fService = null;
  }
```

```
    public void serviceChanged(ServiceEvent ev) {
      ServiceReference sr = ev.getServiceReference();
      switch (ev.getType()) {
        case ServiceEvent.REGISTERED: {
          IAnalyserMonitor monitor = (IAnalyserMonitor) fContext
              .getService(sr);
          fService.addAnalyser(monitor);
        }
        break;
        case ServiceEvent.UNREGISTERING: {
          IAnalyserMonitor monitor = (IAnalyserMonitor) fContext
              .getService(sr);
          fService.removeAnalyser(monitor);
        }
        break;
      }
    }
}
```

A bundle is deployed as a Java archive file (jar) containing java classes, native code, resource files, and a manifest containing additional, OSGi-specific manifest headers. These additional headers are used for example to declare the bundle activator and dependencies to other bundles, among others. Listing 2.5 gives an example manifest file. Among other details, the bundle activator as well as imported and exported packages are defined herein.

Listing 2.5: OSGi manifest for the analyser example

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Osgi
Bundle-SymbolicName: example.osgi
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: example.osgi.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.eclipse.osgi.framework.console;version
   ="1.0.0",
 org.osgi.framework;version="1.3.0",
 org.osgi.util.tracker;version="1.3.1"
Require-Bundle: example.datagenerator;bundle-version="1.0.0"
```

OSGi's most important impacts on a system and its codebase code-base are twofold:

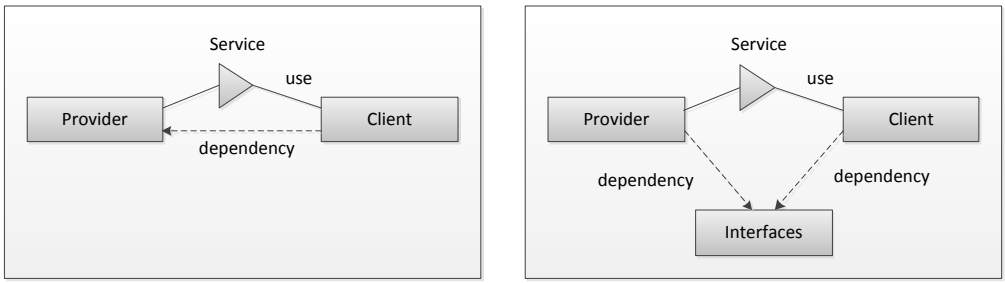


Figure 2.6: Reducing bundle dependencies by introducing an interface bundle

- *Providing a clear module and service concept.* This enables software engineers to define the accessibility of their bundle's java code base package-wise, and let OSGi enforce the specified visibility constraints. Additionally, the service concepts is largely seen as having the most impact on the software structure and quality. However, the service concepts make it necessary for software engineers to embrace the dynamicity of OSGi. Taking into account the possibility of services becoming unavailable at any point forces software developers to code much more defensively than usually done in desktop and even in server environments [92]. The bundle activator in listing 2.4 shows drastically the overhead involved in reacting to OSGi service dynamics.

- *Allowing code-swapping at runtime.* OSGi allows indeed to exchange a code base without the need to shut down the whole OSGi framework and all bundles it hosts. Instead, the OSGi module system allows to shut down only the affected bundles, that is to say, the bundle to exchange and the bundles that directly depend on it. While on a first thought, this policy seems like requiring to shut down a large part of a system, the number of affected bundles can be reduced significantly if proper service-oriented bundle organisation is used, as shown in Figure 2.6, on the right.

OSGi is a solid service-oriented module platform that allows to enforce clear separation of Java code into modules that can be swapped at runtime. The OSGi framework introduces a module system to the Java programming language that was missing before the introduction of the OSGi platform. The component-based software framework that is presented in this thesis

leverages the facilities OSGi provides for the creation of code separation.

2.4 Component-based formal analysis

The goal of this thesis is not only to provide an implementation framework, but also a theoretical framework for the verification of correctness of reconfigurations, as reconfigurations are a useful tool in physiological computing applications, but are also hard to get right. In this section, we introduce the terminology and background for formal analysis of component-based systems, which constitutes the basis to the verification framework for systems undergoing reconfigurations that is introduced in Chapter 4.

For formal analysis, *components* [117, 81] are here considered to be black boxes, making explicit only their communication requirements by means of *required* and *provided* ports. The distinction between provided and required ports is made as a need to attribute control over the port to a component. While a component controls its provided ports, it has no control over its required ports.

The behaviour of a single component as well as of an entire system can be specified by assume-guarantee pairs, where the assumptions address the system environment, and the guarantees the behaviour of the component. Assumptions and guarantees are specified in terms of real-time temporal logic, which is extended to support reconfigurations in Chapter 4. The use of real-time temporal logic is motivated by the fact that physiological computing systems are systems interacting with the real world, and these interactions are best modelled with continuous-time specifications. Due to the decision to use real-time linear temporal logic, we use a continuous trace semantics as for Metric Interval Temporal Logic (MITL) as found in [8].

For the definition of the formal semantics of components and of the temporal logic, the following notation is used: $C = A \uplus B$ means that $C = A \cup B$ and that A and B are disjoint; Given a subset A from a domain of discourse Ω, $\neg A$ denotes $\Omega \setminus A$.

The formal assume-guarantee framework for reconfiguration presented here is based on work of Benveniste et al. [26]. In the following, we introduce the necessary terminology and basic concepts introduced in [8] and [26], as well as the basis for combining the (semantic-level) assume-guarantee framework with metric interval temporal logic [8] specifications.

First of all, following the systematic approach of [121] and [21], we set

the domain of discourse by defining component signatures.

Definition 1 (Component signature)**.** *A* component signature Σ *consists of two disjoint finite sets* R_Σ, P_Σ *of provided and required ports, respectively:* $\Sigma = (R_\Sigma, P_\Sigma)$.

In the following, signature-indexed components are used to refer to the components of signatures: R_Σ denote the required ports of the signature Σ. In a next step, the relations of subsignature and supremum on signatures can be defined.

Definition 2 (Subsignature and signature supremum)**.** *Let* Σ *and* Θ *be component signatures. The notion of a* subsignature $\Sigma \subseteq \Theta$ *is defined by component-wise set inclusion.*

$$\Sigma \subseteq \Theta \text{ iff } R_\Sigma \subseteq R_\Theta \text{ and } P_\Sigma \subseteq P_\Theta$$

Given two signatures Σ *and* Θ*, the* supremum *of* Σ *and* Θ *is defined by component-wise set union.*

$$\sup(\Sigma, \Theta) = (R_\Sigma \cup R_\Theta, P_\Sigma \cup P_\Theta)$$

The semantics of a metric interval temporal logic formula is defined in terms of runs. Runs are also used to represent behaviours of components with signature Σ.

Definition 3 (Run)**.** *Let* Σ *be a component signature. A* Σ-run *is a function* $\rho : \mathbb{R}_0^+ \to (R_\Sigma \cup P_\Sigma)$ *which satisfies the properties of* finite-variability *[8]: for all* $t, t' \in \mathbb{R}_0^+$*, there are only finitely many different states between time* t *and* t'*:*

$$\forall t, t' \in \mathbb{R}_0^+ . t < t' \implies |\{\rho(t'') \mid t \leq t'' \leq t'\}| < \infty.$$

The class of all Σ*-runs is denoted by* $\mathcal{R}(\Sigma)$.

Note that the definition of runs is different to that found in [8]; it can be easily shown however that the definitions given here and the definition of [8] are equivalent. The definition of runs specifies the ports that are activated at time $t \in \mathbb{R}_0^+$: all required and provided ports that are contained in $\rho(t)$ are active. The finite variability required in Definition 3 is necessary to avoid Zeno's paradox [16], and to make the logic operating on these runs decidable. Finite variability prohibits runs that change port valuations infinitely often in finite time.

Assertions can now be defined as sets of runs; a system can be said to satisfy an assertion E if its behaviours M is a subset of the property E.

Definition 4 (Assertion)**.** *Let Σ be a component signature. A* Σ-assertion *E, in the following also denoted by $E : \Sigma$, is a set of Σ-runs: $E \subseteq \mathcal{R}(\Sigma)$.*

Assertions over component signatures are used for specifying *assumptions*, *guarantees*, and *implementations*. As oftentimes, assumptions, guarantees and implementations are assertions over different signatures, there must be a way to transform an assertion from one signature to another. For this, we introduce the concepts of lifting and restriction. First, the lifting of runs and assertions from a signature Σ to a signature $\Theta \supseteq \Sigma$ is defined.

Definition 5 (Lifting)**.** *Let Σ, Θ be component signatures such that $\Theta \supseteq \Sigma$, let $\rho \in \mathcal{R}(\Sigma)$ be a Σ-run. The* lifting *of ρ to Θ, $\rho{\uparrow}^{\Theta}$ is defined as follows.*

$$\rho{\uparrow}^{\Theta} = \{\rho' \in \mathcal{R}(\Theta) \mid \forall t \in \mathbb{R}_0^+ . \rho'(t) \cap (R_\Sigma \cup P_\Sigma) = \rho(t)\}$$

Let $E : \Sigma$ be a Σ-assertion. The lifting *of E to Θ, $E{\uparrow}^{\Theta}$, is defined as follows.*

$$E{\uparrow}^{\Theta} = \bigcup \{\rho{\uparrow}^{\Theta} \mid \rho \in E\}$$

Restriction is then defined as the inverse operation for runs and assertions from a signature $\Theta \supseteq \Sigma$ to Σ.

Definition 6 (Restriction)**.** *Let Σ, Θ be component signatures such that $\Theta \supseteq \Sigma$, let $\rho \in \mathcal{R}(\Theta)$ be a Θ-run. The* restriction *of ρ to Σ, $\rho{\downarrow}_\Sigma$ is defined as follows.*

$$\rho{\downarrow}_\Sigma = \{\rho' \in \mathcal{R}(\Sigma) \mid \forall t \in \mathbb{R}_0^+ . \rho(t) \cap (R_\Sigma \cup P_\Sigma) = \rho'(t)\}$$

Let $E : \Theta$ be a Θ-assertion. The restirction *of E to Σ, $E{\downarrow}_\Sigma$ is defined as follows.*

$$E{\downarrow}_\Sigma = \bigcup \{\rho{\downarrow}_\Sigma \mid \rho \in E\}$$

In this abstract semantics-based framework, parallel composition of implementations is straightforward, and is defined by parallel composition of assertions.

Definition 7 (Parallel composition of assertions)**.** *Let Σ_E, Σ_F be component signatures. Let $E : \Sigma_E$ and $F : \Sigma_F$ be two assertions. The* composition *of E and F is defined by $E \| F = E{\uparrow}^{\Sigma} \cap F{\uparrow}^{\Sigma}$, where $\Sigma = \sup(\Sigma_E, \Sigma_F)$.*

Note that since $E \| F$ satisfies finite variability, it is also a Σ-assertion.[2]

In order to specify not only the behaviour of a component, but also assumptions expressed towards its environment, assume-guarantee contracts are used.

Definition 8 (Assume-guarantee contract)**.** *Let Σ be a component signature. An* assume-guarantee contract *is a pair of assertions $(A : \Sigma, G : \Sigma)$.*

Contract satisfaction is defined by inclusion of runs – more precisely, for an implementation M, every run in M which is in A (i.e. *satisfies* A) must be in G (i.e. *satisfies* G).

Definition 9 (Contract satisfaction)**.** *Let $M : \Sigma_M$ be an implementation, and $(A : \Sigma, G : \Sigma)$ be an assume-guarantee contract. M* satisfies *$(A : \Sigma, G : \Sigma)$, denoted by $M \vDash^{c}_{\Sigma_M} (A, G)$, if and only if $\Sigma \subseteq \Sigma_M$ and $M \cap A\uparrow^{\Sigma_M} \subseteq G\uparrow^{\Sigma_M}$.*

The *canonical form* of contracts can be defined as $(A\uparrow^{\Sigma}, (\neg A)\uparrow^{\Sigma} \cup G\uparrow^{\Sigma})$, over the signature $\Sigma = \sup(\Sigma_A, \Sigma_G)$. The rationale for the canonical form is that non-circular composition of contracts is easier to define, and that an implementation M satisfies a contract if and only if it satisfies its canonical form.[3] To simplify the presentation, it is assumed from now on that contracts are in canonical form. $(A, G) : \Sigma$ can be written instead of $(A : \Sigma, G : \Sigma)$, since the signatures of A and G are the same.

Now, parallel composition of contracts is defined. While parallel composition of implementations is defined by intersection, parallel composition of contracts is more complex, as it produces a new contract whose assumptions consists only of those assumptions that are not satisfied by any element of the composition, and whose guarantee should consist of a combination of the single guarantees. At the same time, parallel composition must create a contract in canonical form, and avoid circular reasoning: if, for instance, the first contract C assumes A and guarantees B while the second contract D assumes B and guarantees A, the assumption of the composed contract should not be the set of all runs (i.e. making no assumption on the environment), but the complement of $\neg A \cap \neg B$, i.e. $\neg(\neg A \cap \neg B)$.

[2]Lifting of assertions preserves finite variability by definition, as does intersection of assertions.

[3]For a discussion of circularity issues in parallel composition, see [1].

Definition 10 (Parallel composition of contracts)**.** *Let $C = (A_C, G_C) : \Sigma_C$, $D = (A_D, G_D) : \Sigma_D$ be two contracts in canonical form with $P_{\Sigma_C} \cap P_{\Sigma_D} = \emptyset$. The* parallel composition *of C and D, $C \| D$ is defined as follows.*

$$C \| D = ((A_C \cap A_D) \cup \neg(G_C \cap G_D), G_C \cap G_D) : \Sigma$$

Note that the composed contract is in canonical form, since $\neg((A_C \cap A_D) \cup \neg(G_C \cap G_D)) = \neg(A_C \cap A_D) \cap (G_C \cap G_D) \subseteq (G_C \cap G_D)$. Furthermore, it can be directly shown that composition of implementations and contracts is commutative and associative, as it is defined by set intersections. Note also that the restriction over provided ports included in Definition 10 requires that the control of provided ports is well-defined: only one component may define the valuation of a provided port. It is therefore possible to define a mapping from provided ports to its controlling component.

We can show now that parallel composition of implementations preserves contract satisfaction.[4]

Theorem 1 (Composition preserves contract satisfaction)**.** *Let Σ_M and Σ_N be component signatures, $M : \Sigma_M$, $N : \Sigma_N$ be two implementations such that* $\sup(\Sigma_M, \Sigma_N) \subseteq \Sigma$. *Let $C = (A_C, G_C) : \Sigma_M$, $D = (A_D, G_D) : \Sigma_N$ be two contracts. Then it holds that*

$$\textit{if } M \models^c_{\Sigma_M} C \textit{ and } N \models^c_{\Sigma_N} D \textit{ then } M \| N \models^c_{\Sigma} C \| D.$$

The proof of this theorem makes use of basic set arithmetic.

Proof of Theorem 1. Let $(A, G_C \cap G_D) = C \| D$. We have to show that $M \| N \cap A \subseteq G_C \cap G_D$. We know that $M \cap A_C \subseteq G_C$, which holds iff $M \subseteq G_C \cup \neg A_C$. Since $\neg A_C \subseteq G_C$, it follows $M \subseteq G_C$. Same holds for N and (A_D, G_D). Hence, $M \cap N \cap A \subseteq G_C \cap G_D \cap A$, which is a subset of $G_C \cap G_D$. □

One missing piece in an assume-guarantee framework is left: the refinement of contracts. When building component-based systems, it is desirable to check whether the contract resulting from composition satisfies a global system specification. Therefore, a refinement relation on contracts is introduced that allows assumptions to be weakened and guarantees to be strengthened: refinement is covariant for guarantees and contravariant for assumptions.

[4] Using the canonical form allows to compute an assumption even for cyclic parallel composition, but with the cost of a possibly weaker guarantee than with the initial, non-canonical contract.

Definition 11 (Contract refinement)**.** *Let* Σ *be a signature.* $(A', G') : \Sigma$ *refines* $(A, G) : \Sigma$*, denoted by* $(A, G) \succeq (A', G')$*, if* $A \subseteq A'$ *and* $G' \subseteq G$*.*

A major requirement for refinement relations is its compatibility with the satisfaction relation for implementations, i.e. whenever an implementation satisfies a refined contract, it satisfies the original contract.

Lemma 1 (Contract refinement preserves contract satisfaction)**.** *Let* $(A, G) : \Sigma$ *and* $(A', G') : \Sigma$ *be contracts and* $M : \Sigma_M$ *be an implementation such that* $\Sigma \subseteq \Sigma_M$*. If* $M \vDash^c_\Sigma (A', G')$ *and* $(A, G) \succeq (A', G')$ *then* $M \vDash^c_\Sigma (A, G)$*.*

The proof of this lemma is as follows.

Proof of Lemma 1. Let $M \vDash^c_\Sigma (A', G')$ and $(A, G) \succeq (A', G')$. We have to show that $M \vDash^c_\Sigma (A, G)$, i.e. that $M \cap A\!\uparrow^\Sigma \subseteq G\!\uparrow^\Sigma$. We know that $M \cap A'\!\uparrow^\Sigma \subseteq G'\!\uparrow^\Sigma$, and $A \subseteq A'$ and $G' \subseteq G$, which extends to $A\!\uparrow^\Sigma \subseteq A'\!\uparrow^\Sigma$ and $G'\!\uparrow^\Sigma \subseteq G\!\uparrow^\Sigma$. Therefore $M \cap A\!\uparrow^\Sigma \subseteq M \cap A'\!\uparrow^\Sigma \subseteq G'\!\uparrow^\Sigma \subseteq G\!\uparrow^\Sigma$. It hence holds that $M \vDash^c_\Sigma (A, G)$. □

So far, the difference between provided and required ports have been mostly ignored. However, the role of ports is asymmetric in this assume-guarantee framework [26]. While provided ports are under the control of the providing component, a component may not control the valuations of required ports. Therefore, implementations and contracts must be designed such that they do not constrain port valuations illegally. To specify the requirements towards contracts and implementations, the notion of receptivity is defined first.[5]

Definition 12 (Receptivity)**.** *Let* $E : \Sigma$ *be a* Σ*-assertion.* E *is* Req-receptive *iff* $E{\downarrow_{\Sigma_{Req}}} = \mathcal{R}(\Sigma_{Req})$*, where* $\Sigma_{Req} = (R_\Sigma, \emptyset)$*.* E *is* Prov-receptive *iff* $E{\downarrow_{\Sigma_{Prov}}} = \mathcal{R}(\Sigma_{Prov})$*, where* $\Sigma_{Prov} = (\emptyset, P_\Sigma)$*.*

If a Σ-assertion E is Req-receptive, it does not constrain the valuations of its required ports. If it is Prov-receptive, it does not constrain the valuations of its provided ports, respectively.

Considering that an assume-guarantee framework is concerned with allowing the decomposition of system specifications, it is clear that receptivity

[5]Note that in [26], required ports amount to uncontrolled ports, and provided ports amount to controlled ports. We renamed these notions to provide a natural connection to the rest of this thesis.

requirements need to be imposed over contracts and implementations. Contracts and implementations that satisfy their receptivity requirements are called *valid*.

Definition 13 (Validity)**.** *Let $M : \Sigma$ be an implementation. M is called* valid implementation *iff M is Req-receptive. Let $C = (A, G) : \Sigma$ be an assume-guarantee contract. C is called* valid contract *iff A is Prov-receptive, and G is Req-receptive.*

The validity of implementations is easy to grasp: an implementation may not constrain the valuations of its required ports, as it is deemed to have no control over them. Of course, the specification is expected to respond to valuations of its required ports by changing the valuations of its provided ports. An implementation must provide a valuation of its provided ports for any valuation of its required ports.

For assume-guarantee contracts, the rationale for its validity requirements are more involved. It is straightforward that the guarantee of an assume-guarantee contract must satisfy the same constraints as for implementations – a guarantee is a loose specification of the behaviour of an implementation, and as such it may not specify any constraints on required ports. Assumptions are different, however, as they represent requirements towards the component's environment. An assumption specifies constraints towards the environment of the component to be implemented, and hence must constrain the allowed valuations of required ports. Conversely to guarantees and implementations however, assumptions may not restrict the behaviour of the specified component – they must focus on specifying valid environments in which the component must operate.

Now that the assume-guarantee framework for continuous-time traces was introduced, we continue with the syntactic level by introducing metric-interval temporal logic (MITL) [8]. First of all, the basic time structure used in MITL formulas, time intervals, need to be introduced.

Definition 14 (Time interval)**.** *As time domain we use the non-negative real numbers $\mathbb{R}_0^+$. A* time interval *is a nonempty convex subset of $\mathbb{R}_0^+$ that has one of the following forms: $[a, b], [a, b), [a, \infty), (a, b], (a, b), (a, \infty)$ where $a < b$ for $a, b \in \mathbb{R}_0^+$.*

Note that the definition of time intervals requires *non-singular* intervals. In the following, some basic arithmetic operations on intervals are used. The

expression $t + I$, for $t \in \mathbb{R}_0^+$, denotes the interval $\{t + t' \mid t' \in I\}$, and analogously, $t - I$ denotes the interval $\{t - t' \mid t' \in I\}$. Finally, the expression $I + J$ denotes the addition of two intervals I, J and is defined by $\{t_i + t_j \mid t_i \in I, t_j \in J\}$.

Definition 15 (MITL-formulas)**.** *The set of* MITL formulas *over a component signature Σ is inductively defined by*

$$\phi ::= \top \mid p \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U}_I \phi_2 \mid \phi_1 \mathcal{S}_I \phi_2$$

where $p \in R_\Sigma \cup P_\Sigma$ is a port, and I is a nonsingular time interval. For brevity, a MITL*-formula ϕ over a signature Σ is also called a Σ-formula.*

While until operator $\phi_1 \mathcal{U}_I \phi_2$ denotes a real-time variant of the until operator that requires ϕ_2 to be true in the time interval I, the since operator $\phi_1 \mathcal{S}_I \phi_2$ is an operator that looks to the past, and requires that ϕ_2 must be true in the past and within the time interval I.

The usual abbreviations for the temporal operators "eventually" and "always" can be introduced. The defined operators $\Diamond_I \phi$ and $\Box_I \phi$ stand for *true* $\mathcal{U}_I \phi$ and $\neg\Diamond_I \neg\phi$, respectively. Similarly, past operators can be defined with the since-operator, see e.g. [104]. Moreover, $\phi_1 \vee \phi_2$ is defined by $\neg(\neg\phi_1 \wedge \neg\phi_2)$, and as usual $\phi_1 \rightarrow \phi_2$ stands for $\neg\phi_1 \vee \phi_2$, and $\phi_1 \leftrightarrow \phi_2$ stands for $(\phi_1 \rightarrow \phi_2) \wedge (\phi_2 \rightarrow \phi_1)$.

The semantics of MITL-formulas is given in the form of sets of runs over a signature. Each run provides, for every point in time, an interpretation of the ports of the given signature.

Definition 16 (Satisfaction of MITL-formulas)**.** *A Σ-run $\rho \in \mathcal{R}(\Sigma)$* satisfies *a Σ-formula ϕ (or ϕ* is valid in *ρ), denoted by $\rho \models_\Sigma \phi$, iff $(\rho, 0) \models_\Sigma \phi$, where the satisfaction relation $\models_\Sigma$ between pairs (ρ, t), $t \in \mathbb{R}_0^+$, and Σ-formulas ϕ is inductively defined as follows:*

$$
\begin{array}{lll}
(\rho,t) \models_\Sigma \top & & \\
(\rho,t) \models_\Sigma p & \textit{iff} & p \in \rho(t); \\
(\rho,t) \models_\Sigma \neg\phi & \textit{iff} & (\rho,t) \not\models_\Sigma \phi; \\
(\rho,t) \models_\Sigma \phi_1 \wedge \phi_2 & \textit{iff} & (\rho,t) \models_\Sigma \phi_1 \textit{ and } (\rho,t) \models_\Sigma \phi_2; \\
(\rho,t) \models_\Sigma \phi_1 \mathcal{U}_I \phi_2 & \textit{iff} & \exists t' \in \mathbb{R}_0^+, t' \in t + I.(\rho,t') \models_\Sigma \phi_2 \\
 & & \textit{and } \forall t'' \in \mathbb{R}_0^+, t < t'' < t'.(\rho,t'') \models_\Sigma \phi_1; \\
(\rho,t) \models_\Sigma \phi_1 \mathcal{S}_I \phi_2 & \textit{iff} & \exists t' \in \mathbb{R}_0^+, t' \in t - I.(\rho,t') \models_\Sigma \phi_2 \\
 & & \textit{and } \forall t'' \in \mathbb{R}_0^+, t' < t'' < t.(\rho,t'') \models_\Sigma \phi_1.
\end{array}
$$

A Σ-formula ϕ is called a consequence of *a set Γ of Σ-formulas, denoted by $\Gamma \models_\Sigma \phi$, if $\rho \models_\Sigma \phi$ holds for every ρ such that $\rho \models_\Sigma \psi$ for all $\psi \in \Gamma$. ϕ is called* universally valid*, denoted by $\models_\Sigma \phi$, if $\emptyset \models_\Sigma \phi$.*

Note that non-strict versions of $\mathcal{U}$ and $\mathcal{S}$ (i.e. versions allowing ϕ_2 to be true immediately, and not only in the future) can be defined from the versions above just as in Schobbens et al. [104].

Given a set Γ of Σ-formulas, the semantics of Γ is defined by the set of all runs satisfying Γ, i.e. $[\![\Gamma]\!] = \{\rho \in \mathcal{R}(\Sigma) \mid \rho \models \Gamma\}$; if $\Gamma = \{\phi\}$, then $[\![\phi]\!] = [\![\Gamma]\!]$.

This concludes the background needed for the introduction of a formal framework for the verification of systems undergoing reconfigurations. This section introduced a semantic domain for assume-guarantee reasoning on real-time specifications, and showed the existing results on the compatibility of composition and refinement, as well as existing approaches to the specification of components with required and provided ports. Additionally, metric interval temporal logic was introduced as candidate syntax for the specification of assumptions and guarantees. The connection between semantics and syntax is established in Chapter 4, where the assume-guarantee framework is also extended to allow for modelling and verifying systems under reconfiguration.

2.5 Software development methodology

Transitioning from a vague idea of a software system to a satisfactory and robust realisation of that idea is the underlying theme of virtually all software engineering research. A significant part of that research is focused on providing specific and highly focused solutions that improve single aspects of software creation activities. In order to be effectively applicable, however, these focused solutions needs to be embedded in a working software development methodology. In this section, we therefore discuss the software development methodology that we intend to use as the frame for our engineering approach to physiological computing: the agile software development methodology Scrum.

2.5.1 Agile development methodologies

Scrum is a software development methodology from the set of agile methodologies which are difficult to characterise precisely. Their common ground is

their focus on individuals, interaction, working software, collaboration with customers, and embracing of environmental change, which they value more than processes, tools, documentation, legal contracts, and plans [23]. However, it must not be assumed that agile methodologies were designed to be a means to abolish the principle of software development methodologies and software engineering in general; in fact, quite the contrary is true:

> *"The agile methodology movement is not anti-methodology; in fact, many of us want to restore credibility to the word. We also want to restore a balance: We embrace modelling, but not merely to file some diagram in a dusty corporate repository. We embrace documentation, but not to waste reams of paper in never-maintained and rarely-used tomes."* [58]

In a way, the agile development movement is a rupture with previous and existing software development methodologies; at the same time, however, the agile development methodologies are a logical consequence of the experiences that were made with traditional methodologies.

The experiences and insights gained when applying traditional methodologies led to several fundamental differences between agile and heavyweight development methodologies. These differences emerge in several aspects of the software development philosophy; in the following, we cover two fundamental ones which are easily depicted: the way process rules are prescribed and the way planning is understood and performed.

2.5.2 Process rules

Previous to the agile movement, software development methodologies tried to provide a complete set of rules and prescriptions on how to organise a software development process, i.e. which documents to create, when to create them, and how to create them. This approach has been coined by Martin Fowler the monumental approach [56], because they are said to entail a monumental and bureaucratic form of organisation. Monumental software development methodologies often provide means for tailoring the process: by following given guidelines; it is therefore possible to simplify and modify the methodology and create a set of prescriptions that are a good match for the project at hand. Agile methodologies follow a different approach to creating a set of organisational rules for a project. Instead of trying to cover every

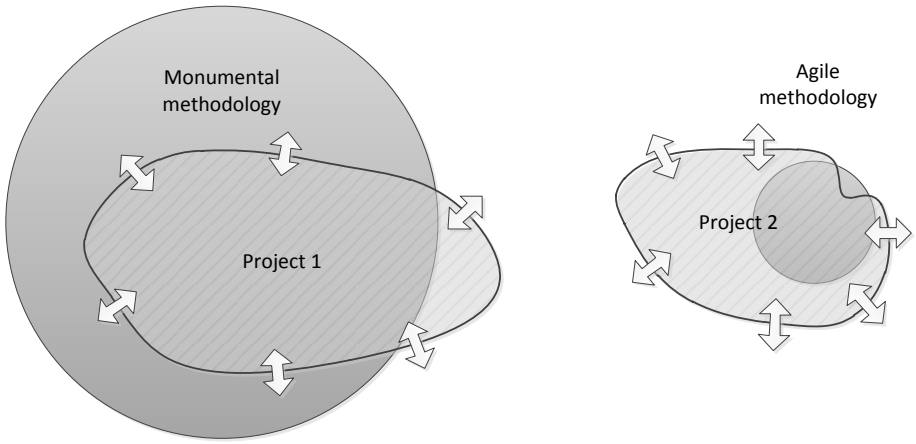


Figure 2.7: Monumental and agile software development methodology compared by rule prescription

rule that may be needed for very large, distributed, or high safety requirements projects, they describe a minimal kernel of rules that virtually every project will need; agile methodologies often try to provide a least common denominator that is needed in all projects.

Figure 2.7 depicts this difference using a Venn diagram between the set of rules prescribed by a hypothetical software development methodology of each family and a hypothetical project. Since a hypothetical monumental approach on the left provides a comprehensive rule set, project 1 needs to reduce the rule set provided through tailoring, but needs to invent only few process rules necessitated by the specific project environment. The agile methodology on the right specifies only a small kernel of process rules. Project 2 hence needs to invent more process rules to cope with the specific project environment, and ignores only a few rules from the agile methodology – those that are not applicable for the specific project context. In total, Project 2 is executed with a smaller process rule set than Project 1. The double arrows on the border line of each project rule set indicate that the rule set may be changed during the project due to deliberate decisions, external forces, or process erosion that is naturally occurring if the process is not maintained; the process rule set that is followed by a project's team is subjected to constant re-negotiation and change.

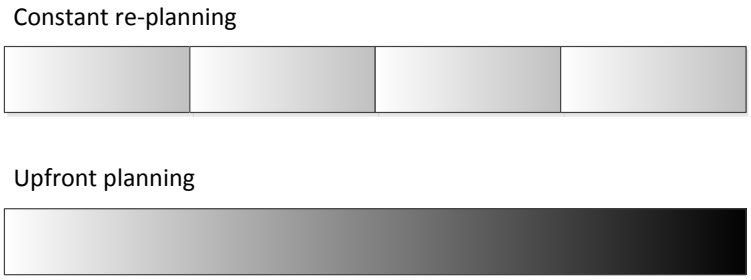


Figure 2.8: Uncertainty of upfront and iterative planning compared

2.5.3 Planning and controlling

Classical software development methodologies like the waterfall model [102], Rational Unified Process (RUP) [79] and the V-Model [70] try to establish a plan for the full software system right at the beginning of the project development work (in RUP, the full plan is established in the elaboration phase). The goal of the plan creation process is to create a stable project roadmap to follow; it may be changed, but its primary purpose is to be followed.

However, experience with software development has shown that software development is a process involving significant amounts of disturbances and noise [107]. When creating plans in such environments, the uncertainty increases with the distance of the planning horizon. Plans for the development of software systems that reach far in the future need a significant amount of contingencies in order to tackle the accumulating uncertainties and still remain meaningful. Figure 2.8 visualises this effect; brighter colours indicate higher confidence, and darker colours higher uncertainty and lower confidence about the project plan. With a single upfront planning activity, the end of the plan corresponds to the end of the project (which is, arguably, an important phase of the project), and features high degrees of uncertainty. With constant re-planning, the uncertainty can be kept comparatively low for the whole project duration. This is the reason why agile software development processes formalise constant re-planning of activities.

Connected with the aspect of project planning is the aspect of project controlling, i.e. the monitoring of the project's adherence to the established plan, and the general supervision of a project's progress. The difference

with regards to project controlling between classical software development methodologies and agile methodologies in what concerns project controlling can be discussed in terms of process control theory [107]. In process control theory, there are two fundamentally different models of process control: defined process control, and empirical process control. The former can be applied when the process under control is simple, well-understood and features a very low amount of disturbances. Here, the definition of the activities to perform and the order in which they have to be performed can be written down upfront, and the results will be very similar each time the process is implemented. This is very much how classical waterfall software development methodologies promotes planning: create a plan upfront, and follow it. Unfortunately, the defined process model is inappropriate as most software development endeavours feature more disturbances and noise than a defined approach can manage. For this reason, an empirical approach to software development seems more appropriate. In an empirical control model, frequent inspection and adaptive response are the dominant control mechanism; where it is not possible to predict the exact reaction of the process to the set of initial conditions and the disturbances it will experience, frequent feedback is used to keep the process on track.

2.5.4 Scrum process development methodology

Scrum [107, 108] is an agile software development methodology that, in contrast to the well-known extreme programming [22] approach, puts its emphasis on management disciplines instead of programming disciplines. In the following, we discuss the principles underlying Scrum, the roles, and the meetings that are defined in Scrum.

The overall process structure of Scrum is depicted in Figure 2.9. Scrum organises the project work into sprints of four weeks length. Each sprint is planned separately by retrieving work items from the product backlog and assembling them to a sprint backlog right before the sprint starts. The product backlog itself is simply a list of work items (sorted by priority) that need to be completed for the product to become reality. The sprint backlog is worked on during the sprint, and daily meetings of 15 minute length are used to provide brief status information from the team. Each sprint is committed to creating a product increment of high-level quality; the goal of each product increment is to be "potentially shippable" to the customer. The product increment is inspected in the sprint review meeting and the team can gather

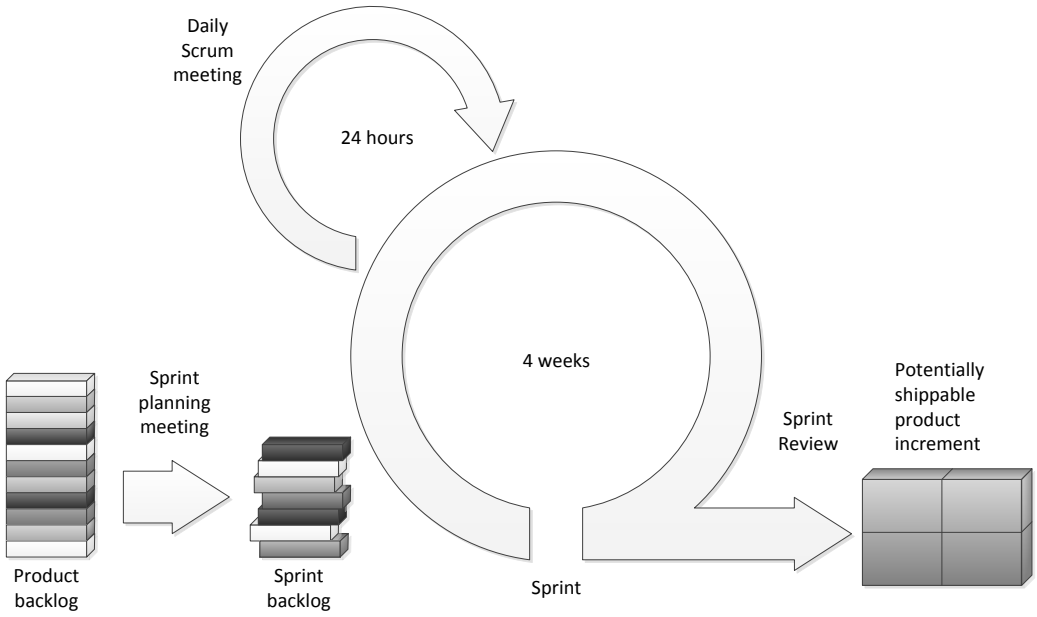


Figure 2.9: Scrum process

feedback on the work it performed on a regular basis (i.e. every four weeks). Once the feedback is gathered, the team is ready to plan the next sprint by assembling to a new sprint planning meeting.

In Scrum, creating a product increment does not necessarily mean delivering (or shipping) the product increment to the customer. Nevertheless, the total product with the latest increment can be inspected and reviewed at the end of each sprint and serves as the planning baseline for the next sprint.

The Scrum software development methodology is organised by a set of underlying principles that guide how process rules are to be enforced and extended.

- **Empirical management.** Scrum relies on the empirical process control model, which means that constant inspection and adaptive corrections are the core concepts of the methodology. It also means that the empirical understanding should underlie each and every decision and action performed. Scrum is also an experimental approach; teams are encouraged to try out new ways to improve their software development approach, and to empirically evaluate the results.

- **Time-boxing.** Each activity in Scrum must have a time limit – each

sprint must have a deadline, and each meeting must have a maximum allowed time. This is required for both allowing frequent feedback and to increase focus on the task to be performed within the time-box. When running out of time, Scrum encourages reducing the scope of work instead of extending the deadline or the available time.

- **Ownership and rights.** Scrum defines clear ownership on the artefacts (and entailed rights) that are managed withing Scrum: the product backlog, the sprint backlog, and the product. The ownership and rights are used to empower the development team, and to keep the sprint scope and product under the control of the team while the team is working. The spirit of the ownership model is explained in Scrum through a small parable of a chicken and a pig, in which the chicken wants to start a restaurant named "Ham n' Eggs", upon which the pig replies "No thanks. I'd be committed, but you'd only be involved!" [107] In this parable, the distinction between involvement and being committed is made clear; the distinction between "pigs" and "chickens" is necessary to shield the team from illegitimate intervention through management forces who think they are committed, but are actually only involved.

- **Transparency.** The goal of Scrum is to make the software development process clearly visible to the team. All practices that make the team productive as well as all impeding practices are made visible in Scrum through its feedback cycles. Planning errors, lack of focus as well as bad engineering practices are continuously exposed and made available for reflection and correction through daily Scrum meetings, sprint reviews, and sprint retrospectives. For additional visibility of a project's current state, Scrum uses a burn-down chart that each day sums up the remaining effort to be performed. Figure 2.10 shows an example burn-down chart of a completed sprint. The chart shows how the remaining efforts of the sprint evolved over time. Through this chart, the team's current situation in terms of past achieved effort and remaining effort is made fully transparent to the team itself. The transparency that is achieved through the burn-down chart, daily Scrum meetings, sprint reviews, and sprint retrospectives enables the team to take corrective actions early to keep the project on track.

- **Self-organising teams.** Scrum does not define how the team must be

structured in order to work efficiently. In fact, prescribed role structures are often hard to map to the needs of a single software development effort. Instead, Scrum encourages the management to proceed empirically, and to encourage behaviours and structures that proved efficient during the project itself. Conversely, behaviours and structures that emerged and proved to impede the team's productivity must be discouraged as well. Striving for self-organising teams also means to give control and authority to the team to self-organise around the challenges the project offers. Management has to make careful tradeoffs between subtle intervention and control on the one side, and handing off control to the team so that it can discover new ways to organise work efforts.

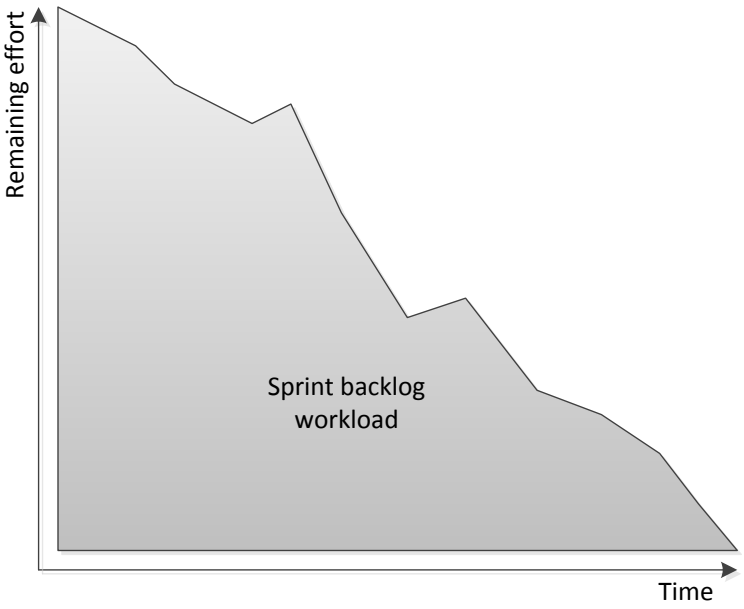


Figure 2.10: Scrum burn-down chart

Scrum defines a set of roles that has to be assigned for the methodology to work: team member, product owner, and Scrum master.

- **Team member.** The team member's main duty is to work on the product to work off the sprint backlog, and to create a potentially shippable product increment. For that, the team member is responsible for assembling the sprint backlog from the product backlog made

available, working on backlog items, and presenting the working result at the end of the sprint.

- **Product owner.** The product owner controls the product backlog, and prioritises the backlog items within it. He is the interface to the stakeholders of the product. The main activities of the product owner are a) to integrate all legitimate requirements towards the product, and b) to prioritise the items in the product backlog to achieve maximum business revenue. The product owner hence manages the product backlog for the team, from which the team assembles its sprint backlog by obeying priority and technical constraints. The product owner is also the person the team contacts on detailed questions about the product; the product owner should be able to answer most of those questions, or otherwise, to organise a meeting with the person who can.

- **Scrum master.** The Scrum master is the coach of the team, and monitors the adherence to established process rules. These rules will mostly originate from Scrum, but they may be modified, and extended to fit the current project. Secondly, the Scrum master is bound to remove any impediments that hinders the team. Most of the time, these impediments will be of organisational nature, which means that the Scrum master is also committed to transform the organisation to a more lean and productive form.

For a Scrum team to be correctly sized, it needs five to nine team members, one product owner, and one scrum master. It is possible for a team member to adopt the role of the product owner or scrum master. It is however very strongly discouraged to assign the role of product owner and scrum master to one and the same person, since the roles must exert different and often conflicting forces on the team.

The Scrum team consists of the team members, their product owner, and their Scrum master. the Scrum process defines a set of meetings that guides the team through the process of sprint creation, execution, and sprint completion. In total, Scrum defines five types of meetings.

- **Sprint planning.** In the Sprint planning meeting, the sprint backlog for the next sprint is assembled from the product backlog. The team decides on how much of product backlog can be taken into the sprint based on estimation techniques such as e.g. planning poker [22]. At the

end of the sprint planning meeting, a sprint backlog is assembled, and a burn-down chart is created. The timebox is eight hours for a four week sprint [108].

- **Daily Scrum meeting.** The daily Scrum meeting is a 15-minute meeting performed by the team and for the team, primarily. It is a brief status report meeting in which each team member reports on the activities performed since the last meeting, the activities planned up to the next meeting, and the impediments that the team member experiences.

- **Sprint review.** The sprint review is used to provide the product owner and stake holders with information about the current state of the product. Also, the sprint review allows the team to gather feedback on the work they performed, and to understand the priorities of customer and product owner. The sprint review is performed at the end of each sprint, with a timebox of four hours for a four week sprint [108].

- **Sprint retrospective.** After each sprint review, and before the next sprint planning meeting, a sprint retrospective of three hours length (for a four week sprint) is used to review and improve the software development approach of the team; that is to say, the team evaluates the practices it follows and tries to improve the way it turns product backlog items into working software, looking for possible improvements to the process.

- **Release planning.** The release planning meeting is an optional meeting that can be performed to plan sprints. The purpose of the release planning meeting is to establish a product roadmap for the team and the rest of the organisation [108]. The release plan contains a set of release dates that guide priorities in the product backlog as well as the nature of the assembled sprints: sprints that are performed right before a release date will be focused heavily on quality improvements, bug fixing and cosmetic improvements of the product itself, while sprints that follow a release will contain more risky activities and changes.

The Scrum principles, roles, meetings, and artefacts are the entities that constitute the Scrum methodology. These entities define the structure and rules of Scrum to a large extent, and describe how a productive and efficient

working environment for the development of software systems can be set up if applied properly.

The Scrum approach has some drawbacks and problems of its own, however. First and foremost, Scrum does not offer any software engineering practices; it is left to the team to discover which design principles, testing approach, and amount of documentation are appropriate, to name a few. As stated previously, agile methodologies tend to focus on a kernel of rules, instead of providing all possible rules, and Scrum is no exception to that rule. Fortunately, the focus of Scrum on project organisation aspects makes it easy to extend with engineering principles such as the Extreme Programming principles [22].

Another drawback of Scrum that is often exposed is its reliance on the integrity and capabilities of individuals. In fact, the autonomy and authority that is given to Scrum teams requires technical qualities, honesty, management and communication capabilities. Otherwise, the team will most probably suffer from wrong or bogus decisions that individual team members make during the sprint planning meeting and the sprint itself. The high transparency that Scrum provides requires openness, the ability to deal with negative feedback, as well as sensitivity towards others. Lack of such competences will have corrosive effects on the team, as the team morale may enter a self-reinforcing downward spiral through the fearful, blameful, and negative atmosphere that may emerge. However, thinking that a project can be brought to success with any significant amount of unskilled people or people with poor integrity (as some proponents are proposing [32]), is arguably naive – or at least, it is worth posing the question if creating such environments is morally justifiable towards the more skilled team members.

An often-mention criticism of agile development methodologies in general and Scrum in particular is that it does not scale to big projects and big teams. This is true to the extent that Scrum teams are limited to a maximum size of nine developers. Scrum defines means to run several Scrum teams in parallel [107, 80], although no consensus has been yet established on the rule set to follow. To give a brief sketch of the principle for scaling up, the idea is to introduce Scrum of Scrums (SoS) to coordinate several parallel Scrums. The SoS team consists of one representative from each parallel Scrum who meet on a regular basis to coordinate the development activities amongst the teams and to provide status reports to each other. In this way, a system development effort of about 80 people (9 times 9) can be coordinated using Scrum. By applying the Scrum of Scrums principle again, the total

development effort can be scaled up to about 700 (9^3), which is already a significant increase above the initial maximum of nine developers.

2.5.5 Other software development methodologies

The first software development methodology, according to Elliot [47], was the systems development life cycle (SDLC) postulated in the 1960's, which equates roughly to the waterfall model [30]. According to [114] and [14], the first complete proposal for this software engineering approach is Winston Royce's paper [102], which suggests to perform activities of requirements elicitation, design, coding, testing, and deployment sequentially for the whole product, and to start the next activity only when the first activity has been completed. However, the waterfall approach is no fitting approach in the vast majority of currently existing software development environments, as pointed out by Krutchen [79, p. 55ff].

The spiral model [31] is a refinement of the waterfall model that proposes a more elaborate approach to planning, requirements elicitation and design of software systems. Both the spiral model and the waterfall model conclude with a linear phase in which development, testing, and deployment are performed sequentially. The spiral model is a risk-driven software development process [14, 114] that allows teams to focus the planning, requirements elicitation, analysis and design work on high-risk areas. It does not require the same amount of effort for the elaboration of all requirements and all sub-components of a software system. However, the spiral model still does not encourage revisions and backtracking from coding and testing to earlier phases of the project. Revisiting and refining the system design and documenting requirement clarifications that did arise during later phases of the project is not included in the methodology.

The Rational Unified Process (RUP) [79] is arguably the most well-known process from the Unified Process family. The processes from the Unified Process family of software development methodologies have several aspects in common, of which the focus on iterative development is, arguably, the most distinctive. The Unified Process extends the principle of iterative development of the spiral model's approach to planning, requirements elicitation and design to the whole development process. It is therefore inspired by the spiral model, but is not a simple "unrolling of the spiral"; it already constitutes a different approach to software development altogether – namely, an iterative one. The Unified Process identifies the development steps of the

waterfall model as disciplines that need to be carried out in all iterations of the project with different weights. For example, early iterations will see a heavier focus on requirements and design, while the team puts much more effort into coding (called implementation in the Unified Process) and deployment in later iterations. The Unified Process adds additional disciplines to the ones known from the waterfall model (requirements, analysis and design, implementation, test, deployment) in order to put more emphasis on the activities that are needed to enable the core disciplines: (1) business modelling (understanding the structure and dynamics of the application domain), (2) configuration and change management (tackling the management of produced artefacts and change requests), (3) project management (project planning, assessment, and control, risk and resource management), and (4) environment (creation and maintenance of necessary tooling and organisational setups). RUP defines a total of 63 artefacts that need to be created – of course, containing running code as some of those artefacts. From that number alone, it becomes clear that even though RUP did discontinue the linear approach to software development that was advertised in the waterfall model – and to some extend in the spiral model as well – it still puts a heavy emphasis on the creation of supplementary documentation. For many projects, especially smaller ones, the RUP software development methodology therefore needs extensive tailoring; unfortunately, RUP tailoring is still an activity with little provided guidance. RUP textbooks write too little about tailoring [67] (the most content on tailoring are ten pages available in [112, pp. 229 – 239]), and research papers on RUP tailoring provide only limited insight and guidelines [95, 67]. Arguably, the number of project managers who can systematically tailor RUP from a solid body of knowledge and experience must therefore remain limited.

