Michael Barth

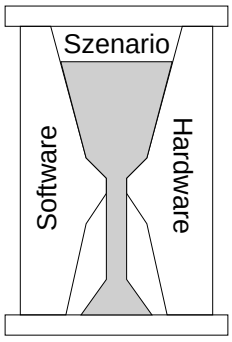
Entwicklung und Bewertung zeitkritischer Softwaremodelle

Dissertation an der Fakultät für
Mathematik, Informatik und Statistik der
Ludwig-Maximilians-Universität München
vorgelegt am 14. November 2007

1. Gutachter: Prof. Dr. Martin Wirsing
2. Gutachter: Prof. Dr. Walter Dosch
Tag der Prüfung: 7. Januar 2008

Michael Barth

Entwicklung und Bewertung zeitkritischer Softwaremodelle



Simulationsbasierter Ansatz und Methodik

Zusammenfassung

In der vorliegenden Arbeit wird eine Technik vorgestellt, die Software bereits aufgrund ihrer Modellbeschreibung in ihrem zeitlichen Verhalten bewertbar macht. Dazu wird eine Technik zur Beschreibung dynamischer Ausführungsmodelle für Zeitverhalten, sogenannte dynamische Performanz-Modelle, eingeführt. Ein Modellierungtechnik für Umgebungen wird beschrieben. Schließlich erlaubt die Definition einer zeitbehafteten Semantik die Bewertung einzelner Bearbeitungsabläufe.

Mittels einer Petrinetzsemantik wird einerseits das Modellverhalten exakt festgelegt und andererseits ein Wohlgeformtheitsbegriff eingeführt, der einer späteren Werkzeugimplementierung dient. Die vorteilhaften technischen Eigenschaften dieses Entwurfes erlauben zudem eine Übersetzung aus UML-Aktivitätsdiagrammen.

Anhand der Bewertung einzelner Abläufe wird das zeitliche Verhalten eines dynamischen Modells analysiert. Diese einzelnen Abläufe, sogenannte Szenarien, werden durch das Werkzeug bewertet. Jeweils ein Tripel aus einem dynamischen Modell, einem Szenario und einer passenden Umgebung läßt sich mit einen Zeitwert bewerten. Dynamischer Modelle können direkt aus UML-Aktivitätsdiagrammen erzeugt werden. Szenarien können graphisch editiert und Last- und Ressourcenmodellen aus Meßwerttabellen generiert werden. Die ermittelten Zeitwerte können dann graphisch aufbereitet und zur Beurteilung der dynamischen Eigenschaften herangezogen werden.

Die hier entwickelten Modelle und eingesetzten formalen Spezifikationstechniken werden in der Werkzeugstudie vollständig gekapselt und müssen vom Werkzeuganwender nicht beherrscht werden. Somit stellt diese Arbeit auch ein Musterbeispiel für den Software-Entwurf unter Einbeziehung formaler Methoden dar.

Abstract

This book introduces a new technology to access performance properties of software based on its design artifacts. A technology of description of performance models and a technique of environment description is introduced. Finally time assessment of single runs is defined by a time semantics.

A Petri nets semantics is introduced to grant a specific dynamic behaviour at one hand, and on the other hand the semantics grants a well formedness that is helpful for a later tool implementation. The advantageous semantics design allows a translation from UML activity graphs.

One analyses temporal behaviour by assessment of distinct runs of the dynamic model. These runs are called *scenarios* and are evaluated in a simulation environment. Each tuple of a dynamic model, a scenario and an simulation environment defines a distinct execution time. Dynamic performance models can be translated direct from UML activity diagrams. Scenarios cab be developed graphically. Load and ressource models can be generated based on measurement tables. Obtained results can be edited graphically and can be utilised for the assessment of the models dynamic behaviour.

All models introduced in this approach are completely capsuled by the toolbench case study and need not to be handled by the user. Hence this approach represents also a paradigm for a modern software enigineering utilising formal methods.