2.5.6 Conclusion

We introduced Scrum, an agile software development methodology that promises to deliver constant feedback and the creation of a healthy and productive development team through a set of simple underlying principles, roles, meetings, and artefacts.

We have also briefly discussed the waterfall model and the spiral model from a historic perspective, before looking more closely at RUP, a more recent software development methodology. The waterfall and spiral model of software development are not a good match for the development of physio-

logical computing systems, as the process model they propose do not fit well with modern software development environments and constraints.

Looking at the drawbacks and benefits of the more modern methods, i.e. RUP and Scrum, we conclude that the flexibility and empirical approach of Scrum is a better match for the development of physiological computing systems. The Rational Unified Process, with its heavy focus on documentation and comparatively extensive process, lacks the flexibility that researching and developing activities in the new application domain of physiological computing.

The Scrum process is a software development methodology that is very well suited for risky or research-oriented software development approaches. Furthermore, the time-boxed structure of Scrum that entails regular feedback on the project's progress and achievements is a crucial factor for success in a volatile environment. Especially, periodic user tests with the developed system are necessary in physiological computing settings, as the impact of introducing such complex systems in the environment of the user may only manifest when applying them in half-real settings. For these reasons, Scrum is a perfect match for the development of physiological computing applications, and serves as the basis for the methodology that is proposed in Chapter 5.

2.6 Conclusion

In this chapter, the foundation for the software engineering approach to the development of physiological computing systems have been settled. The application domain, physiological computing, was introduced and explained in details, and the need for a more systematic approach to their development was identified. Furthermore, we discussed the terminology that is used in component-based software frameworks and implementations, and presented the platform (OSGi) on which the component-based framework for physiological computing is implemented. Chapter 3 presents the software framework that we created on top of OSGi. As the approach to formal verification of reconfigurations is contract-based and uses a dense time model, we also presented the formal theory of real-time contracts, their composition, and refinement which we extend in Chapter 4. Finally, we discussed software engineering methodologies and identified Scrum as the methodology to assemble the single bits of a software engineering approach to a consistent

methodology itself. Chapter 5 discusses the how our contributions can be combined within Scrum.

Chapter 3

Component framework for application development

Creating physiological computing systems is a challenging task that can benefit from both a systematic and structured approach and a framework for implementation. While taking a component-based or modular approach to software creation is undeniably sensible, physiological computing applications feature specific requirements that benefit from a modular approach, but requires the addition of specific features. The addition of these specific features require careful design and consideration of the impact on the usage patterns and usability of the framework, as well as the impact on consistency of the provided capabilities. The resulting component-based framework for the implementation of physiological computing software systems that is presented in this chapter is called the REFLECT framework, as it constituted the core framework of the implementation work in the REFLECT project [110].

In order to provide true implementation support, we first need to take one step back and think about the requirements that software engineers developing physiological computing applications have towards a software framework (Section 3.1). These requirements may be expressed genuinely by the software engineers, or may arise through the needs of the final users of physiological computing systems. In the following requirements elicitation process, therefore, the viewpoints of both parties, the software engineers and the final users, need to be taken into account. Once the requirements have been established, we elaborate on the concepts that form the basis of the framework design in Section 3.2, before discussing the usage of the REFLECT framework for implementation and the rationale behind the interface the

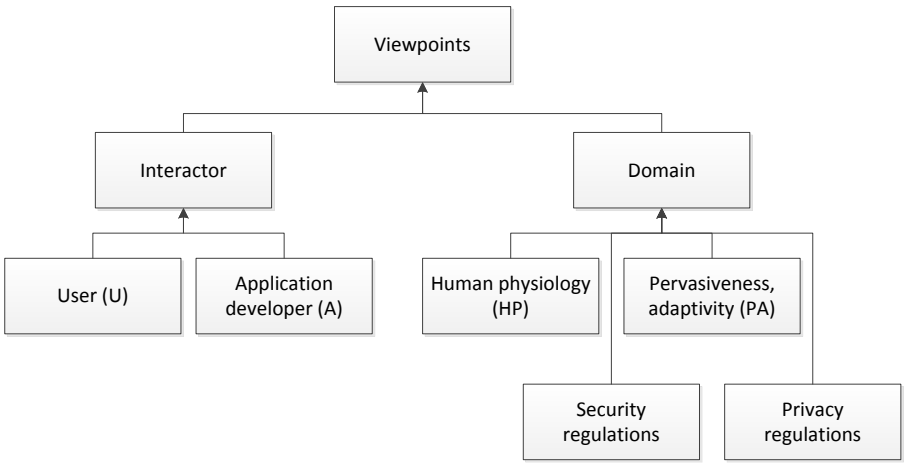


Figure 3.1: Identified viewpoints

framework offers to its users in Section 3.3.

Section 3.4 explores other existing component-based OSGi frameworks, compares the REFLECT framework with those, and highlights different design decisions that exist between the OSGi frameworks and the REFLECT framework, before Section 3.5 concludes this chapter.

3.1 Requirements

Requirements elicitation needs to follow a process, as does the overall software development. In this Section, we first describe the selected process before we proceed with the discussion of the requirements that were found.

3.1.1 Requirements elicitation process

From the multiple existing approaches to requirements engineering, we decided to use the viewpoint-based approach [115], as one of its major strengths is the consideration of domain specific requirements, the thorough analysis of possibly contradicting requirements by different stakeholders and application domain aspects, and their reconciliation. The viewpoints identified for physiological computing systems are pictured in Figure 3.1; they include two interactors, namely the user (U) and the application developer (A). The

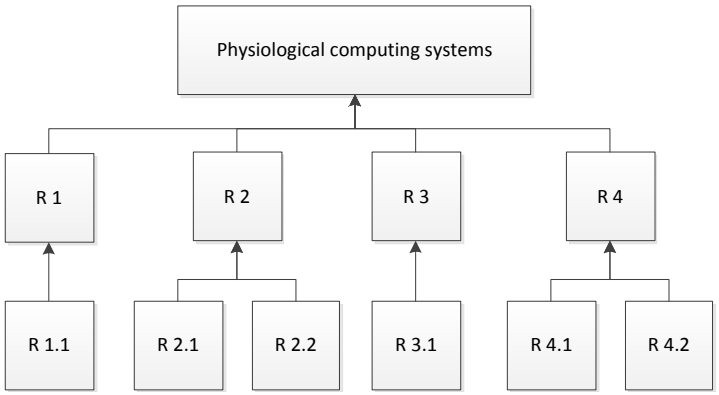


Figure 3.2: Identified requirements structure

user is understood as the person interacting with an environment of devices which constitute the physical manifestation of a physiological computing system, while application developers are all persons involved in designing and constructing software for physiological computing systems.

Beyond these interactors, four domain viewpoints were identified: pervasive adaptation (PA), which is the key concern for physiological computing system, is of course one of the domain viewpoints identified. Another domain viewpoint is human physiology (HP), which subsumes all requirements stemming from the use of physiological measures and constructs as well as from the application of physiological analysis methods and theories in physiological computing systems.

It must be noted that two of four identified domain viewpoints were purposefully left aside, namely security and privacy concerns, as they are not part of the research focus of this work.

Starting from the identified viewpoints, the requirements elicitation process used the ISO/IEC 9126 software quality model [74] as input for potential quality features that the system should exhibit. In addition, the application domain knowledge as well as a list of sample sensors and actuators, their quantitative properties and their intended use for the derivation of higher-level constructs guided the requirements elicitation.

3.1.2 Identified requirements

In the process of requirement elicitation, we started with envisioning the key features that users will expect from physiological computing systems. Then, we investigated the domain-specific challenges and requirements that software developers will face when creating physiological computing systems, and the support they may demand from a software framework in order to guide them and to simplify their task. This approach led us to the identification of four top-level requirements, R 1 to R 4 (see Figure 3.2 and Table 3.1).

View	No.	Requirement
Behavioural adaptation		
U, PA, HP	R 1	Applications should be able to adapt their behaviour on a strategic level
D, PA, HP	R 1.1	The framework should support modular and reconfigurable systems
A, PA, HP	R 2	Applications should be context-aware
D, PA, HP	R 2.1	The framework should support the creation of context-aware applications
D, PA, HP	R 2.2	The framework should support coping with changes in the computing infrastructure
Application evolution and experimentation		
A, PA	R 3	Applications should be easy to extend
A, PA	R 3.1	The systems should be modular for easier integration of new functionality
A, HP, PA	R 4	The framework should allow for experimentation and rapid prototyping of applications
A, HP, PA	R 4.1	Experimentation with various system setups, algorithms, and parameters should be supported
A, HP, PA	R 4.2	Offline testing of single algorithms and components should be facilitated through capture/replay support

Table 3.1: Software framework requirements overview

The requirements R 1 and R 2 are requirements that stem from the user viewpoint. Here, we deduced requirements that the application developer may have towards the software infrastructure to simplify the development of applications that satisfy the user requirements (R 1.1, R 2.1, R 2.2). Re-

quirements R 3 and R 4 are requirements that are genuinely expressed by application developers. Requirement R 3, especially, is a software quality requirement that emerges quickly in discussions when thinking about how to create applications in a quickly evolving domain. As experimentation plays a key role in finding working application concepts, requirement R 4 emerged soon, also.

Behavioural adaptation

R 1 Applications should be able to adapt their behaviour on a strategic level The quintessence of a physiological computing system is its ability to adapt its tangible reactions (such as changes to temperature, lighting, and music) to the user's current physical, emotional, and cognitive state. However, as detailed in Section 2.1.4, physiological computing applications need to be able to change their behaviour on a strategic level in order to offer satisfactory and entertaining behaviour to their user; a system that always reacts in the same way and does not take proactive actions can be easily labelled as primitive, even if it isn't.

R 1.1 The framework should support modular and reconfigurable systems In order to allow for change of system behaviour, it is necessary to structure the parts defining the system behaviour in a way that their mode of operation can be altered externally through system components monitoring and altering the system behaviour on a strategic level.

The software framework to create should support this separation of behaviour and meta-behaviour (i.e. monitoring behaviour and changing it), and allow the meta-behaviour to interact easily and consistently with the system it is controlling.

R 2 Applications should be context-aware The correct tangible behaviour of a physiological computing system depends heavily on the user context for adapting their behaviour accordingly [124].

Pervasive intelligent systems tend to behave wrong when they overlook important aspects of the user's context. Having a good grasp on the user's context is especially crucial when performing noticeable and impacting adaptations to the user's environment; without a correct understanding of a user's context, the impacts on the user's environment must remain small and almost unnoticeable.

Using human physiology allows to gather information about the user context from a different angle, namely from the perspective of the user itself. The physiological reaction of the user can be leveraged as hints on how the user perceives his environment, and influence the interpretation of the facts through the software system. This additional channel of information allows to increase the certainty on the user context, and allows to create systems with more positive impact on the user.

An application's context is not only comprised of the user, however. In pervasive systems especially, the computing, sensor, and actuator infrastructure is bound to change permanently. As the user changes from location to location, different capabilities are found in the vicinity of the user. Context-aware pervasive systems need to cope with these changes in the computing infrastructure.

R 2.1 The framework should support the creation of context-aware applications Gathering context, be it physiological or other, is the key means for physiological computing systems to create awareness of the physiological state of their users, by providing the basic data and the context for its interpretation. It is therefore worthwhile to assess how capturing context can be supported on the level of a software framework.

The framework needs to support software developers in gathering physiological data and context data by providing data structures, programming constructs and interfaces for the acquisition and processing of physiological and context data.

R 2.2 The framework should support coping with changes in the computing infrastructure How an application copes with appearing and vanishing capabilities in the environment of a user is very application specific; how the lack of a specific capability may be compensated is specific to each application. However, it is still worthwhile to investigate how the development of compensation strategies may be simplified through the framework.

Application evolution and experimentation

R 3 Applications should be easy to extend As the research in online-processing of psycho-physiological data is still in its infancy, physiological computing systems need means for easy extension.

R 3.1 The systems should be modular for easier integration of new functionality The framework should support a modular programming and design model that makes extending existing applications feasible and as simple as possible. As applications are bound to be revised, modified, and enhanced, this is a major requirement geared towards the framework.

R 4 The framework should allow for experimentation and rapid prototyping of applications As the software framework is intended to support development efforts in a relatively new application domain, experimentation plays a key role. It is hence crucial that new ideas and concepts can be tested easily with simple mock-ups and without having to create a whole application with tangible devices and full-fledged psycho-physiological analysis. The middleware should therefore support means for introducing simulated software components, sensors and actuators thus allowing experimentation and rapid prototyping of applications.

R 4.1 Experimentation with various system setups, algorithms, and parameters should be supported The process of developing an operational physiological computing system is not a straightforward one. Instead, we foresee that developers will have to experiment with varying configurations and test their applicability and efficiency in realising the desired functionality. Therefore, experimenting with various prototypes, changing application configurations, algorithms, system and algorithm parameters easily should be enabled by the framework.

R 4.2 Offline testing of single algorithms and components should be facilitated through capture/replay support Setting up physical experiments is a time-consuming task. Experiment preparations such as finding people willing to perform the experiments, creating experiment protocols, and creating the necessary physical setup can have a significant impact on a project's time and monetary budget. It is therefore worthwhile to investigate means to use an experiment's data to the maximum extent – especially since the generation of test data is not trivial for physiological inputs. For testing algorithm steps, it may be worthwhile to capture sensory data with the goal of replaying it to the system after the experiment. In this way, some parts of a system can be re-tested without having to re-create a full physical experimentation setup. The software framework should provide means to the

developer to create capture and replay setups for testing parts of the system.

3.2 Framework concepts

The requirements stated above call for a consistent and well-balanced set of concepts. In this section, the concepts of the software framework are discussed, and how they relate to the requirements elicited above.

3.2.1 Components

Our basic approach to the elaboration of concepts for the REFLECT framework is to start from component-based methodology. Taking a component-based approach to the development of systems allows to support the satisfaction of several requirements: Components provide a modular programming model for the creation of modular, reconfigurable systems (R 1.1), and for coping with changes in the infrastructure (R 2.2). Using a component approach also simplifies the integration of new functionality (R 3.1) by providing means for the clear structuring of software.

Providing a component approach to software development also offers benefits for experimentation. It is fairly easy to replace a component from a clearly designed system, and to replace it by a different implementation. Therefore, a component-based approach is also in-line with the requirements on easy experimentation with multiple system setups (R 4.1).

When imposing a programming model that is different from the underlying programming language, the first question immediately arising is how to combine or impose the new programming model on the existing programming language. In this case, the question arising is how to embed a component programming model in the object-oriented programming language that is Java.

The solution for this issue was guided by the following three principles.

- **Embed the component model naturally in the Java programming language.** We prefer this approach to creating a new language in itself in order to allow software developers to re-use existing programming environments, instead of creating an own environment from scratch.

- **Try to be supportive to the software developer instead of being restrictive.** Several component models follow a restrictive approach (e.g. FRACTAL [36], ArchJava [4], and JComp [65]) or correct-by-construction approach (e.g. BIP [20]) in order to guarantee a specific set of properties, and sacrifice usability in exchange. Instead, we refrained from being restrictive on the programmer, trusting in her knowing best the approach to take to realise the needed functionality.

- **Provide a model that scales well from restricted ressource environments to powerful multicore systems.** We therefore decide to support user-level multithreading, as following a "one component equals one thread" or a completely synchronous approach would not scale well over the full range of systems. Instead, we let the developer decide how to partition her application into threads of execution that may span over several components. Also, the insights gained from the early robotic programming models such as the Gat three layer architecture [59] and the Brook substumption architecture [35] show that parallel processing capabilities are of paramount importance for the successful creation of systems interacting directly with the real world.

The natural embedding of the component-based programming model is mainly achieved through three major design decisions.

- **Abstract Java classes are used to define component types.** Depending on which class is extended, the created subclass is interpreted by the model as belonging to a specific kind of component types.[1]

- **Simple unidirectional connectors.** While bidirectional or even more complex connectors are common in theoretical component frameworks (see e.g. [21] and [13]), the natural embedding into an object-oriented language is challenging. Using unidirectional connectors immediately leads to the separation of ports into provided and required ports, where provided ports offers functionality, and required ports declare dependency to a service interface to be offered by a provided port. By following this model, connectors are simplified to framework-managed object pointers.

[1] It is as well possible to use POJOs and Java annotations to achieve similar effects. The difference in usability is however so minor that we decided to stay with the subclassing approach.

- **Required and provided ports defined through Java annotations.** Using required and provided ports also allows a clean embedding of the port concept into the Java language. Provided ports can be mapped to Java getters returning a service object that implements the interface of the provided port. To distinguish a provided port from a regular Java getter, it can be tagged with an Java annotation. Similarly, a required port can be realised as tagged setter, accepting a service object from a provided port of the correct type.

- **Properties defined through Java annotations.** Component properties are simple means to provide mediated access to the internal configuration of a component. Using annotations, it is possible to annotate getters and setters of primitive types to declare inspectable and modifiable properties. Using properties, components can provide access to information about its operation, as well as access to means to tune and change its internal mode of operation.

The resulting component metamodel is shown in Figure 3.3. Components feature provided and required ports that are interconnected through connectors. Each connector is linked to a provided port instance as well as a required port instance. While the provided and required port represent the access and injection points of the component, the instances represent the invocations of the underlying methods: a provided port instance represents one invocation of the annotated getter, and holds a reference to the retrieved service object. Conversely, a required port instance represents a setter injection that was performed by the framework. Both the retrieval of the service object and the injection are linked by a connector. In this way, it is possible to track and manage all injections that were performed by the framework. Figure 3.3 also shows that a component (besides having parameters) is also associated to a Manager instance. The manager instance holds reference to all connectors and components that are managed by a framework instance and hence provides access to all entities the framework knows of.

Multithreading support is offered through active components. Active components feature an own thread of execution, allowing component e.g. to pull information from external sources (like sensors), perform computations, monitor and control physical devices, and actively initiate communication with other components.

While concurrency allows to create convenient solution designs, it needs proper thread-safe code. For this, we rely on the software developer to make

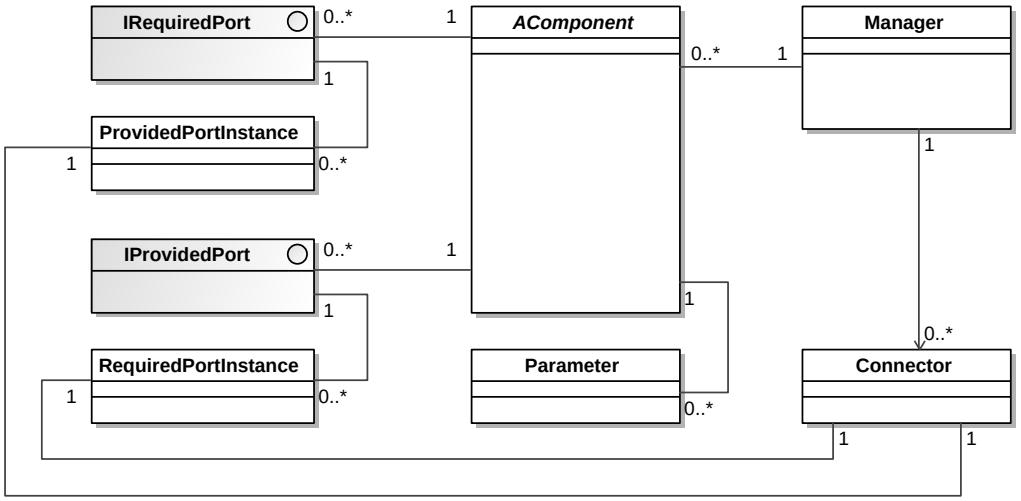


Figure 3.3: REFLECT framework component model

use of the available synchronization facilities that the Java standard library provides [62]. Since these facilities offer extensive support through a plethora of concurrent data structures, we believe that it is not necessary for the framework to add its own implementation on top. It is nevertheless worthwhile to provide support for proper synchronization of reconfigurations and normal operation through a life-cycle concept that allows to halt the normal operation of components before they undergo reconfigurations (see also Section 3.3.8).

To give an example of how the concurrent data structures available in Java are re-used, consider the communication between active components. Clearly, access to information provided by one component must be safely published to the other for access. A common approach to realise safe publication between communicating threads is to use blocking queues. For active components, the REFLECT framework therefore relies on the same concept, making use of the blocking queue interface and implementations provided by Java since version 1.5. In order to safely publish data, one component provides input queues to which other active components can safely publish data through submitting objects to the queue.

3.2.2 Reconfiguration

Physiological computing application can benefit greatly from both a component-based approach as well as reconfigurations for the creation of adaptive behaviour (R 1.1, R 2.2). With the help of reconfiguration, it is possible to model and implement the reaction to disappearing services with relative ease, as well as to adapt the behaviour of physiological computing applications to environmental changes.

Software infrastructures for performing reconfigurations often introduce abstraction layers between the actual components and the reconfiguration interface, so that the components themselves are not directly reachable through the reconfiguration interface. This often has the effect that type safety is lost, since the reconfiguration interface becomes so generic that components, component parameters and port types are referenced through untyped entities such as character strings. In order to avoid the problems of untyped references (e.g. typing and refactoring safety), we instead offer direct query and manipulation access to components through the framework's management interface.

Another issue in performing reconfigurations is the coordination with the normal flow of computation. For the REFLECT framework, we decided to implement and rely on the concept of quiescence introduced in [78]. By leveraging quiescence, it becomes feasible to change a system's configuration without having to know the exact details of the implementations of all components. Instead, it is possible to separate the concerns of reconfiguration and normal operation through a simple lifecycle interface between components and reconfiguration controller.

Finally, the question on where to put reconfiguration rules naturally emerges. Often, reconfiguration rules are put in a separate part of the framework, sometimes in a control layer. As it may depend on the application at hand which separation and structure makes the most sense (e.g. separating a global control layer into several partitions, or bundling reconfigurations with the affected systems), we decided to allow component containers to perform reconfigurations based on signals emitted by any component, and let the software developer decide how component containers featuring reconfiguration actions should be organised.

3.2.3 Distribution

Pervasive adaptive systems are by definition distributed (see also R 2.1); one of their key distinguishing properties is that they consist of several computing-enabled artefacts that blend with the physical environment of the user. It is therefore of paramount importance for the framework to support means for distribution; however, we do not follow the approach of seamless distribution as propagated by many ubiquitous computing frameworks. Instead, we follow a seamful approach to distribution as advocated by Bell and Dourish in [24]. In such an approach, awareness of the underlying topology and distribution is made available to the application instead of making it fully transparent and seamless. The benefits of this approach are that the application can leverage this distribution awareness for making tradeoffs between load on individual nodes and network stress based on the application load profile, instead of relying on the framework to make the right choice for the application.

Providing a seamful approach to distribution is a deliberate design decision that impacts the way applications are written. In the REFLECT framework, the awareness level is primitive. So far, a system knows only existing remote framework instances and their location; properties of the network such as topology, bandwidth and jitter are not visible to the local system.[2] The REFLECT framework provides a model of all known REFLECT nodes and their interconnections within a global system container. The global system container holds the local system container representing the local REFLECT node, as well as remote system stub components that represent remote REFLECT nodes, their service provisions and the services they use. The local system container references all bundle containers. Bundle containers represent OSGi bundles that are declared as potentially containing REFLECT components. Figure 3.4 shows a simple example of the object graph available for reflection on the system topology. In this example, three remote REFLECT framework instances exist that are represented by the remote system components 1, 2, and 3. The local component A that is contained within a bundle container, in turn contained by the local system, uses a service provided by remote system 1, while local component B provides a service that is used by remote system 3.

The object graph reflects changes made to the remote systems as well as

[2]Adding such properties to the network representation is simple, however. They were excluded as no necessity for them was found in the physiological computing case studies.

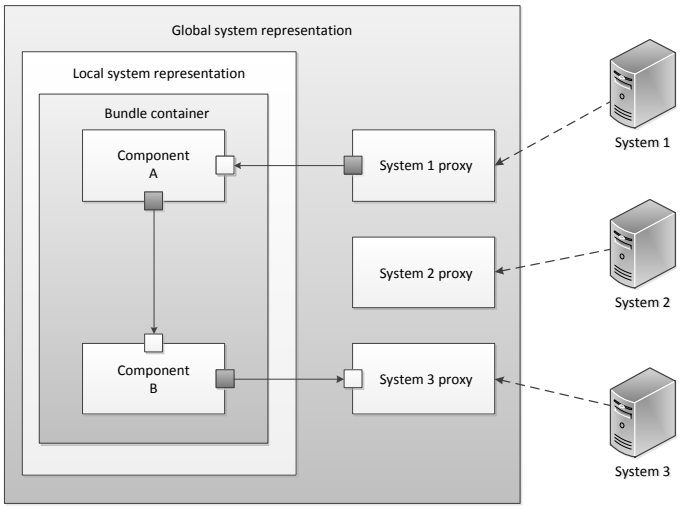


Figure 3.4: Distribution example

changes in the availability of remote systems: if a remote framework instance becomes unavailable due to a network connection loss, the remote system will disappear from the local model within a short time frame. Similarly, as a network connection is established between two framework instances, the remote systems will appear in the local model of both framework instances.

Communication between remote components, i.e., components that reside in different framework instances, is supported by the REFLECT framework through queues. Input queues exported by one component are made available to remote components, and every data object submitted to the local replica of the remote input queue is serialized, transmitted, and finally submitted to the real, remote input queue of the remote component. The decision for restricting remote communication to queue-based communication implies that distributing a system over several computation nodes does not alter the concurrency structure of the system. Two active components running locally and communicating via queues have the same concurrent behaviour as two remote components communicating remotely via queues. Contrast this with, for example, Java's remote method invocation (RMI, [101]) in which the threading behaviour is altered when the system is distributed using RMI – the call is executed in a thread different from the caller, and the call may be executed concurrently to arbitrary many other calls, and therefore the

code needs to be thread-safe where it wasn't needed before. In contrast to the RMI approach, the queue-based multithreading approach followed in the REFLECT framework allows for a more independent decision of deployment from the implementation.

3.2.4 Data analysis services

Building context-awareness (R 2.1) requires a significant amount of data processing. The software framework should therefore provide data structures and services that simplify the creation of data processing subsystems. Furthermore, the data analysis services provide a central access point for storing and retrieving data: a data context service (DCS). Providing such a central service for accessing and distributing data allows developers to focus on the creation of data processing algorithms instead of working out how to access data produced by other components. Also, the data context service facility makes capturing and replaying of sensory data easily feasible through the infrastructure (R 4.2).

The data processing facilities provided by the REFLECT framework are intended to prevent software developers from creating the same data structures over and over again. In the course of the development of several case studies, we identified that most analysis processes store a history of incoming data to perform statistical analysis on the most recent data. This statistical analysis is often performed either periodically or based on incoming events.

What is therefore needed are data structures that are capable of storing recent periods of data: data windows. A data window is similar to a ring buffer in that it is able to store a limited amount of data items. A concept distinguishing data windows from ring buffers is that the length of the period stored is determined in terms of time instead of being determined in terms of item count. A data window contains timed data items, that is to say, an item that also contains a timestamp. A data window guarantees that the difference of the newest timestamp and the oldest timestamp always stays below a predefined window length; if it does not, the older data items are discarded from the data window. In addition, data windows work just like concurrent blocking queues: they prepare the analysis components for distribution by separating the data processing steps into a chain of independent threads of execution. The goal of the data windows infrastructure is to make it easier to think of analysis tasks as a sequence of independent data producing and consuming steps.

In that sense, the separation of data analysis into several steps prepares the software architecture of the application well for distribution. It is almost trivial to distribute analysis steps over several machines once the structure of the analysis steps uses data windows as inputs and outputs.

Additionally, the concept of data windows allows to define computational operations in subclasses or decorator classes. For example, data window subclasses containing numeric data can allow to compute statistical features such as mean and standard deviation.

3.2.5 Management access

Run-time access to component parameters and to the configuration allows to play with configuration parameters and systems at run-time, thus simplifying experimentation with varying system setups (R 4.2). In fact, experimentation also requires being able to inspect the system at run-time in order to observe the internal behaviour of the system. This in turn gives developers the opportunity to gain more insights about the system than would be possible by observing the external behaviour only, or by relying on post-hoc inspection of logs.

3.2.6 Requirements coverage

To wrap up, the three basic concepts of the REFLECT framework are component-orientation, reconfiguration support, distribution services and data analysis services. The framework itself implements these concepts in the Java programming language and on top of the OSGi platform (see section 2.3), leveraging the functionality and services provided by both.

The concepts of the REFLECT framework, namely component-orientation, reconfiguration, distribution and data analysis services, cover all requirements discussed in Section 3.1.2. Table 3.2 shows how each requirement is covered through at least one concept.

3.3 Implementation

The REFLECT component framework was built with the intent to support the creation of physiological computing applications, and was tested and extended during the implementation of several case studies (Chapter 6). In

Id	Short name	Coverage
R 1	behaviour adaptation	Entailed by satisfaction of R 1.1
R 1.1	modularity and reconfiguration	Component-orientation, reconfiguration
R 2	Context-aware applications	Entailed by satisfaction of R 2.1 and R 2.2
R 2.1	support for context-aware applications	Distribution, data analysis services
R 2.2	support for coping with changes	Component-orientation, reconfiguration
R 3	easy extension and maintenance	Entailed by satisfaction of R 3.1
R 3.1	Easy integration of new functionality	Component-orientation
R 4	Support for experimentation and prototyping	Entailed by satisfaction of R 4.1 and R 4.2
R 4.1	Experimentation system setups	Component-orientation, management access
R 4.2	Offline testing	Data analysis services, management access

Table 3.2: Requirements coverage through concepts

this section, we present details of the REFLECT framework, the capabilities it provides, how they interact and how they can be leveraged. This is performed by discussing the available features and the forces that shaped them.

3.3.1 Component type definition

Defining a component type in the REFLECT framework is easily performed by creating a Java class that extends the type `de.lmu.ifi.pst.reflect.core.AComponent`. The single-argument constructor of the `AComponent` class thereby defines the identifier of the created component – component identifiers must be unique within the local REFLECT node. A simple component that does not define any behaviour or ports is shown in Listing 3.1.

Listing 3.1: Component type definition

```
public class SimpleComponent extends AComponent {

  /**
   * Default component constructor.
   *
   * @param identifier
   */
  public SimpleComponent(String identifier) {
    super(identifier);
  }
}
```

3.3.2 Component instantiation

Components need to be instantiated in order to be part of a configuration. To instantiate a component type, a container needs to create a component instance in its configuration. The configuration is created as the bundle container is instantiated. A configuration is defined in the `configure()` method of the `BundleContainer` subclass that must be declared as activator of a bundle (see Listing 3.2).

Listing 3.2: Bundle container definition

```
public class Activator extends BundleContainer {

  @Override
  protected void configure() {
    create("simple").ofType(SimpleComponent.class);
  }
}
```

Bundle Manifests are created by the Eclipse Plugin Development Tools (PDT), and do not need to be created manually. The software developer using the REFLECT framework interacts via the Manifest Editor provided by Eclipse PDT (see Figure 3.5). The manifest file itself shows the static code dependencies and the declaration of the bundle activator as seen in Listing 3.3.

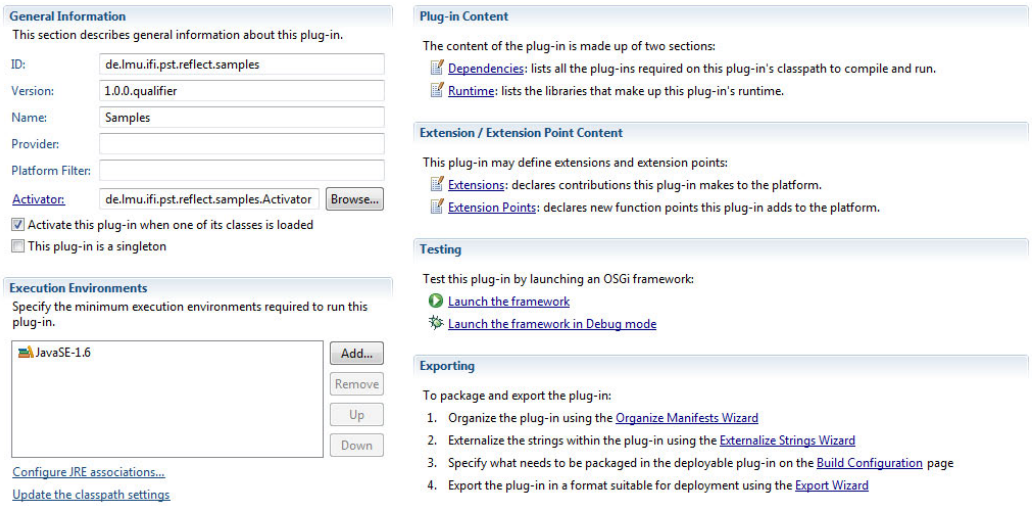


Figure 3.5: Eclipse RCP manifest editor

Listing 3.3: Manifest declaration

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Samples
Bundle-SymbolicName: de.lmu.ifi.pst.reflect.samples
Bundle-Version: 1.0.0.qualifier
Bundle-Activator: de.lmu.ifi.pst.reflect.samples.Activator
Bundle-ActivationPolicy: lazy
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Import-Package: org.osgi.framework;version="1.3.0"
Require-Bundle: de.lmu.ifi.pst.reflect.core;bundle-version
    ="1.0.0"
```

Here again, the approach of the REFLECT framework is to rely on already existing and well-tested tools instead of providing its own. This is especially advised since the REFLECT framework uses the OSGi platform, and the manifest files contain information relevant to the OSGi platform only.

3.3.3 Declaring and satisfying dependencies

Provided and required ports of components can be declared by adding getters and setters with the Java annotations `@Provided` and `@Required`. An example of how to declare a required and a provided port is shown in Listing 3.4. The parameter and return types of port methods need to be interfaces, as they may be proxied through the component framework.

Listing 3.4: Declaring required and provided ports

```
public class SimpleComponent extends AComponent {

  private IProvidedService fProvidedService;

  private IRequiredService fRequiredService;

  /**
   * Default component constructor.
   *
   * @param identifier
   */
  public SimpleComponent(String identifier) {
    super(identifier);
  }

  @Required
  public void setRequiredService(IRequiredService service) {
    fRequiredService = service;
  }

  @Provided
  public IProvidedService getProvidedService() {
    return fProvidedService;
  }
}
```

How component ports are to be connected to each other is defined by bundle containers using the configuration programming interface (Listing 3.5 shows an example use). The configuration programming interface is the interface provided to the software developer and allows to define the connections between components by means of binding rules. By using binding rules, the software developer can define the connections in a lenient or strict way, depending on the configuration needs of the concrete application.

A binding rule defines a set of required ports to which it applies, as well as a set of provided ports that may be used to satisfy the dependencies given through required ports. An example of a connection rule is shown in Listing 3.5.

A connection rule can potentially connect any required port to any provided port when their types match; the type of a provided port matches the type of a required port if the provided port type is the same or a subtype of the required port type. This matching rationale corresponds with the subtype substitutability of Java.

Listing 3.5: Defining binding rules

```
public class Activator extends BundleContainer {

  @Override
  protected void configure() {
    create("simple").ofType(SimpleComponent.class);

    connect().ports().ofComponent("simple").toAPort(name("
        provider"), ofComponent("complex"));
  }
}
```

Offering binding rules to the programmer and system developer does allow them to use very precise connector specifications as in rigorous component-based approaches. At the same time, offering binding rules allow developers to implement loose, service-style binding mechanisms that pick the first available matching port to satisfy a dependency. Binding rules also allow to choose an arbitrary position between strict component-based connectors and service-style binding regimes; they do not force to use a specific binding regime, but allow to chose the most appropriate for the situation at hand.

3.3.4 Declaring and adjusting properties

Components can declare properties to allow access to their internal configuration, which can either be read-only or write access. Read-only access can be used to give a monitoring component access to information about the current operational state of the component (e.g. it can give information about the current workload). Listing 3.6 shows how a modifiable property is declared by a set of annotated methods: the getter `getDomain` and the setter `setDomain` define the property access means, while the `getDelayDomain`

method is used to define the allowed domain for the delay property. By returing `range(0, 10000)`, the method specifies that the value of the delay property can only be set to values within the range from zero to 10,000.

Listing 3.6: Declaring a delay properties

```
public class SimpleProperty extends AComponent {

  private int fDelay;

  public SimpleProperty(String identifier) {
    super(identifier);
  }

  @Property
  public void setDelay(int delay) {
    fDelay = delay;
  }

  @Property
  public int getDelay() {
    return fDelay;
  }

  @PropertyDomain("delay")
  public RangeDomain<Integer> getDelayDomain() {
    return range(0, 10000);
  }
}
```

Changing a property can be done either directly by accessing the component setter as shown in Listing 3.7, which can be done by accessing the manager, as e.g. possible during reconfigurations. Properties can be modified through the management console as well, however, which allows to inspect and change component parameters at run-time. An example interaction with the `SimpleProperty` component on the OSGi console is shown in Listing 3.8.

Listing 3.7: Changing a delay property

```
SimpleProperty simple = (SimpleProperty) manager.getComponent
    ("simple");
int newDelay = computeAdjustedDelay(simple);
simple.setDelay(newDelay);
```

Listing 3.8: Changing a delay property via console

```
osgi> rproperties simple
      simple(ACTIVE, class de.lmu.ifi.pst.reflect.samples.
         SimpleProperty)
[
  delay=10 (int, [0..10000])
]

osgi> rsetproperty simple.delay 20
Property "delay" of component "simple" set to 20 (int)

osgi> rproperties simple
      simple(ACTIVE, class de.lmu.ifi.pst.reflect.samples.
         SimpleProperty)
[
  delay=20 (int, [0..10000])
]
```

Allowing to inspect and change component properties is important in the experimentation with component-based systems. When creating physiological computing systems, this quick interactions with component configuration allowed to give more confidence about the correct behaviour of the system to the software developer and consequently to speed up the development of the case studies.

3.3.5 Active components

Components come in different flavours, depending on whether they initiate communications on their own, or only react to invocations through other components. In the first case, the component is an active component, while in the second case, it is a regular (passive) component with no own thread of execution. The framework offers two kinds of active components: one active component that offers full management of a single thread associated with the component, and one that serves only as a shell for custom-provided threads (through either the developer, libraries, or other frameworks) and declares its active nature to the REFLECT framework.

Managed active components must subclass `AManagedActiveComponent`, and therefore implement a `step` method. The step method is a convenience the managed active component provides for creating (waiting) loops. Classic thread loops usually are structured as a while-loop that check whether to

continue the loop, and perform a blocking operation waiting for input in the loop body, and subsequently process the received input (see Listing 3.9).

Listing 3.9: Classic active loop

```
// check condition
while(conditionHolds()) {
  try {
    // blocking operation
    Object item = queue.take();
    // data processing
    process(item);
  } catch(InterruptedException ex) {
    // reset interruption status.
    Thread.currentThread().interrupt();
    break;
  }
}
// perform cleanup.
doCleanup();
```

The REFLECT framework has to provide windows of opportunities for reconfigurations. In the classic loop however, there is no window for reconfiguration that can be naturally provided. In order to address this issue, the framework reorganises the classical processing loop as shown in Listing 3.10. Here, the body of the loop is contained in the `step` method that additionally returns whether the loop condition was true in the end, indicating whether the step method should be executed another time. As there is often no other condition to check besides checking whether the component thread was interrupted, the if-statement can be eliminated, and the `step` method can simply return `true`; the managed active component code takes care of checking this condition before calling the `step` method. Furthermore, Listing 3.10 shows that the programmer does not need to deal with thread interruption and resetting the thread interruption status; this can all be handled through the framework. For cleanup tasks, the "step-loop approach" needs an additional callback that is invoked in the end.

Through the introduction of step-loops, the framework can perform reconfigurations outside of the loop step and cleanup methods, thus allowing to keep reconfigurations orthogonal to the regular processing of active components, while keeping the programming interface relatively simple and clean.

Listing 3.10: Step-loop approach

```
protected boolean step() throws InterruptedExcpetion {
  // check condition.
  if(conditionHolds()) {
    // blocking operation
    Object item = queue.take();
    // data processing
    process(item);
    return true;
  } else {
    return false;
  }
}

protected void stopInternal() {
  // perform cleanup.
  doCleanup();
}
```

3.3.6 Active component synchronization

Active components are components featuring threads of execution, and hence the communication between them needs to be properly synchronized, or at least, data passed from one component to the other must be safely published [62]. As the Java API provides several means for safe publication of data, we refrain from introducing own means for structuring concurrent code in the REFLECT framework. Developers are allowed to use any means suitable for their application: synchronized or concurrent collections, atomic references, and queues are all valid approaches for what the REFLECT framework is concerned. Still, as is discussed in Section 3.3.12 below, the component-based distribution facilities of the REFLECT framework rely on the use of queues for communication. Here, the queue interfaces and data structures used are the ones provided by the Java API.

To allow for simple, thread-safe and smooth communication between active components, it is best to use bounded queues in which one of the communication partners inserts messages while the other partner retrieves them. For communication via queues, the concepts of provided and required ports are re-used. For communication over queues, the component owning the input queue must feature a provided port allowing access to the input queue.

Conversely, the component wanting to submit a message to the input queue of another component must feature a required port allowing to inject the input queue of another component. The framework offers additional support for this communication style by providing repliable messages. With repliable messages, it is possible to pass the result of a computation back to the component that posted that message.

The following listings show an example of communication through bounded queues using repliable messages. Listing 3.11 shows an example Pi computation server that provides a simple services to compute Pi up to a requested precision on a remote machine. Listing 3.12 shows the message type that is used: a subtype of `RepliableMessage` whose reply is a `BigDecimal` instance. The Pi computation server is a subclass of `AManagedActiveComponent` and uses its `step` method to process the computation queue. The computation queue itself is created by the Pi computation server and made available through a provided port.

Listing 3.11: Pi computation server component

```
/**
 * Component offering computation resources. Pi Computation
 * requests can be submitted through its input queue, see
 * {@link #getQueue()}.
 */
public class PiComputationServer
        extends AManagedActiveComponent {

  public final BlockingQueue<Computation> fQueue;

  public PiComputationServer(String identifier) {
    super(identifier);
    fQueue = new ArrayBlockingQueue<Computation>(10);
  }

  @Provided(messageType=Computation.class)
  public BlockingQueue<PiComputation> getQueue() {
    return fQueue;
  }

  @Override
  protected boolean step() throws InterruptedException {
    // retrieve computation request.
    PiComputation computation = fQueue.take();
    // perform the computation.
```

```
      BigDecimal result = computePi(computation.getPrecision());
      // set the result.
      computation.set(result);
      // continue with the next.
      return true;
    }
  }
```

The Pi computation client shown in Listing 3.13 requires a Pi computation queue and submits Pi computations to it in its `step` method. Note that the Pi computation client waits two seconds for the result of a computation and decides to stop (by returning `false`) as soon as a computation exception occurred or as soon as the deadline is reached. In this way, the computation clients holds all values of π up to a precision that is computable within two seconds by the computation server.

Listing 3.12: Pi computation message type

```
/**
 * A computation request for computing Pi up to a specified
    precision.
 */
public class PiComputation
        extends RepliableMessage<BigDecimal> {

  private static final long serialVersionUID = 1L;

  private final int fPrecision;

  public PiComputation(int precision) {
    fPrecision = precision;
  }

  /**
   * @return the precision of the requested computation
   */
  public int getPrecision() {
    return fPrecision;
  }

}
```

Listing 3.13: Pi computation client component

```
/**
 * Component using a {@link PiComputation} queue to compute Pi
    .
 */
public class PiComputationClient extends
   AManagedActiveComponent {

  private BlockingQueue<Computation> fQueue;

  private final List<BigDecimal> fResults;

  private final AtomicInteger fPrecision;

  public ClientComponent(String identifier) {
    super(identifier);
    fPrecision = new AtomicInteger(0);
    fResults = synchronizedList(new ArrayList<BigDecimal>());
  }

  @Required(messageType=Computation.class)
  public synchronized void setQueue(
      BlockingQueue<Computation> queue) {
    fQueue = queue;
  }

  private synchronized BlockingQueue<Computation> getQueue() {
    return fQueue;
  }

  /**
   * @return all computations of Pi done so far.
   */
  @Provided
  public List<BigDecimal> getResults() {
    return unmodifiableList(fResults);
  }

  @Override
  protected boolean step() throws InterruptedException {
    // get Pi with the next higher precision.
    PiComputation computePi = new PiComputation(fPrecision.
        incrementAndGet());
    // put computation request in the queue.
    getQueue().put(computePi);
```

```
      try {
        // wait for the result (max 2 secs), and store it.
        fResults.add(computePi.get(2, SECONDS));
        // continue with the next computation.
        return true;
      } catch (ExecutionException e) {
        getLogger().warn("Exception during execution", e.
            getCause());
      } catch (TimeoutException e) {
        // computation timed out, cancel the last computation.
        computePi.cancel(true);
      }
      // computation failed, stop the loop.
      return false;
    }
  }
```

The examples show how the communication between active components can make use of existing synchronization data structures available in the Java API. The approach of the REFLECT framework to the synchronization of concurrent threads in active components is that of re-using the synchronization facilities provided by the Java platform.

3.3.7 Event-based communication

When creating software systems in general, having an event-based, publish/-subscribe style of communication instead of a direct point-to-point communication allows for a more loose means of communication. In event-based communication, the responsibility for wiring is different than in point-to-point communication. In point-to-point communication, the container of the communicating components is deemed responsible for establishing (and removing) communication links. In the event-based communication scheme, the subscriber is responsible for the wiring, and the wiring is established by subscribing to a specific event topic. Event topics are referred to by publishers to notify their subscribers that a specific event just happened.

The REFLECT framework offers support for an event-based publish/-subscribe style of communication by providing base classes for events, simple event publishing and event subscription means. Distribution of events is provided through an event bus operating in the background. The event bus features a queue to which every event is enqueued, and a thread working off

the event queue and distributing events to their recipients.

Listing 3.14 shows a simple event class for sending notifications about vanishing sensors. It declares a common event topic for all instances of the event class that is used in the constructor of every event instance. Furthermore, events take an event source as parameter to guarantee that the sender of the event is not notified even if it subscribed to the topic of the event it sent; specifying `null` as event source is also valid, and entails the notification of all topic subscribers of the event.

Listing 3.14: Example sensor removed event

```
public final class SensorRemovedEvent extends AEvent {

  public static final String TOPIC = "sample.
      SensorRemovedEvent";

  private final String fSensorId;

  public SensorRemovedEvent(Object src, String sensorId) {
    super(src, TOPIC);
    fSensorId = sensorId;
  }

  public String getSensorId() {
    return fSensorId;
  }
}
```

Listing 3.15 shows a very basic sensor tracker component. By invoking the `event` method with a newly created `SensorRemovedEvent` instance in the `sensorRemoved` method, a new event is submitted to the event bus for distribution to all subscribers of the event topic.

Listing 3.15: Example sensor tracker component

```
public class SensorTracker extends AComponent {
  public SensorTracker(String identifier) {
    super(identifier);
  }

  private void sensorRemoved(String sensorId) {
    event(new SensorRemovedEvent(this, sensorId));
  }
}
```

Subscribing to events is done by declaring a single-parameter method with an `@EventListener` annotation. The annotation includes the event topic to use for subscribing the method. As the method declares an event type as parameter, the subscription is furthermore narrowed down to events featuring the declared event topic and being assignable to the specified parameter type – in the example shown in Listing 3.16, these constraints guarantee that the method is only invoked with instances of `SensorRemovedEvent`.

Listing 3.16: Example sensor event receiver component