Entwicklung und Bewertung zeitkritischer Softwaremodelle

Simulationsbasierter Ansatz und Methodik

Michael Barth

Für Jutta

in Liebe und Anerkennung

Soles occidere et redire possunt
Nobis cum semel occidit brevis Lux
Nox est perpetua una dormienda

C. Valerius Catullus, Carmine

Danksagung

Mein erster Dank gilt meinem Doktorvater Prof. Dr. Martin Wirsing, der mir die Forschungsarbeit an der Universität München ermöglicht und mir bei Wahl und Ausgestaltung des Themas weitgehend freie Hand gelassen hat. Weiterhin danke ich Herr Prof. Dr. Walter Dosch für seine Geduld und die Bereitschaft sich in ein so umfangreiches Werk einzuarbeiten. Schließlich danke ich meinen Kollegen Dr. Matthias Hölzl und Prof. Dr. Alexander Knapp, von denen ich so viel gelernt habe dank ihrer großen Geduld, bei vielen Tassen Kaffee und manchem Gespräch bis in den Morgen.

Inhaltsverzeichnis

Teil 1. Spezifikation und Bewertung dynamischer Modelle 15

Kapitel 1. Einleitung 17
1.1. Motivation 17
1.2. Verwandte Arbeiten 20
Spezifikationstechniken 20
Quantitative Modelle 22
Methodik 28
1.3. Unser neuer Ansatz 30
Eine simulationsbasierte Technik 30
Methodik. 34

Kapitel 2. Dynamische Modelle 37
2.1. Grundlagen 37
2.2. Dynamisches Kontrollflußmodell 42
2.3. Kontrollflußsemantik 66
2.4. Erreichbarkeitsgraph 86
2.5. Zusammenfassung 89

Kapitel 3. Szenarien 91
3.1. Grundlagen 91
3.2. Dynamische Modelle während der Simulation 98
3.3. Kommunikation 112
3.4. Zusammenfassung 117

Kapitel 4. Umgebungen 119
4.1. Lasten, Ressourcen und Umgebung 119
4.2. Anwendung 123
4.3. Zusammenfassung 128

Kapitel 5. Simulation 129
5.1. Grundlegende Begriffe 129
5.2. Simulationssteuerung 130
5.3. Laufzeitsemantik 136
5.4. Zusammenfassung 147

Teil 2. Entwicklung zeitkritischer Software 149

Ein Entwurf zu performanzzentrierter Softwareentwicklung 151

Kapitel 6. Prozeßgrundlagen 153
6.1. Prozeßintegration 156
6.2. Prozeßablauf 160
6.3. Rollen 168

Entwicklung und Bewertung von Software-Modellen in der Praxis 175

Kapitel 7. Betrachtungen zur Anwendbarkeit 177
7.1. UML - Eine Anwendung 177
7.2. Werkzeugimplementation 182
7.3. Anwendungsbeispiel 198
7.4. Auswertungsklassen 205
7.5. Zusammenfassung 208

Risikomanagement im Prozeßablauf 211

Kapitel 8. Risikozentrierter Prozeßentwurf 213
8.1. Einführung 213
8.2. Prozeß mit Fehlschlagsrisiko 214
8.3. Zusätzliche Aspekte des Prozeßentwurfs 220
8.4. Zusammenfassung 229
Schlußwort 233

Schlagworte und Begriffe 235

Abbildungsverzeichnis 239

Tabellenverzeichnis 245

Literaturverzeichnis 247

Teil 1

Spezifikation und Bewertung dynamischer Modelle

KAPITEL 1

Einleitung

1.1. Motivation

Bei Software spricht man aus Sicht eines Entwicklers meist von einem Quellcode-Dokument, welches in einer Programmiersprache verfasst ist. Diese Verkürzung ist deshalb interessant, weil man die Abbildung eines syntaktisch korrekten Programmes auf eine ausführbare Binärdatei durch einen Compiler praktisch nicht weiter hinterfragt. Heute ist dies meist ein automatischer, problemloser und wiederholbarer Vorgang. Dabei ist die vollständige Delegation des Übersetzungsvorgangs einiger Programmiersprachen an einen Compiler erst seit wenigen Jahrzehnten kein Gegenstand der Diskussion mehr.