```
public class SensorUser extends AComponent {

  public SensorUser(String identifier) {
    super(identifier);
  }

  @EventListener(topic=SensorRemovedEvent.TOPIC)
  public void eventHappened(SensorRemovedEvent event) {
    // process the event.
  }
}
```

The event-based, publish/subscribe style of communication found wide adoption in the case studies (see also Chapter 6), which is a good argument supporting the case for different styles of interaction and communication amongst components. If the communication means were restricted to solely point-to-point connections, it would have been up to the application developer to create a publish/subscribe management component that replicated the functionality of the event bus the framework provides.

3.3.8 Component lifecycle

Components need to be put into a quiescent mode before they can be subdued to reconfigurations. For this, it must be possible to start and stop active components, as well as to wait for the active component to reach the quiescent state. While this is sufficient in theory, it is advisable to introduce a more complex lifecycle for all components. Components may want to perform some initial computation steps as soon as they are becoming part of the system configuration, or after all their dependencies have been satisfied. Therefore, it is advisable to introduce activation and deactivation callbacks, as well as initialisation callbacks. The resulting component lifecycle with initialisation

and activation callbacks as well as transitional states is shown in Figure 3.6. The transitional states are introduced as the components may be queried for their states during the transition from one stable state to another.

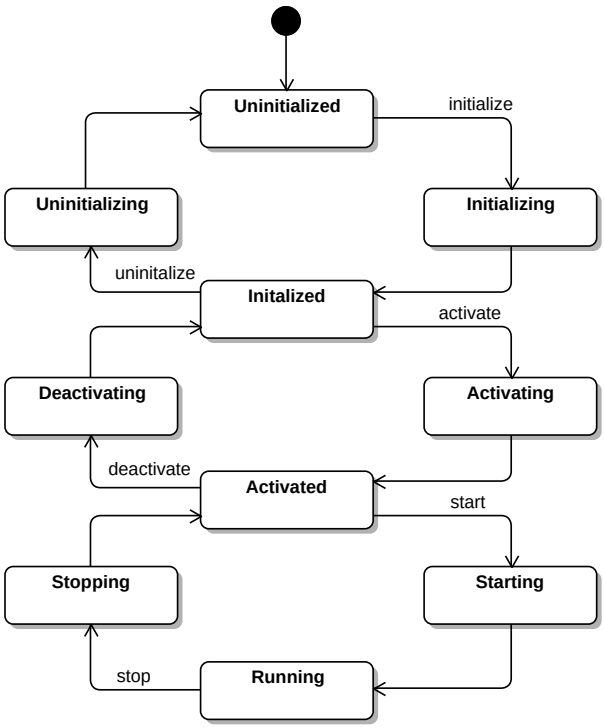


Figure 3.6: Component lifecycle state machine

The component lifecycle of REFLECT components is more complex than one may initially expect, but by being so complex, it not only allows to put active components into a quiescent state, but also allows components to perform actions on initialisation (with no guarantee on bindings of required ports) and activation (after all required ports were bound), which is a necessity in real-life software systems.

3.3.9 Reconfiguration support

Reconfigurations can constitute a helpful feature in the development of physiological computing applications (see also Section 2.1.4). Therefore, the RE-

FLECT framework offers reconfiguration support through its bundle containers. Bundle Containers can submit configuration actions to the framework. Configuration actions get access to a configuration manager instance that allows to query components and connectors. A configuration action's code can navigate from a component to all provided and required ports it contains as well as to the connectors that are outgoing or incoming at a port.

The configuration manager allows configuration actions to query the system configuration to retrieve existing components and connectors, and by this also to perform the canonical changes to the configuration of a system (as discussed in [78]):

- **Create and delete components.** Created components are added to the component configuration, while connectors from and to deleted components are removed, and components that depend on the deleted component are made quiescent.

- **Adding and removing connectors.** Adding connectors means retrieving a service object from the providing component, and injecting it into the requiring component. Removing connectors makes all components that depend on the connection quiescent before removing the connector.

Additionally, the configuration manager allows to perform the following actions:

- **Querying and changing the component lifecycle state.** Components offer methods for changing their lifecycle state. It is therefore possible to start, stop, activate, or deactivate a component that is queried.

- **Querying and changing component parameters.** Components can offer parameters that affect their mode of operation if changed (cf. Figure 3.3). Obviously, once a component is queried, all component parameters can be accessed and altered.

- **Direct interaction with components.** Since the REFLECT framework provides a direct pointer to the component object instead of a representation, configuration actions can interact directly with components through classical means (i.e. method invocations) that are directly processed by the component.

- **Make component quiescent.** It is sometimes necessary to manually put component in a quiescent state before setting parameters or interacting with the component. For such situations, the configuration manager allows to makes single components quiescent for the time the modifications are performed.
- **Schedule component for restart.** Components that were stopped or deactivated manually can be scheduled for restart after the configuration action has been completed. This is a task that must be performed in virtually every configuration action, and therefore support for this is offered through the manager.

In order to understand how these features can be used in reconfiguration code, it is worthwhile to look at an example. The following listings give an example of a basic connection switching reconfiguration. First, let us introduce the components and the configuration that should undergo reconfigurations. Listing 3.17 shows an active component that processes an input queue by emitting the content of the queue to the system console, while Listing 3.18 shows an active component putting messages in an output queue it is bound to through a connector.

Listing 3.17: Example message receiver

```
public class Receiver extends AManagedActiveComponent {
  private final BlockingQueue<String> fInput;

  public Receiver(String identifier) {
    super(identifier);
    fInput = new ArrayBlockingQueue<String>(10);
  }

  @Provided(messageType = String.class)
  public BlockingQueue<String> getInput() {
    return fInput;
  }

  @Override
  protected boolean step() throws InterruptedException {
    String string = fInput.take();
    System.out.println(getIdentifier() + ": " + string);
    return true;
  }
}
```

Listing 3.18: Example message sender

```
public class Sender extends AManagedActiveComponent {

  private BlockingQueue<String> fOutput;

  private final AtomicInteger fCounter;

  public Sender(String identifier) {
    super(identifier);
    fCounter = new AtomicInteger(0);
  }

  @Override
  protected boolean step()  throws InterruptedException {
    fOutput.put("" + fCounter.incrementAndGet());
    return true;
  }

  @Required(messageType=String.class)
  public void setOutput(BlockingQueue<String> target) {
    fOutput = target;
  }
}
```

The system configuration consists of one sender and two receivers with the sender connected to the first receiver, as shown in Listing 3.19. The listing also shows that the bundle container starts a reconfiguration every half second, switching from one receiver to the next. Listing 3.20 finally shows the reconfiguration action itself. The reconfiguration action looks up the output port of the sender component, and disconnects the connector. Next, it looks up the input provided port of the component to connect to, and connects both required and provided ports directly.

Listing 3.19: Example reconfiguration bundle container

```
public class Activator extends BundleContainer {
  @Override
  protected void configure() {
    create("sender").ofType(Sender.class);
    create("receiver 1").ofType(Receiver.class);
    create("receiver 2").ofType(Receiver.class);
    connect().ports().ofComponent("sender").toAPort().
        ofComponent("receiver 1");
  }
```

```
  @Override
  public void start(BundleContext context) throws Exception {
    super.start(context);
    Thread t = new Thread() {
      public void run() {
        try {
          while(true) {
            Thread.sleep(500);
            reconfigure(new ReconnectAction("receiver 2"));
            Thread.sleep(500);
            reconfigure(new ReconnectAction("receiver 1"));
          }
        } catch(InterruptedException ex) {
          // aborting the loop.
        }
      };
    };
    t.start();
  }
}
```

Note that the reconnection action deactivates the sender component by disconnecting the connector between sender and receiver, thereby leaving the required port of the sender component unsatisfied. By connecting the required port of the sender component again at the end of the configuration action, the sender component is automatically checked for dependency satisfaction, and started since its dependencies are again satisfied.

Listing 3.20: Example reconfiguration action

```
public class ReconnectAction extends AConfigurationAction {

  private String fTarget;

  public ReconnectAction(String target) {
    super("to " + target);
    fTarget = target;
  }

  @Override
  protected void reconfigure(ReconfigurationReflectManager
      manager) {
    IRequiredPort sender = manager.getComponent("sender").
        getRequiredPort("output");
```

```
      if(sender.getInstances().isEmpty()) {
        return;
      }
      AConnector connector = sender.getInstances().get(0).
          getConnector();
      try {
        connector.disconnect();
      } catch (ReflectException e) {
        // disconnecting failed.
        return;
      }
      IProvidedPort receiver = manager.getComponent(fTarget).
          getProvidedPort("input");
      manager.connect(sender, receiver);
    }
  }
```

In the REFLECT framework, a reconfiguration action is imperative code, in contrary to many other frameworks that offer reconfiguration support. Often, reconfigurations need to be expressed in terms of rules or in a very constrained declarative language [65, 37, 38]. The benefits of a rule-based or declarative approach is that the reconfiguration is made directly amenable to formal analysis. However, implementing reconfigurations as imperative code makes it easier for software developers to realise reconfigurations in all details and to decide on how to group the complexity of the process in intelligible units of code. Even though obviously a direct formal analysis becomes very hard, it is still possible to create abstractions of the reconfiguration code in a declarative form and to formally verify the correct behaviour of the abstraction. This approach is common for regular programs, and can be applied to reconfigurations as well.

Indeed, a systematic means for the verification of correctness of reconfigurations is advised, as it becomes very hard to predict the effects of the totality of all reconfiguration rules on a running system. This is especially true if the reconfiguration rules do not rely on component identities but rather on types and component structure for identifying areas of change. In this scenario especially, it becomes increasingly difficult to predict the result of repeated applications of configuration rules on the system configuration, as the correct systematic enumeration of all possible system configurations is difficult to perform without tooling support. For this reason, Chapter 4 introduces a means for the formal verification of systems under reconfiguration.

3.3.10 Management console

The REFLECT framework offers a management console to the developer that allows to interact with the system configuration at run-time. For experimenting with different system configuration, the management console allows to inspect the current system configuration (Listing 3.21 gives an example console output for the example of Section 3.3.9), and to manipulate the current system configuration by changing component parameters, changing a component life-cycle state, or executing reconfigurations registered by name to the manager.

Listing 3.21: Console output for status command

```
osgi> rss
GlobalSystem
[
  LocalSystem
  [
    uuid: e1405bd5-d7b3-4c34-b1a0-87c2d7140593,
    required
    [
    ],
    provided
    [
    ]
    Activator(class de.lmu.ifi.pst.reflect.sample.
       reconfiguration.Activator)
    [
      sender(RUNNING, class de.lmu.ifi.pst.reflect.sample.
         reconfiguration.Sender)
      receiver 1(RUNNING, class de.lmu.ifi.pst.reflect.sample.
         reconfiguration.Receiver)
      receiver 2(RUNNING, class de.lmu.ifi.pst.reflect.sample.
         reconfiguration.Receiver)
    ]
    sender.output -> receiver 1.input
  ]
]
```

The REFLECT management console is realised an extension of the OSGi console provided by the Equinox framework. A key factor that allowed for rapid adding of new commands to the REFLECT management console was the central manager; with a central configuration manager in place having access to all components and connectors, as well as to the representation

of the overall system topology, it became easy to create new commands as needed.

3.3.11 Data services

In physiological computing systems, the analysis or state abstraction subsystems often involve complex data processing. In these data processing subsystems, the need for sharing and re-use of intermediate computation results among several analysis and decision processes often arises: the same results are needed in more than one process further down the process chain [105]. This fact makes the configuration of component-based data processing subsystems very involved, as connections between all consumers and the producer of the data need to be established. Instead, an organisation approach inspired by the blackboard pattern [68] can reduce the configuration overhead significantly. In the REFLECT framework, we therefore provide a data service layer that roughly follows the blackboard pattern. The core of the data service infrastructure is the data context service (DCS) that manages data in a centralised and shared data storage. Through the centralised architecture, data can be easily published and made available to multiple consumers. The data managed by the DCS is organised in *data channels*. Each data channel consists of a timed series of data stored in a *data window*. Figure 3.7 illustrates the architecture by showing the data context service with a nexus sensor device creating raw skin conductance and skin temperature data, which is queried by the `FeatureExtractor` component that creates from that data the skin conductance[3] and skin temperature features, which are pushed into the respective channels.

The channels that are managed in the data context service are both data sources and targets, and therefore implement both the `ITimedDataTarget` and `ITimedDataSource` interfaces (see Figure 3.8). In fact, the data context service uses data windows to store data items. Data windows implement `IDataWindow` and store a list of data items sorted by time of acquisition or creation – data items are time-stamped. Using this design, it is possible to store a satisfactory amount of data into memory, while keeping the management of the data structure fairly simple. Data windows are self-managed, ordered sequences of a defined length. However, the length of data windows

[3]To be precise, the feature that is computed is the skin conductance level, in contrast to the skin conductance response.

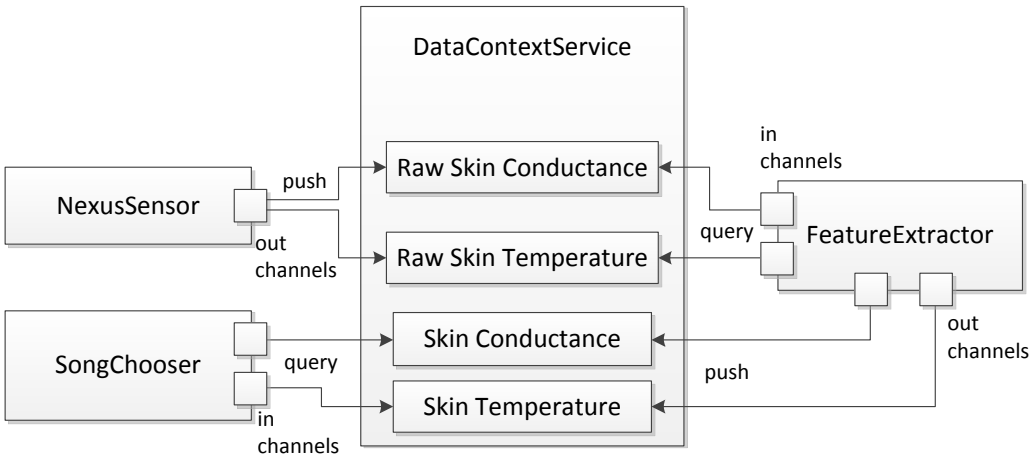


Figure 3.7: Data service architecture

is not described in terms of data item count, but in terms of time. Hence, it is possible to create a data window containing only the most recent five minutes worth of data that discards older data items as new data is appended, similarly as a ring buffer operates. Different to ring buffers, the physical capacity is adapted dynamically to store all received data items whose age is below the maximum age limit.

`IDataWindow` also allows creating a window slice, i.e. a new data window that references only to a part of the original data window. Slices can be created by referencing points in time relative to both the beginning and the end of the underlying window using positive or negative integers. Interestingly, zero can be effectively used to reference both to the beginning and the end of the slice depending on whether the start or the end of the slice is specified: for specifying the start of the slice, zero and positive integers represent instants of time relative to the beginning of the window, while negative integers represent instants of time relative to the end of the underlying window. It is not necessary to be able to reference the end of the window, as any slice starting at the end of the data window would be empty. Conversely, when defining the slice end, zero and negative integers can be used to reference instants relative to the end of the data window. Again, every slice ending at the start of the data window would be empty, which is why zero can now be used for referencing the end of the slice instead of its start.

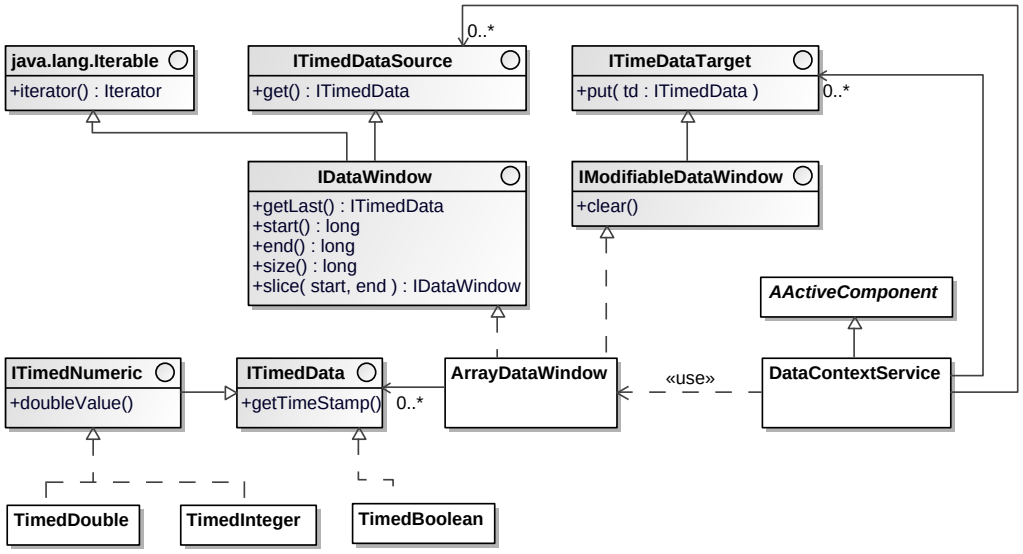


Figure 3.8: Data service data structures

Note also that the data analysis layer also defines the `ITimedNumeric` interface (see Figure 3.8) and the data types `TimedDouble` and `TimedInteger`, as numerical data is the most common data type in physiological computing applications. In the example shown in Figure 3.7, all data windows contain timed double data.

The REFLECT framework offers additions to simplify the use of the data analysis layer in general and channels in particular. Subclasses of `AComponent` exist that offer annotation-based support for accessing data channels in the data context service: `AnalysisComponent`, `AListeningAnalysisComponent` and `APeriodicAnalysisComponent`. Using the `@Channel` annotation on correctly typed setters, it is possible to get data windows injected directly for either querying them (as data windows) or pushing data to them (as data targets). Listing 3.22 an example usage of the `@Channel` annotation as well as the slicing that data windows provide.

The `MeanDiff` template method `compute()` is periodically called every second, and computes the difference between the last minute's mean and the last five minute's mean of the input data. On the injection of the input window in the `setInput` method, a slice with the last minute of data is created as well as a slice with the last five minutes of data. The `setOutput` method

with its `ITimedDataTarget` parameter indicates that the injected channel is used only for putting computation results into; the `compute` method shows how a new timed double item is added to the output channel each time it is called.

Listing 3.22: Mean diff analysis component

```
public class MeanDiff extends APeriodicAnalysisComponent {

  private IDoubleWindow fLastMinute;

  private IDoubleWindow fLastFiveMinutes;

  private ITimedDataTarget<TimedDouble> fOutput;

  public MeanDiff(String identifier) {
    super(identifier);
    setIntervalTime(1, SECONDS);
  }

  @Channel("input")
  public synchronized void setInput(IDoubleWindow window) {
    fLastMinute= window.slice(-1, 0, MINUTES);
    fLastFiveMinutes= window.slice(-5, 0, MINUTES);
  }

  @Channel("output")
  public synchronized void setOutput(
      ITimedDataTarget<TimedDouble> output) {
    fOutput= output;
  }

  @Override
  public void compute() throws InterruptedException {
    double result =
        fLastMinute.mean() - fLastFiveMinutes.mean();
    fOutput.put(new TimedDouble(result, now()));
  }
}
```

In addition to the `APeriodicAnalysisComponent` that was used in the example of Listing 3.22, the framework offers a `AListeningAnalysisComponent` whose `compute` method is called every time a new data item is inserted into a data window that is injected into a `@Channel` setter that is additionally

annotated with `@Listen`.

Listing 3.23 finally shows how channels and analysis components are created in a bundle container configuration. The REFLECT framework offers extensions to the basic rule-based configuration API (see Sections 3.3.2 and 3.3.3) by offering a `loop`-rule allowing to create or reference analysis components and a `map`-subrule to map the local names (given in the `@Channel` annotation) to channels of the data context service (given through channel descriptors).

Listing 3.23: Analysis activator for mean diff example

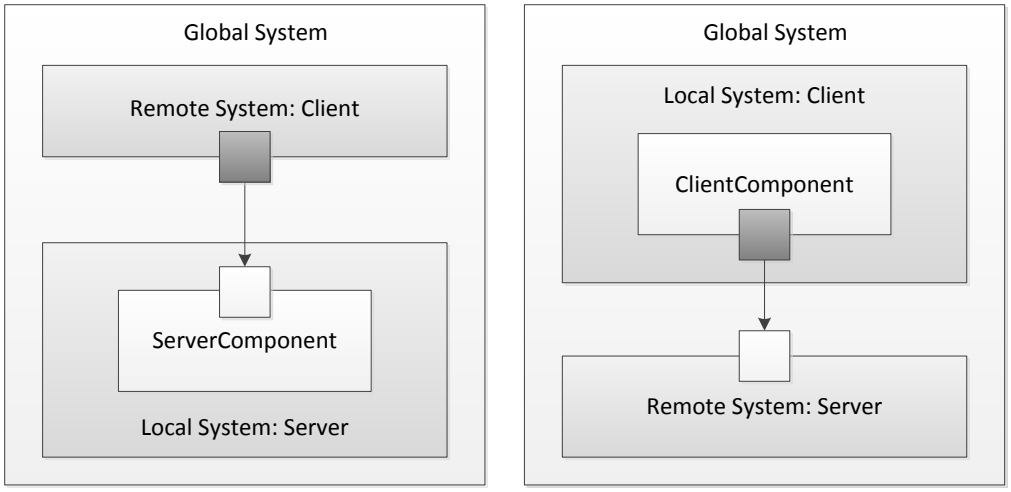
```
public class Activator extends AnalysisContainer {

  @Override
  protected void configure() {
    ChannelDescriptor hrv = doubleChannel("hrv", 1, HOURS, 10)
        ;
    ChannelDescriptor meanDiff = doubleChannel("meanDiff", 10,
         MINUTES, 1);

    loop("meanDiff", MeanDiff.class).map("input", hrv).map("
        output", meanDiff);
  }
}
```

The data analysis layer offered a powerful and helpful frame for the creation of several case studies (see Chapter 6). Especially the channel approach helped re-structuring the process of data analysis in single units of code that allowed for a) independent testing and b) easy distribution. In fact, the data analysis layer's functionality was extended by using the REFLECT framework's distribution infrastructure (see Section 3.3.12) to great effects for realising transparent remote access to data channels.

3.3.12 Distribution

Both in the general concept of physiological computing systems and in most case studies that were created with the REFLECT framework, distributed operation, communication and cooperation of distributed software systems played a key role.

In the automotive case study for example (see Section 6.1.3), the final computation setup consisted of three ultra-mobile PCs, with various sensors



(a) Remoting server view (b) Remoting client view

Figure 3.9: Remoting client and server view

and actuators attached to them, and with three screens to show information to the user on different levels of detail. It is therefore of paramount importance to distribute data and information across the three available computing nodes.

The REFLECT framework therefore supports several means of distribution that are based on the awareness of the existence of remote instances of the framework. In this sense, the distribution means provided by the framework are not fully transparent and seamless, but require awareness of the system topology from developers. This allows the system developer to leverage this exact topology for performance optimisations. It is possible to reduce the amount of data serialized, sent over the network, and deserialized if one knows which computation nodes require the data.

The REFLECT framework stays aware of remote nodes by discovering them either through querying a pre-defined list of IP addresses or by broadcasting queries using the service location protocol (SLP) [64]. Once a remote system is discovered, it is queried for the services it provides. The remote system is represented locally as a component with its offered services as provided ports. The services remote hosts consume from local components are represented as required ports that are bound to the respective provided ports of the local system.

Figure 3.9 shows the computation example from Section 3.3.6 as deployed within two instances of the REFLECT framework. Figure 3.9a shows the view of the server as one component remotely accessing the provided port of the server: a required port of a remote system is connected to the local server component. Figure 3.9b shows the client view: here, a remote system provides a computation queue port, and the required port of a local client component is connected to that remote port. Note that the distribution structure and hence also the remote communication remains transparent for the component developer (although not for the creator of system configurations): the framework takes care of transferring the queued messages to the remote system, as well as to transfer replies to messages back to the initial sender. Queue-based communication is not the only means of remote data transfer that the framework supports. To begin with, the event bus was extended to transparently propagate events to all connected systems. Furthermore, data windows containing data produced and consumed by analysis components, sensors, and TCP/UDP clients and servers can be easily replicated through the REFLECT framework. For this, the domain specific configuration language was extended to allow to specify the need for local replication of remotely available data windows. The software developer needs only to specify the name, type, and location of the remote window. Additionally, the amount of data to be replicated locally as well as the update rate can be specified, that is to say, how often data updates has to be sent to the local data window replica. In this way, the network traffic as well as the CPU load for data transmission can be fine-tuned. Note also that by providing an independent window length specification, it is possible to replicate only a subwindow, or a history of the original remote data window.

Listing 3.24 shows a snipplet from the real automotive demonstrator configuration, and how data from the second ultra mobile PC (UMPC) is replicated on the third UMPC to be shown graphically on the third UMPC display.

Listing 3.24: Distributed data channel configuration

```
public class Activator extends ReflectContainer {

  @Override
  protected void configure() {

    // create data context service
    createDataContext();
    ChannelDescriptor ST = distributedChannel("skin
        temperature sensor", TimedDouble.class, 10, TimeUnit.
        MINUTES,
         200, UMPC.TWO);
    ChannelDescriptor SC = distributedChannel("skin
        conductance sensor", TimedDouble.class, 10, TimeUnit.
        MINUTES,
         200, UMPC.TWO);
    ChannelDescriptor ppv = distributedChannel("ppv",
        PPVMeasurement.class, 1, TimeUnit.MINUTES, 200, UMPC.
        TWO);

    // the channel descriptors can be used just as regular
        descriptors.
  }
}
```

3.3.13 Capture and replay support

In physiological computing applications, testing often means deploying the software on a hardware testbed, and running the application with one or several test persons, reading physiological data from the test persons, monitoring the output of the physiological computing system, and recording the behaviour for further off-line investigation. The overhead that is induced by this procedure makes quickly clear that the physiological input to the system (as well as other inputs from the environment) is valuable. Recording system inputs allows to perform quick off-line tests directly on developer machines (or, more generally, developer-friendly environments such as developer-site test environments) by replaying the recorded inputs.

Here, the design decision of having a centralised data storage for all input data can be leveraged to great effects: since the input data is under control of the Data Context Service, it provides a single point of access for tapping

into data streams and recording them. Similarly, the Data Context Service can be used to replay the data to the system without needing to alter the analysis algorithms for testing purposes.

Listing 3.25: Capture and replay configuration example

```
public class Activator extends AnalysisContainer {

  @Override
  protected void configure() {
    ChannelDescriptor hrv = doubleChannel("hrv", 1, HOURS, 10)
        ;
    ChannelDescriptor meanDiff = doubleChannel("meanDiff", 10,
         MINUTES, 1);

    loop("meanDiff", MeanDiff.class).map("input", hrv).map("
        output", meanDiff);
  }
}
```

In the REFLECT framework, it is possible to create capture configurations for recording the input data using the management console; we demonstrate this feature using the configuration given in Listing 3.25 as the basis. Listing 3.26 shows how a capture configuration is created that captures the data from the `hrv` data window to the `capture.txt` file. The capture configuration specifies the capture file to which the data is recorded, and specifies which channels are to be recorded to the file, and how the channel data has to be serialized. Finally, the `capturestart` and `capturestop` commads tell the Data Context Service to start and stop capturing data, respectively.

Listing 3.26: Capture configuration example

```
osgi> capturewindow hrv double
capture for data window "hrv" using converter
   TimedDoubleConverter added.

osgi> capturefile "capture.txt"
capture file "capture.txt" created

osgi> capturestart
capture started

osgi> capturestop
capture stopped
```

Listing 3.27 shows how to configure a replay run through the management console by similar means: the file to replay from is specified, which channels are replayed from the file, and how the data is parsed from the file to create data objects.

Listing 3.27: Replay configuration example

```
osgi> replayfile "capture.txt"
replaying from file "capture.txt"

osgi> replaywindow hrv hrv double
replay from identifier "hrv" to data window "hrv" using
   converter TimedDoubleConverter added.

osgi> replaystart
replay started

osgi> replaystop
replay stopped
```

Note that the recording and replaying does not necessarily have to take place on the node where the analysis components reside. Using the distribution facilities presented in Section 3.3.12, it is very well possible to replicate data to a remote node for capturing, or replicating replayed data to the node performing the data analysis.

In total, the capture and replay support that the REFLECT framework provides shows the benefits of a central data management facility, and how simple it is to provide services that alter the flow of data – by injecting recorded data, or by forking data to a file – once the data is managed through the framework.

3.4 Related OSGi component frameworks

The REFLECT component framework is not the only OSGi-based component framework; others have also created component frameworks on top of OSGi with the goal of defining a simpler programming model. As already hinted in Section 2.3, OSGi's biggest strength is its major weakness: the highly dynamic service model of the OSGi platform forces the software developer to introduce a significant amount of defensive boilerplate code for each service usage: before each usage, it must be ensured that the service

reference has not gone stale, and each service used must be released immediately after its use.

In this section, we describe four other popular approaches that exist to mitigate this shortcoming: The OSGi declarative services specification [91, 41], OSGi Blueprint Container (formerly known as Spring Dynamic Modules) [91, 61], iPojo [48], and Google Guice with Peaberry [119]. All four frameworks add a simplified component and service model on top of OSGi, and are discussed in the following.

3.4.1 OSGi Declarative Services

The OSGi Declarative Services specification [41, 91, pp. 297] is a conservative, component-based extension of OSGi. Its main focus is providing a simpler and declarative programming model to the software developer, while at the same time leveraging declarative metadata for reducing startup time and memory footprint of OSGi-based systems.

The basic approach that OSGi declarative services follow to attain these goals is to introduce an external XML-based component description. This component description specifies the dependencies and provisions of components.

The dependency specification also includes the policy to follow if a service that is used for dependency satisfaction disappears; components that are able to deal with a disappearing dependency specify a dynamic dependency policy. If the component is not able to handle disappearing services, a static dependency policy will force the deactivation of the component as soon as the used service is unregistered.

Furthermore, OSGi Declarative Services allow to configure immediate and delayed components. While an immediate component is created as soon as all its dependencies are satisfied (and all provided services are registered to the OSGi service registry), a delayed component registers proxies for all its provided services as soon as all its dependencies are satisfied. The component itself is only instantiated if one of its services is actually used. By using this technique, the instantiation of the component can be deferred. Delaying the instantiation of the component allows to defer the creation of the bundle class loader, which is a resource consuming process in OSGi that significantly impacts system startup time and memory consumption. Avoiding this impact is hence in line with the main goals of OSGi Declarative Services to reduce startup time and memory footprint.

Listing 3.28: AnalyserMonitorService component

```
public class AnalyserMonitorService
    implements CommandProvider {

  private final List<IAnalyserMonitor> fAnalysers;
  private final NumberFormat fFormat;

  public AnalyserMonitorService() {
    fAnalysers = new CopyOnWriteArrayList<IAnalyserMonitor>();
    fFormat = NumberFormat.getNumberInstance();
  }

  public void _analysis(CommandInterpreter ci) {
    ci.println("Current snapshot of analysis results");
    for (IAnalyserMonitor analyser : fAnalysers) {
      String name = analyser.getName();
      double value = analyser.getResult();
      ci.println(name + ": " + fFormat.format(value));
    }
  }

  public String getHelp() {
    StringBuilder buffer = new StringBuilder();
    buffer.append("---Analysis Monitor Help---\n");
    buffer.append("\tanalysis - lists the current analysis
       results\n");
    return buffer.toString();
  }

  public void addAnalyser(IAnalyserMonitor d) {
    fAnalysers.add(d);
  }

  public void removeAnalyser(IAnalyserMonitor d) {
    fAnalysers.remove(d);
  }
}
```

Listing 3.28 gives an example of a declarative service component implementing a monitoring command line interface by aggregating all available analysis monitor services. The example chosen here is the same as in Section 2.3: a monitor service following a plug-in architecture aggregates the current analysis state of the system. The `addAnalyser` and `removeAnalyser`

methods are callback methods that are used to notify the component of the binding or unbinding of the component dependency to the service. The component, dependency and the callback methods of the `AnalyserMonitorService` component are specified in the `component.xml` shown in listing 3.29.

Listing 3.29: Monitor service component declaration

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component
   xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
   name="Command Provider for Analyser Service">
   <implementation class="example.ds.AnalyserMonitorService"/>
   <service>
      <provide interface="org.eclipse.osgi.framework.console.
         CommandProvider"/>
   </service>
   <reference bind="addAnalyser" cardinality="0..n"
      name="Analysers" policy="dynamic"
      unbind="removeAnalyser"
      interface="example.ds.IAnalyserMonitor"/>
</scr:component>
```

The analyser monitor interface and implementation are the same as in the initial OSGi example and were given in listings 2.2 and 2.3, respectively. The bundle activator shown in listing 3.30 is a vastly simplified version of the initial bundle activator (listing 2.4). It creates and registers an analyser service, while the service component providing the console monitor service is instantiated through the Declarative Services runtime. The service tracking seen in Listing 2.4 is now performed by the Declarative Service runtime.

Listing 3.30: Monitor service component activator

```
public class Activator implements BundleActivator {

  private BundleContext fContext;
  private Analyser fAnalyser;

  public void start(BundleContext context) throws Exception {
    fContext = context;
    fAnalyser = new Analyser("Mood Valence");
    fAnalyser.start();
    fContext.registerService(IAnalyserMonitor.class.getName(),
        fAnalyser, null);
  }
```

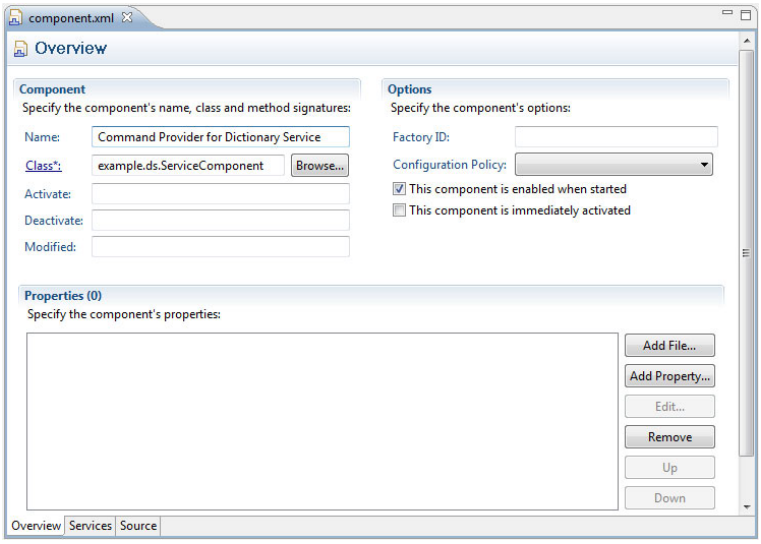
```
  public void stop(BundleContext context) throws Exception {
    fContext = null;
    fAnalyser.stop();
    fAnalyser = null;
  }
}
```

While the OSGi Declarative Services approach clearly satisfies its goals of providing a simpler programming model and reducing startup time and memory footprint, it also has some drawbacks. First of all, the use of XML for the definition of components introduces problems that are generally found when using external configuration files:

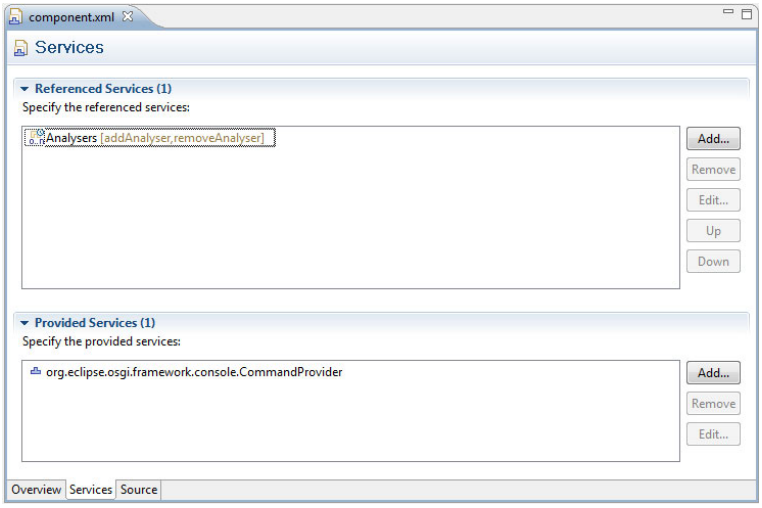
- **Type safety.** Using external configuration files makes it impossible to enforce type constraints (such as e.g. the specified type must be an interface), and may necessitate an additional build step for checking type constraints that are imposed on the configuration.
- **Typo safety.** Without proper editor support, the software developer is uninformed about typing errors that she makes, be it due to misspelling or wrong assumptions on her side.
- **Refactoring safety.** When moving types or changing the names of types, methods or fields that are referenced in the external configuration, it is imperative to also change this external configuration to preserve the link between code and configuration.

All of these problems can be circumvented by providing tool support through editors for the external configuration files. These editors should be aware of the existing code base, the framework's constraints, and should also plug itself into the refactoring actions that can be performed on the code base. The refactoring actions need to be extended to take external configuration files into account on renamings of classes and packages if the configuration is to remain consistent with the code base.

For OSGi Declarative Services, the Eclipse Plug-in Development Environment (PDE) offers editors for service component declarations, but as of Eclipse version 3.7, no refactoring support for service component declarations exist, i.e., renaming of Java classes are not propagated to the component definition XML. Changes to the component definitions must be performed



(a) Component view



(b) Services view

Figure 3.10: PDE declarative services editor

manually. Figure 3.10 shows the basic views of the PDE declarative services editor for the general specification of the component (Figure 3.10a), and the provided services and dependencies (Figure 3.10b).

Another approach followed by the Apache Felix SCR plugin [12] is to use Java annotations for component declarations, and use an annotation processor in a pre-compile step to generate the component declaration XML. As listing 3.31 shows, refactoring robustness is improved by using annotations, but the Apache Felix SCR annotations library does not provide full refactoring robustness: bind and unbind methods are specified as `java.lang.String` values in the `@Reference` annotation instead of being defined through annotations of the respective bind and unbind methods. The rationale behind this design decision may be that annotating the methods directly is misleading: The method lookup algorithm specified in the declarative services specification [91, p. 304] can choose a different method than the one annotated, if another method exists with the same name as the annotated method having a higher precedence.

Listing 3.31: Analyser monitor service component with annotations

```
@Component
@Service
public class AnalyserMonitorService
    implements CommandProvider {

  @Reference(bind = "addAnalyser", unbind = "removeAnalyser",
     cardinality = OPTIONAL_MULTIPLE, policy = DYNAMIC)
  private final List<IAnalyserMonitor> fAnalysers;
  private final NumberFormat fFormat;

  public AnalyserMonitorService() {
    fAnalysers = new CopyOnWriteArrayList<IAnalyserMonitor>();
    fFormat = NumberFormat.getNumberInstance();
  }

  public void _analysis(CommandInterpreter ci) {
    ci.println("Current snapshot of analysis results");
    for (IAnalyserMonitor analyser : fAnalysers) {
      String name = analyser.getName();
      double value = analyser.getResult();
      ci.println(name + ": " + fFormat.format(value));
    }
  }
```

```
    public String getHelp() {
      StringBuilder buffer = new StringBuilder();
      buffer.append("---Analysis Monitor Help---\n");
      buffer.append("\tanalysis - lists the current analysis
          results\n");
      return buffer.toString();
    }

    public void addAnalyser(IAnalyserMonitor d) {
      fAnalysers.add(d);
    }

    public void removeAnalyser(IAnalyserMonitor d) {
      fAnalysers.remove(d);
    }
  }
```

To summarise, the major issue in developing components with OSGi Declarative Services is the managing and maintaining the consistency of the external component definition XML files and the code base. The exact amount of effort that is involved depend on the tool support available for these tasks.

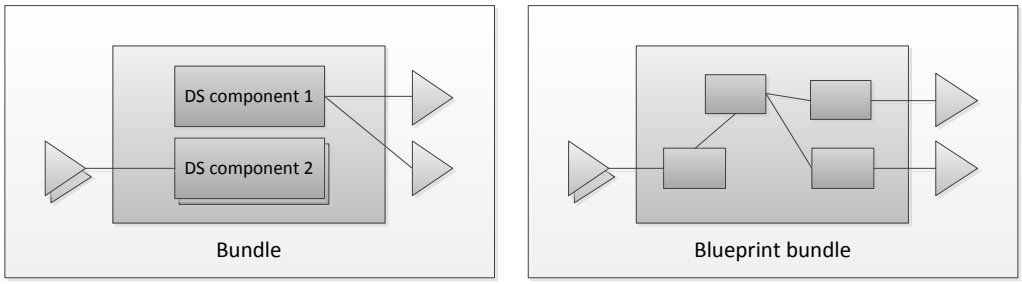
Another issue of OSGi Declarative Services is that its specification misses to describe the synchronization requirements that OSGi Declarative Services imposes on components. For example, it misses to provide information about which threads access the service component's life cycle methods. As various on-line tutorials [18] discuss, thread-safety is a major issue in OSGi in general. Therefore, access to injected services need to be synchronized using a `CopyOnWriteArrayList` as done for the injected `IAnalysisMonitor` service in listings 3.31.

Putting all criticisms aside, however, the Declarative Services specification offers a simple component model that operates well on top of OSGi. Other specifications, as the following OSGi Blueprint Container, feature a significantly more complex component model.

3.4.2 OSGi Blueprint Container

The OSGi Blueprint Container specification [91, pp. 633] is another compendium specification that defines a component layer on top of OSGi, but is different from the Declarative Service model: The Declarative Service spec-

ification follows a model in which an OSGi bundle may define independent component types that provide and depend directly on OSGi services. In the Declarative Services component model, components are opaque entities whose internal structure is defined solely through Java code. Blueprint Container on the other hand allows to specify internal wirings of Java Beans and to export only some of their interfaces as OSGi services. It offers the possibility to define a rich internal structure of OSGi bundles with wirings between Java Beans that are not managed as OSGi services. These different programming models are illustrated in Figure 3.11. Figure 3.11a shows one bundle with two Declarative Services components each interacting directly with the OSGi service registry to retrieve and publish services. Figure 3.11b on the right depicts one blueprint bundle that is made up of beans and wirings that are managed by the blueprint container; only few of these beans use or provide services to the OSGi registry.



(a) Declarative Services (b) Blueprint container

Figure 3.11: Declarative Services vs. Blueprint Container

The Blueprint Container specification relies again heavily on XML, with all the involved issues discussed in Section 3.4.1: type safety, typo safety, and refactoring safety. However, as of 2011, there is no editor support for the creation of blueprint definitions.

Blueprint Container focuses heavily on dependency injection and is mainly focused on the creation and injection of Java Beans. Consequently, the largest part of a blueprint definition XML revolves around the specification of a component blueprint and its instantiation by creating objects and injecting them appropriately. This focus of Blueprint Container on dependency injection is due to its roots in the Spring project [116], which is a dependency injection

framework operating outside of OSGi. The Spring project focuses heavily on server-side Java programming and therefore offers a plethora of functionality for server-side tasks such as persistence, security, and http request processing.

Listing 3.32: Analyser monitor blueprint definition

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="analyser" class="example.bp.Analyser"
        init-method="start" destroy-method="stop">
    <argument value="Mood Valence"/>
  </bean>
  <bean id="analyserMonitor"
        class="example.bp.AnalyserMonitorService">
    <property name="analysers">
      <list value-type="example.bp.IAnalyserMonitor">
        <ref component-id="analyser"/>
      </list>
    </property>
  </bean>
  <service
    id="analyserService"
    ref="analyserMonitor"
    interface="org.eclipse.osgi.framework.console.
       CommandProvider"/>
</blueprint>
```

Listing 3.32 gives an example blueprint definition for the analyser monitoring service example that is used as a running example. The blueprint defines two Java beans and one OSGi service. As in the previous example, the `AnalyserMonitorService` registers an OSGi service that allows to inspect the current state of the system's analysis layer by using the OSGi console command `analysis`. Note that all object instances are created through the given blueprint definition. Consequently, the bundle activator does not need to create and register any service instances.

The `blueprint.xml` example shows several features of the blueprint container specifications. The declaration of the `analyser` bean makes use of the initialisation and destruction lifecycle callbacks, as well as constructor arguments that can be specified for the construction of objects. Furthermore, the declaration of the `analysers` property of the `analyserMonitor` bean shows how lists for injection can be constructed using blueprint con-

tainer. Note that the configuration does not query the OSGi service registry for `IAnalyserMonitor` services; all analyser monitors to be plugged in must be specified in the `blueprint.xml`. Note that using blueprint container, the system can be configured differently, as shown in listing 3.33. By using a `reference-list` element instead of `list` and by registering the mood valence sensor as a service, the blueprint configuration makes use of OSGi services for publishing and fetching `IAnalyserMonitor` instances.

Listing 3.33: Blueprint definition using OSGi services

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="analyser" class="example.bp.Analyser"
        init-method="start" destroy-method="stop">
    <argument value="Mood Valence"/>
  </bean>
  <service
    id="moodService"
    ref="analyser"
    interface="example.bp.IAnalyserMonitor">
  </service>
  <bean id="analyserMonitor"
        class="example.bp.ServiceComponent">
    <property name="analysers">
      <reference-list
        interface="example.bp.IAnalyserMonitor"
        availability="optional"/>
    </property>
  </bean>
  <service
    id="analyserService"
    ref="analyserMonitor"
    interface="org.eclipse.osgi.framework.console.
       CommandProvider"/>
</blueprint>
```

An interesting concept included in the Blueprint Container specification is the concept of *damping* [91, p. 696–697], which is a strategy for the management of OSGi service dynamics. The Bluprint Container specification requires that the extender (i.e. the bundle that takes care of expanding blueprint definitions into objects) must inject service proxy objects into beans once the respective OSGi service is available. The service proxy must

take care of acquiring the backing service upon invocation and release the service as soon as the invocation is finished. By using a service proxy, the blueprint extender is able to hide the deregistration of an OSGi service from the managed Java Bean, thereby damping the service dynamics for the dependant Bean. This constitutes an approach to the management of service dynamics that differs radically from the concepts that the Declarative Services specification offers. In OSGi Declarative Services, a policy is specified with service dependencies, and the component is deactivated if a static policy was specified and the bound service is unregistered. In Blueprint Container, Java Beans keep their service proxy even if the bound service is unregistered, but every call to the service is blocked for a specified amount of time (with a default timeout of five minutes). If the specified timeout is reached, an unchecked exception is thrown. In this way, most Java Beans are able to work in a completely OSGi-agnostic fashion. If a Bean must react to the unbinding of a service, however, the programmer must take care that all uses of the respective service are surrounded by a `org.osgi.service.blueprint.container.ServiceUnavailableException` try-catch block.

Blueprint Container is a specification that is different from the Declarative Services specification in several aspects including default instantiation mechanism, and handling of service dynamics. The Blueprint Container specification follows the idea that most of an application configuration should be done in an OSGi-agnostic way, while only relying on OSGi services for dependencies whose bindings may change over time.

3.4.3 iPOJO

iPOJO is a service component framework that is implemented on top of the Apache Felix OSGi implementation, but runs on other OSGi R4.1 implementations as well such as Eclipse Equinox. The main focus of iPOJO is to simplify the development of OSGi-based applications as far as possible and to offer an extensible component model for experimentation with new features.

The first goal is achieved by removing the burden for dealing with service dynamics in a similar way as OSGi Declarative Services do. iPOJO additionally uses bytecode manipulation, a powerful tool in the Java world. However, bytecode manipulation in iPOJO is not performed at runtime. Instead, it requires an additional build step that is sometimes difficult to incorporate in existing building processes. Furthermore, bytecode manipulation introduces

a level of behaviour that is difficult to communicate to the uninformed software developer, as will become apparent in the following analysis component example.

Listing 3.34: Analyser component type definition

```
@Provides
@Component(name="analyser")
public class Analyser implements IAnalyserMonitor {

  private final DataGenerator fGenerator;
  private final String fName;
  private double fResult;

  public Analyser(String name) {
    fName = name;
    fGenerator = new DataGenerator(new IDoubleTarget() {
      @Override
      public void push(double value) throws
          InterruptedException {
        synchronized(Analyser.this) {
          fResult = value;
        }
      }
    });
  }

  public void start() {
    fGenerator.start();
  }

  public void stop() {
    fGenerator.stop();
  }

  @Override
  public synchronized double getResult() {
    return fResult;
  }

  @Override
  public String getName() {
    return fName;
  }
}
```

Listing 3.34 shows how annotations are used in iPOJO to define the `analyser` component type using the `Analyser` Java class. The `@Provides` annotation declares that all interfaces that are implemented by the `Analyser` class are registered as OSGi services.

The `service` component type is declared in the `AnalyserMonitorService` class as shown in Listing 3.35. Note that the bind and unbind methods are annotated with `@Bind` and `@Unbind` annotations. However, iPOJO allows also to inject object directly to fields. iPOJO uses bytecode manipulation to create setter methods in the class file. Furthermore, note that a concurrent collection is used for the `fAnalysers` fields. This is although iPOJO manipulates the bytecode of the class to inject locking and service lookup statements to guarantee the availability of a services for the duration of a method call. This approach is problematic: while Java code using Declarative Services or Blueprint Container are thread-safe in all contexts, as the developer has to implement his components in a thread-safe way, iPOJO code relies on the framework to provide thread safety. The code is hence unsafe when executed in other contexts or on other middlewares.