Die fehlerfreie Übersetzbarkeit belegt aber nicht, daß ein Programm die von der Software erwarteten Aufgaben erfüllt, also *funktionale Anforderungen* realisiert. Es wurden viele Techniken entwickelt, um solche Anforderungen ausdrücklich zu spezifizieren. In der industriellen Software-Fertigung hat sich dabei besonders die Sprachfamilie der *UML* zur Spezifikation durchgesetzt. Die verschiedenen Notationen haben ihren Erfolg einerseits der intuitiv leicht verständlichen Notationen zu verdanken, andererseits aber auch der *weichen* Semantik, die oft eine freie Interpretation im lokalen Kontext zuläßt. Dies ist im industriellen Bereich ausdrücklich erwünscht, auch wenn das einer Entwicklung von Werkzeugen hinderlich ist, die Software-Eigenschaften in die Ebene der Programmiersprache überträgt. Compiler, die vollständig und problemlos - wie bei höheren Programmiersprachen - eine automatische Übersetzung von Softwarespezifikationen in Quellcode ermöglichen, bleibt aus vielen Gründen Zukunftsmusik.

Die Trennung der Ebene der Implementation und der Ebene der Spezifikation der Anforderungen führte zwangsläufig zur Verbreitung eines Prozeßverständnisses der Software-Entwicklung, die Abschnitte von Analyse, Entwurf, Implementierung und Test unterscheidet. Jeder Abschnitt weist dabei typische Artefakte auf. Abhängig davon, wie bewußt in der Praxis die Trennung dieser Abschnitte betrieben wird, erfolgt die Entwicklung von Software in Iterationen dieser Abschnitte.

Mit dem Fortschritt im Entwicklungsprozeß lassen sich typische Problembereiche identifizieren, die den Inhalt der Artefakte bestimmen und einen spezifischen Aufwand erfordern. Dazu wurden Prozeßmodelle entwickelt, die verschiedene Phasen in der Kette an Iterationen unterscheiden. Beispielsweise der *Unified Process* [**40**], der *Inception-*, *Elaboration-*, *Construction Phase* und andere unterscheidet. Ein weiterer Grund der weiten Verbreitung der UML ist die Durschgängigkeit in den verschiedenen Phasen eines Projektes. Wird dieser Einsatz bewußt durchgeführt, skalieren die Techniken gut für die verschiedenen Stufen der Abstraktion.

In den letzten Jahren kann man Anstrengungen beobachten, die Aufwendungen im Prozeß quantifizierbar und damit den Prozeß wirtschaftlich besser planbar machen. Da die Entwicklung von Software kostspielig ist, sind belastbare Vergleichsstudien nicht in ausreichendem Umfang verfügbar. Der Prozeß selbst und das Wissen um sein Management werden als schützenwerte Güter betrachtet und unterliegen der Geheimhaltung. Wirklich befriedigende Maßfunktionen bleiben problematisch.

Der größte Teil der Arbeiten zur Spezifikation und der Umsetzung von Software-Projekten befaßt sich mit diesen *funktionalen* Eigenschaften. Eigenschaften wie Sicherheit, Verfügbarkeit, Stabilität und Performanz wurden meist implizit von den ersten Interview-Protokollen bis zur fertigen Software mitgeführt. Trend ist eine gesonderte Beschäftigung mit diesen *nicht-funktionalen* Eigenschaften. Fragestellungen zu einer geeigneten Spezifikation und einer Thematisierung im Software-Entwicklungsprozeß müssen dabei behandelt werden.