Listing 3.35: Analyser monitor service component type definition

```
@Provides
@Component(name="monitor")
public class AnalyserMonitorService
    implements CommandProvider {

  private final List<IAnalyserMonitor> fAnalysers;
  private final NumberFormat fFormat;

  public AnalyserMonitorService() {
    fAnalysers = new CopyOnWriteArrayList<IAnalyserMonitor>();
    fFormat = NumberFormat.getNumberInstance();
  }

  public void _analysis(CommandInterpreter ci) {
    ci.println("Current snapshot of analysis results");
    for (IAnalyserMonitor analyser : fAnalysers) {
      double value = analyser.getResult();
      String name = analyser.getName();
      ci.println(name + ": " + fFormat.format(value));
    }
  }
```

```
  public String getHelp() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("---Analysis Monitor Help---\n");
    buffer.append("\tanalysis - lists the current analysis
        results\n");
    return buffer.toString();
  }

  @Bind(aggregate=true)
  public void bindAnalyser(IAnalyserMonitor d) {
    fAnalysers.add(d);
  }

  @Unbind
  public void unbindAnalyser(IAnalyserMonitor d) {
    fAnalysers.remove(d);
  }
}
```

Going back to the analyser example, the instance definitions are still missing for a full working system. The missing declarations are provided through a simple XML file (see listing 3.36) that references the component types previously declared.

Listing 3.36: iPOJO component instance definitions

```
<ipojo>
  <instance component="analyser"/>
  <instance component="monitor"/>
</ipojo>
```

The iPOJO framework is also easily extensible through the concept of *handlers*. The framework itself is built in a modular way, and several independent concepts such as composite components and runtime manipulation that are not discussed here are implemented as independent handlers that are plugged into the core framework.

To summarize, the iPOJO framework offers a compact programming model which relies on bytecode manipulation, however. The programming model is extensible through the concept of handlers that can be added to the framework core.

3.4.4 Guice and Peaberry

Peaberry is an OSGi extension of the Java-annotation-based dependency injection framework Google Guice [119]. The goal of Peaberry is to make OSGi services usable for dependency injection in Guice and make services offered by a Guice module available through OSGi. Guice and Peaberry offer their functionality through annotation processing facilities and fluent APIs [57]. The aversion against external configuration files stems from the drawbacks that are entailed from the use of XML configuration files in the Spring framework [83].

The basic Guice approach revolves around a three concepts:

1. **Injection.** The single Java annotation `@Inject` annotates constructors, fields or setters to mark them accessible for dependency injection through Guice.

2. **Scope.** Guice provides annotations to control the number of instances it creates. By default, Guice will create a new instance for each injection. By defining a broader validity scope, Guice can be forced to re-use instances. The notion of scope that Guice relies on stems from the web framework community. The use of scopes – and the term scope itself – is dominant in web programming: object with the lifetime of a request (e.g. transactions) or session (e.g. authentication and authorisation information) are omnipresent in web applications.

3. **Binding.** Modules define injection bindings. A binding consists of two parts, the *bind* part and the *to* part: the *bind* part specifies a set of injection points and narrows it through constraints such as the type of the injection point, the class it is defined in, and other present annotations. The *to* part defines which instances are to be injected into the injection points. By default, a new instance is created for each injection point, but through the use of scope, re-use of injected instance can be enforced.

Peaberry extends Guice by introducing additional bindings allowing to use OSGi services and export objects managed by Guice as OSGi services. Furthermore, Peaberry introduces an service export and import management layer which allows to interact programmatically with the OSGi/Peaberry service binding life cycle.

Listing 3.37 shows how injection points are defined with Guice. A single `@Inject` annotation is added to the `setAnalysers` method to declare it as injection point.

Listing 3.37: Analyser monitor service component with Guice injection

```
public class AnalyserMonitorService
    implements CommandProvider {

  private Iterable<IAnalyserMonitor> fAnalysers;
  private final NumberFormat fFormat;

  public AnalyserMonitorService() {
    fFormat = NumberFormat.getNumberInstance();
  }

  public void _analysis(CommandInterpreter ci) {
    ci.println("Current snapshot of analysis results");
    for (IAnalyserMonitor analyser : getAnalysers()) {
      double value = analyser.getResult();
      String name = analyser.getName();
      ci.println(name + ": " + fFormat.format(value));
    }
  }

  public String getHelp() {
    StringBuilder buffer = new StringBuilder();
    buffer.append("---Analysis Monitor Help---\n");
    buffer.append("\tanalysis - lists the current analysis
       results\n");
    return buffer.toString();
  }

  private synchronized Iterable<IAnalyserMonitor>
      getAnalysers() {
    return fAnalysers;
  }

  @Inject
  public synchronized void
      setAnalysers(Iterable<IAnalyserMonitor> analysers) {
    fAnalysers = analysers;
  }
}
```

The module definition in listing 3.38 shows a pure Guice binding for `IAnalyserMonitor` injections to one instance of the class `Analyser`. This binding is used to inject an instance in the `AnalyserMonitorService` class given in listing 3.37. The second binding shows how a set of services is referenced. It constructs an iterable object that represents the set of services available in the registry that implement the `IAnalyserMonitor` interface.

Finally, the third binding specifies that `AnalyserMonitorService` provides an OSGi service of the type `CommandProvider`. The `export` call within the `bind` method declares that the injection point is the OSGi service registry. The method chain `toProvider(service(AnalyserMonitorService.class).export())` creates an exportable service from a `ServiceComponent` instance.

Listing 3.38: Peaberry-based analyser module

```
public class AnalyserModule extends AbstractModule {

  @Override
  protected void configure() {
    Analyser analyser = new Analyser("Mood Valence");
    analyser.start();
    bind(export(IAnalyserMonitor.class)).toProvider(service(
        analyser).export());

    bind(iterable(IAnalyserMonitor.class)).toProvider(service(
        IAnalyserMonitor.class).multiple());
    bind(export(CommandProvider.class)).toProvider(service(
        AnalyserMonitorService.class).export());
  }
}
```

The activator shown in listing 3.39 kick starts the instance module creation through Guice and Peaberry. The `start` OSGi callback method requests the member injection on itself, causing Peaberry to create an instance of `AnalyserMonitorService` and `Analyser`, which are also automatically registered to the OSGi service registry. The `stop` method takes care of unregistering the services as the bundle is stopped.

Listing 3.39: Peaberry-based activator

```
public class Activator implements BundleActivator {

  @Inject
  Export<IAnalyserMonitor> fMoodAnalyserHandle;

  @Inject
  Export<CommandProvider> fConsoleCommandHandle;

  public void start(BundleContext context) throws Exception {
    createInjector(osgiModule(context), new AnalyserModule())
      .injectMembers(this);
  }

  public void stop(BundleContext context) throws Exception {
    fConsoleCommandHandle.unput();
    fMoodAnalyserHandle.unput();
  }
}
```

Guice allows very precise control of injections through the bindings concept and the accompanying mechanisms for narrowing the scope of bindings. Since it does not involve any additional building steps and does not rely on external configuration files, it integrates well into existing build processes and leverages the power of existing IDEs such as Eclipse. Guice configuration code is fully type safe and refactoring safe. However, embedding configuration code in Java has its drawbacks too, as every change in the configuration requires a full release cycle.

Interestingly, Peaberry follows a similar approach to Blueprint Container when it comes to the management of service dynamics. Peaberry proxies OSGi services before injecting them to Guice-managed Java objects similarly to Blueprint Container. Although Peaberry uses the same service dynamic damping concept as Blueprint Container, the damping is not as strong as the service unavailable exception is immediately thrown if the service is unavailable at invocation time.

Altogether, Peaberry has a development path and origin that is similar to Blueprint Container. Both frameworks stem from a pure Java dependency injection framework, and try to extend their reach to OSGi services. Peaberry itself seems to be in development still, but builds upon the very mature Guice dependency injection framework.

3.4.5 Comparative summary

The above discussion of the four OSGi-based component frameworks Declarative Services, Blueprint Container, iPOJO and Peaberry focused on the key features of each framework. In this section, we give a comprehensive comparison of the four frameworks (Declarative Services, Blueprint Container, iPOJO and Peaberry), and the REFLECT framework. The comparison operates along a defined set of aspects in the following. A tabular summary of the comparison can be found in table 3.3.

- **Origin.** The context in which the component framework was initially developed. Blueprint Container and Peaberry are two frameworks that were created on top of sophisticated dependency injection frameworks and constitute extensions of these frameworks to OSGi. Declarative Services, iPOJO, and the REFLECT framework, in contrast, were created directly on top of OSGi.

- **Configuration.** Which languages are used to describe and instantiate components. All other frameworks than Peaberry and the REFLECT framework support pure XML configuration; iPOJO support configuration through a mixture of XML and Java annotations. Peaberry and the REFLECT framework are the only frameworks that do not rely on XML for configuration.

- **Service dynamics.** How disappearing services are handled through the framework and propagated to developer code. Interestingly, component frameworks that stem from dependency injection frameworks rely solely on damping of service dynamics (i.e. injection of proxies that wait for the availability of the service and throw runtime exceptions on timeout). These frameworks motivate their approach with the potentially high cost of re-creation of a full system configuration. Frameworks that were build on top of OSGi define a component lifecycle that allows to re-use already created components: the REFLECT framework is one of the frameworks following a lifecycle and re-use approach. In contrast, Peaberry and Blueprint Container allow to be notified about vanishing service bindings, but do not provide a lifecycle model. The client code must implement a component lifecycle on its own every time it is needed.

	Declarative Services	**Blueprint Container**	**iPOJO**	**Guice Peaberry**	**REFLECT framework**
Origin	OSGi	Spring	OSGi	Guice	OSGi
Configuration	XML or annotations	XML	XML or annotations	Java and annotations	Java and annotations
Service dynamics	Deactivation	Damping	Deactivation or Damping	Damping	Deactivation
Lifecycle	Yes	No	Yes	No	Yes
Wiring without OSGi	No	Yes	No	Yes	Yes
Injection	method	constructor, property	method, field	constructor, method, field	method
Default instantiation	Singleton	Multiple instances	Singleton	Multiple instances	Singleton
Extensions	No	No	Yes	Yes	No
Thread-safety	User	User	Framework	User	User

Table 3.3: Component framework feature comparison

- **Lifecycle.** Whether the component framework define a component lifecycle. This aspect is tightly related to the handling of service dynamics. Dependency injection frameworks do not specify a component lifecycle, and have therefore little means to notify client code about vanishing services. Instead, they rely on damping service dynamics.

- **Wiring without OSGi.** Whether it is possible to wire Java instances without relying on OSGi service bindings. As expected, Blueprint Container and Peaberry allow wirings that bypass OSGi, while Declarative Services and iPOJO do not. Relying solely on OSGi service bindings involves the overhead of service registration, discovery and binding through OSGi. The REFLECT framework constitutes a remarkable exception, as it is built upon OSGi, but allows to bypass the OSGi service registry – which is, in fact, its default mode of wiring. The REFLECT framework follows a more fine-grained wiring approach that is closer to the wiring approach of Peaberry and Blueprint Container.

- **Default instantiation.** The default instantiation scheme that is defined through the component framework. In this aspect, again, the split between "OSGi-native" and "dependency injection extending" component frameworks is apparent. Dependency injection framework create new instances for each injection, and reducing the number of instantiation is achieved by defining validity scopes for instances such as e.g. singleton scope. In "OSGi native" frameworks (Declarative Services, iPOJO and the REFLECT framework), the default instantiation is singleton, and creating multiple instances involves some overhead, such as defining and registering a component factory instead in addition to the component itself.

- **Extensions.** Whether the component framework can be extended. Interestingly, the frameworks that are included in the OSGi compendium do not define means for extending the features and semantics of the framework. This may be due to the complexity involved in fully specifying such extensible frameworks. iPOJO and Peaberry are both extensible, and their extension mechanisms are used for modularisation of features and experimentation. The REFLECT framework on the other hand does not feature means for user-level extensions to the framework core through plug-in or other mechanisms; it was not necessary, as extensions to the functionality of the framework core could be provided

by additional components made available by default on system start, like for example the data context service (see Section 3.3.11).

- **Thread-safety.** Whether the framework or the developer (i.e. the user of the framework) is responsible for providing thread-safety. Interestingly, only iPOJO takes responsibility for providing a thread-safe implementation, while all other impose own thread-safety requirements on client code. The decision of taking the burden of thread-safety from the developer or imposing it on her is not as easy as expected, and is a decision that has far-reaching consequences. From an abstract point of view, one may think that providing as much service to the developer can only lead to better client code. Looking further into the issue reveals however that taking the responsibility for thread-safety means that the client code is inherently unsafe, and becomes safe only through relying on the framework. This makes the client code dependent on the framework, which contradicts the idea of POJOs. Furthermore, taking the responsibility from the developer will make it harder for her to understand the notions of concurrency, and to appreciate the programming of concurrent systems. The approach of providing a clear model of shared-memory concurrency, concurrency patterns and concurrency building blocks as done in the Java concurrency library [62] seems to be a more promising approach, as it leaves the responsibility for dealing with concurrency and deciding on the design of concurrent behaviour with the developer. The REFLECT framework follows a mixed approach to guaranteeing thread-safety of applications: the framework takes care of synchronizing changes to port bindings and the component lifecycle state without having the programmer to take care of ensuring thread-safety. It also guarantees that ports may not change during the execution of the step method, and during the processing of method invocations, as components are made quiescent before reconfigurations take place.

From the discussion of the existing component frameworks and the comprehensive comparison above, one can conclude the following.

- Different models for the creation of applications and systems on top of OSGi exist, of which two can be identified: the “everything is a service” model, and the “variation points are services” model. In the “everything is a service” model, each dependency injection is performed

via an OSGi service. In contrast, the "variation points are services" model, POJOs are instantiated, of which some can be registered as OSGi services, and OSGi services can be injected into POJOs via service proxies. The two models imply different approaches to service dynamic management, the existence of a component lifecycle, and a different default instantiation strategy. Intuitively, the "everything is a service" model encourages the management of coarse-grained, heavyweight components, whereas the "variation points are services" model encourages the management of lightweight POJOs with the framework that are assembled to create a complex, heavyweight system. In this dichotomy, the REFLECT framework follows the "variation points are services", but with the twist that the internal structure of OSGi modules can be altered through reconfigurations at runtime.

- Making a framework understandable by the less informed developers is not easily achieved. Dependency injection, runtime reflection, service dynamics and the involved concurrency aspects are topics that must be clearly presented to the software developer. The component frameworks achieve this task with varying degree of success. Arguably, Guice and Peaberry provide the clearest introduction for software developers, as well as the most accessible means for configuration, as they are based on Java and Annotations. The REFLECT framework follows the same approach to configuration as Guice and Peaberry in order to make it easier for newcomers to start developing applications with the REFLECT framework.

- The configuration definition can be done using external configuration files using e.g. XML, or with Java code. While XML files are harder to maintain and demand for additional editors, changing the configuration does not require recompilation. Configurations using Java code have the inverse properties: they require recompilation for changing the configuration, but the editor support offered by existing IDEs is sufficient for effective management of configurations.

- Additional processing steps such as bytecode manipulation as performed by iPOJO are sometimes difficult to integrate into existing IDE building processes. These difficulties can make the adoption of component frameworks significantly harder.

- No component framework offers control over both instantiation and binding, while providing configuration through Java code. Control of instantiation is only given in iPOJO, but it does not provide control over binding as Guice does. iPOJO allows only autowiring of dependencies in a service oriented way. Selection of matching services is done via property maps and filters, whereas Guice introduces a binding rule API which allows fine-grained control over the bindings. This is where the REFLECT framework proposes a unique set of features: it allows the control of both instantiation and binding, and allows to configure systems through Java code. The wiring regime can be arbitrarily defined through the rule-based wiring framework that ressembles the approach of Guice (see Section 3.3.3).

In summary, the OSGi component framework ecosystem has developed interesting, yet contradicting approaches to component frameworks. It is not yet clear whether the definitive answer to the needs of developers of OSGi-based systems is already found. Especially, it is interesting to note that no component framework other than the REFLECT framework offers fine-grained control over both instantiation and binding, or allows to select a binding regime from a continuum from a service oriented binding regime to a component-oriented binding regime.

3.5 Conclusion

This chapter introduced the REFLECT framework – a component-based software framework for the creation of physiological computing applications. This section also documents how the framework was developed to satisfy high-level requirements that users have towards physiological computing applications, and requirements developers have towards a framework for physiological computing applications. With the given requirements (Section 3.1), a set of concepts were selected (Section 3.2) that together satisfy the requirements. A detailed discussion of the implementation (Section 3.3) then showed how the concepts can be used, what were the forces that emerged from the implementation of these concepts, and how they interact with each other in the implementation. In this chapter, we have therefore shown how the implemented concepts of the framework can be traced to the initial requirements, and how the concepts relate to the framework implementation.

The REFLECT framework design revolved around three basic design decisions:

- Providing a *natural embedding* of a component model into the Java programming language, and therefore avoiding introducing external configuration code such as XML.
- Choosing an approach to multithreading that **scales** from restricted resource environments to mutlicore systems.
- Providing *support* for the development of physiological computing applications, especially for the implementation and testing of physiological computing systems. This also included allowing for easy and powerful implementations of *reconfigurations* by choosing an imperative approach to reconfiguration.

In our opinion, the REFLECT framework was successful in satisfying the requirements imposed on it by following the above-mentioned basic design decisions. This opinion is supported by the case studies that were developed using the REFLECT framework, as elaborated in Chapter 6. However, it became also apparent that dealing with reconfigurations is not as easy one might initially expect. This is why the next Chapter (Chapter 4) focuses on providing a formal verification framework for systems undergoing reconfigurations.

In total, the REFLECT framework encompasses a set of carefully selected features that interact well with each other, and provide some beneficial synergy effects, as showcased by the the Data Context Service and distribution support in Section 3.3.12. Furthermore, when comparing the REFLECT framework with other OSGi based component frameworks, it becomes apparent that the REFLECT framework offers a distinctive combination of features and means of control that are not found in any other framework.

Chapter 4

Specifying and verifying systems under reconfiguration

A major challenge when creating systems capable of performing autonomous runtime reconfigurations is to guarantee that they operate correctly even when undergoing structural changes. This is especially true in real-time environments, where the duration of a reconfiguration or other timing aspects may lead to unstable back-and-forth structure changes. As it happen to be, physiological pervasive adaptive computing systems are exactly such real-time environments.

The design of correct reconfiguration behaviour is a challenging task. While the initial structure of a system may be easily known in the beginning, that structure can be altered drastically once multiple reconfiguration steps were performed, and tracking all possible chains of reconfiguration steps is not as easy. Working towards a formal proof of correct behaviour of the system under all possible reconfigurations is therefore worthwhile.

In this chapter, it is investigated how a formal proof can be established on the correct realisation of a formal specification through a system undergoing reconfigurations. For this, a component-based assume-guarantee reasoning framework is introduced that is able to handle both reconfigurations and real-time specifications natively. The work presented in this chapter have been published in [106], albeit in a more condensed form; this chapter presents the work more extensively.

Following the component-based assume-guarantee framework from Benveniste et al. [26], *components* are considered to be black boxes, making explicit only their communication requirements by means of *required* and

provided ports (see Section 2.4). We go beyond the framework presented by Benveniste et al. by introducing the concept of a system *configuration*, that makes the description of bindings from required ports to suitable provided ports. By changing the bindings of component ports, the system's behaviour can be changed. This is how reconfiguration is mapped into the formal verification framework.

In the proposed framework, the specification of the overall system and the single components is given by *assume-guarantee contracts*, where the assumptions address the system environment, and the guarantees the output produced. The conformance of a component configuration to a global specification can be checked off-line, and invalid application designs can be detected early. A novelty of the proposed framework compared to the assume-guarantee framework of Benveniste et al. is that the system configuration is taken into account in the composition of contracts; that is to say, possible changes of bindings are considered when composing a system from its components.

The assume-guarantee framework presented in this thesis is based on a real-time, linear time semantics that allows to specify both safety and liveness properties. Building on the preliminaries given in Section 2.4, an extension of the linear-time semantics (Section 4.1) and temporal logic syntax (Section 4.2) to cover configurations and connectors is introduced. For the specification of configurations, we extend Metric Interval Temporal Logic [8] to reMITL (Metric Interval Temporal Logic for Reconfigurable Components) which features an additional binary predicate $\sim$ that allows to specify the connection status of component ports. Section 4.3 will introduce a simple example of the application of the framework. More complex examples can be found in the case studies Chapter 6. Section 4.4 presents related work, and Section 4.5 finally gives the summary of the results achieved through the proposed formal framework.

4.1 Assume-guarantee reasoning and reconfiguration

The basic idea of the reasoning framework is to be able to deduce whether a composition of assume-guarantee-style component specifications under a given reconfiguration specification can satisfy a global specification given as

assume-guarantee contract.

In order to create this framework, we take a look at the preliminaries provided in Section 2.4 and perform the necessary modification for supporting real-time reconfiguration specifications. In the following, the definitions of signatures, states, runs, assertions and contracts as well as the operations defined on them are re-visited and extended to account for reconfigurations.

4.1.1 Signatures, states and runs

This section introduces an abstract assume-guarantee framework that is formulated on the semantic domain of runs over a given signature. Note that in the following, we use shared variable semantics instead of message passing, although the terminology of components and connectors may suggest otherwise. This decision allows us to simplify the underlying semantics, and to keep systems describable with regular (metric interval) temporal logic.

Definition 17 ((Composite) component signature)**.** *A* component signature Σ *consists of two disjoint finite sets* R_Σ, P_Σ *of provided and required ports, respectively:* $\Sigma = (R_\Sigma, P_\Sigma)$.

A composite component signature $\Sigma = ((R_\Sigma^{Ext}, P_\Sigma^{Ext}), (R_\Sigma^{Int}, P_\Sigma^{Int}), (C_\Sigma^i, C_\Sigma^d))$ *consists of one external and one internal component signature with mutually disjoint port sets, and a connector signature consisting of internal connectors* $C_\Sigma^i \subseteq R_\Sigma^{Int} \times P_\Sigma^{Int}$ *and delegating connectors* $C_\Sigma^d \subseteq (R_\Sigma^{Ext} \times R_\Sigma^{Int}) \cup (P_\Sigma^{Ext} \times P_\Sigma^{Int})$.

Component signatures can obviously be embedded into composite component signatures, e.g. for a component signature Σ, $((R_\Sigma, P_\Sigma), (\emptyset, \emptyset), (\emptyset, \emptyset))$ is its corresponding composite component signature. The sets C_Σ^i and C_Σ^d can be used to model constraints on connections resulting from e.g. type (in-)compatibilities between ports. All port sets of signatures must be pairwise disjoint. A *signature* is either a component signature or a composite component signature; both are commonly denoted by letters Σ and Θ.

Definition 18 (Signature functions *ports* and *conns*)**.** *Let* $\Sigma = (R_\Sigma, P_\Sigma)$ *be a component signature. The set of ports of* Σ, $ports(\Sigma)$ *is defined as* $ports(\Sigma) = R_\Sigma \uplus P_\Sigma$. *The set of connectors of* Σ, $conns(\Sigma)$ *is defined as* $conns(\Sigma) = \emptyset$. *Let* $\Sigma = ((R_\Sigma^{Ext}, P_\Sigma^{Ext}), (R_\Sigma^{Int}, P_\Sigma^{Int}), (C_\Sigma^i, C_\Sigma^d))$ *be composite component signature. The set of ports of* Σ, $ports(\Sigma)$ *is defined as* $ports(\Sigma) = R_\Sigma^{Ext} \uplus P_\Sigma^{Ext} \uplus R_\Sigma^{Int} \uplus P_\Sigma^{Int}$. *The set of connectors of* Σ, $conns(\Sigma)$ *is defined as* $conns(\Sigma) = C_\Sigma^i \uplus C_\Sigma^d$.

Composite component signatures are richer in structure than simple component signatures. It is therefore useful to define funtions that retrieve the internal and external sub-signatures they feataure.

Definition 19 (External and internal signature)**.** *Let* Σ *be a composite component signature,* $\Sigma = ((R_\Sigma^{Ext}, P_\Sigma^{Ext}), (R_\Sigma^{Int}, P_\Sigma^{Int}), (C_\Sigma^i, C_\Sigma^d))$ *be composite component signature. The* external signature *of* Σ*,* $ext(\Sigma)$*, is defined as* $ext(\Sigma) = (R_\Sigma^{Ext}, P_\Sigma^{Ext})$*. The* internal signature *of* Σ*,* $int(\Sigma)$*, is defined as* $int(\Sigma) = (R_\Sigma^{Int}, P_\Sigma^{Int})$*.*

It is possible to define the notion of a *subsignature* $\Sigma \subseteq \Theta$ by component-wise set inclusion, and *supremum* of signatures $\sup(\Sigma, \Theta)$ by component-wise set union.

Definition 20 (Subsignature and signature supremum)**.** *Let* Σ *and* Θ *be component signatures. The notion of a* subsignature $\Sigma \subseteq \Theta$ *is defined by component-wise set inclusion:*

$$\Sigma \subseteq \Theta \text{ iff } R_\Sigma \subseteq R_\Theta \text{ and } P_\Sigma \subseteq P_\Theta$$

The supremum *of* Σ *and* Θ *is defined by component-wise set union:*

$$\sup(\Sigma, \Theta) = (R_\Sigma \cup R_\Theta, P_\Sigma \cup P_\Theta)$$

Let Σ *and* Θ *be composite component signatures. The notion of a* subsignature $\Sigma \subseteq \Theta$ *is defined by component-wise set inclusion:*

$$\begin{array}{rll} \Sigma \subseteq \Theta & \textit{iff} & R_\Sigma^{Ext} \subseteq R_\Theta^{Ext} \textit{ and } P_\Sigma^{Ext} \subseteq P_\Theta^{Ext} \\ & \textit{and} & R_\Sigma^{Int} \subseteq R_\Theta^{Int} \textit{ and } P_\Sigma^{Int} \subseteq P_\Theta^{Int} \\ & \textit{and} & C_\Sigma^i \subseteq C_\Theta^i \textit{ and } C_\Sigma^d \subseteq C_\Theta^d \end{array}$$

The supremum *of* Σ *and* Θ*,* $\sup(\Sigma, \Theta)$ *is defined by component-wise set union:*

$$\sup(\Sigma, \Theta) = (\sup(ext(\Sigma), ext(\Theta)), \sup(int(\Sigma), int(\Theta)), C_\Sigma^i \cup C_\Theta^i, C_\Sigma^s \cup C_\Theta^s)$$

Since the structure of signatures is more complex than before (cf. Section 2.4), it is worthwhile to define the notion of a Σ-state before defining runs.

Definition 21 (Σ-state)**.** *Let Σ be a composite component signature or component signature. A* Σ-state *σ is a subset of $ports(\Sigma) \uplus conns(\Sigma)$. A state is called* valid *iff the following condition is satisfied: if σ is a Σ state and $(p_1, p_2) \in \sigma$, then $p_1, p_2 \in \sigma$ or $p_1, p_2 \notin \sigma$. The set of all valid Σ-states is denoted by $\mathcal{S}(\Sigma)$.*

The definition of Σ-states implies (for simplicity) that ports have boolean values, and port connectors can be present (instantiated) in σ or not; both is expressed by element relationship. The notion of state validity implies that connected ports are always equal in value.

Definition 22 (Σ-run)**.** *Let Σ be a signature. A* Σ-run *is a function $\rho : \mathbb{R}_0^+ \to \mathcal{S}(\Sigma)$ which satisfies the properties of* finite-variability *[8]: for all $t, t' \in \mathbb{R}_0^+$, there are only finitely many different states between time t and t':*

$$\forall t, t' \in \mathbb{R}_0^+ . t < t' \implies |\{\rho(t'') \mid t \leq t'' \leq t'\}| < \infty.$$

The class of all Σ-runs is denoted by $\mathcal{R}(\Sigma)$.

The restriction to runs with finite-variability must be made in order to be able to prove the completeness of REMITL – runs that do not satisfy finite-variability may exhibit properties that cannot be shown using REMITL.

Using the above definition, assertions can be defined as sets of runs.

Definition 23 (Assertion)**.** *Let Σ be a signature. A* Σ-assertion *E, in the following also denoted by $E : \Sigma$, is a set of Σ-runs: $E \subseteq \mathcal{R}(\Sigma)$.*

Similarly as before, assertions over signatures are used for specifying *assumptions*, *guarantees*, and *implementations*. Additionally, *assembly specifications* are modeled with assertions over composite component signatures.

As oftentimes, assumptions, guarantees, implementations, and assembly specifications are assertions over different signatures, there must be a way to transform an assertion from one signature to another. First, the lifting of runs and assertions from a signature Σ to a signature $\Theta \supseteq \Sigma$ is defined.

Definition 24 (Lifting)**.** *Let Σ, Θ be component signatures such that $\Theta \supseteq \Sigma$, let $\rho \in \mathcal{R}(\Sigma)$ be a Σ-run. The* lifting *of ρ to Θ, $\rho{\uparrow^\Theta}$ is defined as follows.*

$$\rho{\uparrow^\Theta} = \{\rho' \in \mathcal{R}(\Theta) \mid \forall t \in \mathbb{R}_0^+ . \rho'(t) \cap (ports(\Sigma) \uplus conns(\Sigma)) = \rho(t)\}$$

Let $E : \Sigma$ be a Σ-assertion. The lifting *of E to Θ, $E{\uparrow^\Theta}$ is defined as follows.*

$$E{\uparrow^\Theta} = \bigcup \{\rho{\uparrow^\Theta} \mid \rho \in E\}$$

Restriction is then defined as the inverse operation for runs and assertions from a signature $\Theta \supseteq \Sigma$ to Σ.

Definition 25 (Restriction)**.** *Let Σ, Θ be component signatures such that $\Theta \supseteq \Sigma$, let $\rho \in \mathcal{R}(\Theta)$ be a Θ-run. The* restriction *of ρ to Σ, $\rho{\downarrow_\Sigma}$ is defined as follows.*

$$\rho{\downarrow_\Sigma} = \{\rho' \in \mathcal{R}(\Sigma) \mid \forall t \in \mathbb{R}_0^+ . \rho(t) \cap (\mathit{ports}(\Sigma) \uplus \mathit{conns}(\Sigma)) = \rho'(t)\}$$

Let $E : \Theta$ be a Θ-assertion. The restriction *of E to Σ, $E{\downarrow_\Sigma}$ is defined as follows.*

$$E{\downarrow_\Sigma} = \bigcup \{\rho{\downarrow_\Sigma} \mid \rho \in E\}$$

Note that restriction and lifting can be applied between assertions over both component and composite component signatures.

For the parallel composition of components and contracts, we use interconnection specifications that are called *assembly specification*. An assembly specification defines a dynamic evolution of the system structure, i.e. how the connectors between ports of components change over time.

Definition 26 (Assembly specification)**.** *Let Σ be a composite component signature. An* assembly specification *is a Σ-assertion that satisfies the following properties.*

1. *A required port $r \in R_{int(\Sigma)}$ may be bound to at most one provided port at a time, i.e., for all $r \in R_{int(\Sigma)}$, for all $\rho \in \mathcal{A}$ and for all $t \in \mathbb{R}_0^+$ it holds that $|\{p \in P_{int(\Sigma)} \mid (r, p) \in \rho(t)\}| \leq 1$.*

2. *Each external provided port $p_{ext} \in P_{ext(\Sigma)}$ must be delegated to exactly one internal provided port, i.e., for all $p_{ext} \in P_{ext(\Sigma)}$, for all $\rho \in \mathcal{A}$ and for all $t \in \mathbb{R}_0^+$ it holds that $|\{p_{int} \in P_{int(\Sigma)} \mid (p_{ext}, p_{int}) \in \rho(t)\}| = 1$.*

The constraint on the binding of required ports to provided is different to the constraint on the delegation of external provided ports to an internal provided port: while external provided ports must be connected continuously, required ports of internal components may be disconnected, as internal components may remain unused in an assembly for specific periods. In these periods, all ports of the component get disconnected, and the component remains orphan in the assembly until it is needed.

In this abstract semantics-based framework, parallel composition of implementations is straightforward. Note that parallel composition can be applied to implementations as well as assembly specifications.

Definition 27 (Parallel composition of assertions)**.** *Let* Σ_E, Σ_F *be component signatures. Let* $E : \Sigma_E$ *and* $F : \Sigma_F$ *be two assertions, and* $\Sigma = \sup(\Sigma_E, \Sigma_F)$*. The* composition *of* E *and* F *is defined by* $E \| F = E{\uparrow^{\Sigma}} \cap F{\uparrow^{\Sigma}}$*.*

4.1.2 Assume-guarantee contracts

For a general theory of specification composition and refinement, using assertions alone is not satisfactory. In order to support the separation between assumptions towards the environment and specification of component behaviour, it is worthwhile to introduce assume-guarantee contracts for the abstract specification of behaviour to be implemented.

Definition 28 (Assume-guarantee contract)**.** *Let* Σ *be a component signature or a composite component signature. An* assume-guarantee contract *is a pair of assertions* $(A : \Sigma, G : \Sigma)$*.*

Note that assume-guarantee contracts can be defined for both components and composite components. For composite components, this means that assumptions may specify the internal behaviour of the component, i.e. how port interconnections and internal ports behave, and the externally visible behaviour of the component, i.e. how the valuation of external provided ports correspond to the valuation of external required ports. Still, contract satisfaction can be defined by inclusion of runs for both components and composite components for the simple reason that $\mathcal{S}(\Sigma)$ is defined for both.

Definition 29 (Contract satisfaction)**.** *Let* $M : \Sigma$ *be an implementation, and* $(A : \Sigma_C, G : \Sigma_C)$ *be an assume-guarantee contract.* Σ *and* Σ_C *are either both component signatures or composite component signatures. The implementation* M satisfies $(A : \Sigma_C, G : \Sigma_C)$*, denoted by* $M \models^{c}_{\Sigma} (A, G)$*, if and only if* $\Sigma_C \subseteq \Sigma$ *and* $M \cap A{\uparrow^{\Sigma}} \subseteq G{\uparrow^{\Sigma}}$*.*

The *canonical form* of contracts is defined as $(A, (\neg A) \cup G)$. The rationale for the canonical form is still the same as before: non-circular composition of contracts is avoided, and the canonical form is a valid replacement for a contract as an implementation M satisfies a contract if and only if it satisfies its canonical form. To simplify presentation, it will again be assumed from now on that contracts are in canonical form, and write $(A, G) : \Sigma$ instead of $(A : \Sigma, G : \Sigma)$.

Now, we define parallel composition of contracts. While parallel composition of contracts in the simple assume-guarantee framework presented

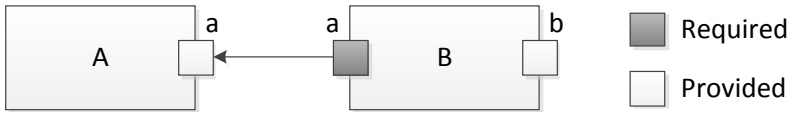


Figure 4.1: Contract composition example

in Section 2.4 was relatively simple, parallel composition of contracts under consideration of assembly specifications becomes more complex. It still applies that assumptions of the composition must a) consist only of assumptions that are not satisfied mutually, b) avoid circular reasoning, and c) create contracts in canonical form. Since dynamic systems are considered, the parallel composition operator for contracts becomes ternary, taking two contracts (C and D) and an assembly specification ($\mathcal{A}$) as parameters. The assembly specification is an assertion over a composite component signature specifying how C and D are assembled, i.e. when connectors between component ports are established and removed. The assembly specification is provided by the composite component, and constitutes a blueprint in which components and contracts are plugged in to create the complete behaviour specification of the composite component.

Definition 30 (Parallel composition of contracts)**.** *Let Σ_C and Σ_D be component signatures, let $C = (A_C, G_C) : \Sigma_C$, $D = (A_D, G_D) : \Sigma_D$ be two assume-guarantee contracts in canonical form, and let $\mathcal{A} : \Sigma$ be an assembly specification over a composite component signature Σ with $\sup(\Sigma_C, \Sigma_D) \subseteq int(\Sigma)$. The contracts C and D are* composable *iff $ports(\Sigma_C) \cap ports(\Sigma_D) = \emptyset$. The* composition *of C and D through $\mathcal{A}$, $C \| ^{\mathcal{A}} D$, is defined as*

$$C \| ^{\mathcal{A}} D = ((A_C \cap A_D \cap \mathcal{A}) \cup \neg(G_C \cap G_D \cap \mathcal{A}), G_C \cap G_D \cap \mathcal{A}) : \Sigma.$$

Note that since the assembly specification $\mathcal{A}$ defines how and when components are interconnected, it is necessary to include $\mathcal{A}$ in both guarantee and assumption of the composed contract. Note also that the composed contract is in canonical form, since $\neg((A_C \cap A_D \cap \mathcal{A}) \cup \neg(G_C \cap G_D \cap \mathcal{A})) = \neg(A_C \cap A_D \cap \mathcal{A}) \cap (G_C \cap G_D \cap \mathcal{A}) \subseteq (G_C \cap G_D \cap \mathcal{A})$. Furthermore, it can be directly shown that composition of implementations and contracts is commutative and associative, as it is defined by set intersections.

Example 1. *We consider the simple system shown in Figure 4.1 consisting of one component A having one provided port A.a, a component B having a*

required port $B.a$ and a provided port $B.b$, and a static connector binding the port $B.a$ to $A.a$. The signature of A is $\Sigma_A = (\emptyset, \{A.a\})$, the signature of B is $\Sigma_B = (\{B.a\}, \{B.b\})$. The component signature of the assembly specification is $\Sigma = ((\emptyset, \emptyset), (\{B.a\}, \{A.a, B.b\}), (\{(B.a, A.a)\}, \emptyset))$. The assumptions of A and B and the static assembly specification $\mathcal{A} : \Sigma$ can be defined as follows.

$$\begin{array}{rl} A_A = & \mathcal{R}(\Sigma_A) \\ G_A = & \{\rho \in \mathcal{R}(\Sigma_A) \mid \forall t \in \mathbb{R}_0^+.\ A.a \in \rho(t)\} \\ A_B = & \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.a \in \rho(t)\} \\ G_B = & \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.b \in \rho(t)\} \\ \mathcal{A} = & \{\rho \in \mathcal{R}(\Sigma) \mid \forall t \in \mathbb{R}_0^+.\ (B.a, A.a) \in \rho(t)\} \end{array}$$

Now, the contract composition $(A_A, G_A)\|^{\mathcal{A}}(A_B, G_B) = C$ consists of the following assumption A_C and guarantee G_C:

$$\begin{array}{rcl} A_C & = & (A_A \cap A_B \cap \mathcal{A}) \cup \neg(G_A \cap G_B \cap \mathcal{A}) \\ & = & (A_B \cap \mathcal{A}) \cup \neg(G_A \cap G_B \cap \mathcal{A}) \\ & = & (A_B \cap \mathcal{A}) \cup \neg(G_A \cap \mathcal{A}) \cup \neg G_B \\ & \overset{(i)}{=} & \mathcal{R}(\Sigma) \\ G_C & = & G_A \cap G_B \cap \mathcal{A} \end{array}$$

For step (i) note that the assembly $\mathcal{A}$ enforces that the ports $B.a$ and $A.a$ have the same value over all runs, and hence $G_A \cap \mathcal{A} = A_B \cap \mathcal{A}$. ■

In this new assume-guarantee framework, with potentially dynamic assembly specifications, parallel composition of implementations still preserves contract satisfaction.

Theorem 2 (Composition preserves contract satisfaction)**.** *Let Σ_M and Σ_N be component signatures, let $\mathcal{A} : \Sigma$ be an assembly specification, $M : \Sigma_M$, $N : \Sigma_N$ be two implementations with $\sup(\Sigma_M, \Sigma_N) \subseteq int(\Sigma)$. Let $C = (A_C, G_C) : \Sigma_M$, $D = (A_D, G_D) : \Sigma_N$ be two composable contracts. Then it holds that*

$$\text{if } M \vDash^c_{\Sigma_M} C \text{ and } N \vDash^c_{\Sigma_N} D \text{ then } M\|N\|\mathcal{A} \vDash^c_{\Sigma} C\|^{\mathcal{A}}D.$$

Poof of Theorem 2. Let $(A, G_C \cap G_D \cap \mathcal{A}) = C\|^{\mathcal{A}}D$. We have to show that $M\|N\|\mathcal{A} \cap A \subseteq G_C \cap G_D \cap \mathcal{A}$. We know that $M \cap A_C \subseteq G_C$, which holds iff $M \subseteq G_C \cup \neg A_C$. Since $\neg A_C \subseteq G_C$, it follows $M \subseteq G_C$ (same holds for N and (A_D, G_D)). Hence, $M \cap N \cap \mathcal{A} \cap A \subseteq G_C \cap G_D \cap \mathcal{A} \cap A$, which is a subset of $G_C \cap G_D \cap \mathcal{A}$. □

Assume-guarantee contracts, both over composite and simple component signatures, can be refined by using the following definition of contract refinement. As before, refinement is covariant for guarantees and contravariant for assumptions: refinement allows assumptions to be weakened and guarantees to be strengthened.

Definition 31 (Contract refinement)**.** *Let Σ be a signature. $(A', G') : \Sigma$ refines $(A, G) : \Sigma$, denoted by $(A, G) \succeq (A', G')$, iff $A \subseteq A'$ and $G' \subseteq G$.*

Contract refinement is compatible with contract satisfaction, i.e. whenever an implementation satisfies a refined contract, it also satisfies the original contract.

Lemma 2 (Contract refinement preserves contract satisfaction)**.** *Let $(A, G) : \Sigma$ and $(A', G') : \Sigma$ be contracts and $M : \Sigma_M$ be an implementation such that $\Sigma \subseteq \Sigma_M$. If $M \models^c_\Sigma (A', G')$ and $(A, G) \succeq (A', G')$ then $M \models^c_\Sigma (A, G)$.*

The proof of Lemma 2 is the same as for Lemma 1.

Example 2. *Going back to Example 1, we can investigate valid refinements and abstractions of component B, whose assumption and guarantees were given as follows:*

$$\begin{aligned} A_B = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.a \in \rho(t)\} \\ G_B = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.b \in \rho(t)\} \end{aligned}$$

Creating parameterised variants of A_B and G_B, we can investigate which combinations constitute valid refinements:

$$\begin{aligned} A_B(n) = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \leq n.\ B.a \in \rho(t)\} \\ G_B(n) = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \geq n.\ B.b \in \rho(t)\} \end{aligned}$$

Intuitively, the contract $C_B(n) = (A_B(n), G_B(n))$ assumes $B.a$ up to instant n, and guarantees $B.b$ starting from n. For the parameterised assertions, it holds that $A_B(n') \subseteq A_B(n)$ iff $n \leq n'$, and $G_B(n) \subseteq G_B(n')$ iff $n \leq n'$. Therefore, a refinement relation between contracts $C_B(n)$ exists as follows: $C_B(n') \succeq C_B(n)$ iff $n' \geq n$. ■

4.1.3 Receptivity requirements

So far, we do not consider the special role of required and provided ports in contracts and implementations. However, the distinction between required and provided ports plays a crucial role for the engineering of component-based systems: with the help of this distinction, a component structure defines how responsibilities are assigned to individual components. Therefore, we need to restrict the domain of discourse to sensible contracts and implementations: Component implementations may control the behaviour of its provided ports, but should not be given control over required ports. We address this requirement with the notion of *receptivity* [26].

Definition 32 (Receptivity)**.** *Let $E : \Sigma$ be an assertion over a component signature Σ. E is* Req-receptive *iff $E{\downarrow_{\Sigma_{Req}}} = \mathcal{R}(\Sigma_{Req})$, where $\Sigma_{Req} = (R_\Sigma, \emptyset)$. E is* Prov-receptive *iff $E \downarrow_{\Sigma_{Prov}} = \mathcal{R}(\Sigma_{Prov})$, where $\Sigma_{Prov} = (\emptyset, P_\Sigma)$. Let $E : \Sigma$ be an assertion over a composite component signature Σ. E is* Req-receptive *iff $E{\downarrow_{ext(\Sigma)_{Req}}} = \mathcal{R}(ext(\Sigma)_{Req})$. E is* Prov-receptive *iff $E{\downarrow_{ext(\Sigma)_{Prov}}} = \mathcal{R}(ext(\Sigma)_{Prov})$.*

The requirements for an implementation M and for an assume-guarantee contract are the same as in the initial assume-guarantee framework. If an assertion E is Req-receptive, it does not constrain the valuations of its required ports. For a composite component signature Σ, receptivity is required only w.r.t. the external signature $ext(\Sigma)$.

We must also constrain assume-guarantee contracts. Just as with implementations, guarantees should make no assumptions about the valuation of required ports. In fact, a contract with a non-Req-receptive guarantee would forbid all Req-receptive implementations. Conversely, assumptions are not allowed to restrict the valuation of provided ports. In this way, assumptions can be made about the environment of a component, but not on the behaviour of the component. Assumptions of valid assume-guarantee contracts must be *Prov-receptive*: As with Req-receptiveness, Prov-receptiveness for composite component signatures concerns provided ports of the external signature $ext(\Sigma)$ only. We say an assume-guarantee contract $(A, G) : \Sigma$ is *valid* iff A is Prov-receptive and G Req-receptive.

Definition 33 (Validity)**.** *Let $M : \Sigma$ be an implementation. M is called* valid implementation *iff M is Req-receptive. Let $C = (A, G) : \Sigma$ be an assume-guarantee contract. C is called* valid contract *iff A is Prov-receptive, and G is Req-receptive.*

Level	Assertion	Requirement
Implementation	Implementation	Req-Receptiveness
Contract	Assumption	Prov-Receptiveness
	Guarantee	Req-Receptiveness
Assembly	Assembly Specification	Σ-Receptiveness for all contained component signatures and for the external signature

Table 4.1: Receptivity requirements.

Additionally, we must require that assembly specifications $\mathcal{A} : \Sigma_{\mathcal{A}}$ do only define the interconnections between components, and do not restrict internal component behaviours. Therefore, we introduce the notion of Σ-receptivity for component signatures, allowing to specify that an assembly specification does not constrain the behaviour of the components it assembles.

Definition 34 (Σ-receptivity)**.** *$\mathcal{A} : \Sigma_{\mathcal{A}}$ be an assembly specification, let Σ be a component signature such that $\Sigma \subseteq \Sigma_{\mathcal{A}}$. $\mathcal{A}$ is Σ-receptive iff $\mathcal{A}{\downarrow_{\Sigma}} = \mathcal{R}(\Sigma)$.*

Altogether, we impose receptivity requirements on implementations, contracts and assembly specifications; Table 4.1 summarises these requirements.

Note that the composition of two valid assume-guarantee contracts is not necessarily valid; in fact, if the composition is inconsistent, the contract guarantee resulting from composition becomes empty, which is not Req-receptive for any signature having a non-empty set of required ports.

Example 3. *In order to give an example of receptivity requirements, we go back to Example 1, and investigate the receptiveness of the contract C_B of component C, which was given as $C_B = (A_B, G_B)$ by the following assertions:*

$$\begin{aligned} A_B = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.a \in \rho(t)\} \\ G_B = &\ \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.b \in \rho(t)\} \end{aligned}$$

In order for C_B to be a valid contract, A_B must be prov-receptive and G_B must be req-receptive. In order to show the prov-receptivity of A_B, we must prove that $A_B{\downarrow_{(\emptyset,\{B.b\})}} = \mathcal{R}((\emptyset, \{B.b\}))$. As the left hand side is obviously a subset of the right, we must only show that every run $\rho \in \mathcal{R}((\emptyset, \{B.b\}))$ is also

contained in $A_B{\downarrow_{(\emptyset,\{B.b\})}}$. *For this, let us consider a run* $\rho \in \mathcal{R}((\emptyset, \{B.b\}))$, *and construct the following run* ρ' *over the signature* Σ_B*:*

$$\rho'(t) = \rho(t) \cup \{B.a\} \qquad \forall t \in \mathbb{R}_0^+$$

Obviously, this run is in A_B, *as* $\forall t \in \mathbb{R}_0^+.\ B.a \in \rho'(t)$ *holds. At the same time,* $\rho \in \rho'{\downarrow_{(\emptyset,\{B.b\})}}$ *holds, as for all* $t \in \mathbb{R}_0^+.\rho'(t) \cap \{B.b\} = \rho(t)$. *Therefore,* $\rho \in A_B{\downarrow_{(\emptyset,\{B.b\})}}$.

The Prov-receptiveness of G_B *can be shown conversely, and verifying the receptiveness requirements for the assembly specification* $\mathcal{A}$ *of Example 1 is more interesting. The assembly specification itself is given as*

$$\mathcal{A} = \{\rho \in \mathcal{R}(\Sigma) \mid \forall t \in \mathbb{R}_0^+.\ (B.a, A.a) \in \rho(t)\}$$

This assertion needs to be receptive for the signature Σ_A *and* Σ_B *of components* A *and* B *in order to ensure that it is a valid assembly specification and does not interfere with the implementation of the components* A *and* B. *Hence, it must be proved that* $\mathcal{A}{\downarrow_{\Sigma_A}} = \mathcal{R}(\Sigma_A)$ *and* $\mathcal{A}{\downarrow_{\Sigma_B}} = \mathcal{R}(\Sigma_B)$. *Starting with* Σ_A, $\mathcal{A}{\downarrow_{\Sigma_A}} \subseteq \mathcal{R}(\Sigma_A)$ *is again obviously true. Left to be shown is* $\mathcal{R}(\Sigma_A) \subseteq \mathcal{A}{\downarrow_{\Sigma_A}}$. *Let hence* $\rho \in \mathcal{R}(\Sigma_A)$ *and* $\rho' \in \mathcal{R}(\Sigma)$[1] *be such that* $(B.a, A.a) \in \rho'(t)$ *for all* $t \in \mathbb{R}_0^+$, *and* $A.a \in \rho'(t)$ *iff* $A.a \in \rho(t)$ *for all* $t \in \mathbb{R}_0^+$. *By construction of* ρ', *it holds that* $\rho' \in \mathcal{A}$ *and* $\rho \in \rho'{\downarrow_{\Sigma_A}}$. *Therefore,* $\mathcal{A}{\downarrow_{\Sigma_A}} = \mathcal{R}(\Sigma_A)$. *The proof of* $\mathcal{A}{\downarrow_{\Sigma_B}} = \mathcal{R}(\Sigma_B)$ *uses a very similar construction and reasoning, and is therefore omitted.*

Now that all shown assertions so far satisfy the receptivity requirements imposed on them, the question arises which assertions actually violate receptivity requirements. We give two examples of assertions violating receptivity requirements; first, a simple contract assertion, and then an assembly specification. First, let us verify that the guarantee of B, $G_B = \{\rho \in \mathcal{R}(\Sigma_B) \mid \forall t \in \mathbb{R}_0^+.\ B.b \in \rho(t)\}$ *is not Prov-receptive for* Σ_B. *We show that* $\mathcal{R}((\emptyset, \{B.b\})) \not\subseteq G_B{\downarrow_{(\emptyset,\{B.b\})}}$ *by choosing* $\rho \in \mathcal{R}((\emptyset, \{B.b\}))$ *such that* $\rho(t) = \emptyset$ *for all* $t \in \mathbb{R}_0^+$. *Obviously,* $\rho \notin G_B{\downarrow_{(\emptyset,\{B.b\})}}$, *as* $G_B{\downarrow_{(\emptyset,\{B.b\})}} = \{\rho_b\}$ *with* $\rho_b(t) = \{B.b\}$ *for all* $t \in \mathbb{R}_0^+$.

Now, let us define a variation of $\mathcal{A}$ *that does not satisfy* Σ_A*-receptiveness by requiring that* $A.a$ *must always be contained in the run:*

$$\mathcal{A}' = \{\rho \in \mathcal{R}(\Sigma) \mid \forall t \in \mathbb{R}_0^+.\ (B.a, A.a) \in \rho(t) \text{ and } A.a \in \rho(t)\}$$

[1]Remember that in Example 1, Σ refers to the composite component signature of the whole system.

Looking at $\mathcal{A}'\downarrow_{\Sigma_A}$*, it is clear that the run* $\rho \in \mathcal{R}(\Sigma_A)$ *with* $\rho(t) = \emptyset$ *for all* $t \in \mathbb{R}_0^+$ *is missing, as all runs in* $\mathcal{A}'$ *contain* $A.a$ *in every state, and therefore* $\mathcal{A}'\downarrow_{\Sigma_A} = \{\rho_a\}$ *with* $\rho_a(t) = \{A.a\}$ *for all* $t \in \mathbb{R}_0^+$*. It does not satisfy* Σ_A*-receptiveness, as it constrains the possible behaviours of* Σ_A *components. Specifying such restrictions is not the intent of assembly specifications; it is therefore mandatory to rule out assembly specifications that do restrict the behaviour of component as invalid.* ■

4.1.4 Components and composite components

Now that the full assume-guarantee framework for dynamically reconfigurable component-based systems is in place, we can define simple components and composite components, allowing to assemble components into a composite structure. Furthermore, we define correctness criteria for simple and composite components.

Definition 35 (Components)**.** *A* component *is a triple* $C = (\Sigma, (A, G), M)$ *consisting of a component signature* Σ*, a contract* $(A, G) : \Sigma$*, and an implementation* $M : \Sigma$*. A component is* correct *iff* $M \vDash^c_\Sigma (A, G)$*.*

A composite component *is a tuple* $C = (\Sigma, (A, G), \mathcal{A}, \mathcal{C})$ *consisting of a composite component signature* Σ*, an external contract* $(A, G) : ext(\Sigma)$*, an assembly specification* $\mathcal{A} : \Sigma$*, and a set of components* $\mathcal{C}$*. A composite component is* correct *iff (1) all its constituents are correct, (2) all component signatures are disjoint, (3) the internal signature* $int(\Sigma)$ *is a superset of the supremum of the component signatures it contains, (4) the internal contract refines the contract of the composite component, (5)* $\mathcal{A}$ *is* Σ*-receptive for all sub-component signatures and the external signature, and (6) the composition of the implementations* M *is Req-receptive:*

$$
\begin{array}{ll}
\forall c \in \mathcal{C}.\ c \text{ is correct} & (1)\\
\forall c, d \in \mathcal{C}.\ (c \neq d) \Rightarrow ports(\Sigma_c) \cap ports(\Sigma_d) = \emptyset & (2)\\
(\sup_{c\in\mathcal{C}} c) \subseteq int(\Sigma) & (3)\\
(A, G)\uparrow^\Sigma \succeq \|^{\mathcal{A}}_{c\in\mathcal{C}}(A_c, G_c) & (4)\\
\mathcal{A} \text{ is } \Sigma\text{-receptive for all } \Sigma_c \text{ such that } c \in \mathcal{C} \text{ and for } ext(\Sigma) & (5)\\
M = (\|_{c\in\mathcal{C}} M_c \| \mathcal{A})\downarrow_{ext(\Sigma)} \text{ is Req-receptive} & (6)
\end{array}
$$

The external view *of the composite component is given by* $C_{Ext} = (ext(\Sigma), (A, G), M)$*, which is again a component.*

Theorem 3 (Composite component external views are valid)**.** *Let* $C = (\Sigma, (A, G), \mathcal{A}, \mathcal{C})$ *be a composite component. If* C *is correct, then the external view* C_{Ext} *is correct.*

Proof of theroem 3. Since C is correct, it follows from (1) that all $c \in \mathcal{C}$ are correct, and hence $M_c \vDash^c_{\Sigma_c} (A_c, G_c)$. Let $M_{Int} = \|^{\mathcal{A}}_{c \in \mathcal{C}} M_c$. By Lemma 2, it follows that $M_{Int} \vDash^c_{\Sigma} \|^{\mathcal{A}}_{c \in \mathcal{C}} (A_c, G_c)$. By (4) and Lemma 2, it follows that $M_{Int} \vDash^c_{\Sigma} (A, G)\uparrow^{\Sigma}$. Hence, $M = M_{Int}\downarrow_{ext(\Sigma)} \vDash^c_{ext(\Sigma)} (A, G)$. Furthermore, M is Req-receptive by (6). □

To verify that a composite component C satisfies its external contract, we therefore have to verify that the external contract (A, G) is refined by the composition of the contracts of its internal components $\mathcal{C}$ under the assembly specification $\mathcal{A}$.

Up to now, we introduced a semantic assume-guarantee framework. In the next section, we discuss how contracts and their satisfaction can be verified on the syntactic level by using real-time temporal logic.

4.2 Real-time logic for reconfiguration

In this section a real-time temporal logic is introduced for the specification of component-based systems that are subject to reconfiguration. This logic provides a syntax for specifying assertions, guarantees and assembly specifications in the described semantic assume-guarantee contract framework. Our starting point is the real-time temporal logic MITL (Metric Interval Temporal Logic) which was introduced in [8]. In general, real-time temporal logics are not decidable [77, 84, 6, 10]. This drawback has often been tackled by sacrificing continuous time in order to achieve decidability and hence a feasible verification [93, 10, 11]; for a discussion on this trade-off see also [9]. The reason for undecidability lies in punctual timing constraints as for example in the formula $\Box(p \rightarrow \Diamond_{=3.7} q)$ expressing that whenever p is true, q must hold after exactly 3.7 time units. In [8] it is shown that allowing only relaxed versions of this formula as in $\Box(p \rightarrow \Diamond_{(3.6,3.8)} q)$, which expresses that q must hold between 3.6 and 3.8 time units after p, yields the decidable logic MITL for a continuous time model with non-singular time intervals. We believe that removing the ability to specify punctuality properties shall not be seen as a drawback since punctuality is hard to achieve in practice, and also not required in general.

To be expressive enough for the purpose of specifying reconfigurable systems and reconfigurations in particular, we extend MITL by a binary predicate $\sim$ on port sets in order to express connectivity, i.e. given two ports p_1 and p_2, $p_1 \sim p_2$ states the existence of a connector between p_1 and p_2. The predicate $\sim$ is used in two manners: either $\sim$ states connectivity between required and provided ports of components by an internal connector, or it states that a delegate connector connects an external provided (required) port with an internal provided (required) port, respectively. The resulting logic reMITL (Metric Interval Temporal Logic for Reconfigurable Components) is used for specifying assertions and assembly specifications, i.e., interconnections between components and their dynamic evolution; Moreover, we introduce a significant set of derived inference rules for use when verifying global system properties.

In the following definition of reMITL formulas, we use the same time intervals as defined in Section 2.4, Definition 14.

Definition 36 (reMITL-formulas)**.** *The set of* reMITL-formulas *over a (simple or composite component) signature Σ is inductively defined by*

$$\phi ::= \top \mid p \mid p_1 \sim p_2 \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \mathcal{U}_I \phi_2 \mid \phi_1 \mathcal{S}_I \phi_2$$

where $p \in ports(\Sigma)$ is a port, $(p_1, p_2) \in conns(\Sigma)$ is a (delegate) connector, and I is a non-singular time interval. For brevity, a reMITL*-formula ϕ over a signature Σ is also called a Σ-formula.*

As before (cf. Section 2.4), the abbreviations $\Diamond_I \phi$, $\Box_I \phi$, $\phi_1 \vee \phi_2$, $\phi_1 \rightarrow \phi_2$, and $\phi_1 \leftrightarrow \phi_2$ are defined.

The semantics of reMITL-formulas is again given as sets of runs over the formula signature. Each run provides, for every point in time, an interpretation of the ports and the connectors of the given signature.

Definition 37 (Satisfaction of reMITL-formulas)**.** *A Σ-run $\rho \in \mathcal{R}(\Sigma)$* satisfies *a Σ-formula ϕ (or ϕ* is valid in *ρ), denoted by $\rho \models_\Sigma \phi$, if $(\rho, 0) \models_\Sigma \phi$, where the satisfaction relation $\models_\Sigma$ between pairs (ρ, t), $t \in \mathbb{R}_0^+$, and Σ-formulas ϕ is*

inductively defined as follows:

$$
\begin{array}{lll}
(\rho,t) \models_\Sigma \top & & \\
(\rho,t) \models_\Sigma p & \textit{iff} & p \in \rho(t); \\
(\rho,t) \models_\Sigma p_1 \sim p_2 & \textit{iff} & (p_1,p_2) \in \rho(t); \\
(\rho,t) \models_\Sigma \neg\phi & \textit{iff} & (\rho,t) \not\models_\Sigma \phi; \\
(\rho,t) \models_\Sigma \phi_1 \wedge \phi_2 & \textit{iff} & (\rho,t) \models_\Sigma \phi_1 \textit{ and } (\rho,t) \models_\Sigma \phi_2; \\
(\rho,t) \models_\Sigma \phi_1 \mathcal{U}_I \phi_2 & \textit{iff} & \exists t' \in \mathbb{R}_0^+, t' \in t + I.(\rho,t') \models_\Sigma \phi_2 \\
 & & \textit{and } \forall t'' \in \mathbb{R}_0^+, t < t'' < t'.(\rho,t'') \models_\Sigma \phi_1; \\
(\rho,t) \models_\Sigma \phi_1 \mathcal{S}_I \phi_2 & \textit{iff} & \exists t' \in \mathbb{R}_0^+, t' \in t - I.(\rho,t') \models_\Sigma \phi_2 \\
 & & \textit{and } \forall t'' \in \mathbb{R}_0^+, t' < t'' < t.(\rho,t'') \models_\Sigma \phi_1.
\end{array}
$$

A Σ-formula ϕ is called a consequence of *a set Γ of Σ-formulas, denoted by $\Gamma \models_\Sigma \phi$, if $\rho \models_\Sigma \phi$ holds for every ρ such that $\rho \models_\Sigma \psi$ for all $\psi \in \Gamma$. ϕ is called* universally valid*, denoted by $\models_\Sigma \phi$, if $\emptyset \models_\Sigma \phi$.*

Again, non-strict versions of $\mathcal{U}$ and $\mathcal{S}$ can be defined from the versions above just as in Schobbens et al. [104].

Given a set Γ of Σ-formulas, the semantics of Γ is defined by the set of all runs satisfying Γ, i.e. $[\![\Gamma]\!] = \{\rho \in \mathcal{R}()\Sigma \mid \rho \models \Gamma\}$; if $\Gamma = \{\phi\}$, then $[\![\phi]\!] = [\![\Gamma]\!]$.

Lemma 3 (Semantic Equivalences)**.** *For any Σ-formulas ϕ and ψ, and any set Γ of Σ-formulas it holds*

1. $[\![\neg\phi]\!] = \mathcal{R}(\Sigma) \setminus [\![\phi]\!]$*, and* $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$*,*
2. $\Gamma \models_\Sigma \phi \to \psi$ *iff* $[\![\Gamma]\!] \cap [\![\phi]\!] \subseteq [\![\psi]\!]$*.*

Proof of Lemma 3.
First, we prove that $[\![\neg\phi]\!] = \neg[\![\phi]\!]$.
Proof of $[\![\neg\phi]\!] \subseteq \neg[\![\phi]\!]$: Let $\rho \in [\![\neg\phi]\!]$. By Definition 37, $(\rho,0) \not\models_\Sigma \phi$. Therefore, $\rho \notin [\![\phi]\!]$, and hence $\rho \in \neg[\![\phi]\!]$.
Proof of $[\![\neg\phi]\!] \supseteq \neg[\![\phi]\!]$: Let $\rho \in \neg[\![\phi]\!]$. Hence, $\rho \notin [\![\phi]\!]$. By Definition 37, it follows that $(\rho,0) \not\models_\Sigma \phi$. Hence, $\rho \in [\![\neg\phi]\!]$.
Now, we prove that $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cap [\![\psi]\!]$.
Proof of $[\![\phi \wedge \psi]\!] \subseteq [\![\phi]\!] \cap [\![\psi]\!]$: Let $\rho \in [\![\phi \wedge \psi]\!]$. By Definition 37, it holds that $(\rho,0) \models_\Sigma \phi$ and $(\rho,0) \models_\Sigma \psi$. Therefore, $\rho \in [\![\phi]\!] \cap [\![\psi]\!]$.
Proof of $[\![\phi \wedge \psi]\!] \supseteq [\![\phi]\!] \cap [\![\psi]\!]$: Let $\rho \in [\![\phi]\!] \cap [\![\psi]\!]$. I.e., by Definition 37, $(\rho,0) \models_\Sigma \phi$ and $(\rho,0) \models_\Sigma \psi$. Thus, $(\rho,0) \models_\Sigma \phi \wedge \psi$. Finally, $\rho \in [\![\phi \wedge \psi]\!]$.
Next, we prove that $\Gamma \models_\Sigma \phi \to \psi$ iff $[\![\Gamma]\!] \cap [\![\phi]\!] \subseteq [\![\psi]\!]$ holds.

Proof direction "⇒": Let $\Gamma \models_\Sigma \phi \to \psi$, i.e. $\forall \rho \in \mathcal{R}(\Sigma).(\forall \gamma \in \Gamma.\rho \models_\Sigma \gamma) \Rightarrow \rho \models_\Sigma \phi \to \psi$. This is equivalent to $\forall \rho \in [\![\Gamma]\!].\rho \models_\Sigma \phi \to \psi$. Let $\rho \in [\![\Gamma]\!]$. By Definition 37, it follows that $\rho \not\models_\Sigma \psi$ or $\rho \models_\Sigma \phi$. Assuming that $\rho \models_\Sigma \psi$ therefore implies that $\rho \models_\Sigma \phi$. Hence, $[\![\Gamma]\!] \cap [\![\psi]\!] \subseteq [\![\phi]\!]$.
Proof direction "⇐": Let $[\![\Gamma]\!] \cap [\![\phi]\!] \subseteq [\![\psi]\!]$, i.e. $\forall \rho \in [\![\Gamma]\!] \cap [\![\phi]\!].\rho \models_\Sigma \psi$. Let $\rho \in [\![\Gamma]\!]$. For this ρ, it holds that $\rho \not\models_\Sigma \phi$ or $\rho \not\models_\Sigma \psi$. Therefore, $\rho \models_\Sigma \phi \to \psi$. It follows that $\forall \rho \in \mathcal{R}(\Sigma).(\forall \gamma \in \Gamma.\rho \models_\Sigma \gamma) \Rightarrow \rho \models_\Sigma \phi \to \psi$, and hence $\Gamma \models_\Sigma \phi \to \psi$. □

With the help of Lemma 3, it is possible to specify assume-guarantee contracts and implementations with the help of MITL formulas over the correct signature.

Contract satisfaction can now be expressed by satisfaction of reMITL-formulas which means that we have decidability of contract satisfaction as long as all assertions (contract, assumption and guarantee) are given as reMITL-formulas. By applying Definition 29 and Lemma 3 we obtain the following corollary.

Corollary 1 (Contract satisfaction expressed by model satisfaction)**.** *Let ϕ_A and ϕ_B be Σ-formulas, and Γ a set of Σ-formulas. Then it holds that $\Gamma \models_\Sigma \phi_A \to \phi_G$ if and only if $[\![\Gamma]\!] \models^c_\Sigma ([\![\phi_A]\!], [\![\phi_B]\!])$.*

Furthermore, contract refinement can expressed by satisfaction of reMITL-formulas.

Corollary 2 (Contract refinement expressed by model satisfaction)**.** *Let ϕ_{A_1}, ϕ_{A_2}, ϕ_{G_1}, and ϕ_{G_2} be Σ-formulas. Then it holds that $([\![\phi_{A_2}]\!], [\![\phi_{G_2}]\!]) \succeq ([\![\phi_{A_1}]\!], [\![\phi_{G_1}]\!])$ if and only if $\models_\Sigma (\phi_{A_2} \to \phi_{A_1}) \wedge (\phi_{G_1} \to \phi_{G_2})$.*

Finally, parallel composition of contracts can be computed directly on reMITL-formulas.

Corollary 3 (Contract composition in reMITL)**.**
Let ϕ_{A_1}, ϕ_{A_2}, ϕ_{G_1}, ϕ_{G_2}, and $\phi_{\mathcal{A}}$ be Σ-formulas. Then it holds that $([\![\phi_{A_1}]\!], [\![\phi_{G_1}]\!]) \|^{[\![\phi_{\mathcal{A}}]\!]} ([\![\phi_{A_2}]\!], [\![\phi_{G_2}]\!]) = ([\![\phi_A \vee \neg\phi_G]\!], [\![\phi_G]\!])$, where $\phi_A = \phi_{A_1} \wedge \phi_{A_2} \wedge \phi_{\mathcal{A}}$ and $\phi_G = \phi_{G_1} \wedge \phi_{G_2} \wedge \phi_{\mathcal{A}}$.

Now, we discuss soundness and completeness results for reMITL. As shown by Schobbens et al. [104] MITL with nonsingular intervals has a sound and complete proof system (consisting of one rule and 59 axiom schemata); we call this proof system Π_{MITL}.

(1) all propositional tautologies are derivable $\quad (2)\ \dfrac{\phi \in \Gamma}{\Gamma \vdash_\Sigma \phi}$

$$(3)\ \frac{\Gamma \cup \{\phi\} \vdash_\Sigma \psi}{\Gamma \vdash_\Sigma \phi \to \psi} \qquad (4)\ \frac{\Gamma \vdash_\Sigma \psi}{\Gamma \cup \{\phi\} \vdash_\Sigma \psi}$$

$$(5)\ \frac{\Gamma \vdash_\Sigma \phi \to \psi \qquad \Gamma \vdash_\Sigma \phi}{\Gamma \vdash_\Sigma \psi} \qquad (6)\ \frac{\Gamma \vdash_\Sigma p \sim q}{\Gamma \vdash_\Sigma q \leftrightarrow p}$$

$$(7)\ \frac{\Gamma, \phi \vdash_\Sigma \psi \qquad \Gamma, \neg\phi \vdash_\Sigma \psi}{\Gamma \vdash_\Sigma \psi} \qquad (8)\ \frac{\Gamma \vdash_\Sigma \phi_1 \to (\phi_2 \to \psi)}{\Gamma \vdash_\Sigma \phi_1 \wedge \phi_2 \to \psi}$$

$$(9)\ \frac{\Gamma \vdash_\Sigma \phi}{\Box_I \Gamma \vdash_\Sigma \Box_I \phi} \qquad (10)\ \frac{\Gamma \vdash_\Sigma \Box_I \phi}{\Gamma \vdash_\Sigma \Diamond_I \phi} \qquad (11)\ \frac{\Gamma \vdash_\Sigma \Box_I \phi}{\Gamma \vdash_\Sigma \phi}\ 0 \in I$$

$$(12)\ \frac{\Gamma \vdash_\Sigma \Box_I(\phi \to \psi) \qquad \Gamma \vdash_\Sigma \Diamond_J \phi}{\Gamma \vdash_\Sigma \Diamond_J \psi}\ I \supseteq J$$

$$(13)\ \frac{\Gamma \vdash_\Sigma \Box_I \phi \qquad \Gamma \vdash_\Sigma \Box_I(\phi \leftrightarrow \psi)}{\Gamma \vdash_\Sigma \Box_I \psi} \qquad (14)\ \frac{\Gamma \vdash_\Sigma \Box_I \phi \qquad \Gamma \vdash_\Sigma \Box_I \psi}{\Gamma \vdash_\Sigma \Box_I(\phi \wedge \psi)}$$

$$(15)\ \frac{\Gamma \vdash_\Sigma \Box_J \phi}{\Gamma \vdash_\Sigma \Box_I \phi}\ J \supseteq I \qquad (16)\ \frac{\Gamma \vdash_\Sigma \Diamond_J \phi}{\Gamma \vdash_\Sigma \Diamond_I \phi}\ J \subseteq I$$

$$(17)\ \frac{\Gamma \vdash_\Sigma \Box_I \Box_J \phi}{\Gamma \vdash_\Sigma \Box_{I+J} \phi} \qquad (18)\ \frac{\Gamma \vdash_\Sigma \Diamond_I \Diamond_J \phi}{\Gamma \vdash_\Sigma \Diamond_{I+J} \phi}$$

$$(19_i)\ \frac{\Gamma \vdash_\Sigma \Box_I(\phi_1 \wedge \phi_2)}{\Gamma \vdash_\Sigma \Box_I \phi_i}\ i \in \{1,2\} \qquad (20)\ \frac{\Gamma \vdash_\Sigma \Diamond_I \phi \qquad \Gamma \vdash_\Sigma \Box_I \psi}{\Gamma \vdash_\Sigma \Diamond_I(\phi \wedge \psi)}$$

$$(21)\ \frac{\Gamma \vdash_\Sigma \Box_I(p \sim q)}{\Gamma \vdash_\Sigma \Box_I(p \leftrightarrow q)} \qquad (22)\ \frac{\Gamma \vdash_\Sigma \Box_I(p \sim q)}{\Gamma \vdash_\Sigma \Box_I(\neg p \leftrightarrow \neg q)}$$

$$(23)\ \frac{\Gamma \vdash_\Sigma \Box_I \phi \wedge \Box_J \phi}{\Gamma \vdash_\Sigma \Box_{I \cup J} \phi}\ I \cup J \text{ convex}$$

Figure 4.2: Useful axiom schemata and derivation rules of reMITL

For the completeness proof, Schobbens et al. translate MITL formulas into formulas of a real-time temporal logic over clocks, called EventClockTL, and then prove the completeness of EventClockTL w.r.t. a proof system we call Π_{EvCTL}. We use this result for proving the completeness of reMITL as follows.

For any composite component signature Σ we add the following axiom schema *ConnAx* of propositional formulas of the form

$$(p \sim q) \to (p \leftrightarrow q)$$

where p, q are ports of Σ to the proof systems and call these extensions $\Pi^{\Sigma}_{\mathrm{reMITL}}$ and $\Pi^{\Sigma}_{\mathrm{reEvCTL}}$.

Theorem 4 (Completeness of reMITL)**.** *For any composite component signature Σ the proof system $\Pi^{\Sigma}_{\mathrm{reMITL}}$ is a complete proof system for* reMITL*.*

Proof of Theorem 4. For reasons of succinctness we discuss only a sketch of the proof; in particular, all omitted definitions can be found in [104].

Obviously, the propositional axioms *ConnAx* are consistent with the axioms of propositional Metric Interval Temporal Logic MITL; thus the proof system $\Pi^{\Sigma}_{\mathrm{reMITL}}$ is sound. For the completeness we observe that the translation of MITL into EventClockTL is the identity for propositional formulas of the form *ConnAx*. Thus it suffices to show the completeness of $\Pi^{\Sigma}_{\mathrm{reEvCTL}}$. This proof is completely analogous to the completeness proof for Π_{EvCTL} in [104]. It suffices to extend the notion of propositional consistency in [104] by the clause

$$(p \sim q) \in F \text{ implies } ((p \wedge q) \in F \text{ or } (\neg p \wedge \neg q) \in F)$$

for any set F of formulas (which is contained in the closure of a formula α). Then the model construction for α is the same as in [104]. □

For the soundness of $\Pi^{\Sigma}_{\mathrm{reMITL}}$, it suffices to show that the axiom schemata and additional, derived rules introduced in Figure 4.2 are sound.

Theorem 5 (Soundness of reMITL)**.** *The axiom scheme ConnAx and the inference rules in Figure 4.2 are sound: $\Gamma \vdash_{\Sigma} \phi$ implies $\Gamma \models_{\Sigma} \phi$ for every finite set of Σ-formulas Γ and every Σ-formula ϕ.*

Proof of Theorem 5. The axiom scheme *ConnAx* is trivially true, as it

We prove the most interesting rules, other rules are proved analogously.

Rules (1) refers to basic propositional logic and is therefore not expanded here.

Rules (2) – (5) are straightforward to show, and are therefore omitted.

(6) : Assume $\Gamma \models_\Sigma p \sim q$, let $\rho \models_\Sigma \Gamma$, hence $\rho \models_\Sigma p \sim q$ where $(p, q) \in \mathit{conns}(\Sigma)$. By definition $(p, q) \in \rho(0)$. Since $\rho(0)$ is a valid state (see Sect. 4.1.1) we either have $p, q \in \rho(t)$ or $p, q \notin \rho(t)$ hence $\rho \models_\Sigma p \leftrightarrow q$.

(9) : Assume $\Gamma \models_\Sigma \phi$, i.e. for all $\rho \in \mathcal{R}(\Sigma)$ it holds that if $(\rho, 0) \models_\Sigma \Gamma$ then $(\rho, 0) \models_\Sigma \phi$ (*). Assume $(\rho, 0) \models_\Sigma \Box_I \Gamma$. Let $t \in I$ and assume $(\rho, t) \models_\Sigma \Gamma$. Define $\rho'(t') = \rho(t' + t)$. We know $(\rho', 0) \models_\Sigma \Gamma$ by definition of ρ'. Then by (*) it follows that $(\rho', 0) \models_\Sigma \phi$, and hence $(\rho, t) \models_\Sigma \phi$. Since t was arbitrarily chosen from I, it follows that $(\rho, 0) \models_\Sigma \Box_I \phi$.

(10) : Assume $\Gamma \models_\Sigma \Box_I \phi$, let $\rho \models_\Sigma \Gamma$, hence $\rho \models_\Sigma \Box_I \phi$. Then for all $t \in I$, $(\rho, t) \models_\Sigma \phi$. Hence, there is $t' \in I$ such that $(\rho, t') \models_\Sigma \phi$.

(12) : Assume $\Gamma \models_\Sigma \Box_I(\phi \to \psi)$, $\Gamma \models_\Sigma \Diamond_J \phi$, and $J \subseteq I$. We need to show hat $\Gamma \models_\Sigma \Diamond_J \psi$. Let therefore $\rho \models_\Sigma \Gamma$, and hence $\rho \models_\Sigma \Box_I(\phi \to \psi)$ and $\rho \models_\Sigma \Diamond_J \phi$. We know that there is $t \in J$ such that $(\rho, t) \models_\Sigma \phi$. Since for all $t' \in I$ it holds that $(\rho, t') \models_\Sigma \phi \to \psi$ and $J \subseteq I$, we know that $(\rho, t) \models_\Sigma \phi \to \psi$. Using modus ponens, we get $(\rho, t) \models_\Sigma \psi$, and hence $\rho \models_\Sigma \Diamond_J \psi$.

(15) : Assume $\Gamma \models_\Sigma \Box_J \phi$ and $J \supseteq I$. Let $\rho \models_\Sigma \Gamma$, hence $\rho \models_\Sigma \Box_J \phi$. It holds that for all $t \in t + J$ $(\rho, t) \models_\Sigma \phi$. Since $I \subseteq J$, it follows that for all $t' \in I$ $(\rho, t') \models_\Sigma \phi$. Hence, $\rho \models_\Sigma \Box_I \phi$.

(17) : Assume $\Gamma \models_\Sigma \Box_I \Box_J \phi$, and let $\rho \models_\Sigma \Gamma$, hence $\rho \models_\Sigma \Box_I \Box_J \phi$. It follows that for all $t \in I$, $t' \in t + J$ it holds that $(\rho, t') \models_\Sigma \phi$, and hence for all $t'' \in (I + J)$ it holds that $(\rho, t'') \models_\Sigma \phi$, which implies that $\rho \models_\Sigma \Box_{I+J} \phi$.

(19_1) : Assume $\Gamma \models_\Sigma \Box_I(\phi \wedge \psi)$, and let $\rho \models_\Sigma \Gamma$, hence $\rho \models_\Sigma \Box_I(\phi \wedge \psi)$. We know that for all $t \in I$ it holds that $(\rho, t) \models_\Sigma \phi \wedge \psi$. It hence also holds that $(\rho, t) \models_\Sigma \phi$, and hence $\rho \models_\Sigma \Box_I \phi$.

(20) : Assume $\Gamma \models_\Sigma \Diamond_I \phi$ and $\Gamma \models_\Sigma \Box_I \psi$, and let $\rho \models_\Sigma \Gamma$. It follows that $\rho \models_\Sigma \Diamond_I \phi$ and $\rho \models_\Sigma \Box_I \psi$. We know hence that $(\rho, 0) \models_\Sigma \Diamond_I \phi$ $(\rho, 0) \models_\Sigma \Box_I \psi$. It follows directly that for there is $t \in I$ $(\rho, t) \models_\Sigma \phi$. Since for all $t' \in I$ it holds that $(\rho, t') \models_\Sigma \psi$, it also holds for t. We get $(\rho, t) \models_\Sigma \phi \wedge \psi$, and hence $(\rho, 0) \models_\Sigma \Diamond_I(\phi \wedge \psi)$. □

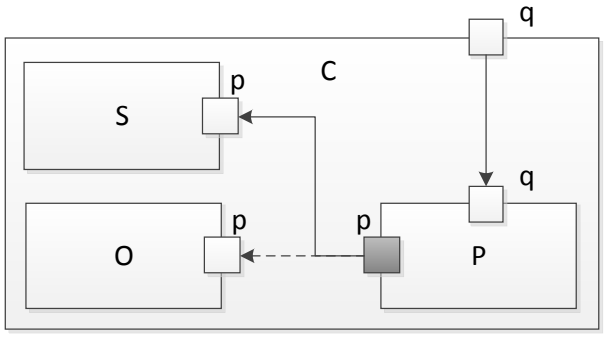


Figure 4.3: Example reconfiguration scenario

4.3 Simple framework application case study

We give a simple example to show how the assume-guarantee framework for reconfigurations can be applied. The scenario consists of one specification that simply requires a port q to be always true, and three simple components: S, O and P. In this example, which is also shown in Figure 4.3, the "processing component" P is providing a port $P.q$ and guaranteeing to maintain its truth only if its single required port, $P.p$ is always true. P would be a natural fit for satisfying the system specification if a component would be available that is bound to $P.q$ and always maintains its truth. However, in this example, we assume that the component O can always guarantee $O.p$ during the operation of the system except for a bootstrap phase – in this example here, the bootstrap phase is assumed to take four time units. In order to guarantee $P.p$, the second providing component S, informally called the "startup component", is used to bridge this bootstrap phase. S is able to provide $S.p$ for exactly four time units.

We start to delve into the formal description of the system by investigating the global system specification that requires a q to be always true. It can be given as the following contract C_G:

$$C_G = (\llbracket \top \rrbracket, \llbracket \Box q \rrbracket)$$

The component signature of this contract is $\Sigma_G = (\emptyset, \{q\})$.

The design of the system makes use of a processing component P that guarantees q only if its required port p is always true. The signature of P is

therefore $\Sigma_P = (\{P.p\}, \{P.q\})$. Its contract can be specified as

$$C_P = (\llbracket \Box P.p \rrbracket, \llbracket \Box P.q \rrbracket)$$

To satisfy the property $\Box P.p$, we make use of the two components S and O, where S takes care of guaranteeing p during the bootstrap phase (four time units) and O guarantees p during the normal operation of the system. The signature of S and O are similar, as they both provide one port: $\Sigma_S = (\emptyset, \{S.p\})$, and $\Sigma_O = (\emptyset, \{O.p\})$. The contracts of S and O are as follows.

$$C_S = (\llbracket \top \rrbracket, \llbracket \Box_{[0,4]} S.p \rrbracket) \qquad C_O = (\llbracket \top \rrbracket, \llbracket \Box_{[4,\infty)} O.p \rrbracket)$$

Additionally, an assembly specification is needed that assembles the three components in a way that the assumption A_{C_P} is satisfied, and guarantees $\Box p$ such that the global contract holds. The composite component signature for the composite component C that hosts $\mathcal{A}_C$ can be defined as $\Sigma_C = (\Sigma_C^{Ext}, \Sigma_C^{Int}, C^i, C^d)$ where $\Sigma_C^{Ext} = (\emptyset, \{q\})$, $\Sigma_C^{Int} = (\{P.p\}, \{P.q, S.p, O.p\})$, $C^i = \{(P.p, S.p), (P.p, O.p\}$, and $C^d = \{(P.q, q)\}$. The following assembly specification is proposed.

$$\mathcal{A}_C = \llbracket (\Box q \sim P.q) \wedge (\Box_{[0,4]} P.p \sim S.p) \wedge (\Box_{(4,\infty)} P.p \sim O.p) \rrbracket$$

Now, we bundle the three components P, S, and O together into one composite component C. The resulting composite component is given as

$$C = (\Sigma_C, (A_C, G_C), \mathcal{A}_C, \{P, S, O\})$$

The component contract is specified as $C_C = (A_C, G_C) = (\llbracket \top \rrbracket, \llbracket \Box q \rrbracket)$, which is, obviously, a refinement of the specification C_G. Figure 4.3 shows the complete example scenario with the three components S, O, P boxed in the composite component C.

Fist, let us have a look at the receptivity of the contracts C_G, C_S, C_O, and C_P to make sure that all contracts are consistent. Considering C_G, C_S and C_O it is clear that the assumptions of all three contracts are Prov-receptive, as the assumptions are $\llbracket \top \rrbracket$, i.e. the set of all runs. The guarantees of C_G, C_S and C_O are Req-receptive for the simple reason that the set of required ports in the signatures of C_G, C_S and C_O is empty. It remains to investigate C_P. The assumption of C_P, $\llbracket \Box P.p \rrbracket$, is Prov-receptive, as $\llbracket \Box P.p \rrbracket \downarrow_{(\emptyset, \{P.q\})} = \{\rho \in \mathcal{R}(\Sigma_P) \mid \forall t \in \mathbb{R}_0^+ . P.p \in \rho(t)\} \downarrow_{(\emptyset, \{P.q\})} = \mathcal{R}((\emptyset, \{P.q\}))$. Similarly,

the guarantee of P, $[\![\Box P.q]\!]$ is Req-receptive. Obviously, $[\![\Box P.q]\!]\downarrow_{(\{P.p\},\emptyset)} = \mathcal{R}((\{P.p\},\emptyset))$.

Next, it is worthwhile to investigate the S-receptivity of the assembly specification $\mathcal{A}_C$. It must be shown that for each $\Sigma \in \{\Sigma_S, \Sigma_O, \Sigma_P\}$, it holds that $\mathcal{A}_C\downarrow_\Sigma = \mathcal{R}(\Sigma)$. As $\mathcal{A}_C = [\![(\Box q \sim P.q) \wedge (\Box_{[0,4]} P.p \sim S.p) \wedge (\Box_{(4,\infty)} P.p \sim O.p)]\!]$ specifies only connectivity requirements, it is obvious that $\mathcal{A}_C$ alone does not restrict the runs of S, O, and P. Looking more closely to the semantics, $\mathcal{A}_C$ reduces the set of runs to those that contain specific connectors in specific states, which – due to the restriction of valid runs – does restrict the possible valuations of ports in each state by requiring that valuations of connected ports must be the same. At first, the semantics of $\mathcal{A}_C$ can be given as $\mathcal{A}_C = \{\rho \in \mathcal{R}(\Sigma_C) \mid (\forall t \in \mathbb{R}_0^+ \ (P.q, q) \in \rho(t)) \text{ and } (\forall t \leq 4 \ (P.p, S.p) \in \rho(t)) \text{ and } (\forall t > 4 \ (P.p, O.p) \in \rho(t))\}$. Now, we need to show that restricting $\mathcal{A}_C$ to Σ_S is equal to the set $\mathcal{R}(\Sigma_S)$. Since it obviously holds that $\mathcal{A}_C\downarrow_{\Sigma_S} \subseteq \Sigma_S$, it suffices to show that $\Sigma_S \subseteq \mathcal{A}_C\downarrow_{\Sigma_S}$. Let hence $\rho \in \Sigma_S$ be a run. By the definition of restriction (Definition 25), we need to find a run $\rho' \in \mathcal{A}_C$ such that $\forall t \in \mathbb{R}_0^+ . \rho'(t) \cap \mathcal{S}(\Sigma_S) = \rho(t)$. We take the run

$$\rho'(t) = \begin{cases} \{(P.p, S.p), P.p, (P.q, q)\} \cup \rho(t) & \text{if } t \leq 4 \text{ and } S.p \in \rho(t) \\ \{(P.p, S.p), (P.q, q)\} \cup \rho(t) & \text{if } t \leq 4 \text{ and } S.p \notin \rho(t) \\ \{(P.p, O.p), (P.q, q)\} \cup \rho(t) & \text{if } t > 4. \end{cases}$$

Obviously, $\forall t \in \mathbb{R}_0^+ . \rho'(t) \cap \mathcal{S}(\Sigma_S) = \rho(t)$, since ρ' contains ρ in every state, and does not add any predicates belonging to Σ_S. At the same time, it holds that $\rho' \in \mathcal{A}_C$, since $P.p$ is connected to $S.p$ are connected in the time interval $[0, 4]$, and $P.p$ is connected to $O.p$ in the time interval $(0, \infty)$, and $P.q$ is always connected to q. Hence, $\mathcal{A}_C$ is Σ_S-receptive.

Similarly, it can be shown that $\mathcal{A}_C$ is Σ_O-receptive. In order to show that $\Sigma_O \subseteq \mathcal{A}_C\downarrow_{\Sigma_O}$, we choose for $\rho \in \Sigma_O$ the following ρ':

$$\rho'(t) = \begin{cases} \{(P.p, S.p), (P.q, q)\} \cup \rho(t) & \text{if } t \leq 4 \\ \{(P.p, O.p), (P.q, q), P.p\} \cup \rho(t) & \text{if } t > 4 \text{ and } O.p \in \rho(t) \\ \{(P.p, O.p), (P.q, q)\} \cup \rho(t) & \text{if } t > 4 \text{ and } O.p \notin \rho(t). \end{cases}$$

As before, it holds obviously that $\forall t \in \mathbb{R}_0^+ . \rho'(t) \cap \mathcal{S}(\Sigma_O) = \rho(t)$. At the same time it holds that $\rho' \in \mathcal{A}_C$, since ρ' satisfies the connectivity requirements of $\mathcal{A}_C$.

Finally, the proof that $\mathcal{A}_C$ is Σ_P-receptive is just the same as the proofs of Σ_S- and Σ_O-receptivity of $\mathcal{A}_C$.

It remains to be shown that C is consistent, however. For this, it must be shown that the conditions (1)-(6) specified in Definition 35 are satisfied. We assume that P, S, O are correct, and therefor condition (1) is satisfied. Conditions (2) and (3) are satisfied as the naming of ports and components implies so. Condition (5) was just investigated before: $\mathcal{A}_C$ is $\mathcal{S}$-receptive. Condition (6) is trivially true, as there is no externally visible required port. Hence, it only remains to show condition (4): The parallel composition of component contracts is indeed a refinement of the composite component contract:

$$C_C \succeq C_S \|^{\mathcal{A}_C} C_O \|^{\mathcal{A}_C} C_P$$

First, we compute the result of the parallel composition

$$(A, G) = C_S \|^{\mathcal{A}_C} C_O \|^{\mathcal{A}_C} C_P$$

Using Corollary 3, this can be performed on the syntactic level. The formula characterising the assumption resulting from the parallel composition of C_S and C_O, the startup component and the operation component, is given in the following, and immediately simplified.

(1)	$\mathcal{A}_C \vee \neg(\mathcal{A}_C \wedge G_S \wedge G_O)$	
(2)	$(\mathcal{A}_C \vee \neg\mathcal{A}_C) \vee \neg(G_S \wedge G_O)$	from (1), by rule (1) and (3)
(3)	$\top \vee \neg(G_S \wedge G_O)$	from (2), by rule (1) and (3)
(4)	$\top$	from (3), by rule (1) and (3)

While the assumption of the composition of C_S and C_O under $\mathcal{A}_C$ can be simplified to $\top$, the guarantee remains the conjunction of assembly specification and the guarantees of C_S and C_O:

$$\mathcal{A}_C \wedge G_S \wedge G_O$$

The composition of all three components finally yields the following assumption, which is also immediately simplified:

(1) $(\mathcal{A}_C \wedge \Box P.p) \vee \neg(\mathcal{A}_C \wedge \Box_{[0,4]} S.o \wedge \Box_{[4,\infty)} O.p \wedge G_P)$

(2) $(\mathcal{A}_C \wedge \Box P.p) \vee \neg(\mathcal{A}_C \wedge \Box_{[0,4]} P.p \wedge \Box_{[4,\infty)} P.p \wedge G_P)$
from (1), by rule (1) and (21)

(3) $(\mathcal{A}_C \wedge \Box P.p) \vee \neg(\mathcal{A}_C \wedge \Box P.p \wedge G_P)$ from (2), by rule (1) and (23)

(4) $(\mathcal{A}_C \wedge \Box P.p) \vee \neg(\mathcal{A}_C \wedge \Box P.p) \vee \neg G_P$ from (3), by rule (1)

(5) $(\mathcal{A}_C \wedge \Box P.p) \vee \neg(\mathcal{A}_C \wedge \Box P.p) \vee \neg G_P$ from (4), by rule (1)

(6) $\top$ from (5), by rule (1)

The composition of guarantees following Corollary 3 yields directly

$$G = (\mathcal{A}_C \wedge \Box P.q \wedge \Box_{[0,4]} S.p \wedge \Box_{[4,\infty)} O.p)$$

Now that the composition is constructed, it has to be verified that the composition is indeed a refinement. This can be done on the syntactic level again by using Corollary 2, in which we need to show two entailments. Both entailments are trivial: the assumption of the composite component contract C_C, $\top$, entails the assumption of the composition, $\top$. Similarly obvious, the guarantee of the composite component, $\Box p$ can be shown from $(\mathcal{A}_C \wedge \Box P.q \wedge \Box_{[0,4]} S.p \wedge \Box_{[4,\infty)} O.p)$ by applying rule (22).

Altogether, we have shown that the composite component is valid by proving all six requirements of validity of composite components. Since the contract of the composite component is a refinement of the initial specification (it is indeed identical), we have shown that the composite system undergoing reconfigurations that adhere to the assembly specification $\mathcal{A}_C$ is a design satisfying the specification initially imposed.

4.4 Related work

Scientific analysis and understanding of dynamically re-structuring systems have been a field of active research in recent years; The paper of Bradbury et al. [33] gives a decent overview of various approaches. The approaches closest to the work presented here are from Basso et al. [19] and from Barros et al. [17]. In the paper from Basso et al. [19], a CTL-based deduction approach to the specification and verification of Fractal component models [36] undergoing reconfigurations. The work combines several tooling platforms and techniques for the extraction of CTL formulas from Fractal component models, and the creation of proofs for desired system properties. As Basso et al. use an extension of CTL, ECTL$^+$, it is possible to specify liveness properties of the form $\Box\Diamond p$ and $\Diamond\Box p$ in their framework. However, they do not provide an assume-guarantee framework, and consider systems with discrete time.

Barros et al. [17], propose a model-checking approach based on the μ-calculus for verifying distributed Fractal component models. Again, the approach is based on an set of tools that support the creation of the needed formal models (labelled transition systems) from Fractal component models. By capturing reconfiguration behaviour as part of the formal model, properties of the system under reconfiguration can be expressed and verified. While the approach Barros et al. propose in [17] is backed up by tools supporting the individual steps required in the verification process, it does not provide assume-guarantee reasoning, and considers systems with discrete time only.

Other approaches to specification and verification of systems under reconfiguration include [50], offering a means for moving parts of the needed type checking for reconfigurations from runtime to compile-time, [40] proposing a contract-based approach to the composition and configuration of component-based systems using a process calculus and type system. A more recent approach is [113], presenting a formal specification of the semantics of the Fractal component model without providing tractable means for verification (see [33, 65] for an overview of component models supporting reconfiguration). One earlier approach to the verification of dynamic software architectures was Wright [5], using a mapping to classical process algebras (CSP [71] in that case) to leverage existing deadlock analysis tools. This approach is common in earlier work on reconfiguration; all work so far focused on discrete-time semantics for software systems, as they were sufficient for the specification of the systems that were considered at that time.

Assume-guarantee reasoning for distributed systems was initially introduced by Misra and Chandy in [86]. Our work was inspired by [3] and [76], both featuring a temporal logic approach to dynamic software architectures with a binary relation for the modelling of connections. Again, our approach differs from the works by using a continuous-time semantics. Similarly, our work differs from the work of Hooman [72, 73] by introducing dynamic architectures instead of static ones, and by relying on decidable real-time logic.

Our contract framework is heavily inspired by the assume-guarantee contract framework presented in [26], and extended in [46]. However, while Delahaye and Caillaud have shown in [46] how the assume-guarantee contract framework can be extended to probabilistic contracts, we discussed here an extension of this framework to dynamically evolving systems with real-time constraints, by introducing a temporal logic adapted for the specification of real-time assertions.

The work presented in this chapter is based on [106], which was extended

in [122], in which the authors show how the theoretical framework for verifying systems under reconfiguration that is presented here can be implemented (in a simplified version) using Real-Time Maude [90], an extension of the rewrite-based modelling and verification language and tool Maude [42]. Real-Time Maude also features an LTL model checker that can be used to check basic real-time properties.

4.5 Conclusion

In this chapter, an assume-guarantee-based verification framework for reconfiguring component-based systems was introduced. By providing refinement and composition operations, the verification framework allows to verify the design of a component-based system undergoing reconfigurations early in the development process; it is possible to specify an abstract desired property in terms of an assume-guarantee contract, and to specify a concrete component-based design consisting of reconfiguration rules and abstract specification of components. The more concrete design allows to divide the system into simpler, interacting components with well-defined contracts. By using the results of composition and refinement presented in this chapter, the framework allows to verify whether the chosen design satisfies the abstract desired property specified as assume-guarantee contract.

The framework itself consists of a semantic-level framework in which composition and refinement results were achieved, and a syntactic layer that allows the specification of assumptions, guarantees and assembly specifications with metric interval temporal logic (MITL). It was shown that reMITL, the extension of MITL for reconfigurations, is sound and complete, and a set of derived rules was presented that alleviates manual proofs of refinement and simplification of compositions.

The approach presented here is not unique in its single elements; As discussed in Section 4.4, formal treatments of reconfigurations exist as well as dense real-time reasoning frameworks and assume-guarantee reasoning frameworks. The approach presented here is, however, unique in the way it combines all three elements to create an assume-guarantee, dense real-time framework for the verification of reconfigurations. In that respect, it constitutes a novel contribution to the domain of formal verification of software systems.