In der vorliegenden Arbeit wird das Problem der zeitbezogenen Performanz von Software behandelt. Möglichkeiten zur Beschreibung dieser nicht-funktionalen Eigenschaften existieren zwar bereits, einige in der Praxis relevanten Eigenschaften können aber nicht oder noch nicht gut genug ausgedrückt werden. Unser Ansatz soll mehrere wichtige Ziele erreichen:

- Unterstützung gängiger Techniken (zB. UML)
- Nahtlose Integrierbarkeit in gängige Softwareentwicklungs- oder Prozeß-Modelle (zB. RUP)
- Legung eines Fundamentes für die Entwicklung eines qualitativ hochwertigen Werkzeugs

Die entscheidende Hemmschwelle für die Einführung neuer Technologien in einen Prozeß sind selten die Anschaffungskosten eines Werkzeuges. Eine Technik, die eine Übertragung in einen eigenen Formalismus verwendet, in den übersetzt werden muß, setzt meist ein umfassendes

Verständnis dieses neuen Formalismus voraus. Verlangt dieser Formalismus auch noch mathematische oder andere gehobene fachliche Qualifikation ist er bei nicht wissenschaftlich ausgebildetem Personal ein Ausschlußkriterium. Andernfalls aber immer noch ein entscheidendes Gegenargument gegen den Einsatz der Technik. Darum ist das vierte und vielleicht wichtigste Ziel eine Technik zu entwickeln, die durch ein intuitiv anwendbares Werkzeug erschlossen ist und zu deren Nutzung kein Verständnis einer komplizierten Theorie nötig ist.

Die Grundlagen des hier präsentierten Ansatzes werden im ersten Teil der Arbeit herausgearbeitet. Dazu werden im zweiten Kapitel Aufbau und Darstellung dynamischer Modelle, die zur Simulation herangezogen werden, exakt spezifiziert. So wird sichergestellt, daß alle nach dieser Spezifikation wohlgeformten Modelle eindeutig interpretiert werden können. Außerdem werden Modelleigenschaften sichergestellt, die eine spätere Umsetzung in ein Simulationswerkzeug begünstigen und große Ausdruckskraft begünstigen.

Im dritten Kapitel wird zum besseren Verständnis zunächst die Menge der möglichen Abläufe eines solchen Modells, Szenarien genannt, in die existierenden Theorie eingeordnet. Die Entwicklung eines Editors für Testabläufe im Simulationswerkzeug wird durch das Verständnis der Struktur von Szenarien erheblich begünstigt.

Im vierten Kapitel werden die Mindestanforderungen zur Beschreibung einer virtuellen Umgebung durch Zeitfunktionen entwickelt. Dadurch wird im Simulationswerkzeug eine Beschreibung des zeitlichen Verhaltens einer Ressource durch eine beliebig zu gestaltende Funktion ermöglicht, die für eine konkrete Lastsituation eine Bearbeitungszeit ermittelt. Solche Werte können in der Praxis durch eine Messung meist leichter bestimmt werden als beispielsweise eine „mittlere Verweildauer in einer Prozeßschlange".

Im fünften Kapitel wird eine Bewertungsfunktion für einzelne Abläufe, jeweils bestehend aus einem Tripel aus dynamischem Modell, Szenario und einer Umgebung eingeführt. Die definiert einen Zeitwert, der zur Ausführung des Ablaufes benötigt wird. Somit greift das fünfte Kapitel die Ergebnisse der drei vorigen Kapitel auf.

Durch Änderung dieser Schätzwerte im Laufe des Entwicklungsprozesses kann auf Erfolge oder Mißerfolge geschlossen werden.

Im zweiten Teil der Arbeit werden die unter verschiedenen Blickrichtungen formulierten Ziele dieser Technik in der Anwendung erprobt. Im sechsten Kapitel wird die Technik in ein klassisches Prozeßverständnis eingebettet. Die bekannten Phasen, Artefakte und Rollenbeschreibungen werden ergänzt, soweit es die neue Technik notwendig macht.