Chapter 5

Methodology

Providing a component-based software framework (Chapter 3) and means for the formal verification of component reconfigurations (Chapter 4) alone does not provide clear guidance for the creation of physiological computing applications. This chapter aims at filling the gaps by providing a software development methodology that makes use of both the formal verification framework (Chapter 4) and the component-based software infrastructure (Chapter 3) and embeds them into a software development process that fits the requirements of physiological computing applications. For this, the verification artefacts as well as means for the informal modelling of system designs needs to be defined.

Creating physiological computing applications is an endeavour that includes a significant amount of research efforts: experimenting with psychological concepts, available hardware sensors and actuators, and experiments with different control strategies are all day-to-day activities. The software development methodology needs to be adequate for such development efforts; Scrum [107], as discussed in Section 2.5.6, is a good choice, also because its focus lies on the organisational aspect of task management than on the details of the software creation techniques used (compare with Section 2.5.4), which gives freedom in the definition of the latter.

In this chapter, we therefore begin with discussing how verification of reconfigurations is performed independently from any framing methodology in Section 5.1. Next, we proposing an approach for combining the tools introduced so far with the Scrum methodology. This integration is discussed in details in Section 5.2. In the subsequent Section 5.3, we introduce the notations that are missing for the execution of the formal verification method-

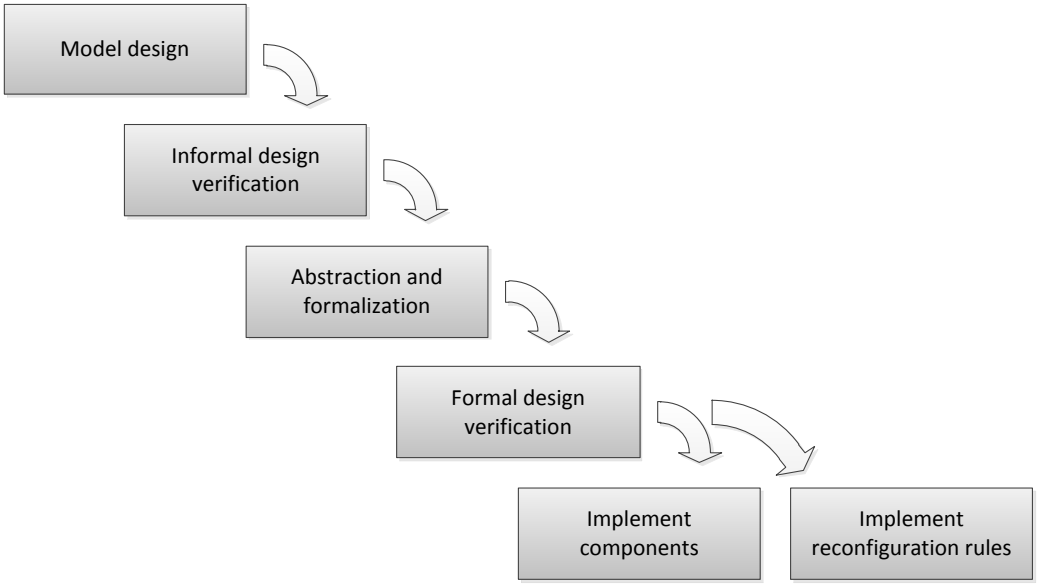


Figure 5.1: Standalone verification and implementation process

ology presented in Section 5.1, i.e. notations for designing component-based systems in the style of the REFLECT framework (especially supporting required and provided ports as well as the component lifecycle), as well as event-condition-action style reconfiguration rules.

Furthermore, we discuss how reconfigurations can be beneficially used for the development of physiological computing applications. Section 5.4 shows how reconfigurations can be used to respond to the volatile nature of devices in pervasive settings as well as for the adaptation of behaviours on a strategic level. By this, both examples of the methodology, as well as two widespread usage patterns for reconfigurations in physiological computing systems are presented.

Finally, we present related work in Section 5.5 before concluding this chapter in Section 5.6.

5.1 Standalone verification of reconfigurations

First, we discuss how the process of formal verification of reconfiguration is performed – independently from a framing methodology. Generally, the

verification and subsequent creation of structurally adaptive systems is performed along the following steps (compare with Figure 5.1):

1. **Design.** The component design needs to be modeled based on initial requirements (or change requests). The design consists of the following artefacts:

   (a) An informal, textual description of the desired system behaviour that should be achieved through the design.

   (b) A list of all needed component types with their ports.

   (c) An informal description of the expected component behaviour.

   (d) The initial component configuration, i.e. the initial component instances and connectors.

   (e) The reconfiguration rules that should be applied to the configuration as an informal textual description. A scheme that can be used and transitions nicely into temporal logic is the event-condition-action (ECA) scheme [2].

2. **Informal verification.** As a filter step, the informal description and diagrams should be verified informally to check that the design does not contain any obvious errors. The informal verification step is a lightweight filter step for the subsequent resource intensive formal verification, and is used to identify and single out problems early in the design, so that the design can be improved.

3. **Abstraction and formalisation.** First, the global system properties that need to be formally verified must be extracted from the informal textual description and formulated in reMITL. Together with this activity, the component definitions with their complex ports and data types need to be simplified to Boolean predicates that allow to formulate the behaviour of the components that is relevant for the global system properties. Once the abstraction to Boolean predicates is completed, the component signatures can be defined. Next, the informal component behaviour is translated into assume-guarantee contracts in reMITL. The initial component configuration and the reconfiguration descriptions are converted into the assembly specification, again using reMITL.

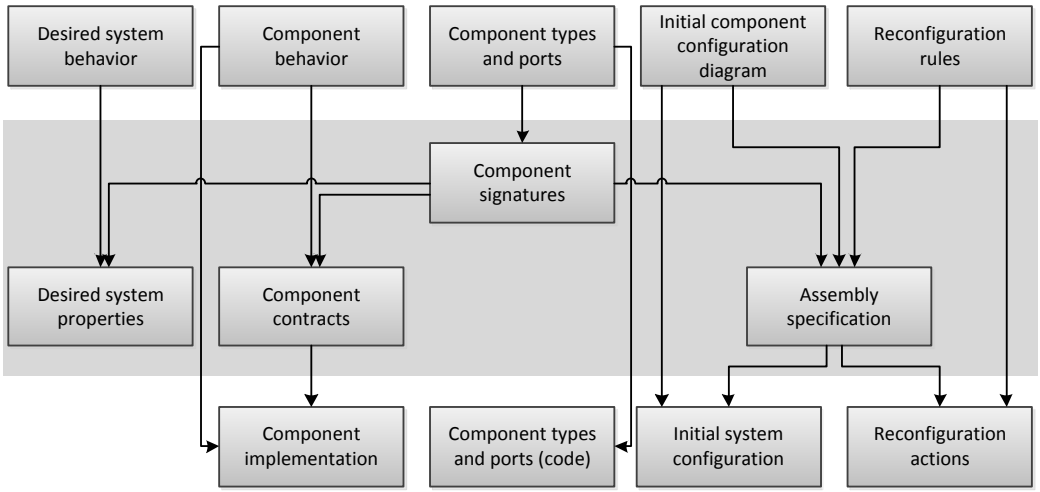


Figure 5.2: Verification and implementation artefacts

4. **Formal verification.** The existence of a refinement relation between the global system contract and the assume-guarantee contracts can be verified by using the formal verification framework described in Chapter 4.

5. **Implementation.** The initial, informal system design (consisting of components, ports and types) that served as the baseline for formal verification can be used directly as the skeleton for the implementation. At the same time, the initial configuration can be mapped directly to an initial system assembly. Finally, the formal and informal descriptions of component behaviour serve as specifications for the implementation of the component code. Similarly, the informal event-condition-action rules together with the assembly specification constitute the specification for the reconfiguration rules to be created, as well as for the initial system configuration.

Figure 5.2 depicts the artefacts that are created in the process together with their interdependencies. The upper third of the figure shows the artefacts created in the informal design phase: the textual description of the system behaviour and component behaviour, the set of component types and ports, the initial configuration, and a textual description of the reconfiguration rules.

The middle part of Figure 5.2, depicts the artefacts that are created in the phase of formalisation and abstraction, i.e. after the initial artefacts successfully passed the informal verification step. First, the component signatures are defined by abstracting from the component port and types given by the initial design. The signatures are crucial, as they define the alphabet for the specification of all system properties. Using the component signatures, the desired system behaviour can be translated into a set of global contracts, and the component behaviours into component contracts. Similarly, the information given by the initial configuration diagram and the reconfiguration rule descriptions are condensed to create the assembly specification. With these artefacts created, the formal verification step can be started to establish a formal proof of correctness.

In the final step, the implementation (shown in the lower third part of Figure 5.2), the code for the REFLECT framework is created: from the component ports and types of the initial design, the code for the static structure (i.e. the components with their portS) of the adaptive system can be created. Similarly, the component implementation is derived from the initial component behaviour and the formal component contracts. The initial system configuration is created from information available in the initial component configuration diagram and the assembly specification, and the reconfiguration actions are created from the textual description of the reconfiguration rules and from the assembly specification.

5.2 Combining formal verification and Scrum

A software development methodology for the development of physiological computing should be agile, as the development of physiological computing systems is often an undertaking that involves a significant amount of research work. We propose using Scrum [107] since it is a software development methodology that is well suited for the development of new products and software (cf. Section 2.5.6). Scrum has its roots in the empirical process model, and as a consequence, uses frequent inspection and adaptive feedback as its main means of control (compare with Section 2.5.4). Scrum is a generic software development methodology that can be used to create software, but also for research and development activities that do not primarily focus on software; hence, it is most suitable for the creation of physiological computing applications.

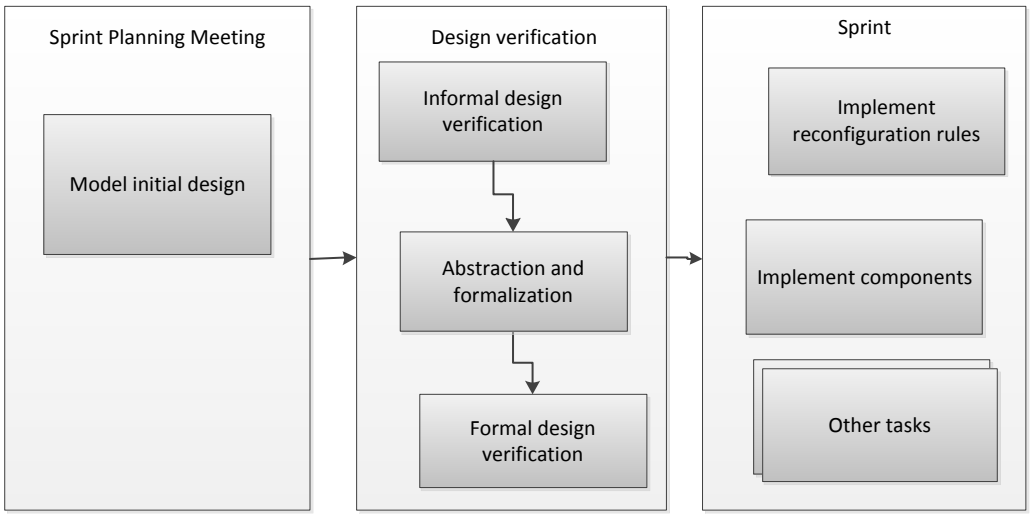


Figure 5.3: Delaying combination of reconfiguration verification and Scrum

However, the formal verification and implementation work needs to be carefully embedded in Scrum without diminishing its adaptive nature. When integrating the verification process described in Section 5.1 in Scrum, the question arises how the five steps defined for formal verification can be naturally embedded. Overall, there are two forces that guide the integration of the verification process into the agile methodology of Scrum:

- **Early verification.** The formal verification of a system structure and its reconfiguration rules is a step that needs to be performed early; the failure of a verification proof often entails changes in the structure of a system that require modifications of implementation code if that implementation code was already created.

- **Staying agile.** Scrum suggests to include early design discussion in the sprint planning meeting, but not to perform any detailed design activities in that meeting. Introducing design verification steps between the sprint planning and sprint execution causes a delay of implementation work that commemorates the waterfall approach. Such an approach should be avoided, as it would stall the whole development team, and reduce the adaptation capabilities of the development teams.

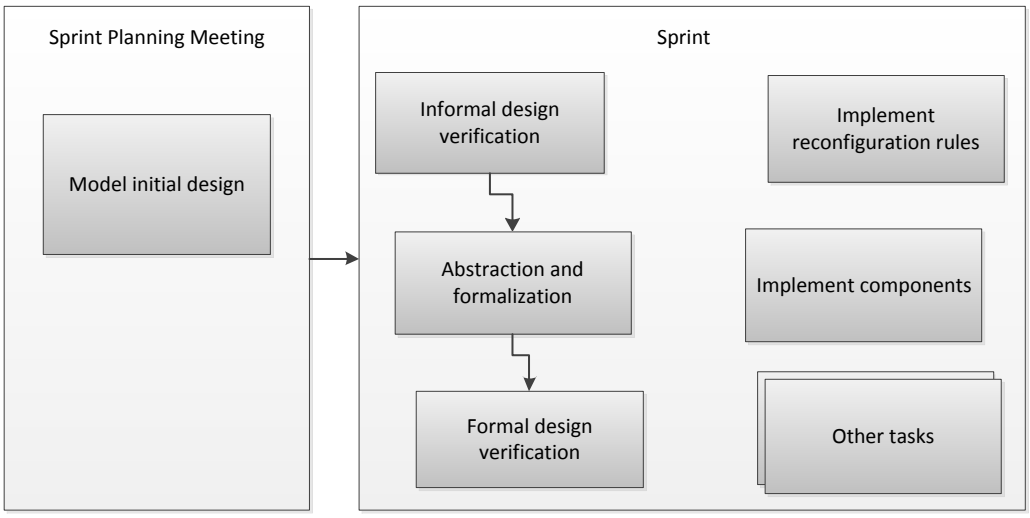


Figure 5.4: Agile combination of reconfiguration verification and Scrum

Figure 5.3 shows how the process would be organised if we followed solely the force of early verification. The modelling, verification, and implementation activities would be integrated in Scrum by introducing another phase before the start of a sprint. During the sprint planning meeting, the design would be elaborated and informally verified, while the the formal verification would be moved to an own phase of the project. Once the formal proof of correctness is achieved, the sprint starts, and the component and reconfiguration design is implemented along with other sprint backlog items that need completion.

In order to prevent progress delay, it is necessary to find a compromise between early verification and agility. Figure 5.4 shows a different approach to the integration of reconfiguration verification in Scrum. Here, the design verification activities are added as regular, additional sprint backlog items to the sprint itself. This implies that the reconfiguration rules and component behaviour implementation may start before a formal proof of correctness have been established. However, parallel implementation and verification allows to feed back insights gained from the implementation into the specification, and by this to rule out problematic designs that would have been hard to detect otherwise. Additionally, performing the formal validation of the design within the sprint allows the team to tackle product backlog items that do not

affect the design or component behaviour to be verified. This may include bug fixes and development in other parts of the system, experiments with new technologies, code cleanup and refactorings.

In the described approach, the early verification force seems to be blatantly violated, but it is not; it is only subordinated to the force of staying agile. Of course, the team should tackle the issue of component and reconfiguration design verification early on; partial implementation of an invalid design may lead to time-consuming changes and revisions that impede the progress of the sprint. At the same time, the team is empowered to make the correct trade-off between the forces of agility and early verification based on the current project context, and other existing forces.

In total, the second embedding shown in Figure 5.4 provides a more natural and realistic embedding of the formal verification of component and reconfiguration design into a system development effort. Rather than imposing a strict ordering on how to proceed, the development team is empowered to decide how to intertwine the formal verification process with other activities in the sprint for maximum benefit.

5.3 Informal design modelling

The development process described in Section 5.1 requires several models to be created, one of these being the initial design, consisting – according to Figure 5.2 – of the overall desired system behaviour, component behaviour, component types and ports, the initial component configuration diagram, and informal reconfiguration rules.

In this section, we present the notations that are missing for performing the formal verification methodology previously described; propose therefore propose a diagram type and meta-model for the the creation of first informal system configurations and architectures as supported by the REFLECT framework. The diagram type is primarily geared fro supporting rapid sketching of system configurations. Additionally, we propose a notational frame for describing reconfiguration rules based on the ECA rule scheme [2].

First, we introduce a metamodel (Figure 5.5) and graphical notation for system configurations. The metamodel (shown as UML class diagram) defines entities for components, ports and connectors contained within a system configuration. The metamodel includes definition and instance en-

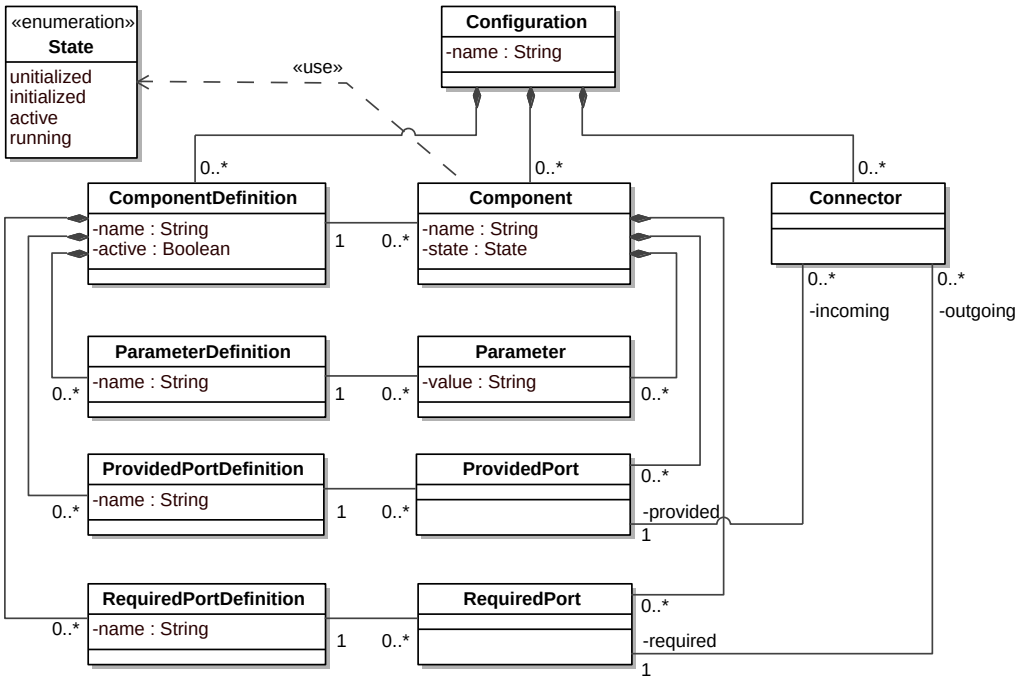
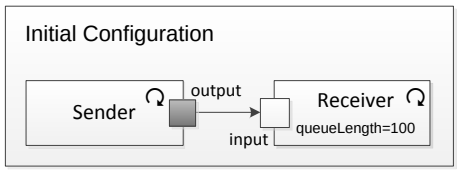


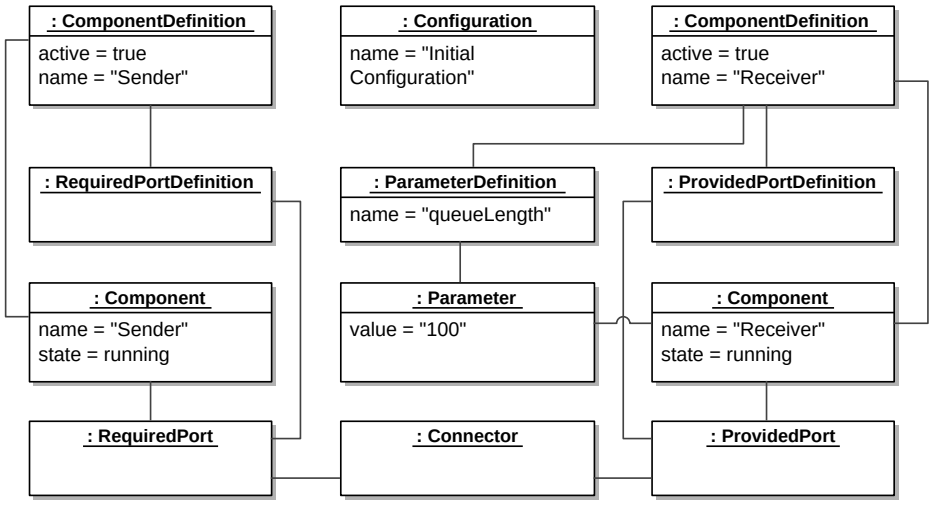
Figure 5.5: System configuration diagram metamodel

tities for components, ports and parameters, indicating that components, ports, and parameters have a specific type that is declared in code, while their instances are managed by the software framework at runtime. Components, ports and parameters are directly found in system configuration diagrams, while their declarations are only referenced indirectly through the specification of instances. The type declaration entities of the system configuration metamodel are `ComponentDefinition`, `ParameterDefinition`, `ProvidedPortDefinition`, and `RequiredPortDefinition`. These are the entities that are mapped to component classes (cf. Section 3.3.1), parameter accessor methods (cf. Section 3.3.4), and port accessor methods (cf. Section 3.3.3), respectively. The instance entities, i.e. `Component`, `Parameter`, `ProvidedPort`, `RequiredPort`, and `Connector` are found in system configurations; they are either in the initial system configuration specified in a bundle container (cf. Section 3.3.2 and Section 3.3.3), or created through reconfiguration actions that transform the system configuration.

The graphical notation for system configuration diagrams was already



(a) Graphical design notation



(b) Metamodel instance

Figure 5.6: System configuration diagram example

used throughout this work; Figure 5.6 shows both an example as a model instance of the metamodel given in Figure 5.5, and a system configuration diagram in its (compact) graphical notation. Both diagrams show the same information, one using the graphical notation, and one using UML object diagrams. For clarity, the links between the configuration and all its constituents were omitted in the UML object diagram shown in Figure 5.6b.

System configuration diagrams are used to specify existing components, component (lifecycle) states, component parameters, ports and their interconnections in a system state, but also define component types, ports and interfaces implicitly through their reference in the system configuration diagram. This approach is intended to support rapid sketching of system configurations and architectures, as this is the approach that was found to be the most useful in the creation of the case studies (Chapter 6). Note that the port type (i.e. required vs. provided port) is inferred from the colouring

of the port: a required port is dark gray, while a provided port is left white. The notation used for denoting active components is a round arrow as shown on both Sender and Receiver components. The notation for component parameters is shown on the Receiver component that defines a `queueLength` parameter with a current value of 100.

Component declaration names are directly inferred from the name of the component instance; where component declaration and component should not coincide in names, a notation following the UML can be used for separating component name and declaration name through a colon, hence using the scheme `component name : component declaration name`.

System configuration diagrams can be used to define the initial component configuration, as well as component, parameter, and port declarations by creating a system configuration diagram with one component per component declaration with all its parameters as well as all its required and provided ports. While this approach may seem overly complicated at first, it is indeed the natural approach that emerged during the design of the case studies (see Chapter 6) for specifying component types and their ports.

What is missing still is the specification of the global system behaviour and component behaviour, as well as the reconfiguration rules. For the behaviour specifications, we propose to resort to classical UML activity diagrams, state machines, or just plain text, depending on the complexity of the behaviour at hand – we did focus on researching specific behaviour notations that allowed for simpler specification of physiological computing system behaviours. For reconfiguration rules, we also propose to resort to existing description schemes, namely event-condition-action (ECA) [2].

With the help of ECA, reactive rules are described using the form *On* event *If* condition *Do* action. The interpretation of an ECA rule is that when the event occurs and the condition holds true at that time, the action is executed. ECA rules allow a natural structuring of reconfiguration rules into three compartments: an *event* describing when the *condition* of a rule needs to be evaluated, and what reconfiguration *action* needs to be performed if the condition is evaluated to be true.

The three compartments of ECA reconfiguration rules can be written down on using the following guidelines:

- **Events** are best described using free text. Events will mostly be signals emitted by one component to the container. Therefore, the emitting component should be stated, as well as how the component generates

the event – e.g. by repeatedly checking a resource, by reacting to an externally generated signal, or by detecting specific conditions in the incoming physiological data.

- **Conditions** can be described using free text and system configuration diagrams where appropriate. Conditions describe the prerequisites that must be satisfied for a reconfiguration action to be performed. The prerequisites may concern the system configuration as well as conditions to be evaluated on individual components.

- **Actions** are the more involved parts of ECA reconfiguration rules. Reconfiguration actions need to be expressed using the basic manipulation means on component-based systems, which are: component creation and deletion, connector creation and disconnection (these are the four listed in [78]), as well as changing component (lifecycle) state, and changing component parameters. In addition, actions need also to specify the components, component ports and connectors to manipulate. These queries are implicit in the naming of components, ports and connectors, which is fine for a first informal specification of reconfiguration actions. Nevertheless, these implicit queries should be kept in mind when describing reconfiguration actions.

It is advised to also provide examples in all three compartments for illustration purpose: sample event generating conditions, as well as sample contexts in which the condition should hold, and before-after-pairs of configurations for reconfiguration actions. Providing such examples also simplifies the first informal consistency checking that is performed after the creation of the initial design.

Let us have a look at an example ECA reconfiguration rule based on the system shown in Figure 5.6a. The reconfiguration rule shown in Table 5.1 is intended to switch the connector from the currently existing receiver to a newly created receiver. This is supposed to happen as soon as the current receiver sends the event that its queue has run full.

The sample ECA reconfiguration rule shows how lightweight the specification format is; it imposes only few constraints and rules on the description of reconfigurations. In total, system configuration diagrams, textual and/or UML-based behaviour specifications and ECA rules can be used to describe all the artefacts that the informal system design modelling phase needs. In the next section, we clarify how the formal specification is obtained from

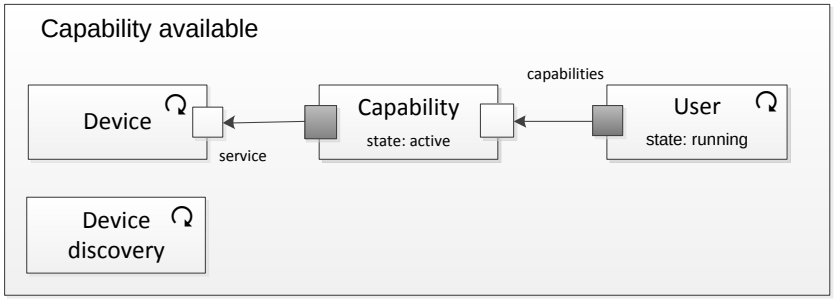
Event
Receiver connected to `sender.output` sends queue full event
Condition
Only one `receiver` instance exists
Action
• Create new `receiver` instance • Stop `sender` component • Disconnect `sender.output` connector • Connect `sender.output` port to new `receiver.input` port • Start new `receiver` component • Start `sender` component

Table 5.1: Example ECA reconfiguration rule

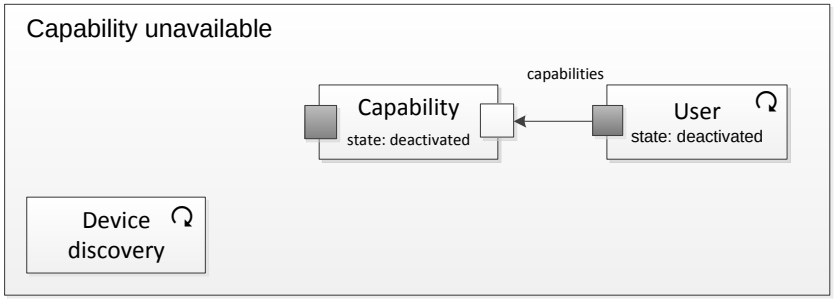
the initial design model, and how implementation skeletons can be created from both the initial design and the formal specification. We discuss this aspect by presenting three patterns of use for reconfigurations in physiological computing systems in Section 5.4.

5.4 Reconfiguration patterns in physiological computing

In physiological computing settings, reconfigurations can be used in several ways, of which two application areas emerge most prominently. These are discussed in more detail in the following: device tracking (Section 5.4.1) and adjusting the behaviour of physiological computing systems to the current context (Section 5.4.2). We describe how these patterns make use of the informal modelling, formal modelling and verification, and the software framework described in Chapter 4, Chapter 3, and Section 5.1, respectively. In each pattern description, we present schemes for each artifact described in the development methodology (Section 5.2) in addition to a discussion of the pattern's intent, its forces (how they interact, compete and conflict with each other and repeatedly create problems asking for solutions), and consequences (both positive and negative outcomes of applying the solution proposed).



(a) Configuration with available capabilities



(b) Configuration with unavailable capabilities

Figure 5.7: Configurations for tracking services

5.4.1 Device tracking

System configurations are a natural tool to model and track volatile devices and services that appear and disappear. Formal verification of reconfigurations can be used in these settings to verify the correct response of a software system to all possible reconfigurations initiated through the loss or discovery of devices, which is important as these situations typically have a large impact on the capabilities of a pervasive system. In a component-based system, both losing devices as well as losing capabilities can be modelled and achieved by removing components from the system configuration. Conversely, adding devices and capabilities equates to adding components to the system configuration.

Intent

Keeping track of volatile devices that may or may not be available in the environment is the main intent of the device tracking pattern. It allows

leveraging available devices in the application as soon as they appear.

Forces

We have identified three forces that guide the application of this pattern as well as the further elaboration of the design and implementation.

- **Robustness**. The application's goal is to remain available when parts of the system fail.
- **Leverage**. The application needs to make use of devices that are available only intermittently.
- **Volatility**. Quickly appearing and vanishing devices may lead to perceived inconsistent behaviour of the application and increased usage of computational resources. The stability of the environment and the impact of the availability of devices therefore needs to be considered.

Informal modelling

Typically, the initial model of a system keeping track of devices has a component representing the device, and capability components making use of services the device provides. The capability component itself is providing its services to a user component, which is a component from the application logic making use of the provided capability. Figure 5.7 shows the scheme for both a configuration having devices and capabilities available (Figure 5.7a), and a configuration where the device for the capability is missing (Figure 5.7b). In both configurations, the system must feature a device discovery component that signals to the configuration the discovery or vanishing of devices, so that the configuration can react accordingly. The reaction of the system to a vanishing device is the removal of the device component, and the deactivation of the capability component, and all its depending components. Conversely, once the appropriate device is found, a component for the device is created, and a connection to the capability is created.

The corresponding ECA reconfiguration rules are shown in Table 5.2 and Table 5.3. The rule handling the discovery of a device is straightforward: the new device is connected to the capability and the capability is activated, which cascades to the user component. The vanishing rule shown in Table 5.3 is slightly more involved, as it needs to check whether another device may

Event
Device discovery sends device discovered event.
Condition
Discovered device matches capability requirements, and capability component is not connected to another device component.
Action
• Create new device component configured from the discovery event • Connect `capability.service` to `device.service` • Activate capability • Start user

Table 5.2: Device discovery rule

exist that can satisfy the need of the capability component, and connect that device if so.

The behaviour of the components participating in this design is abstractly defined in Table 5.4. The globally desired behaviour of the service tracking system is described textually as follows: the capability component is always activated and connected to an available device of the type it needs, if such a device exists in the configuration. Obviously, the specified desired behaviour is only a skeleton of the specification that a complete system would encompass. In a complete system, more complex reconfiguration rules and other behaviours may be required as well.

Abstraction and formalisation

The next step in the software development methodology is to abstract and formalise the informal description into a specification in terms of reMITL assume-guarantee contracts for the component behaviours, the global behaviour, and the assembly specification.

The abstraction and formalisation step produces the component signatures, desired system properties, component contracts, and assembly specifications. In the service tracking pattern, the abstraction step reduces the amount of tracked devices to two devices. This reduction allows to prove the system to be correct while still featuring significant complexity. At the same time, all ports are reduced to Boolean-valued ports, meaning that the

Event
Device discovery sends device vanished event.
Condition
None.
Action
• Delete `device`, and disconnect `capability.service` from `device.service` (possibly deactivates `capability` and `user`) • If `capability` is now disconnected, check whether other matching `device` components exist, and if so: – Connect `capability.service` to `device.service` – Activate `capability` – Start `user`

Table 5.3: Device vanishing rule

Component	**Behaviour**
Device	Provide services to the capability component. In the case of sensing devices for example, the service means providing access to sensor data to the capability component.
Device discovery	Notifies the system of new devices and vanishing devices.
Capability	Provides specific capabilities for use through the system.
User	Uses the capability.

Table 5.4: Desired component and system behaviours for device tracking

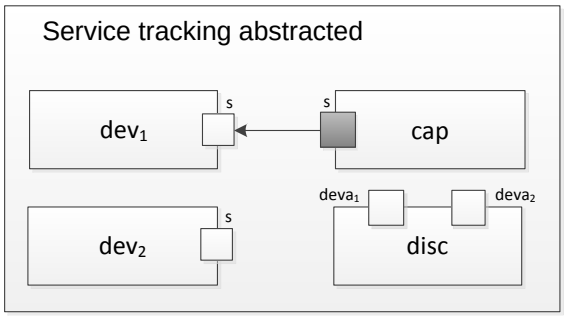


Figure 5.8: Service tracking system abstracted

discovery components now feature one port for the availability of each device.

We now define the component signatures for a system with one capability, one device discovery, and two devices. We omit the user component, as it is not important for the verification of the correct behaviour of the system behaviour. We shorten the names of the components to dev_1 and dev_2 for device components, *cap* for the capability component and *disc* for the device discovery component. The device components feature one provided port $dev_1.s$ and $dev_2.s$ for service, while the capability component has only one required port *cap.s* requiring the service. The discovery component features two provided ports, $disc.deva_1$ and $disc.deva_2$ signalling the availability of each device. Figure 5.8 shows the abstracted system with two devices, one capability, and one discovery with one port for each device. The component signatures can be derived directly from the figure:

$$
\begin{array}{lcl}
\Sigma_{dev1} & = & (\emptyset, \{dev_1.s\}) \\
\Sigma_{dev2} & = & (\emptyset, \{dev_2.s\}) \\
\Sigma_{cap} & = & (\{cap.s\}, \emptyset) \\
\Sigma_{disc} & = & (\emptyset, \{disc.deva_1, disc.deva_2\}) \\
\Sigma_{Sys} & = & ((\emptyset, \emptyset), \Sigma_{Int}, (C^i, \emptyset)) \\
\Sigma_{Int} & = & (\{cap.s\}, \{dev_1.s, dev_2.s, disc.deva_1, disc.deva_2\}) \\
C^i & = & \{(cap.s, dev_1.s), (cap.s, dev_2.s)\}
\end{array}
$$

The system signature Σ_{Int} is the signature over which the assembly specification is defined. Note that due to the connector constraints expressed in C^i, the required port *cap.s* may only be connected to the provided ports $dev_1.s$ and $dev_2.s$.

The desired global system behaviour can be directly formalised from Ta-

ble 5.4. First however, we introduce the following abbreviations:

$$\begin{aligned} cap\_connected &= cap.s \sim dev_1.s \vee cap.s \sim dev_2.s \\ dev\_available &= disc.deva_1 \vee disc.deva_2 \end{aligned}$$

With the help of these abbreviations, the desired assume-guarantee contract for the system global can be specified as $(\llbracket A_S \rrbracket, \llbracket G_S \rrbracket)$. While the assumption is empty (i.e. $A_S = \top$), the desired guarantee is

$$G_S = dev\_available \rightarrow \Diamond_{[0,l]} cap\_connected$$

Where l is a reaction limit that is specific to the concrete system under observation.

In the service tracking pattern, no component contracts are needed in order to satisfy the global system property; service tracking is an activity solely performed by the assembly and its reconfiguration rules which are ultimately triggered by the environment. It is impossible to establish any meaningful assumptions or guarantees on the duration of presence of capabilities and devices, as the discovery and vanishing of devices is fully under control of the environment. The contracts for the components dev_1, dev_2, cap, and $disc$ are therefore $(\llbracket \top \rrbracket, \llbracket \top \rrbracket)$.

The last piece to the puzzle, the assembly specification, can be derived from the ECA reconfiguration rules given in Table 5.2 and Table 5.3. The assembly specification consists of the semantics of the conjunction of four formulas, $\mathcal{A} = disc_1 \wedge disc_2 \wedge van_1 \wedge van_2$, which are defined as follows:

$$\begin{aligned} disc_1 &= disc.deva_1 \wedge \neg cap\_connected \rightarrow \Diamond_{[0,l]} cap.s \sim dev1.s \\ disc_2 &= disc.deva_2 \wedge \neg cap\_connected \rightarrow \Diamond_{[0,l]} cap.s \sim dev2.s \\ van_1 &= \neg disc.deva_1 \wedge cap.s \sim dev_1.s \rightarrow \big[(\Diamond_{[0,l]} \neg cap.s \sim dev_1.s) \\ &\qquad \wedge (disc.deva_2 \rightarrow \Diamond_{[0,l]} cap.s \sim dev_2.s)\big] \\ van_2 &= \neg disc.deva_2 \wedge cap.s \sim dev_2.s \rightarrow \big[(\Diamond_{[0,l]} \neg cap.s \sim dev_2.s) \\ &\qquad \wedge (disc.deva_1 \rightarrow \Diamond_{[0,l]} cap.s \sim dev_1.s)\big] \end{aligned}$$

To prove that the desired system contract follows from the component assembly, it suffices to show that the global system guarantee follows from the assembly specification, which we do in the following.

First, let us assume that $dev\_available$ holds, i.e. $disc.deva_1 \vee disc.deva_2$. We assume that $disc.deva_1$ holds – the case where $disc.deva_2$ holds can be

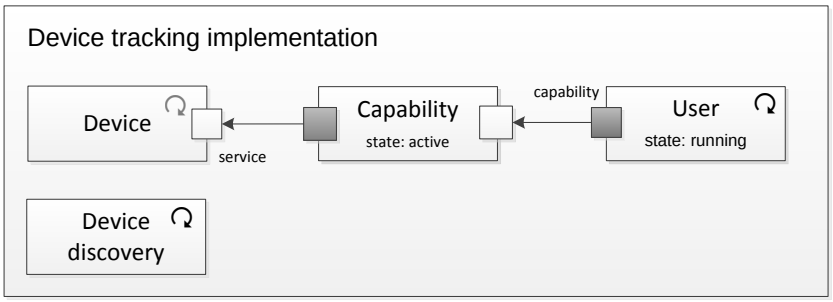


Figure 5.9: Device tracking system as implemented

shown analogously. Furthermore, we assume that $\neg cap\_connected$ holds, as otherwise, the proof that $\Diamond_{[0,l]} cap\_connected$ holds is already established. Since $disc.deva_1 \wedge \neg cap\_connected$ holds, $disc_1$ can be applied, and it follows that $\Diamond_{[0,l]} cap.s \sim dev_1.s$, and hence $\Diamond_{[0,l]}(cap.s \sim dev_1.s \vee cap.s \sim dev_2.s)$.

Implementation

The implementation of the device tracking pattern is very close to the initial design discussed in Section 5.4.1; Figure 5.9 shows the static structure of the implementation. The figure corresponds to the initially modelled configuration except for the fact that the user component references only one capability in the pattern implementation.

We do not discuss all the component code in this section, but only relevant and interesting classes, which are the container (Listing 5.1), and the reconfiguration actions for device discovery (Listing 5.2) and device vanishing (Listing 5.3).

The container, as usual, creates all components and connects the user port to the capability component. Additionally, the container registers itself as a listener for device events, and triggers appropriate configuration actions.

Listing 5.1: Device tracking pattern system container

```
public class Container extends BundleContainer {

  @Override
  protected void configure() {
    create("user").ofType(User.class);
    create("capability").ofType(Capability.class);
```

```
    create("deviceDiscovery").ofType(DeviceDiscovery.class);
    connect().ports().ofComponent("user");
  }

  @EventListener(topic = DeviceEvent.TOPIC)
  public void deviceEvent(DeviceEvent evt) {
    switch (evt.getType()) {
    case DISCOVERED:
      reconfigure(new Discover(Container.this,
          evt.getName()));
      break;
    case VANISHED:
      reconfigure(new Vanish(evt.getName()));
      break;
    }
  }
}
```

The configuration actions take the responsibility for checking the condition, and performing configuration changes if the condition holds. The code given in Listing 5.2 corresponds to the rule defined in Table 5.2 except for the fact that we assume compatibility of device type and capability requirement; the check whether the capability is already connected to a device service is performed in the discovery rule, and the rule is aborted in case the check is successful (i.e. the method returns immediately).

Listing 5.2: Device discovery reconfiguration action

```
public class Discover extends AConfigurationAction {

  private final BundleContainer fContainer;

  private final String fDevice;

  public Discover(BundleContainer container, String device) {
    super("Connect");
    fContainer = container;
    fDevice = device;
  }

  @Override
  protected void reconfigure(ReconfigurationReflectManager
     manager) {
    Device device = new Device(fDevice);
    manager.addComponent(fContainer, device);
```

```
    AComponent capability = manager.
        getComponent("capability");
    IRequiredPort capabilityPort = capability.
        getRequiredPort("service");

    if(capabilityPort.getCardinality() != 0)
      return;

    IProvidedPort devicePort = device.
        getProvidedPort("service");
    AActiveComponent user = (AActiveComponent) manager.
        getComponent("user");
    manager.connect(capabilityPort, devicePort);
    capability.activate();
    manager.scheduleRestart(user);
  }
}
```

The reconfiguration rule that reacts to a vanishing device, shown in Listing 5.3, takes care of deleting the device from the system configuration. If the removed device was connected to the capability component, the system configuration is queried for another device to satisfy the capability dependency.

Listing 5.3: Device vanishing reconfiguration action

```
public class Vanish extends AConfigurationAction {

  private final String fDevice;

  public Vanish(String device) {
    super("Vanish");
    fDevice = device;
  }

  @Override
  protected void reconfigure(ReconfigurationReflectManager
      manager) {
    AComponent device = manager.getComponent(fDevice);
    manager.deleteComponent(device);

    AComponent capability = manager.
        getComponent("capability");
    IRequiredPort capabilityPort = capability.
```

```
          getRequiredPort("service");

      if(capabilityPort.getCardinality() == 1)
        return;

      Device nextDevice = manager.findComponent(Device.class);
      if (nextDevice == null)
        return;

      IProvidedPort devicePort = nextDevice.
          getProvidedPort("service");
      manager.connect(capabilityPort, devicePort);
      capability.activate();
      manager.scheduleRestart((AActiveComponent) manager.
          getComponent("user"));
    }
  }
```

This concludes the discussion of the implementation of the device tracking pattern, which corresponds to its initial design model.

The device tracking pattern is used in physiological computing applications when devices may appear and vanish arbitrarily during the operation of the system. Since the software system has no control over the availability of the device, however, no guarantee about the minimal availability of specific capabilities can be made and verified. For a minimal availability guarantee, it would be necessary to require more elaborate behaviour from the capabilities (such as caching of requested commands), and to make stability assumptions on the environment (i.e. that devices stay available for a minimum duration once they become available).

Consequences

The application of the device tracking pattern has two consequences for the design and implementation code. First, it separates the device discovery and initialisation code from the device wrapper itself, as the logic becomes separated over two components: the device discovery, and the device wrapper component itself. Secondly, as the device discovery operates asynchronously to the application logic, application of the device discovery pattern makes synchronization of device discovery events (as well as device vanishing events) with the main application logic necessary.

5.4.2 Behaviour adjustment pattern

A key benefit the concept of reconfiguration offers when creating physiological computing applications is that it allows changing the behaviour of physiological computing systems. More specifically, it allows separating meta-level monitoring, reasoning, and behaviour modification from the algorithms realising the physiological computing behaviour. With the help of the reconfiguration concept, it is possible to let meta-behaviour and behaviour interact with each other in a defined and meaningful way. Thinking of a physiological computing system as a reconfigurable system may also help to identify opportunities for meta-level monitoring of and reasoning on the physiological computing behaviour.

In this section, we discuss the basic pattern for adapting behaviour of physiological computing systems by using structural reconfiguration (i.e. the substitution of software components). We discuss the pattern by detailing all three steps of informal modelling, abstraction and formalisation, and implementation.

Intent

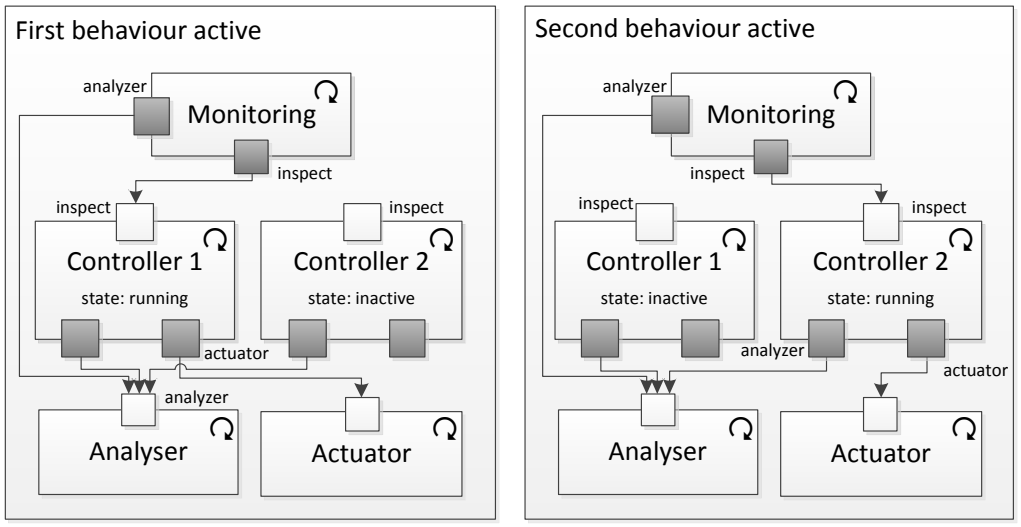
The intent of the pattern is to change the behaviour in new operational contexts, and in order to do so, to separate this reasoning from the main application logic.

Forces

The application of this pattern is mainly driven by two forces.

- **Discoverability.** It is of key importance that contextual changes are reliably discoverable through the system in order to provide adequate (i.e. consistent) user experience through the system. If the contextual change detection is unreliable, the behaviour adaptation must remain subtle and on the fringe of perception.

- **Volatility.** The speed of context changes is also important to relate to the speed of the reconfiguration. If context changes occur in rapid succession, it may not be feasible to adjust the behaviour through reconfigurations. Instead, the behaviour adaptation logic has to be interwoven with the regular behaviour, and the context granularity has to be decreased.



(a) Configuration exhibiting first behaviour (b) Configuration exhibiting second behaviour

Figure 5.10: Configurations for switching between behaviours

Informal modelling

A physiological computing system generally consists of behaviour-determining components, and components providing analysis information and access to physical actuators. For the illustration of the behaviour adaptation pattern, we have chosen three components: an analysis component, an actuator component, and a controller component determining the behaviour of the physiological computing system (cf. Figure 5.10).[1] For altering its behaviour at runtime, the system needs a second behaviour component that it can switch to, and a monitoring component that supervises the current physiological system behaviour, as well as the context in which the system currently operates. The monitoring component emits an event to the container as soon as the need for reconfiguration is detected. The description of the behaviour adaptation pattern remains abstract, with the complexity of detecting the need for changing behaviour being encapsulated in the monitoring compo-

[1]These three components are placeholders; in real physiological systems, the behaviour will consist of more than one component, and the analysis and actuation may consist of several components themselves, or be provided by one single hardware abstraction component, or a combination of both.

Event
Monitoring requests switch to controller 2.
Condition
Controller 1 is active.
Action
• Disconnect `monitoring.inspection` from `controller 1.inspection` • Disconnect `controller 1.actuator` from `actuator.actuator` • Connect `monitoring.inspection` to `controller 2.inspection` • Connect `controller 2.actuator` to `actuator.actuator` • Start `controller 2`

Table 5.5: Rule changing behaviour to the second controller

nent – although the system developers should be aware that the monitoring logic may constitute the major challenge in realising this pattern.

In the informal modelling phase, we have to establish the initial configuration diagram and settle the component and port types; Figure 5.10a shows the initial configuration. The system thus consists of two behaviour controls of different type, both having one provided inspection port, one required analyser port, and one required actuator port. The analyser component and the actuator component are both able to satisfy these required ports by providing one port each. The inspection port of the behaviour control component serves the monitoring component (together with the required analyser port) as input for deciding on changing the system configuration by emitting the reconfiguration-triggering event. The monitoring component emits a reconfiguration event if it detects that the current control component fails to maintain the desired behaviour, and that environmental conditions (through the analyser component) indicate that a switch to the other control component is advised.

The reconfiguration rules themselves are very simple, as the monitoring component takes care of verifying whether the reconfiguration should take place. The reconfiguration rule only needs to verify that the reconfiguration can take place, as can be seen in Tables 5.5 and 5.6.

The behaviour that we expect from the single components must remain

Event
Monitoring requests switch to `controller 1`.
Condition
`Controller 2` is active.
Action
• Disconnect `monitoring.inspection` from controller 2.inspection • Disconnect `controller 2.actuator` from `actuator.actuator` • Connect `monitoring.inspection` to `controller 1.inspection` • Connect `controller 1.actuator` to `actuator.actuator` • Start `controller 1`

Table 5.6: Rule changing behaviour to the first controller

abstract; we can only vaguely require that the monitoring component detects the need for switching between behaviours by using the data available through its inspection and analysis ports, and that the control components are able to establish the desired system behaviour under specific conditions – namely, the conditions the monitoring component uses to determine a necessary reconfiguration. The desired global system behaviour requires the system to always re-establish the correct behaviour within a specific time bound – that is to say, if the monitoring component detects that the current control component fails to establish the correct control behaviour, the system has a grace period for performing the reconfiguration, and re-establishing the correct control behaviour. The behaviour descriptions for all components are given in Table 5.7. The desired global system behaviour is very simple: we require that the system always re-establishes the correct control behaviour within a time bound t_b.

Abstraction and formalisation

The abstraction and formalisation of the behaviour adjustment pattern mainly consists of reducing the number of components, creating the required Boolean ports, and introducing a formal behaviour specification for components and reconfigurations. The actuator component is removed from the formal component system as it is not relevant for the behaviour of reconfigurations. The analysis component on the other hand remains in the abstracted system, as

Component	Behaviour
Monitoring	Detects the need for changing the system behaviour, and notifies the configuration.
Controller 1	Establishes the desired system behaviour under specific conditions.
Controller 2	Establishes the desired system behaviour under specific conditions.
Analyser	Analyses the (physical and physiological) context and provides the results.
Actuator	Provides access to physical actuators.

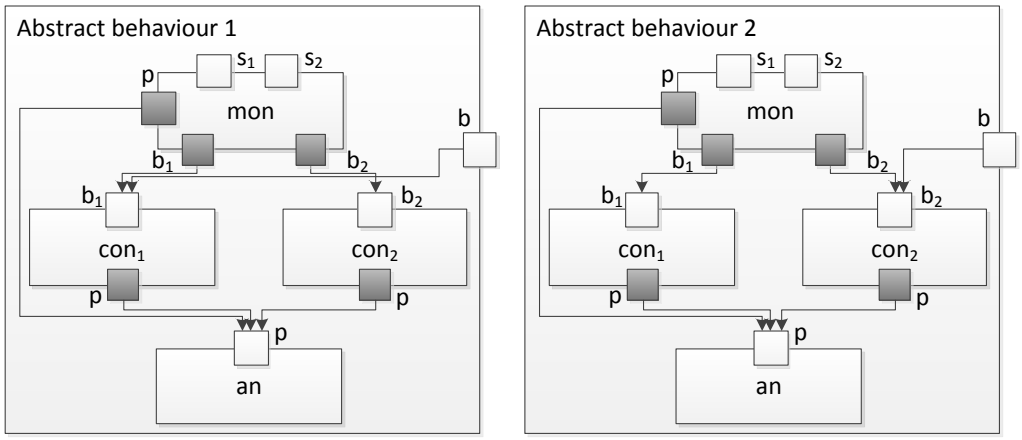
Table 5.7: Desired component and system behaviours for behaviour adaptation

it provides the information about which control component may be used to establish the globally desired behaviour within a certain time bound.

For simpler specification of the component and reconfiguration behaviour, the following abbreviations are used for components and ports:

- con_1 and con_2 for control components,
- mon for the monitoring components,
- an for the analyser component,
- p for the context predicate that determines whether the first or the second controller is suited best,
- b, b_1, b_2 for the behaviour monitoring ports,
- s_1 and s_2 for the ports signalling the reconfiguration of the system to the first controller and the second controller, respectively.

The signatures of the components are as follows (compare also with Fig-



(a) Abstraction for first behaviour (b) Abstraction for second behaviour

Figure 5.11: Behaviour changing system abstraction

ure 5.11):

$$
\begin{aligned}
\Sigma_{con_1} &= (\{con_1.p\}, \{con_1.b_1\}) \\
\Sigma_{con_2} &= (\{con_2.p\}, \{con_2.b_2\}) \\
\Sigma_{an} &= (\emptyset, \{an.p\}) \\
\Sigma_{mon} &= (\{mon.b_1, mon.b_2, mon.p\}, \{mon.s_1, mon.s_2\}) \\
\Sigma_{Sys} &= ((\emptyset, \{b\}), \Sigma_{Int}, (C^i, C^d)) \\
\Sigma_{Int} &= (\{con_1.pcon_2.p, mon.b_1, mon.b_2\}, \\
&\quad \{con_1.b_1, con_2.b_2, mon.s_1, mon.s_2, mon.p\}) \\
C^i &= \{(con_1.p, an.p), (con_2.p, an.p), (mon.p, an.p) \\
&\quad (mon.b_1, con_1.b_1), (mon.b_2, con_2.b_2)\} \\
C^d &= \{(b, con_1.b_1), (b, con_2.b_2)\}
\end{aligned}
$$

Note that in the abstracted version, the monitoring component continuously monitors the behaviour of component 1 and component 2, which serves the simplification of the formalisation of this pattern.

Before giving the behaviour specification of the single components and reconfigurations, we define abbreviations for the configurations depicted in Figure 5.11a (c_1) and Figure 5.11b (c_2). These are used in the assembly

specification:

$$\begin{aligned} c_1 &= b \sim con_1.b_1 \wedge mon.b_1 \sim con_1.b_1 \wedge con_2.b_2 \sim mon.b_2 \\ &\wedge \quad con_1.p \sim an.p \wedge con_2.p \sim an.p \wedge mon.p \sim an.p \\ c_2 &= b \sim con_2.b_2 \wedge mon.b_1 \sim con_1.b_1 \wedge con_2.b_2 \sim mon.b_2 \\ &\wedge \quad con_1.p \sim an.p \wedge con_2.p \sim an.p \wedge mon.p \sim an.p \end{aligned}$$

Now, we need to specify contracts for the controller 1, controller 2, monitoring, and analyser components. The assumptions of all those components are $[\![\top]\!]$, such that we can focus on the specification of the guarantees of the components. For a start, the guarantee of the analyser component, G_{an} is $\top$ as well, as we assume that the analyser component provides information about the environment which is not under (full) control of the system.

$$\begin{aligned} G_{con_1} &= \Box(con_1.p \rightarrow con_1.b_1 \wedge \Diamond_I con_1.b_1) \\ G_{con_2} &= \Box(\neg con_2.p \rightarrow con_2.b_2 \wedge \Diamond_I con_2.b_2) \\ G_{mon} &= \Box(\neg mon.b_1 \wedge \neg mon.p \leftrightarrow mon.s_2) \\ &\wedge \quad \Box(\neg mon.b_2 \wedge mon.p \leftrightarrow mon.s_1) \\ \mathcal{A} &= \Box(c_1 \vee c_2) \wedge \Box(c_1 \wedge mon.s_2 \rightarrow \Box_I c_2) \\ &\wedge \quad \Box(c_2 \wedge mon.s_1 \rightarrow \Box_I c_1) \end{aligned}$$

The guarantees G_{con_1} and G_{con_2} as well as the assembly specification $\mathcal{A}$ are parameterised by a time interval I representing the response time of the reconfigurations, and also determines the time interval in which the desired behaviour can be re-established. The time interval cannot include zero, but is otherwise arbitrary.

The analyser predicate determines whether the controller provides the desired behaviour ($con_1.b_1$ and $con_2.b_2$). In a way, $an.p$ partitions a system run into phases where the first controller would behave correctly (i.e. when $an.p$ is true), and phases where the second controller would behave correctly. Such a partitioning is necessary in order to ensure the desired global system guarantee. Accordingly, the monitoring component initiates a switch from configuration 1 (Figure 5.11a) to configuration 2 (Figure 5.11b) if the analysis component indicates that the second configuration would be more appropriate (i.e. $mon.p$ is false), and at the same time, the current configuration does not satisfy the desired property (i.e. $mon.b_1$ is false as well). Together with the assembly specification that guarantees a transition from

configuration one to configuration two in the time interval I, it is possible for the system to re-establish the desired behaviour, as controller 2 guarantees to behave as desired within that same interval I.

The global desired system behaviour can be specified as the following contract:

$$C_{sys} = (\llbracket \top \rrbracket, \llbracket \Box(b \vee \Diamond_I b) \rrbracket)$$

We carry on to show that the component guarantees in conjunction with the assembly specification entail $\Box\Diamond_I mon.b$ in the following proof.

We perform a case distinction over c_1: first, we assume that c_1 holds. Now, we perform a case distinction over $an.p$: first, we assume that $an.p$ holds. Since c_1 holds, we know that $con_1.p \sim an.p$, and therefore $con_1.p$ holds. G_{con_1} then entails $con_1.b_1$, and since $b \sim con_1.b_1$ follows from c_1, b holds. Now, we assume that $\neg an.p$ holds. We perform a case distinction over $con_1.b_1$. If $con_1.b_1$ holds, so does b (by using c_1). If $\neg con_1.b_1$ holds, we continue as follows: since c_1 holds, $mon.b_1 \sim con_1.b_1$ and therefore $\neg mon.b_1$ holds. Similarly, we can deduce $\neg mon.p$ from $mon.p \sim an.p$ and $\neg an.p$. Since G_{mon} states that $\Box(\neg mon.b_1 \wedge \neg mon.p \leftrightarrow mon.s_2)$, we can deduce that $mon.s_2$ holds. Using $\Box(c_1 \wedge mon.s_2 \rightarrow \Box_I c_2)$ from $\mathcal{A}$, we can derive $\Box_I c_2$. Additionally, we know that $con_2.p \sim an.p$ from c_1, and therefore $\neg con_2.p$ holds. From G_{con_2}, we can deduce that $\Diamond_I con_2.b_2$ holds. Together with $\Box_I b \sim con_2.b_2$ that can be deduced from $\Box_I c_2$, we can show that $\Diamond_I b$ holds.

Left to be shown is the case $\neg c_1$. Since $\mathcal{A}$ holds, we can derive c_2. This case is proved analogously to the case c_1.

Implementation

The implementation of the pattern corresponds to the initial model, although it features some additional implementation details (such as the use of events for communication). The system structure of the implementation is as given in Figure 5.10. Listing 5.4 shows the component and binding creation code that is necessary to set up the system.

Listing 5.4: Behaviour change pattern container action

```
public class Container extends BundleContainer {

  @Override
  protected void configure() {
    create("controller1").ofType(Controller1.class);
    create("controller2").ofType(Controller2.class);
```

```
    create("monitoring").ofType(Monitoring.class);
    create("analyzer").ofType(Analyzer.class);
    create("actuator").ofType(Actuator.class);

    connect().ports().ofComponent("controller1");
    connect().ports().ofComponent("controller2");
    connect().ports().ofComponent("monitor").toAPort().
        ofComponent("controller1");
    connect().ports().ofComponent("monitor").toAPort().
        ofComponent("analyzer");
  }

  @EventListener(topic=ChangeBehaviourEvent.TOPIC)
  public void changeBehaviour(ChangeBehaviourEvent evt) {
    if(evt.getTarget() == Behaviour.BEHAVIOUR_1)
      reconfigure(new ChangeToBehaviour1());
    if(evt.getTarget() == Behaviour.BEHAVIOUR_2)
      reconfigure(new ChangeToBehaviour2());
  }
}
```

The system uses two events types to request behaviour-changing reconfigurations (`ChangeBehaviourEvent`), and to pass around changes in the behaviour that occurred (`BehaviourChangedEvent`). The reconfiguration requests are created by the monitoring component (Listing 5.5), and consumed by the system container to initiate the appropriate reconfigurations (Listing 5.4).

Listing 5.5: Behaviour supervising monitoring component

```
public class Monitoring extends AManagedActiveComponent {

  private Behaviour fActiveBehaviour;

  private IBehaviourInspection fInspect;

  private IContext fContext;

  public Monitoring(String identifier) {
    super(identifier);
    setActiveBehaviour(Behaviour.BEHAVIOUR_1);
  }

  public void setActiveBehaviour(Behaviour behaviour) {
    fActiveBehaviour = behaviour;
```

```
    }

    @Override
    protected boolean step() throws InterruptedException {
      if (!isCurrentBehaviourAdequate() &&
          contextIndicatesBehaviourChange()) {
        Behaviour newBehaviour = null;
        if (fActiveBehaviour == Behaviour.BEHAVIOUR_1) {
          newBehaviour = Behaviour.BEHAVIOUR_2;
        }
        else if (fActiveBehaviour == Behaviour.BEHAVIOUR_2) {
          newBehaviour = Behaviour.BEHAVIOUR_1;
        }
        if (newBehaviour != null) {
          event(new ChangeBehaviourEvent(this, newBehaviour));
        }
      }
      Thread.sleep(100);
      return true;
    }

    @Required
    public void setInpsect(IBehaviourInspection behaviour) {
      fInspect = behaviour;
    }

    private IBehaviourInspection getBehaviour() {
      return fInspect;
    }

    @Required
    public void setContext(IContext context) {
      fContext = context;
    }

    private IContext getContext() {
      return fContext;
    }

    @EventListener(topic=BehaviourChangedEvent.TOPIC)
    public void configurationSet(BehaviourChangedEvent evt) {
      setActiveBehaviour(evt.getBehaviour());
    }
  }
```

In order to emit the correct behaviour change request, the monitoring component needs to keep track of the currently active behaviour. The monitoring component keeps up-to-date concerning the currently active behaviour by registering itself as a listener to behaviour change events. The behaviour change events themselves are published by the reconfiguration rules after successful completion.

Listing 5.6 shows the code for switching to the first behaviour. As informally described in Table 5.6, the reconfiguration rule takes care of verifying whether the desired behaviour is already active, and continues with disconnecting and reconnecting the behaviour inspection port and actuator ports.

Listing 5.6: Reconfiguration rule switching to first behaviour

```
public class ChangeToBehaviour1 extends AConfigurationAction {

  public ChangeToBehaviour1() {
    super(Behaviour.BEHAVIOUR_1.toString());
  }

  @Override
  protected void reconfigure(ReconfigurationReflectManager
      manager) {
    AComponent controller1 = manager.
      getComponent("controller1");
    AComponent controller2 = manager.
      getComponent("controller2");
    AComponent monitor = manager.getComponent("monitoring");
    AComponent actComponent = manager.
      getComponent("actuator");
    IProvidedPort inspect1 = controller1.
      getProvidedPort("inspect");
    IProvidedPort inpsect2 = controller2.
      getProvidedPort("inspect");
    IRequiredPort inspect = monitor.
      getRequiredPort("inspect");

    IRequiredPort actuator1 = controller1.
      getRequiredPort("actuator");
    IRequiredPort actuator2 = controller2.
      getRequiredPort("actuator");
    IProvidedPort actuator = actComponent.
      getProvidedPort("actuator");

    if(inpsect2.getCardinality() == 0)
```

```
        return;

      inpsect2.disconnect();
      actuator2.disconnect();

      manager.connect(actuator1, actuator);
      manager.connect(inspect, inspect1);

      event(new BehaviourChangedEvent(Behaviour.BEHAVIOUR_1));
    }
  }
```

This concludes the description of the behaviour adjustment pattern. It is one of the most promising patterns for leveraging reconfigurations in creating physiological computing applications. In this section, we described the behaviour change pattern in an abstract and simplified form, showing a system altering between two abstract behaviours consisting each of only one component. Nevertheless, while real applications will exhibit more complexity with respect to system structure and number of behaviours to select from, the core of behaviour changing through reconfiguration is covered by the behaviour adjustment pattern.

Consequences

Applying the behaviour adjustment pattern has two main consequences for the structure of the system, namely that the application logic becomes separated along two lines. The first line separates the operational logic from meta-reasoning logic. The operational logic remains in operational components, while the largest part of the meta-reasoning gets extracted into monitoring components serving the sole purpose of observing the behaviour of the system and initiating corresponding corrective actions. The second line of separation divides the operational logic required for each context into different components and configurations. Instead of covering all contexts in one configuration and using only one set of operational components, it is possible to specialise the behaviour of operational components to specific contexts, and to tailor configurations to these contexts.

5.5 Related work

Ramirez and Cheng [100] describe design patterns for distributed adaptive systems, which are similar to the reconfiguration usage patterns discussed here. They present a full set of design patterns for distributed adaptive systems that were extracted from existing middlewares, frameworks and applications. The patterns hence provide design knowledge that is transferable across frameworks and middlewares. Ramirez and Cheng do not focus on providing a software development methodology encompassing formal verification, as this thesis does. Furthermore, while each pattern includes LTL specifications of desired behaviour, the pattern descriptions do not provide a proof template of how the desired behaviour can be attained – in contrast to the patterns given in this chapter.

Previous work from Zhang and Cheng [125] proposed a specification-driven approach to the design and verification of adaptive software. The methodology that underlies the approach consists of six sequential steps: (1) specification of global invariants, (2) enumeration of operational contexts requiring different system behaviours, (3) specification of local properties for each operational context, (4) creation of a state-based model for each operational context and verification against local properties and global invariants, (5) modelling of possible transitions between operational contexts, creating transitional models and verifying that the adaptation models satisfy global invariants, and finally (6) the creation of prototypes from the models. While this approach is richer in terms of specification kinds, it does not provide modular means for specification – in contrast to the assume-guarantee framework presented in Chapter 4. Furthermore, the approach of Zhang and Cheng does not provide any guidance for embedding the six-step process into software development methodologies.

An approach that provides guidance in terms of a software development methodology for adaptive systems is the DiVA methodology [45], which focuses on the requirements engineering and modelling aspects of adaptive features of software systems. The DiVA approach is primarily a model-driven, aspect-oriented approach in which an adaptive system is captured as a *base model* with *adaptation models* applied. The base model constitutes the least common denominator that the system consists of throughout all adaptation contexts, while the adaptation models represent aspects that are added in specific contexts only. While the base model is mapped to object-oriented code, the adaptation models are mapped to aspects that are interwoven with

the object-oriented code at runtime. The focus on aspect-orientation constitutes one major distinction between the DiVA methodology and the approach presented in this thesis. It can be argued that the aspect-oriented approach may fail if the changes to the software system are so drastic that the base model becomes virtually empty – a very extreme situation, but one that may arise in physiological computing if all sensor and actuator devices are considered volatile, and the control logic system is only instantiated if devices are available. One of the strongest similarities between the approach presented in this thesis and the DiVA methodology is that both propose an iterative and incremental approach to the development of adaptive systems. The DiVA methodology iterates the phases of requirements engineering, adaptation modelling, runtime architecture modelling and deployment. In the approach presented in this thesis, the three phases of informal modelling, formalisation and abstraction as well as formal verification and implementation are iterated. The DiVA approach agrees with the findings in this thesis that an iterative and incremental approach is a good match for the construction of adaptive systems (physiological or not).

Multi-agent-system development methodologies can be seen as another, different approach to the development of adaptive systems. In these development approaches (for an overview, see [69]), context modelling has a limited importance compared to the modelling of agent roles and interactions. To shed some light on these differences, we use the Gaia methodology [123] in the following as one representative multi-agent based methodology. In the Gaia methodology, a software system is understood and modelled as an artificial society of cooperating entities (agents) that can fulfil different responsibilities and rights (roles) and interact with different communication partners at different times. In this view, understanding and modelling the agent roles, their associated capabilities and constraints, as well as the intended communication structures of roles, becomes the most significant activity. Other activities such as context understanding and modelling are only means to this end. In contrast to the approach presented in this chapter, the focus on roles entails a more normative and defensive view on the possible interactions of entities – one in which each agent can only be made accountable for its behaviour with respect to its current role. While creating the case studies (cf. Chapter 6), we discovered that this view on adaptive systems proved to be too cumbersome in the development of physiological computing systems; while using concurrent software entities (active components) that change their interaction patterns helps in structuring the design of physi-

ological computing systems, the focus on potential role conflicts and role changes of agents that multi-agent based methodologies provide seemed superfluous. Instead, we found that physiological computing applications are easier to design and understand as a set of cooperating active components whose configuration may be altered at runtime.

The ubiquitous systems research community has also brought up a methodology for the creation of distributed adaptive systems that follows similar goals as the methodology presented in this thesis, namely the MUSIC approach [60, 49], which focuses heavily on capturing the adaptation contexts and the variability that is required in a component-based realisation to satisfy the system requirements. For this, the MUSIC approach introduces a UML profile for variability models – prominently capturing the necessary variability in the system in terms of required component variants for a component supertype using inheritance – and domain models – describing the relations between application contexts and component QoS properties. These models are deduced initially from informal requirements structured as application scenarios: Scenarios are used to distill operational contexts requiring different modes of operation. Contexts can be split up in sub-contexts to create a hierarchy of contexts, each corresponding to a main operational mode for the top-level contexts, and sub-modes for the sub-contexts. Contexts are used to guide the application architecture design (consisting of a component configuration) as well as the adaptation rule design that amounts to the change of a component configuration to another to accommodate a change of context. In the MUSIC approach, the canonical UML component diagrams for system architecture description are complemented by variability models using a specific UML profile. The variability model describes (1) available choices for certain component types using inheritance, (2) the properties of each possible choice using UML tagged values, and optionally (3) an utility function that is associated with each application that helps in guiding the component selection process. The domain model helps in this process by providing dependencies between contexts and (QoS) properties. In a next step, the MUSIC approach uses ontology-based tools to generate code from variability and domain model. The MUSIC approach is hence also a model-driven approach.

In comparison with the methodology for the creation of physiological computing applications presented in this Chapter, it is apparent that the MUSIC approach focuses more on the early elicitation and design steps in which the application modes and contexts are discovered than on the latter

phases in which the reconfiguration rules are designed, specified and implemented. Furthermore, the MUSIC approach provides a development process of its own instead of embedding it into a well-tested software development methodology. The issues and forces that may emerge from a combination of the MUSIC process with existing software development methodologies remain to be discovered. Also, the focus on a model-driven approach that is featured in the MUSIC approach is a two-edged sword. While it provides some support for the creation of adaptive systems, it constrains the developers heavily during the modelling phase and their general approach to the creation of adaptive systems; furthermore, the overhead entailed by a model-driven approach may not pay off in every adaptive application. Finally, the MUSIC methodology does not provide an embedding for formal verification of adaptive systems.

5.6 Conclusion

In this chapter, we have presented a methodology for the development of physiological computing systems that makes use of the formal verification framework and the component framework presented earlier (Chapter 4 and 3, respectively).

First, this chapter has shown how formal verification can be combined with agile methodologies; more specifically, with Scrum. In particular, it has been shown that formal verification and Scrum work well together in the context of physiological computing system development. In fact, as it is of paramount importance to keep physiological computing projects agile, it is necessary to re-think and re-shape the approach to formal verification. In particular, is is necessary to relax the constraints of early verification (i.e. requiring that verification is performed before implementation), and to allow for parallelisation of implementation and verification tasks. How far the early verification force is a driving factor for the organisation of work depends heavily on the project.

Next, we have proposed a metamodel and corresponding graphical notation for the design of component-based architectures. The proposed diagram differs from existing UML diagram types for the design of component-based software systems in that it a) is geared towards the component model supported by the REFLECT framework and b) specifically targets the rapid sketching of component-based software architectures. The latter is achieved

by supporting implicit definition of types through specification of instances. Also, we have provided a minimalistic notational frame for capturing ECA-style reconfiguration rules that has proved itself adequate in the examples.

The examples that are provided to demonstrate each single step of the methodology are indeed recurring patterns: discovery of device capabilities and changing behaviour on a strategic level are usage scenarios for reconfigurations that are constantly emerging in physiological computing applications. For these two patterns, forces, consequences, informal design, abstraction and verification, and implementation were discussed. We have shown how the information from informal modelling influences verification and implementation, and have provided proofs for interesting properties of both patterns that can be modified and adjusted for specific applications of the patterns.

Finally, the discussion of related work has shown the particular strengths of the presented approach. In comparison to the literature, our methodology shares the same common traits, but improves the state of the art in the context of physiological computing.

Chapter 6

Case studies

So far, this thesis has presented only relatively small application examples for its results. This chapter therefore provides more extensive case studies that demonstrate and discuss the consequences and benefits of the REFLECT framework and the real-time assume-guarantee framework.

The REFLECT framework case study revolves primarily around the REFLECT automotive demonstrator, which is the end-result of the REFLECT demonstration activities. The demonstrator software runs on an automotive setup provided by Ferrari S.P.A. as detailed in Section 6.1. In this case study, we primarily discuss how the features of the REFLECT framework were used. The REFLECT framework was developed during the REFLECT project, a "Specific Targeted Research Project" (STREP) that started in 2008 and ended 2011.

The aim of the REFLECT project has been to push the frontier in physiological computing along two major lines:

1. researching means for supporting software development activities involved in the creation of physiological computing systems through software frameworks and practical tools.

2. researching in the creation of closed loops in three themes: emotion, cognitive engagement, and physical comfort. In each theme, the research have been focused on all aspects involved in the creation of the loop: from establishing a general psychological and physiological concept of a closed loop control, down to the selection of the correct sensors, actuators, and the linking of the physical hardware to a software control system.

This thesis presents results from the first research line; the second research line, i.e. the creation of concrete physiological computing applications, provided valuable feedback guiding the development and realisation of the framework and tools produced.

The real-time contract-based verification framework is demonstrated by the adaptive advertising scenario in Section 6.2 that introduces an interactive advertisement display capable of interacting with its viewers, and providing auto-active content as well. In this case study, the real-time contract-based verification framework is used to prove the responsiveness of the advertising application as well as the continued animation of displayed content (see Section 6.2 for details).

After presenting the two case studies and discussing experiences and results for each case study separately, this chapter concludes in Section 6.3.

6.1 Automotive demonstrator case study

The automotive demonstrator integrated the emotional, cognitive, and comfort loops into a vehicular demonstrator, and aims at applying the principles of physiological computing to the automotive domain to the advantage of the driver. By measuring available physiological signals from the driver, and combining this information with information available from the car, adapting the driver's environment to his current context becomes feasible.

The automotive demonstrator case study was the main demonstration activity of the REFLECT project. Therefore, the software that was created for that demonstrator made significant use of the REFLECT framework. More important however, the application of the REFLECT framework gave valuable feedback on the applicability and practicability of the framework to the creation of a larger software system over a prolonged development effort, as the automotive demonstrator was created over a period of roughly 14 months.

6.1.1 Demonstrator concept

The automotive demonstrator is intended to show how the results of the REFLECT project in the domain of physiological computing can be applied to improve human-machine interface and ultimately provide a more pleasurable and safer driving experience. The major goal in the creation of the demon-

strator was hence to enhance the driving experience through the implementation of real-time feedback loops operating on different levels of the driver's psychophisology, according to the three physiological research streams of the REFLECT project (emotion, cognitive, physical comfort), and the application domain of the demonstrator (driving). The four feedback loops that form the automotive demonstrator are operating as follows.

The **Effort loop** is concerned with determining the mental workload of the driver and adjusting the driver's tasks accordingly. In the automotive demonstrator, the effort loop should reduce the number of secondary tasks the driver is asked to perform, i.e. in the automotive context, incoming mobile phone calls must be suppressed, and music currently playing is faded out to allow the driver to focus on the (currently demanding) primary task of driving. The indicators for mental workload that were usable in the automotive setting were determined to be heart rate and heart rate variability. High mental workload translates to a slightly increased heart rate, and a decrease in heart rate variability (i.e. the heart beats become more regular under high mental workload) [88].

The **Emotion loop** is involved in managing the emotion of the driver according to a driver-selected goal. The goal state can be selected from the following three emotional goal states (for simplicity of the user interface): energetic, neutral, and relaxed. The emotional loop continuously monitors the current emotional state of the driver, and uses music from the driver's own music database to guide him to the desired goal state. For monitoring the driver's emotion, skin conductance level and skin temperature level features were selected that must be implemented in the demonstrator. Furthermore, the emotional loop requires the creation of a music database that stores the song effects on the driver, and a music player that selects songs according to the recorded song effects and the selected target mood. The concept of emotional control through personalised music, and the filtering processes required to detect the effects on the emotional state of the user are described by Janssen et al. in [75].

The **Comfort loop** tracks the physical comfort the user is experiencing, and is able to adjust the driver seat moderately to improve on the sitting posture and stability of the driver. Physical comfort of the driver is tracked by the pressure pattern the driver is exerting on the seat, as well as by measuring the accelerations (longitudinal, lateral, and vertical) that the driver is experiencing through either fast driving or driving on a rough road [28, 27]. Improving the sitting posture is achieved by inflating or de-

flating cushions that are available in the driver seat. Inflating the cushions result in more stabilised sitting, while deflated cushions allow more freedom to move. Additionally, the comfort loop monitors the driver's alertness to the road and should give indications of the user's drowsiness. Drowsiness is detected through eye blink duration, as blink durations increase significantly with driver drowsiness [39].

The **Driving loop** gives feedback on the driving style currently expressed by the driver. It deduces the driving style from monitored car telemetry data such as lateral and longitudinal accelerations, speed, as well as actuations of pedals, the steering wheel, and gear shifts. In order to give implicit feedback to the driver, the driving loop should use continuous, moderately changing "ambient" visualisations in the vicinity of the driver's primary viewing direction. Using continuous, unobtrusive visualisations allow to improve the driver's situational awareness once he has learned the meaning of the driving loop feedback. The increased situational awareness is intended to allow the driver to enhance on his driving style accordingly. In order to achieve proximity to the driver's primary viewing direction, the driving loop must use a head-up display allowing projecting information on the car's windscreen. Furthermore, the driver should be given a choice to turn off the system or to select from a set of desired enhanced situational awareness.

In the automotive demonstrator setup, we decided to refrain from using a head-up display due to the involved costs of installation. Furthermore, we focused on a simple version of the driving loop that is easy to demonstrate, and therefore chose a feedback loop that tries to keep the driving style smooth and calm by giving negative feedback on risky driving.

6.1.2 Hardware setup

The sensing and actuation requirements of the four loops that constitute the automotive demonstrator are summarised in Table 6.1. These hardware requirements need to be addressed with a proper physical setup; this setup is described in this section. The physical setup shown in Figure 6.2 provides all required sensing and actuation facilities. Figure 6.1 shows the external and cockpit view of the Ferrari California used and the demonstrator as installed in the car.

The physical setup of the automotive demonstrator in its core consists of three ultra-mobile PCs (UMPCs) installed in a Ferrari California connected to several sensors and actuators available in the car as shown in Fig-

Loop	Sensing requirements	Actuation requirements
Cognitive	Electrocardiogram	Control over phone and media center
Emotion	Skin conductance, skin temperature	Sound / music output
Comfort	Seat pressure, accelerations on the driver, eye blinking	Seat cushion inflation control, driver warnings
Driving	Car telemetry	Ambient visualisation

Table 6.1: Sensing and actuation requirements of the automotive demonstrator

(a) External view (b) Cockpit view

Figure 6.1: Automotive demonstrator deployed

ure 6.2. The three UMPCs (1) communicate via wi-fi, and have various sensory equipment connected via USB: UMPC number three is connected to the car's controller-area network bus (CAN bus) of the Ferrari California (2) via a Kvaser USB can II (3), a CAN-bus to USB adapter. The connection to the CAN-bus allows to read speed, longitudinal and lateral acceleration, and steering wheel actuations (among others) from the car sensors. Furthermore, the sensory setup consists of a polar heart rate monitor (4) sending its readings via a custom protocol to a SparkFun Polar Heart Rate Monitor Interface (5) that in turn is connected via USB to the third UMPC, and skin conductance and skin temperature sensors (6) linked to a Mind Media Nexus 10 device (7) that is paired via Bluetooth to the second UMPC. Additionally, the system features a pressure map (8) that is placed on the

driver seat (9) which also features two pairs of inflatable cushions for seat adjustment on lateral and lumbar positions. Both the pressure map and the seat are connected to a custom-build controller (10) that allows to read out comfort indicators and to manipulate the seat cushions via a USB interface. As additional sensor, the system features a Logitech C905 web camera (11) connected to a laptop (12) running the Fraunhofer SHORE™ face recognition system, and communicating via wi-fi with the three UMPCs. Finally, the demonstrator features Logitech Z7205 loudspeakers (13) that are used to provide a satisfactory sound experience in the car cockpit.

Figure 6.2: Automotive demonstrator setup

As the hardware setup consist of three UMPCs each offering a touch screen interface, a solid quantity of information can be given back to the driver and an accompanying system tester. In the automotive demonstrator, we decided to use the UMPC that is the closest to the driver, i.e. the leftmost, as user interface as in a final product. Figure 6.3 shows the displayed content

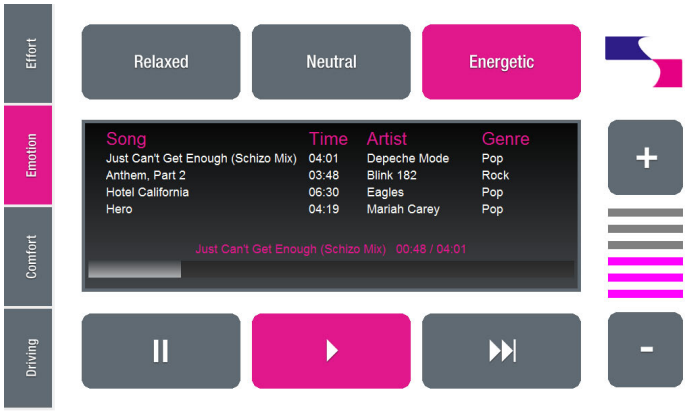


Figure 6.3: Demonstrator user interface view, emotion loop

on the first UMPC during the activation of the emotion loop. As the emotion loop's concept is to influence mood with music, the user interface shows three target mood selection button on its top, a volume control on the right, and a progress panel showing the currently playing song, as well as the next four songs to play. The lower third of the display hosts a set of music player controls, allowing to start, stop, and skip songs.

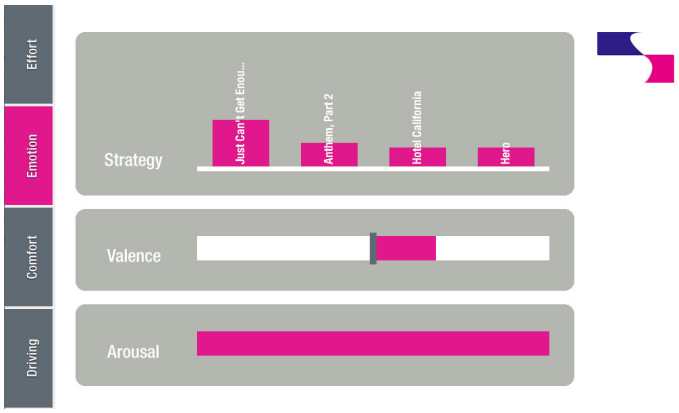


Figure 6.4: Demonstrator details view, emotion loop

The middle UMPC is devoted to showing the internal workings of the system and explaining the rationale behind its actions. On the one hand, the display shows what the currently inspected loop assumes about the driver's

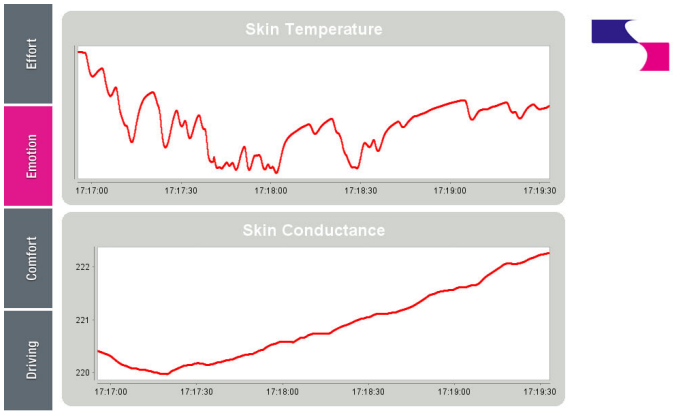


Figure 6.5: Demonstrator raw data view, emotion loop

state in terms of effort, emotion, comfort, and driving. On the other hand, the display also shows an explanation for its current strategy, i.e. the currently executed action, and the next planned actions. Figure 6.4 shows the details view of the emotional loop. In the top panel labelled Strategy, the panel shows the score of the next songs that will be played, clearly showing a decreasing score from the immediate next song to play (which is rated best), to the fourth next song to play (rated worst of the four, but still better than any of the other songs). The lower panels display the currently assumed emotional state in terms of emotional valence (positive or negative) and arousal (low or high) shown on a linear scale.

The rightmost UMPC finally shows the raw data input that the physiological computing system is getting from its sensors. This view is mainly devoted to debugging the sensor connectivity and inspecting the quality of the raw input data. With the help of the raw data view, it is possible to inspect the quality of the physiological data analysis subsystem of the automotive demonstrator on-line, and to build up confidence in the correct operation of the physiological analysis. As shown in Figure 6.5, the emotional loop raw data view displays the input skin temperature and skin conductance data of the last ten minutes, so that e.g. the effect of signal perturbations (also coined artefacts) on the physiological inference system that maps skin conductance and skin temperature to emotional valence and arousal can be reviewed on-line.

Due to the use of the three UMPC displays, the automotive demonstrator

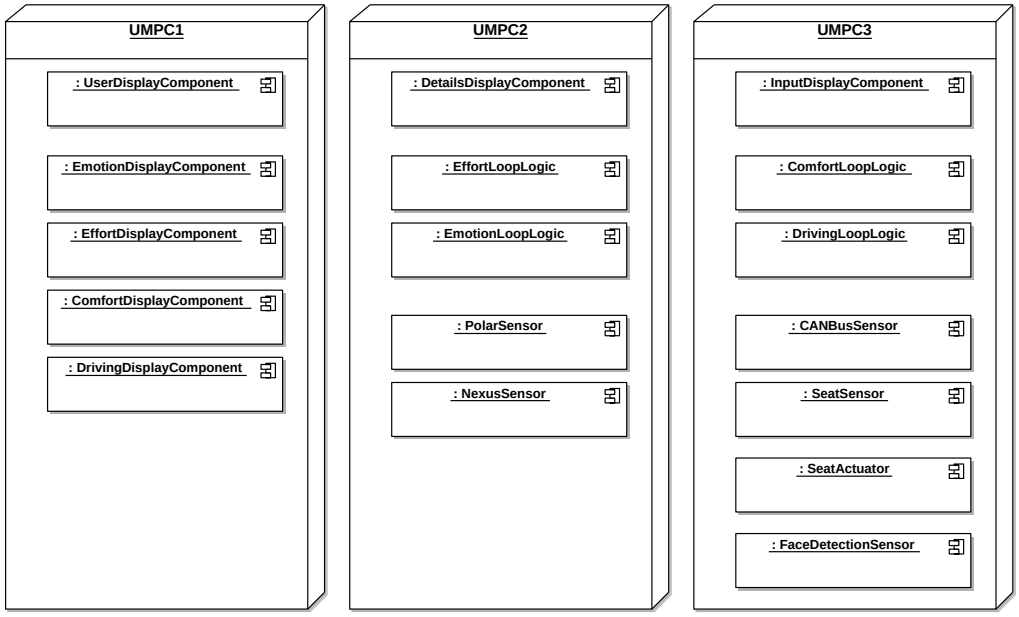


Figure 6.6: Component distribution

software system must operate robustly and distributed. In the next section, the architecture of the automotive demonstrator software is detailed.

6.1.3 Software architecture

As detailed in the previous section, the automotive demonstrator software system must operate in a distributed fashion due to the information displayed to the driver and co-driver of the car. However, the system also needs to operate in a distributed fashion due to the limited available computing resources the single UMPCs provide. Figure 6.6 shows a birds-eye view on the distribution of the automotive demonstrator over the three UMPCs using a UML deployment diagram. Each UMPC contains its respective display component that contains the user interface and features ports for communication with the business logic components. Looking at the emotion loop specifically, the loop is distributed in the following way: The first UMPC contains the emotion display component that contains the music player user interface. The component plugs into the the user display component, and re-

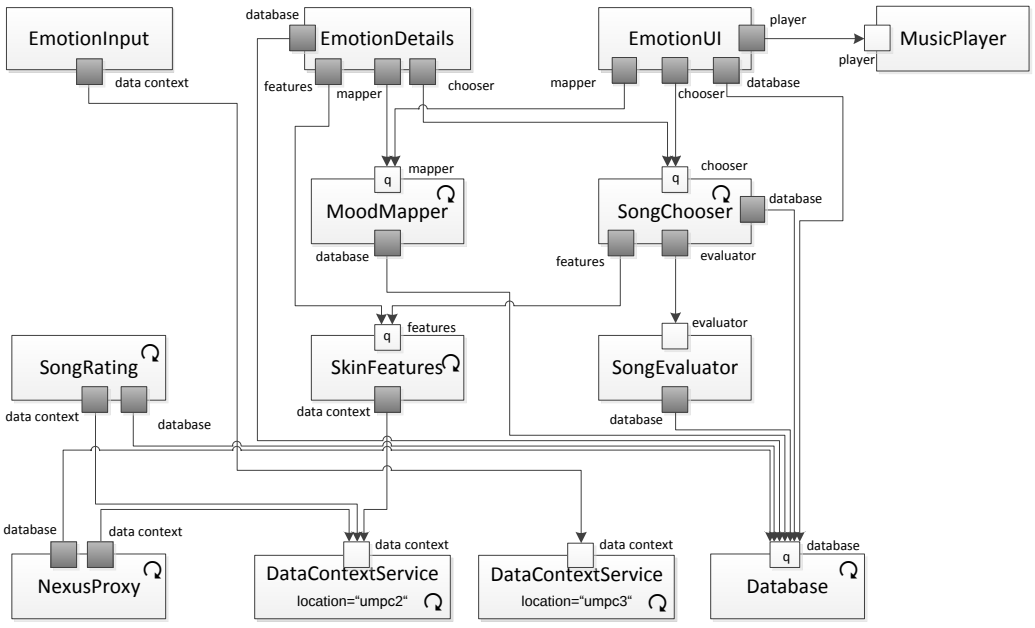


Figure 6.7: Structure of the emotion loop

lays the user commands to the emotion loop logic component that is located on the second UMPC. The emotion loop logic component is only a proxy for several components that together implement the business logic of the emotion loop. The second UMPC also hosts the Nexus sensor component that reads data from the Nexus 10 device, and pushes it into the system (using the data context service, cf. Section 3.3.11). On the third UMPC finally, the input display component taps into the data context services to display the skin conductance and skin temperature that the emotion loop uses as raw physiological input data.

Figure 6.7 shows the system configuration diagram for the emotional loop as implemented on the REFLECT framework, which we describe by roughly following the flow of physiological data through the system. Note that in Figure 6.7, provided ports that offer an input queue as means for asynchronous communication are additionally tagged with a "q". We start with the NexusProxy component on the bottom left, which reads physiological data from the Nexus 10 device, and pushes it into the data context service component that manages the data in its data channels. The data channels are mirrored

to a second instance of the data context service residing on the third UMPC. This data context service instance is used by the EmotionInput component to display the raw physiological input data. The SkinFeatures component is concerned with extracting the features required for extracting information about emotional valence and arousal (i.e. relative skin conductance level and skin temperature, averaged over one minute and normalised over a one hour baseline). The skin features are used by the EmotionDetails component to display the emotional valence and arousal – they are mapped to valence and arousal by using the MoodMapper component. Additionally, the EmotionDetails component needs to display the next songs to play with their score in its Strategy panel. To achieve this, the EmotionDetails component queries the SongChooser component that manages the list of songs to play and their scores. The SongChooser component needs the following information to compute the score of all songs: the current state of the user (in terms of skin temperature and skin conductance), and the estimated change in skin temperature and skin conductance – and transitively, in emotional state – a given song may induce. For this, the SongChooser uses the service provided by the SongEvaluator. The SongEvaluator rates each song by consulting a database featuring a set of recorded songs effects. With the help of machine learning techniques (kernel-based density estimation, see [75] for details), an estimated song effect is condensed from the individual song effects.

The EmotionUI makes use of the SongChooser services to retrieve the song list, and to publish new user goal states when the user decides to change his target mood. The current song to play is submitted to the music player, which also realises the music player command for starting, pausing, and skipping songs.

Finally, as the last component to discuss, the SongRating component populates the database with new song effects as songs are finished. For this, the SongRating component listens to song start and end events emitted by the MusicPlayer components: on song start, the SongRating component records the current skin features, and on song end, the component computes the difference between start and end feature values, and stores them into the database as newly recorded song effect for use by the SongEvaluator.

6.1.4 Experience and results

The automotive demonstrator served primarily as a case study of the REFLECT framework described in Chapter 3. Work on the demonstrator

showed that the REFLECT framework was suitable for the creation of a large, real-world prototype. Especially, the REFLECT framework proved itself useful in the following respects:

- *Simplification of assembly with service-style rules.* As the number of connectors grows significantly, a component-style, explicit connection of ports would lead to hard to read and hard to maintain code in the bundle container. By using service-style wildcard rules, the bundle container code can be simplified significantly.

- *Robustness vs. network failures and node shutdown.* The component life-cycle and the binding rules allow to create systems that are robust again network failures and node shutdown with relative ease. With the automotive demonstrator, a system that responds gracefully to single UMPC shutdowns and re-establishes connections as soon as the required components become available.

- *Capture/replay and confidence establishing inspection.* The means for system inspection that are offered by the framework's management console allowed to inspect and interact with the system quickly. Especially, the inspection means allowed to develop confidence in the proper operation of the system even in the relatively new and complex setup.

- *Thread confinement through event bus.* A relatively surprising outcome of the automotive demonstrator creator was that the event-based communication facilities became adopted very heavily in several loops – in the driving and comfort loop especially. By executing the loop's business logic in the event handling method, a thread-confined subsystem was created within the loops that allowed to ignore inter-thread synchronization issues; fortunately, the event processing latency that is entailed by writing a majority if business code in the event handler was tolerable for the overall system performance.

Altogether, implementing the automotive demonstrator on top of the REEFLECT framework did show that a component-based framework-based approach to the creation of physiological computing systems is promising for simplifying the development efforts at hand.

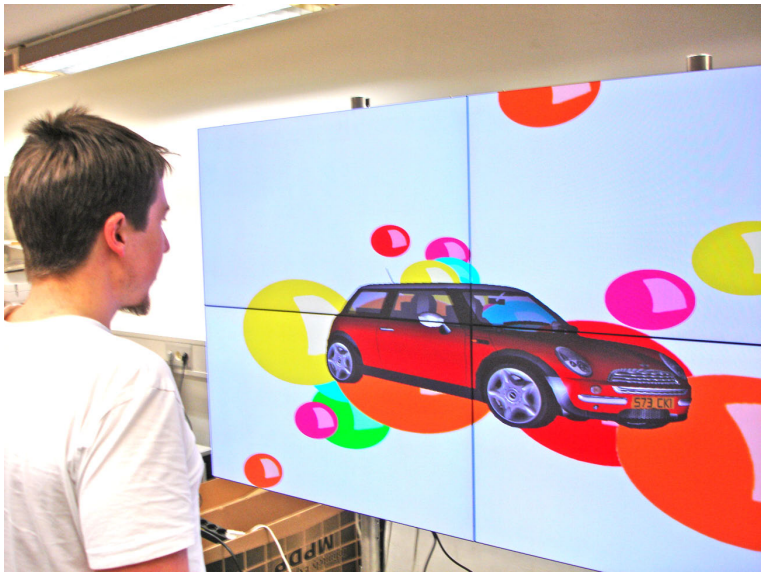


Figure 6.8: Adaptive advertising system interacting with viewer

6.2 Adaptive advertising

The adaptive advertising case study presented in this section showcases the assume-guarantee framework for reconfigurable systems on a larger system.

6.2.1 Case study concept

The general idea of adaptive advertising is to adapt a displayed advertisement to the current situation in front of it – whether there are several people just passing by, a small group of persons watching the ad carefully, or just one person in front of it waiting for someone else [29]. The system uses cameras to observe the passers-by, and enables the ad to react to e.g. the number of passers-by watching the advertisement, to discover their interest in the advertisement by analysing their gaze direction and exposure time, or to enable gesture-based interactions with a passer-by becoming interested in the ad.

A simple scenario within the adaptive advertising setting is an adaptive car advertisement, reacting to gestures of users in front of the display: By moving around the display, pointing at items or looking at them, the viewers influence the contents of the ad. Figure 6.8 shows an example lab installation

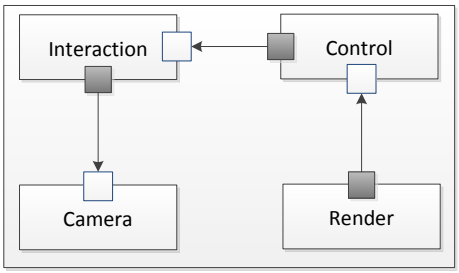


Figure 6.9: Initial adaptive advertising architecture

of an adaptive car advertising.