Damit wird ein Anwendungsvorschlag für den Einsatz in der Praxis vorgestellt. Im siebten Kapitel wird die Anwendbarkeit dieser Technik auf aktuell verwendete Notationen der UML, hier Aktivitäts- und Implementationsdiagramme, untersucht. Eine Übersetzung für UML-Notationen wird definiert. Eine Architektur für ein Werkzeug wird vorgestellt, in der jede Komponente einen Teilformalismus vollständig kapselt. Schließlich wird am Anwendungsbeispiel der prototypischen Werkzeugimplementierung |SEMPER|[1] sichtbar, daß die Kenntnisse aus dem ersten Teil dieser Arbeit zum Verständnis der Funktion des Werkzeuges hilfreich, zu seinem Einsatz in der Praxis aber nicht notwendig sind und das Ziel einer hochgradigen Intuitivität erreicht wurde. Weiterhin wird am Beispiel gezeigt, daß eine Ableitung der Ressourcenmodellierung aus einfach durchzuführenden Messungen unproblematisch ist. Schließlich wird im letzten Kapitel der Einsatz dieser Technik in risikobehafteten Software-Entwicklungs-Prozessen thematisiert. Chancen und Grenzen bei der Verringerung von Fehlschlagsrisiken und der Verbesserung der Qualität bezüglich der Performanz-Eigenschaften werden aufgezeigt.

1.2. Verwandte Arbeiten

In der Vergangenheit sind eine Reihe von Ansätzen zur Bewertung der Performanzeigenschaften von Software entstanden. Diese Ansätze lassen sich nach drei Aspekten klassifizieren. Ein Aspekt betrachtet die Technik oder Notation, in der ein dynamisches Modell spezifiziert wird, ein weiterer Aspekt ist das Modell, welches zur Bewertung der dynamischen Eigenschaften herangezogen wird, und schließlich die Ansätze zur Methodik, mit denen durch verwendete Techniken gewonnene Erkenntnisse den Software-Entwicklungsprozeß beeinflussen sollen.

Spezifikationstechniken

Klassische Techniken zur Beschreibung dynamischer Eigenschaften verwenden Netzgraphen, Transitionssysteme oder Zustandsautomaten. Formalismen sind dann zur Beschreibung dynamischer Modelle geeignet, wenn sich sequentielle und parallele Abhängigkeiten ausreichend ausdrücken lassen.

[1] SEMPER ist ein Akronym für: **S**imulationswerkzeug zur **E**valuierung von **M**odellen auf **P**erformanz-**E**igenschaften und **R**essourcennutzung

Gewichteter Automat. Neben den bekannten deterministischen und nicht-deterministischen Automaten findet man stochastische Zustandsautomaten, die einen nicht-deterministischen Automaten darstellen, dessen Übergangsfunktion mit einer Wahrscheinlichkeitsfunktion annotiert ist. [**38**]

Eine weitere allgemeine Form stellt ein gewichteter Automat dar. Es handelt sich dabei um ein Fünftupel $\mathcal{A} = (Q, \Sigma, \delta, \lambda, \theta)$. Dabei spezifizieren die Übergangsfunktion $\delta : Q \times \Sigma \times Q \to \mathbb{K}$ und die Funktionen $\lambda : Q \to \mathbb{K}$ und $\theta : Q \to \mathbb{K}$ Kosten, wobei die Werte für Kosten aus einem Halbring $\mathbb{K} = (K, +, *, 0, 1)$ entnommen sind. δ und θ spezifizieren Einstiegs- und Ausstiegsgewichte zu den Zuständen und δ Kantengewichte zu jeder Transition. Mit der Summe der Produkte der Einstiegs- und Ausstiegsgewichte sowie der Transitionsgewichte wird ein Wortgewicht des Automaten bewertet. Stephane Gaubert [**32**] beschreibt die Möglichkeit, damit Performanzeigenschaften zu berechnen.

Prozess-Algebra. Zur Spezifikation von Sequentialität und Nebenläufigkeit von Prozessen kann Prozess-Algebra verwendet werden [**31**]. Analysen wie Verhaltensäquivalenz und Verklemmungsfreiheit werden durch diese Spezifikation ermöglicht. Um Performanzuntersuchungen durchführen zu können, sind aber Erweiterungen des Modells notwendig. Hillston und Gilmore [**33, 34, 35**] haben dazu die *Performance Evaluation Process Algebra - PEPA* eingeführt. PEPA stellt eine eigene Sprache dar, mit der Modelle beschrieben werden können und ist mit einem mathematischen Modell unterlegt, welches stochastische Prozesse verwendet. Genauso aber wird PEPA als Formalismus verwendet, in den man von anderen Notationen aus hineinübersetzt [**34**]. PEPA zeigt darin also ein ambivalentes Verhalten. Grundzüge von PEPA werden im folgenden Kapitel eingeführt.

Petrinetze. Besonders ausdrucksmächtig zur Darstellung eines dynamischen Modells sind Petrinetze [**57**]. Ein Petrinetz (S, T, F, m, m_0) besteht aus einem bipartiten, gerichteten Graphen (*Stellen* S, *Transitionen* T und *Flußrelation* F) und einer *Markierung* $m : S \to \mathbb{N}$, welche die Anzahl an *Token* auf jeder Stelle spezifiziert. Die Startmarkierung m_0 definiert dabei die initiale Belegung der Stellen des Netzes.

Ein solcher Graph kann unmittelbar zur Spezifikation eines dynamischen Modells verwendet werden. Es existieren in der Literatur zahlreiche bekannte Varianten höherer Petrinetze, die spezielle Eigenschaften durch Transitionsannotationen spezifizieren. Solche Annotationen könne beispielsweise durch positive ganze Zahlen als Prioritäten

(*prioritäts-basierte Netze*), Zeitwerte (*gezeitete Netze*), Zufallsvariablen (*stochastische Netze*) oder beliebige Attribute realisiert sein. [**17**]

Message Sequence Charts. Zur Spezifikation dynamischer Eigenschaften anhand der Nachrichtenfolgen komunizierender Objekte wurden von der International Telecommunication Union *Message Sequence Charts* (MSC) als Standard eingeführt. Dieser Standard kennt die Darstellung beispielhafter Kommunikationsabläufe in graphischer und textueller Notation. [**4**]

Unified Modelling Language. Die Unified Modelling Language [**62, 61**] kennt eine Reihe an graphischen Notationen, die verschiedene Spezifikationstypen erlauben. Dabei sind *UML-Zustandsautomaten* und *UML-Aktivitätsdiagramme* geeignet, um allgemeine dynamische Modelle zu spezifizieren. *Sequenz-* und *Kollaborationsdiagramme* hingegen erlauben die Spezifikation einzelner Abläufe. Ein vollständige Beschreibung benötigt eine sehr große Anzahl solcher Diagramme. Alle Notationen sind geeignet, Aspekte von Parallelität und sequentieller Abhängigkeit auszudrücken.

Die UML kennt zwei Erweiterungmechanismen. Die eine ist die Entwicklung von *Stereotypen*. Durch Vererbungsmechanismen läßt sich so das Metamodell der UML erweitern. Hier wurden vor allem durch die Object Management Group selbst Erweiterungsstandards [**60**] erarbeitet, die eine Spezifikation von Performanz-bezogenen Eigenschaften erlaubt.

Weiterhin hat man die Möglichkeit, Elemente in einem UML-Modell direkt mit Werten (*tagged values*) zu versehen, für deren Interpretation man eine eigene Semantik vorsehen kann. Dazu ist es nicht nötig das UML-Metamodell zu erweitern.

In jedem Fall aber muss der Spezifikation eines UML-Modells ein weiteres Modell zugeordnet werden, das die quantitative Bewertung dann durchführt, da die Spezifikation der UML dazu keine ausreichenden Formalismen aufweist. Im Folgenden werden einige bekannte Modelle vorgestellt.

Quantitative Modelle

Von einigen dynamischen Modellen wie beispielsweise einem gewichteten Automaten lassen sich Werte ableiten, die man als quantitative Aussagen über Performanz-Eigenschaften interpretiert. Manchmal mag man unmittelbar in diesen Formalismen modellieren, oft sind aber die Spezifikationen dynamischer Modelle in einem weiteren Formalismus

festgehalten, in dem keine quantitative Bewertung durchgeführt werden kann. Dann wird aus diesen in geeignete quantitative Modell übersetzt. Im fogenden sollen einige bekannte Modelle angesprochen und auf die Probleme hingewiesen werden, die durch den hier vorliegenden Ansatz besser bewältigt werden sollen. Zum Verständinisse der Darstellung der Problematik werden dabei Kenntnisse des Lesers bezüglich der angesprochenen Modelle weitgehend vorausgesetzt.