In the following, we consider the application of the assume-guarantee framework to the specification and verification of interesting system properties of adaptive advertising installations. For this, we first need to define a desired global system property. In fact, there are two interesting global system properties that must be satisfied. Expressed as contracts, the two properties can be specified – at first, informally – as guarantees G1 and G2 as follows: (G1) Being an interactive ad, the system should react to a user in front of the display. Furthermore, (G2) The content displayed must change at least every ten seconds: an advertising campaign using a large-scale display should not waste its capabilities by showing static content.

A first realisation of the system consists of four components (see Figure 6.9): a camera component for image acquisition, an interaction detection component detecting gestures, a control component operating the car ad, and a rendering component displaying the ad content. This simple realisation is problematic, however: It cannot provide the guarantee that the displayed content changes every ten seconds, as the car ad will remain static if no one is interacting with it.

We therefore need to introduce a more complex behaviour. In fact, we can apply the behaviour monitoring pattern described in Section 5.4.2. By introducing a monitoring component observing whether someone is interacting with the ad, the system can be made aware that it is about to violate its contract. Then, a behaviour changing reconfiguration can be triggered which alters the system such that it shows auto-active content generated by a presentation component, e.g. an advertising movie or predefined animation sequences (see Figure 6.10; components that are present, but disconnected are shown in light gray). In a way, introducing a monitoring component

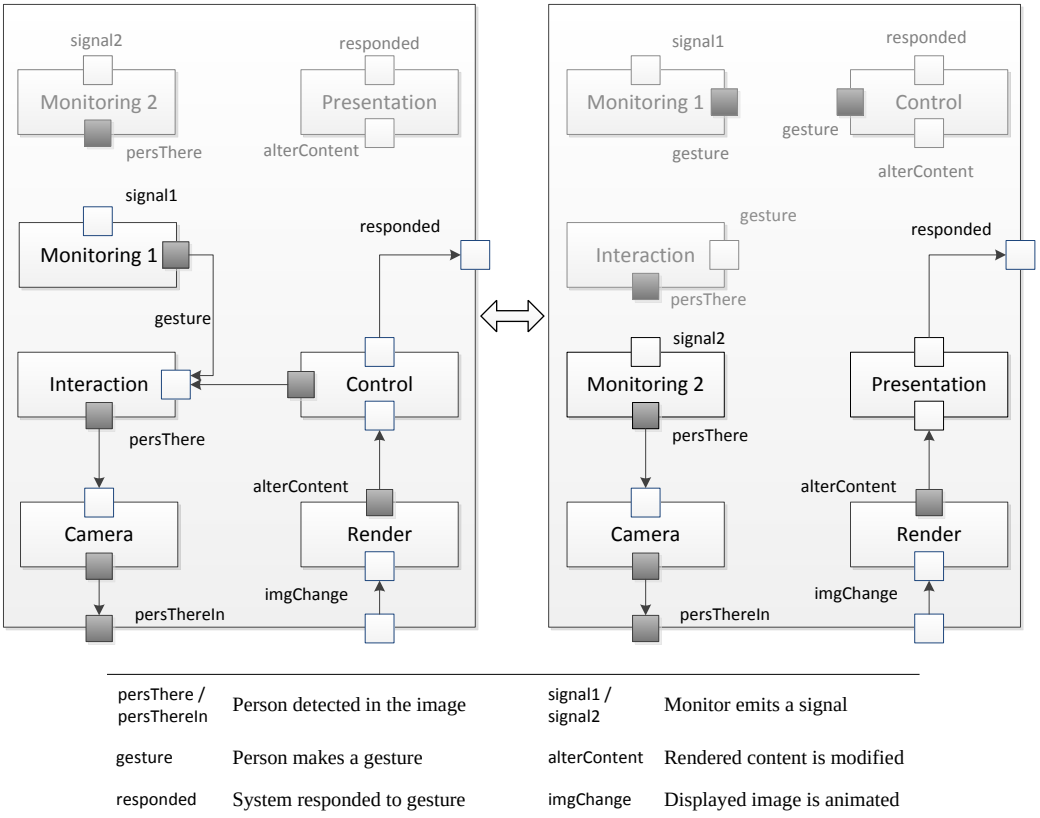


Figure 6.10: Enhanced adaptive advertising architecture using reconfigurations

allows a partial solution to assume that the environment exhibits certain features (e.g. always have someone interacting with the ad) that it does not exhibit in the general case. Note that the second system (Figure 6.10, right) also needs a monitoring component, as it again does not satisfy (G1): The second system provides interactive content to its viewers, and therefore must be changed as soon as a person is in front of the display, interacting with it.

6.2.2 Formalisation and verification

Now, we describe how the adaptive advertising case study can be specified by reMITL-formulas and how the global system contract can be verified using the real-time logic framework described in Chapter 4. For the formalisation

of the adaptive advertising case study, we first define the system signature $\Sigma_{sys} = ((R^{Ext}, P^{Ext}), (R^{Int}, P^{Int}), (C^i, C^d))$:

$$
\begin{aligned}
R^{Ext} &= \{persThereIn\},\\
P^{Ext} &= \{imgChange, responded\},\\
R^{Int} &= \{Cam.persThereIn, Intr.persThere, Mon1.gesture, Mon2.persThere, Cont.gesture,\\
&\quad Rend.alterContent\},\\
P^{Int} &= \{Cam.persThere, Intr.gesture, Mon1.signal1, Mon2.signal2, Pres.responded,\\
&\quad Pres.alterContent, Cont.responded, Cont.alterContent, Rend.imgChange\},\\
C^i &= \{(Mon1.gesture, Intr.gesture), (Cont.gesture, Intr.gesture), (Intr.persThere, Cam.persThere),\\
&\quad (Rend.alterContent, Cont.alterContent), (Mon2.persThere, Cam.persThere),\\
&\quad (Rend.alterContent, Pres.alterContent)\},\\
C^d &= \{(persThereIn, Cam.persThereIn), (imgChange, Rend.imgChange), (responded, Pres.responded)\}
\end{aligned}
$$

The reconfiguration rules are given by formulas where C_1 is a conjunction of connection constraints defining "the system is in configuration 1" (shown in Figure 6.10 on the left), and C_2 denotes similarly "the system is in configuration 2" (shown in Figure 6.10 on the right).

$$
\begin{aligned}
C_1 = \; & persThereIn \sim Cam.persThereIn \wedge Intr.persThere \sim Cam.persThere\\
& \wedge Mon1.gesture \sim Intr.gesture \wedge Cont.gesture \sim Intr.gesture\\
& \wedge Cont.responded \sim responded \wedge Cont.alterContent \sim Rend.alterContent\\
& \wedge Rend.imgChange \sim imgChange\\
C_2 = \; & persThereIn \sim Cam.persThereIn \wedge Cam.persThere \sim Mon2.persThere\\
& \wedge Pres.responded \sim responded \wedge Pres.alterContent \sim Rend.alterContent\\
& \wedge Rend.imgChange \sim imgChange
\end{aligned}
$$

The reconfiguration rules that we specify for the system are the following:

$$\mathcal{A} = \Box(C_1 \vee C_2) \wedge \Box(C_1 \wedge Mon1.signal1 \rightarrow \Diamond_{[0,0.2]} C_2) \wedge \Box(C_2 \wedge Mon2.signal2 \rightarrow \Diamond_{[0,0.2]} C_1)$$

Next, the single components the system consists of are specified by the following contracts (assumptions are $\top$ in each case):

$$
\begin{aligned}
G_{Camera} &= \Box(Cam.persThere \leftrightarrow Cam.persThereIn)\\
G_{Interaction} &= \Box(Intr.gesture \leftrightarrow Intr.persThere)\\
G_{Control} &= \Box(Cont.responded \leftrightarrow Cont.gesture \wedge Cont.alterContent \leftrightarrow Cont.gesture)\\
G_{Render} &= \Box(Rend.imgChange \leftrightarrow Rend.alterContent)\\
G_{Presentation} &= \Box(Pres.alterContent \wedge \neg Pres.responded)\\
G_{Mon1} &= \Box(\Box_{[0,3]} \neg Mon1.gesture \leftrightarrow \Diamond_{[3,3.5]} Mon1.signal1)\\
G_{Mon2} &= \Box(\Box_{[0,0.5]} Mon2.persThere \leftrightarrow \Diamond_{[0.5,0.7]} Mon2.signal2)
\end{aligned}
$$

Note that the guarantees of the monitoring components together with the reconfiguration rules $\mathcal{A}$ express changes in configurations: The first configuration C_1 (reacting to viewers) will be reconfigured in 3.7 seconds to the second configuration C_2 (showing animated content) if the scene in front of the display stays empty for three seconds. This property can be derived from the guarantee G_{Mon1} together with the connection properties of C_1. Similarly, guarantee G_{Mon2} together with $\mathcal{A}$ ensures that the system will be changed within 0.9 seconds if a person is seen in front of the display for at least 0.5 seconds.

The whole system $Sys = (\Sigma_{sys}, (\llbracket A_{sys} \rrbracket, \llbracket G_{sys} \rrbracket), \llbracket \mathcal{A} \rrbracket, \mathcal{C}_{sys})$ is a composite component where $\mathcal{C}_{sys}$ is the set of all (correct) components the system consists of, and $(\llbracket A_{sys} \rrbracket, \llbracket G_{sys} \rrbracket)$ is the global system contract which can be formulated as follows:

$$(A_{sys}, G_{sys}) = (\top, (\Box\Diamond_{[0,10]} \mathit{imgChange}) \wedge \Box(\Box_{[0,0.5]} \mathit{persThere} \rightarrow \Diamond_{[0,1]} \mathit{responded}))$$

In order to prove that Sys is a correct composite component we must prove the conditions (1)-(6) given in Def. 35. We assume that all components in $\mathcal{C}_{sys}$ are correct therefore condition (1) is satisfied. Conditions (2) and (3) are trivially satisfied as we have chosen disjoint naming of components and ports. Condition (5) is satisfied as $\mathcal{A}$ does not link ports of the same component together, and makes no assumptions on the behaviour of each single component. Finally, condition (6) can be seen as follows: The only required port is *persThereIn* which is always connected to the camera component *Cam* which in turn does not make any assumptions on the valuation of this port. Finally, it remains to show condition (4): The inferred contract guaranteed by the composition of all contracts of the single components must be a refinement of the system contract:

$$(\llbracket A_{sys} \rrbracket, \llbracket G_{sys} \rrbracket) \uparrow^{\Sigma_{sys}} \succeq \|^{\llbracket \mathcal{A} \rrbracket}_{c \in \mathcal{C}_{sys}} (\llbracket A_c \rrbracket, \llbracket G_c \rrbracket).$$

First, it can be easily shown that the parallel composition on the left hand side yields the contract $(\llbracket \top \rrbracket, \llbracket \bigwedge_{c \in \mathcal{C}_{sys}} G_c \wedge \mathcal{A} \rrbracket)$. Then, by Cor. 2, we show the refinement

$$(\llbracket A_{sys} \rrbracket, \llbracket G_{sys} \rrbracket) \uparrow^{\Sigma_{sys}} \succeq (\llbracket \top \rrbracket, \llbracket \bigwedge_{c \in \mathcal{C}_{sys}} G_c \wedge \mathcal{A} \rrbracket)$$

by entailment of reMITL-formulas, i.e.

$$\vDash_{\Sigma_{sys}} (\bigwedge_{c \in \mathcal{C}_{sys}} G_c \wedge \mathcal{A}) \rightarrow G_{sys}.$$

For the sake of brevity, we will focus on discussing the system guarantee $\Box\Diamond_{10} imgChange$. To prove this guarantee we have to show

$$\bigwedge_{c \in \mathcal{C}_{sys}} G_c \wedge \mathcal{A} \vdash_{\Sigma_{sys}} \Box\Diamond_{[0,10]} imgChange$$

(by soundness, cf. Thm. 5, and by rules (3) and (8)). In the following the left hand side (the conjunction of all guarantees and $\mathcal{A}$) is abbreviated by Γ_{sys}.

As a first step, we informally prove $\Gamma_{sys} \vdash_{\Sigma_{sys}} \Box(C_2 \rightarrow imgChange)$, i.e., we show that whenever the system is in configuration C_2 also *imgChange* holds: The internal component *Pres* guarantees that the content is continuously altered (i.e. *Pres.alterContent* is true), and since in C_2, port *Pres.alterContent* is connected to *Rend.alterContent* and component *Rend* guarantees *Rend.alterContent* $\leftrightarrow$ *Rend.imgChange*, *imgChange* is always true because of the delegate connector between *imgChange* and *Rend.imgChange* (in C_2).

We proceed this proof by doing a case distinction on the formula

$$F = \Box_{[0,3.5]} C_1 \wedge \Box_{[0,3.5]} \neg Intr.gesture$$

which expresses that always, within the time period of 3.5 seconds, there is no person in front of the display and additionally configuration C_1 is active. From F and its negated version $\neg F$ (together with Γ_{sys}) we will derive $\Diamond_{[0,10]} imgChange$. We first prove $\Gamma_{sys}, F \vdash_{\Sigma_{sys}} \Diamond_{[0,10]} imgChange$.

(1)	$\Box_{[0,3.5]} Intr.gesture \sim Mon1.gesture$	from F, by rule (1) and (19_i)
(2)	$\Box_{[0,3.5]} \neg Intr.gesture \leftrightarrow \neg Mon1.gesture$	from (1) by rule (22)
(3)	$\Box_{[0,3.5]} \neg Intr.gesture$	from F by rule (1)
(4)	$\Box_{[0,3.5]} \neg Mon1.gesture$	from (2),(3) by rule (13)
(5)	$\Box_{[0,3]} \neg Mon1.gesture$	from (4) by rule (15)
(6)	$\Diamond_{[3,3.5]} Mon1.signal1$	from (5), G_{Mon1} by rule (5)
(7)	$\Diamond_{[0,3.5]} Mon1.signal1$	from (6) by rule (16)
(8)	$\Box_{[0,3.5]} C_1$	from F by rule (1)
(9)	$\Diamond_{[0,3.5]} (C_1 \wedge Mon1.signal1)$	from (7),(8) by rule (20)
(10)	$\Box_{[0,3.5]} (C_1 \wedge Mon1.signal1 \rightarrow \Diamond_{[0,0.2]} C_2)$	from $\mathcal{A}$ by rule (1)
(11)	$\Diamond_{[0,3.5]} \Diamond_{[0,0.2]} C_2$	from (9),(10) by rule (12)
(12)	$\Diamond_{[0,3.7]} C_2$	from (11) by rule (18)
(13)	$\Box_{[0,3.7]} (C_2 \rightarrow imgChange)$	(see proof above), by rule (15)
(14)	$\Diamond_{[0,3.7]} imgChange$	from (12),(13) by rule (12)
(15)	$\Diamond_{[0,10]} imgChange$	from (14) by rule (16)

For the other case, i.e. $\Gamma_{sys}, \neg F \vdash_{\Sigma_{sys}} \Diamond_{[0,10]} imgChange$, we just informally describe how $\Diamond_{[0,10]} imgChange$ can be derived. Since we assume that $\neg F$ holds, we know $\neg C_1 \vee \Diamond_{[0,3.5]} Intr.persThere$. In case of $\neg C_1$ it follows by the assembly specification $\mathcal{A}$ that C_2 is true which (as we have shown above) immediately implies *imgChange*. If $\Diamond_{[0,3.5]} Intr.gesture$ holds, then it means that between 0 and 3.5 seconds, there must be *Intr.gesture* true at some point. If the system, at this point, is in configuration C_2 we argue the same way as before; otherwise the system is in configuration C_1, and in this case it is again straightforward since under configuration C_1, *Intr.gesture* is connected to *imgChange*. Finally, by rule (7), we can conclude that $\Gamma_{sys} \vdash_{\Sigma_{sys}} \Diamond_{[0,10]} imgChange$. By rule (9), we get $\Gamma_{sys} \vdash_{\Sigma_{sys}} \Box\Diamond_{[0,10]} imgChange$, as $\Box\Gamma_{sys}$ is semantically the same as Γ_{sys}.

In this section, we have hence shown that the chosen component-based design in fact satisfies the properties that we imposed in the beginning: the system reacts promptly to viewers in front of the display, and the system keeps showing animated content even if no viewer is interacting with it.

6.2.3 Experiences and results

In this section, we established a proof of correctness for a larger case study. This case study has shown how the assume-guarantee framework described in Chapter 4 can be applied to an real-world case study from the domain of physiological computing. The results of the case study are as follows:

- *Application to a larger system.* The established proof of correctness on the adaptive advertising display has shown that the the assume-guarantee framework developed in this thesis is applicable to larger systems.

- *Insights on interactions.* Establishing a proof of correctness in the assume-guarantee framework is a demanding task, but at the same time it allows to gain deep insights about subtle and intricate interactions of component behaviours and reconfigurations. Using reMITL and the accompanying framework allows for a better understanding about the pitfalls that are involved in specified system behaviours. The insights gained during the creation of a formal proof can be subsequently used to improve on the quality of the implementation, and to avoid discovered issues in the potential behaviour of the system.

Overall, the case study has shown that the approach to the formal verification of real-time, reconfigurable systems proposed in this thesis is a valid and valuable approach enhancing on the state of the art on the verification of reconfigurable systems.

6.3 Conclusion

The formal assume-guarantee framework for reconfigurable systems (first presented in Chapter 4) and the component-based software framework that is the REFLECT framework (introduced in Chapter 3) have been applied to two different case studies in this chapter.

The first case study, the REFLECT automotive demonstrator, shows a real use case for the REFLECT framework and have been used to discuss the impact of using the REFLECT framework for creating a physiological computing application. More specifically, we have discussed the entailed system configuration simplification, achieved increased robustness, the benefits of having capture/replay facilities as well as a management console in place, and the thread-confinement consequences of using (or abusing) the event distribution facilities of the framework.

In the second case study, the adaptive advertisement system, we have shown how the formal assume-guarantee framework for reconfigurable systems can be applied to prove that a system satisfies desired properties on a larger system. The insights that were attained in the process of proof construction helped to avoid issues in the behaviour of the implemented system.

Altogether, we have given evidence that both the REFLECT framework and the assume-guarantee framework constitute useful and practical tools whose use can provide advantages over creating physiological computing systems directly from scratch.

Chapter 7

Conclusion

This thesis has focused on the presentation of a comprehensive approach to the software engineering of physiological computing applications. We have considered the practical aspects of implementing physiological computing applications as well as formal aspects for the verification of reconfigurations. Furthermore, we have introduced a methodology that show how our formal verification framework can be combined with implementation activities that use the REFLECT framework. The complete approach, and the combination of our contributions is shown in Figure 7.1.

Our approach addresses the *verification* of component-based systems under reconfigurations. We have introduced a contract-based verification framework that allows to verify whether a refinement between an expected behaviour and the initial system design exists. Furthermore, we have introduced a component-based software framework that simplifies the *implementation* of a given physiological computing system design. Furthermore, we have discussed how verification activities using our formal verification framework should be combined with implementation activities with Scrum, and have proposed a software development *methodology* for the creation of physiological computing applications. Furthermore, we have introduced two reconfiguration patterns that show how both our formal verification framework and the REFLECT framework benefit from each other in the implementation of component-based reconfigurations.

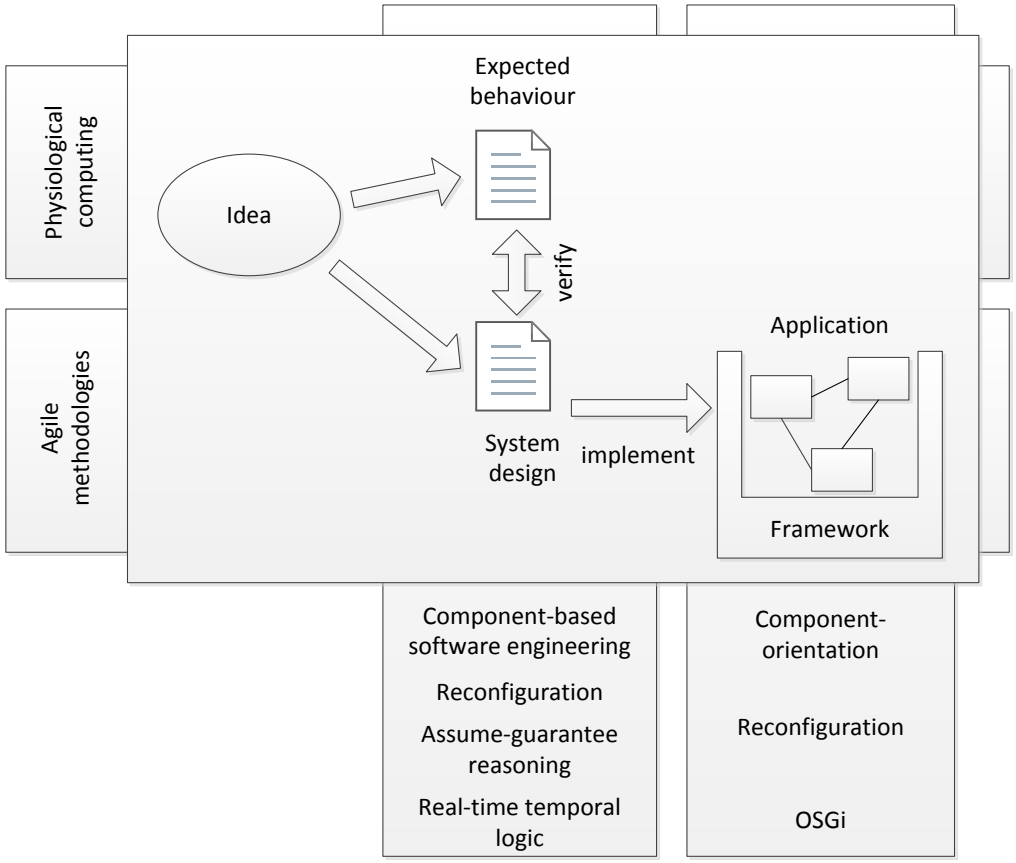


Figure 7.1: Software engineering for physiological computing

7.1 Contributions

We have addressed *implementation* issues in physiological computing by providing a component-based software framework, the REFLECT framework. We have discussed requirements of physiological computing applications, the concepts that we used to satisfy these requirements, as well as details about the implementation that we created, and how the implemented features performed in the end.

Furthermore, we have provided solutions for *verification* issues that arose from the use of reconfigurations in physiological computing applications. We have provided a formal verification framework based on assume-guarantee

contracts. The semantics of the contracts features a dense time domain that allow for a more precise and natural specification of contracts for physiological computing systems tht feature real-time behaviour. For the specification of reconfigurations and contracts, we have introduced reMITL, an extension of MITL with an operator for specifying component port connectivities.

We also investigated software development *methodologies* with the goal of providing a frame for implementation activities using the REFLECT framework and our formal verification framework together. We have shown how formal verification activities should be combined with implementation activities, and discussed two reconfigurations patterns that showcase the use of our formal verification framework as well as the REFLECT framework.

Finally, we have demonstrated that our approach is applicable to larger systems by discussing the application of both our formal verification framework as well as the REFLECT framework to a larger case study each. While both case studies were implemented, the first case study, being the REFLECT automotive demonstrator, focused on showcasing the applicability of the REFLECT framework to a physiological computing application that integrates four control loops (i.e. the emotional, effort, comfort, and driving control loops). The second case study discussed the application of our formal verification framework to the adaptive advertising case study, which represents a system that reacts to the viewers in front of it. Both case studies showcase the successful application of our results to a larger software system.

7.2 Discussion

This thesis has covered a large variety of topics: from implementation to formal verification and software development methodologies, which allowed to gain insights in each topic on its own, but also in the synergies and conflicts that emerged in each topic. In total, we present three findings which we think are specifically worthwhile to share.

7.2.1 Genericity of software frameworks

When creating a software framework, we naturally strive towards the most generic solutions, as our common computer science knowledge tells us that a solution is better the more generic it is. Genericity promises reducing the amount of duplications in code or models, as well as producing simple, clear

and powerful concepts. However, introducing more generic concepts means they need to be specialised for the specific purpose that needs to be solved. In a software framework, the specialisation of a generic programming interface to a concrete application is mainly achieved through a setup phase preceding the actual use of the programming interface, or specification of appropriate parameters on programming interface calls. Needless to say, generic programming interfaces introduce significant amounts of clutter in code using them. This clutter can be reduced with proper use of encapsulation techniques. However, it is also very well possible to introduce wrong abstractions that do not align well with the programming effort at hand, and hence significantly complicate software development efforts. It is therefore advisable to be very careful when introducing new generic concepts.

That being said, it is also very well possible that a software framework is generic without hindering software development activities, but neither helping it. This would have been (almost) the case for the REFLECT framework if we removed any data-specific features. However, the REFLECT framework also shows an approach that bridges between a very generic framework (i.e. component-based), and a specific application domain (i.e. physiological computing). By building specific services on top of a generic framework that allows for a relatively seamless extension of the provided infrastructure, we were capable of offering both a generic and easy to use component framework, and at the same time provide support specifically for physiological computing (e.g. data context services, capture-replay support, and simple distribution facilities).

Providing a generic framework that allows for specialised extensions hence seems to be a feasible and promising approach for introducing powerful generic concepts, and domain-specific services that together simplify implementation efforts.

7.2.2 Reconfigurations in the automotive demonstrator case study

As the avid reader may have already noticed or suspected, the automotive demonstrator makes no use of component reconfigurations, although all other provided features and services of the REFLECT framework were used. Given the fact that this thesis provides an extensive body of work specifically targeted at the formal verification of reconfigurations, this may seem surprising.

The explanation for this fact is very simple, however. The work within the REFLECT project was not only to advance the state of the art with respect to software infrastructures and computer technologies in the domain of physiological computing, but also advance the state of the art with respect to the physiology of emotion, cognitive effort, and physical comfort. The created demonstrator was primarily concerned with demonstrating all results achieved in the REFLECT project, and not only the software infrastructure. Consequently, the logic of the loops that were created in each research stream (emotion, cognitive effort, physical comfort) remained very simplistic, and did not require a separation of normal operation from meta-reasoning that can be achieved through software-level monitoring and reconfiguration. Any addition of monitoring and reconfiguration to the automotive demonstrator would have been artificial, and we hence refrained from superimposing the use of reconfigurations in the automotive demonstrator.

7.2.3 Benefits of constant feedback

In the creation of the automotive demonstrator (presented in Section 6.1), we have experienced the real value that continuous feedback offers to the creation of a product on our own. During the conception and implementation of first prototypes, we introduced joint installation and testing sessions in Maranello on the deployed hardware (i.e. the three UMPCs installed in the Ferrari California) roughly every three months for the first ten months, and every month in the final four months. These installation and testing sessions on the target hardware platform proved to be tremendously valuable for understanding the real challenges of the application to create, as well as highlighting the pressing issues that needed to be focused on. Also, the feedback gained from the test drivers on issues in the system behaviour and suggestions on how to improve the automotive demonstrator were valuable for guiding the development efforts.

To give an example, the test driver hinted that lateral and longitudinal accelerations are far more risky at high speed than at low speed. When driving at 30 kilometers per hour, it is possible to create higher lateral and longitudinal accelerations while still driving safe, as the car is moving slowly, and provides the driver with a larger window of reaction. At higher speeds such as 80 kilometers per hour, the same lateral and longitudinal accelerations may be nearly unfeasible on existing roads, as they require much quicker reactions and much more precise lateral and longitudinal control over the

car, and thus constitute a high risk driving style.

The constant installation and testing sessions – especially towards the end of the development endeavour – had also beneficial organisational effects on the development of the demonstrator overall: the repeated sessions helped to raise the visibility of the efforts on the demonstrator, and allowed the distributed development team to stay committed to the creation of the final demonstrator. In a sense, the repeated sessions allowed the research partners to understand that the created system was indeed real and was really to be created. Furthermore, the documentation of the demonstrator state on each testing session (through testing protocol documents, videos and pictures) allowed the team to appreciate and understand the progress they made towards the final automotive demonstrator that was created and successfully presented.

7.3 Future work

Both the REFLECT framework and our formal verification framework provide grounds for future work and potentials for extensions.

Considering our formal verification framework, it would be possible to extend the semantics to make the number of existing component instances dynamic, following the approach of history-dependent automata [87]. History-dependent automata introduce instance tokens in the state representation of automata, as well as a mapping of token instances from the source state to the target state with a state transition. A similar pooling and mapping approach may be used to keep track of live component instances, as well as to allow to create and destroy component instances. Furthermore, it could be worthwhile to introduce a syntax for specifying models (especially, for assembly specifications) based on timed automata [7]. Timed automata may prove to be a more natural tool for assembly specifications and the reconfigurations they contain. Furthermore, the concept we introduced in our formal verification framework of embedding connectors and connectivity in the semantics of the formalism can be applied not only to dense real-time traces, but to other formalisms as well in order to make changes in connectivity amenable for verification.

Considering the REFLECT framework, it would be worthwhile to investigate how the component, reconfiguration and data service concepts combine with a more elaborate distribution facility that offers more sophisticated

knowledge about network topology, and more sophisticated (e.g. topology-based) queries. Furthermore, it could be interesting to see how the software monitoring aspect could be extended beyond the property-based and event-based approaches our framework supports, and how well the new monitoring approaches integrate with the framework concepts. Finally, it would be worthwhile to investigate how the replay of captured data could be sped up in a systematic way that would not alter the behaviour of the system - apart from performing the operations faster. With captured runs being several hours long, we have found that the only two viable ways are a) off-line data analysis with mathematical simulation tools by replicating the algorithms and behaviour in these tools, or b) using the real system and significantly speed up the replay rate while accepting discrepancies in behaviour. Having a systematic mean for replaying data to a system in a fast-forward logical time while preserving the system behaviour would be a very worthwhile tool for off-line experimentation with physiological computing systems.

7.4 Final words

This thesis has presented a software engineering approach to the creation of physiological computing applications that is inehently component-oriented, as we believe that component-orientation is a strong concept for the creation of software systems. For physiological computing systems especially, we believe that separating normal operation from meta-reasoning that is enabled through reconfigurations that change a system's architecture offer significant benefits. We have contributed both a component-based software framework for the realisation of physiological computing systems, as well as a formal framework for the verification of systems under reconfigurations. By providing a methodology, we have also provided a software development process frame for embedding both frameworks.

Bibliography

[1] M. Abadi and L. Lamport. Composing Specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, 1993.

[2] Corporate Act-Net Consortium. The active database management system manifesto: a rulebase of ADBMS features. *ACM SIGMOD Record*, 25:40–49, 1996.

[3] N. Aguirre and T. S. E. Maibaum. Hierarchical Temporal Specifications of Dynamically Reconfigurable Component Based Systems. *Electronic Notes in Theoretical Computer Science*, 108:69–81, 2004.

[4] J. Aldrich. Using Types to Enforce Architectural Structure. In *Proceedings of the Seventh Working IEEE/IFIP Conference on Software Architecture*, pages 211–220. IEEE Computer Society, 2008.

[5] R. Allen, R. Douence, and D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Fundamental Approaches to Software Engineering*, volume 1382 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 1998.

[6] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[7] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[8] R. Alur, T. Feder, and . A. Henzinger. The Benefits of Relaxing Punctuality. *Journal of the ACM*, 43(1):116–146, 1996.

[9] R. Alur and T. A. Henzinger. Logics and Models of Real Time: A Survey. In *Real-Time: Theory in Practice*, volume 600 of *Lecture Notes in Computer Science*, pages 74–106. Springer, 1992.

[10] R. Alur and T. A. Henzinger. Real-Time Logics: Complexity and Expressiveness. *Information and Computation*, 104(1):35–77, 1993.

[11] R. Alur and T. A. Henzinger. A Really Temporal Logic. *Journal of the ACM*, 41(1):181–204, 1994.

[12] Apache felix maven scr plugin. `http://felix.apache.org/site/apache-felix-maven-scr-plugin.html`, 2010. visited 2011-10-21.

[13] C. Baier, M. Sirjani, F. Arbab, and J. Rutten. Modeling component connectors in reo by constraint automata. *Science of Computer Programming*, 61:75–113, 2006.

[14] H. Balzert. *Lehrbuch der Softwaretechnik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1998.

[15] W. Barfield and T. Caudell. *Fundamentals of wearable computers and augumented reality*. Lawrence Erlbaum, 2000.

[16] H. Barringer, R. Kuiper, and A. Pnueli. A Really Abstract Concurrent Model and its Temporal Logic. In *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 173–183. ACM, 1986.

[17] T. Barros, L. Henrio, and E. Madelaine. Verification of Distributed Hierarchical Components. *Electronic Notes in Theoretical Computer Science*, 160:41–55, 2006.

[18] N. Bartlett and H. Seeberger. Component Oriented Development in OSGi with DS, Spring and iPOJO. EclipseCon 2009 Tutorial. `http://www.eclipsecon.org/2009/sessions?id=245`, 2009. visited 2011-10-21.

[19] A. Basso, A. Bolotov, A. Basukoski, V. Getov, L. Henrio, and M. Urbanski. Specification and Verification of Reconfiguration Protocols in Grid Component Systems. Technical report, Institute on Programming Model (WP3), 2006. CoreGRID Technical Report, TR-0042.

[20] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Components in BIP. In *Proceedings of the Fourth IEEE International Conference on Software Engineering and Formal Methods*, pages 3–12. IEEE Computer Society, 2006.

[21] H. Baumeister, F. Hacklinger, R. Hennicker, A. Knapp, and M. Wirsing. A Component Model for Architectural Programming. *Electronic Notes in Theoretical Computer Science*, 160:75–96, 2006.

[22] K. Beck. *Extreme programming explained: embrace change*. Addison-Wesley Professional, 2000.

[23] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for Agile Software Development. `agilemanifesto.org`, 2001. visited 2011-10-21.

[24] G. Bell and P. Dourish. Yesterday's tomorrows: notes on ubiquitous computing's dominant vision. *Personal and Ubiquitous Computing*, 11(2):133–143, 2006.

[25] S. Bensalem, M. Bozga, J. Sifakis, and T. Nguyen. Compositional Verification for Component-Based Systems and Application. In *Proceedings of the 6th International Symposium on Automated Technology for Verification and Analysis*, pages 64–79. Springer-Verlag, 2008.

[26] A. Benveniste, B. Caillaud, A. Ferrari, L. Mangeruca, R. Passerone, and C. Sofronis. Multiple Viewpoint Contract-Based Specification and Design. In F. S. Boer, M. M. Bonsangue, S. Graf, and W. Roever, editors, *Formal Methods for Components and Objects*, pages 200–225. Springer, 2008.

[27] G. M. Bertolotti, R. Lombardi A. Cristiani and, M. Stanojevic, M. Ribaric, and N. Tomasevic. D5.3 Third Year Report: Evaluation of Case Studies. Technical report, Fraunhofer FIRST, 2011. `http://reflect.pst.ifi.lmu.de/images/pdf/d5.3.pdf`, visited 2011-10-21.

[28] G. M. Bertolotti, A. Cristiani, R. Lombardi, M. Ribaric, N. Tomasevic, and M. Stanojevic. Self-Adaptive Prototype for Seat Adaption.

In *Fourth IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshop*, pages 136–141. IEEE, 2010.

[29] G. Beyer, C. Mayer, C. Kroiß, and A. Schroeder. Person Aware Advertising Displays: Emotional, Cognitive, Physical Adaptation Capabilities for Contact Exploitation. In *1st Workshop on Pervasive Advertising*, 2009.

[30] B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

[31] B. Boehm. A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986.

[32] B. Boehm and R. Turner. Rebalancing Your Organization's Agility and Discipline. In *Proceedings of the Third XP Agile Universe Conference*, volume 2753 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2003.

[33] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*, pages 28–33. ACM, 2004.

[34] D. Brin. *The transparent society: Will technology force us to choose between privacy and freedom?* Basic Books, 1998.

[35] R. A. Brooks. Intelligence without representation. *Artificial intelligence*, 47(1-3):139–159, 1991.

[36] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Software — Practice & Experience*, 36:1257–1284, September 2006.

[37] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, and A. Lluch-Lafuente. Graph-Based Design and Analysis of Dynamic Software Architectures. In *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 37–56. Springer, 2008.

[38] R. Bruni, M. Hölzl, N. Koch, A. Lluch-Lafuente, P. Mayer, U. Montanari, A. Schroeder, and M. Wirsing. A Service-Oriented UML Profile

with Formal Support. In *Proceedings of the 7th International Joint Conference on Service-Oriented Computing and ServiceWave*, volume 5900 of *Lecture Notes in Computer Science*, pages 455–469, 2009.

[39] P. P. Caffier, U. Erdmann, and P. Ullsperger. Experimental evaluation of eye-blink parameters as a drowsiness measure. *European Journal of Applied Physiology*, 89:319–325, 2003.

[40] C. Carrez, A. Fantechi, and E. Najm. Assembling components with behavioural contracts. *Annales des Télécommunications*, 60(7-8):989–1022, 2005.

[41] H. Cervantes and R. S. Hall. Automating Service Dependency Management in a Service-Oriented Component Model. In *Proceedings of the 6th ICSE Workshop on Component-Based Software Engineering: Automated Reasoning and Prediction*, 2003.

[42] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, editors. *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.

[43] A. Damasio. *Descartes' Error*. G.P. Putnam, 1994.

[44] A. Damasio. *The Feeling of What Happens: Body and Emotion in the Making of Consciousness*. Mariner Books, 2000.

[45] V. Dehlen. DiVA methodology. Technical report, SINTEF, 2010.

[46] B. Delahaye and B. Caillaud. A Model for Probabilistic Reasoning on Assume/Guarantee Contracts. Research Report RR-6719, INRIA, 2008.

[47] G. Elliott. *Global Business Information Technology: an integrated systems approach*. Pearson Education, 2004.

[48] C. Escoffier, R. S. Hall, and P. Lalanda. iPOJO: an Extensible Service-Oriented Component Framework. In *IEEE International Conference on Services Computing*, pages 474 – 481. IEEE Computer Society, 2007.

[49] C. Evers, A. Hoffmann, D. Saur, K. Geihs, and J. M. Leimeister. Ableitung von Anforderungen zum Adaptionsverhalten in ubiquitären adaptiven Anwendungen. *Electronic Communications of the EASST*, 37, 2011.

[50] S. Fagorzi and E. Zucca. A Calculus of Components with Dynamic Type-Checking. *Electronic Notes in Theoretical Computer Science*, 182:73–90, 2007.

[51] S. H. Fairclough. Fundamentals of physiological computing. *Interacting with Computers*, 21(1–2):133–145, 2009.

[52] S. H. Fairclough and L. Venables. Psychophysiological candidates for biocybernetic control of adaptive automation. In D. de Waard, K. A. Brookhuis, and C. M. Weikert, editors, *Human Factors in Design*, pages 177–189. Shaker, 2004.

[53] M. Fowler. Pojo. `http://www.martinfowler.com/bliki/POJO.html`, 2000. visited 2011-10-21.

[54] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Longman Publishing, 2002.

[55] M. Fowler. Inversion of control containers and the dependency injection pattern. `http://martinfowler.com/articles/injection.html`, 2004. visited 2011-10-21.

[56] M. Fowler. The New Methodology. `http://martinfowler.com/articles/newMethodology.html`, 2005. visited 2011-10-21.

[57] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[58] M. Fowler and J. Highsmith. The agile manifesto. `http://drdobbs.com/open-source/184414755`, 2001. visited 2011-10-21.

[59] E. Gat. *Three-layer architectures*, pages 195–210. MIT Press, 1998.

[60] K. Geihs, C. Evers, R. Reichle, M. Wagner, and M. U. Khan. Development Support for QoS-Aware Service-Adaptation in Ubiquitous Computing Applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 197–202. ACM, 2011.

[61] Gemini Blueprint. `www.eclipse.org/gemini/blueprint/`. visited 2011-10-21.

[62] B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, and T. Peierls. *Java Concurrency in Practice*. Addison-Wesley Professional, 2006.

[63] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java (TM) Language Specification*. Addison-Wesley, 3 edition, 2005.

[64] E. Guttman, C. Perkins, J. Veizades, and M. Day. Service Location Protocol, Version 2. Technical report, The Internet Engineering Task Force, 1999. RFC 2608 `http://www.ietf.org/rfc/rfc2608.txt`, visited 2011-10-21.

[65] M. Hammer. *How To Touch a Running System – Reconfiguration of Stateful Components*. PhD thesis, Ludwig-Maximilians Universität München, 2009.

[66] P. A. Hancock and J. L. Szalma. The future of neuroergonomics. *Theoretical Issues in Ergonomics Science*, 4(1-2):238–249, 2003.

[67] G. K. Hanssen, H. Westerheim, and F. O. Bjørnson. Tailoring RUP to a defined project type: A case study. In F. Bomarius and S. Komi-Sirviö, editors, *Product Focused Software Process Improvement*, volume 3547 of *Lecture Notes in Computer Science*, pages 209–228. Springer, 2005.

[68] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26(3):251–321, 1985.

[69] B. Henderson-Sellers and P. Giorgini, editors. *Agent-Oriented Methodologies*. Idea Group Publishing, 2005.

[70] R. Höhn and S. Höppner. *Das V-Modell XT: Grundlagen, Methodik und Anwendungen*. Springer, 2008.

[71] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[72] J. Hooman. A compositional proof theory for real-time distributed message passing. In *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 259 of *Lecture Notes in Computer Science*, pages 315–332. Springer, 1987.

[73] J. Hooman and J. Widom. A Temporal-Logic Based Compositional Proof System for Real-Time Message Passing. In *Parallel Architectures and Languages Europe, Volume II: Parallel Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 1989.

[74] ISO/IEC 9126-1:2001 Software engineering – Product quality – Part 1: Quality model, 2001.

[75] J. H. Janssen, E. L. van den Broek, and J. Westerink. Personalized affective music player. In *Proceedings of the IEEE 3rd International Conference on Affective Computing and Intelligent Interaction*, pages 704 – 709. IEEE, 2009.

[76] A. Knapp, G. Marczynski, M. Wirsing, and A. Zawlocki. A heterogeneous approach to service-oriented systems specification. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2477–2484. ACM, 2010.

[77] R. Koymans. Specifying Real-Time Properties with Metric Temporal Logic. *Real-Time Systems*, 2(4):255–299, 1990.

[78] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, 1990.

[79] P. Kruchten. *The Rational Unified Process: An Introduction*. Addison Wesley, 2 edition, 2000.

[80] C. Larman and B. Vodde. *Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum*. Addison-Wesley Professional, 2010.

[81] K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[82] L. Leahu, P. Sengers, and M. Mateas. Interactionist AI and the promise of ubicomp, or, how to put your box in the world without putting the world in your box. In *Proceedings of the 10th International Conference on Ubiquitous computing*, volume 344 of *ACM International Conference Proceeding Series*, pages 134–143. ACM, 2008.

[83] B. Lee. I don't get Spring. `http://blog.crazybob.org/2006/01/i-dont-get-spring.html`, 2006. visited 2011-10-21.

[84] H. R. Lewis. A Logic of Concrete Time Intervals (Extended Abstract). In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 380–389. IEEE Computer Society, 1990.

[85] N. Medvidovic. ADLs and dynamic architecture changes. In *Joint proceedings of the second international software architecture workshop and international workshop on multiple perspectives in software development*, pages 24–27. ACM, 1996.

[86] J. Misra and K. M. Chandy. Proofs of Networks of Processes. *IEEE Transactions on Software Engineering*, 7(4):417–426, 1981.

[87] U. Montanari and M. Pistore. An Introduction to History Dependent Automata. *Electronic Notes in Theoretical Computer Science*, 10(0):170–188, 1998.

[88] L. J. M. Mulder. Measurement and analysis methods of heart rate and respiration for use in applied environments. *Biological Psychology*, 34(2-3):205–236, 1992.

[89] D. A. Norman. *The Design of Future Things*. Basic Books, 2007.

[90] P. C. Ölveczky and J. Meseguer. Semantics and pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation*, 20:161–196, 2007.

[91] OSGi Service Platform Compendium Specification. Release 4, Version 4.2, 2009.

[92] OSGi Service Platform Core Specification. Release 4, Version 4.2, 2009.

[93] J. S. Ostroff. Composition and Refinement of Discrete Real-Time Systems. *ACM Transactions on Software Engineering and Methodology*, 8(1):1–48, 1999.

[94] P. Parizek and F. Plasil. Assume-guarantee verification of software components in SOFA 2 framework. *IET Software*, 4(3):210–211, 2010.

[95] O. Pedreira, M. Piattini, M. R. Luaces, and N. R. Brisaboa. A systematic review of software process tailoring. *ACM SIGSOFT Software Engineering Notes*, 32(3):1–6, 2007.

[96] Pervasive Adaptation - background document. `ftp://ftp.cordis.europa.eu/pub/ist/docs/fet/ie-jan07-peradapt-01.pdf`, 2006. visited 2011-10-21.

[97] R. W. Picard. *Affective Computing*. MIT Press, 1997.

[98] R. W. Picard. Affective computing: challenges. *nternational Journal of Human-Computer Studies*, 59(1–2):55–64, 2003.

[99] R. W. Picard and J. Klein. Computers that recognise and respond to user emotion: theoretical and practical implications. *Interacting with computers*, 14(2):141–169, 2002.

[100] A. J. Ramirez and B. H. C. Cheng. Design patterns for developing dynamically adaptive systems. In *ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pages 49–58. ACM, 2010.

[101] Java™ Remote Method Invocation Specification. `http://download.oracle.com/javase/7/docs/platform/rmi/spec/rmi-title.html`. visited 2011-10-21.

[102] W. W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the 9th international conference on Software Engineering*, pages 328–338. IEEE Computer Society Press, 1987.

[103] M. Satyanarayanan. Pervasive Computing: Vision and Challenges. *IEEE Personal Communications*, 8(4):10–17, 2002.

[104] P. Schobbens, J. Raskin, and T. A. Henzinger. Axioms for real-time logics. *Theoretical Computer Science*, 274(1-2):151–182, 2002.

[105] A. Schroeder, C. Kroiß, and T. Mair. Context Acquisition and Acting in Pervasive Physiological Computing. In *Proceedings of the 7th International ICST Conference on Mobile and Ubiquitous Systems*, 2010.

[106] Schroeder, A. and Bauer, S. S. and Wirsing, M. A contract-based approach to adaptivity. *Logic and Algebraic Programming*, 80:180 – 193, 2011.

[107] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall, 2002.

[108] K. Schwaber and J. Sutherland. Scrum guide. `http://www.scrum.org/scrumguides/`, 2011. visited 2011-10-21.

[109] M. S. Schwartz and F. Andrasik. *Biofeedback: A Practitioner's Guide*. The Guilford Press, 3 edition, 2005.

[110] N. B. Serbedzija. Reflective Computing – Naturally Artificial. In Jeremy Pitt, editor, *This Pervasive Day: The Potential and Perils of Pervasive Computing*. Imperial College Press, 2011.

[111] N. B. Serbedzija and S. H. Fairclough. Biocybernetic loop: From awareness to evolution. In *IEEE Congress on Evolutionary Computation*, pages 2063–2069. IEEE, 2009.

[112] A. K. Shuja and J. Krebs. *IBM Rational Unified Process Reference and Certification Guide: Solution Designer*. IBM Press, 1 edition, 2008.

[113] M. Simonot and V. Aponte. A Declarative Formal Approach to Dynamic Reconfiguration. In *Proceedings of the 1st international workshop on Open component ecosystems*, pages 1–10. ACM, 2009.

[114] I. Sommerville. *Software Engineering*. Addison-Wesley, 9 edition, 2010.

[115] I. Sommerville and P. Sawyer. Viewpoints: principles, problems and a practical approach to requirements engineering. *Annals of Software Engineering*, 3:101–130, 1997.

[116] Spring 3 reference documentation. `http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/pdf/spring-framework-reference.pdf`, 2010. visited 2011-10-21.

[117] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Addison-Wesley Professional, 2002.

[118] D. S. Tan and A. Nijholt, editors. *Brain-Computer Interfaces. Applying our Minds to Human-Computer Interaction*. Human–Computer Interaction Series. Springer, 1 edition, 2010.

[119] R. Vanbrabant. *Google Guice: Agile Lightweight Dependency Injection Framework*. Apress, 2008.

[120] M. Weiser. The computer for the 21st century. *ACM SIGMOBILE Mobile Computing and Communications Review*, 3(3):3–11, 1999.

[121] M. Wirsing. Algebraic Specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 675–788. Elsevier and MIT Press, 1990.

[122] M. Wirsing, S. S. Bauer, and A. Schroeder. Modeling and Analyzing Adaptive User-Centric Systems in Real-Time Maude. In *Proceedings of the First International Workshop on Rewriting Techniques for Real-Time Systems*, volume 36 of *EPTCS*, pages 1–25, 2010.

[123] F. Zambonelli, N. R. Jennings, and M. Wooldridge. Developing multi-agent systems: The gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12:317–370, 2003.

[124] Z. Zeng, M. Pantic, G. I. Roisman, and T. S. Huang. A survey of affect recognition methods: audio, visual and spontaneous expressions. In *Proceedings of the 9th international conference on Multimodal interfaces*, pages 126–133. ACM, 2007.

[125] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of the 28th international conference on Software engineering*, pages 371–380. ACM, 2006.