Markov-Prozesse. Betrachtet werden so genannte Markov-Ketten. Eine solche Kette wird gebildet durch die Zustandsfolge eines stochastischen Prozesses, für den die Einschränkung erhoben werden muss den Folgezustand als nur vom aktuellen Zustand abhängig darzustellen. Die Forderung dieser so genannten Markov-Eigenschaft ist für die gute mathematische Handhabbarkeit entscheidend.

Kritik. Die dadurch bewirkte „Gedächtnislosigkeit" des Prozeßmodells ist zugleich der entscheidende Kritikpunkt. Die Abhängigkeit eines Prozesses von den Vorzuständen ist oft wesentlicher Teil der Fragestellung und kann mit diesem Formalismus schlecht ausgedrückt werden.

Warteschlangenmodelle. Warteschlangennetze sind zusammengesetzte Ressourcen [**54**]. Sie bestehen zunächst aus einzelnen Bedienstationen.

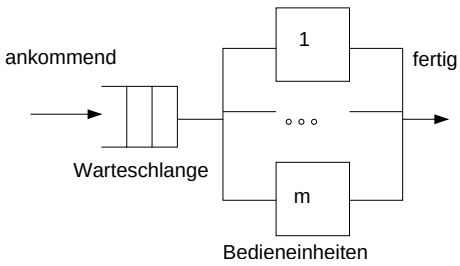


ABBILDUNG 1.2.1. Bedienstation

Abbildung 1.2.1 zeigt schematisch die Vorstellung von einer Bedienstation. Jede Bedieneinheit kann nur einen ankommenden Auftrag gleichzeitig bedienen. Nebenläufigkeit wird modelliert, indem die Bedienstation über mehrere Bedieneinheiten verfügt. Eine Bedieneinheit ist daher entweder *frei* oder *belegt*. Wichtige Charakteristika werden ausgedrückt durch die Zwischenankunftszeit t_δ und die Bedienzeit t_s, die jede Bedieneinheit für einen Auftrag benötigt. Damit ergeben sich die Ankunftsrate zu $r_a = \frac{1}{t_\delta}$ und die Bedienrate zu $r_b = \frac{1}{t_s}$. Da die Ankunft und Bedienung in gewissem Rahmen streuen bzw. Zufallscharakter haben, werden sie meist durch eine Verteilung ausgedrückt. Die

Auslastung der Bedieneinheit ist dabei durch $\eta = \frac{r_a}{r_b}$ gegeben. Weitere grundlegende Festlegungen müssen durch die sogenannte Warteschlangendisziplin getroffen werden. Mögliche Disziplinen sind beispielsweise *First-Come-First-Served* oder *Round-Robin*. Mit dem Durchsatz $d = m * \eta * t_b$ und einem Mittelwert der Anzahl an Aufträgen $\overline{k}$ ergibt sich die Antwortzeit für einen Auftrag gemäß $t = \frac{\overline{k}}{d}$.

Netze von Warteschlangen werden realisiert, indem man aus einer Menge N an Knoten deren Zusammenhang wie folgt betrachtet. Die Bearbeitung durch einen Knoten $i \in N$ und den Wechsel zur Warteschlange des Knotens $j \in N$ wird durch eine Wahrscheinlichkeit p_{ij}, mit der ein fertiger Auftrag von Knoten i zu Knoten j wechselt, modelliert.

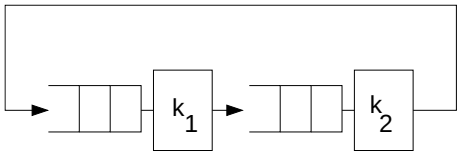


Abbildung 1.2.2. Geschlossenes Netz

Beispiel. Das einfache Warteschlangennetz in Abbildung 1.2.2 ist geschlossen und verfügt über zwei Knoten $N = \{k_1, k_2\}$. Man nimmt an, daß in dem Netz drei Aufträge kreisen. Die Permutation der möglichen Verteilungen ergibt vier Zustände.

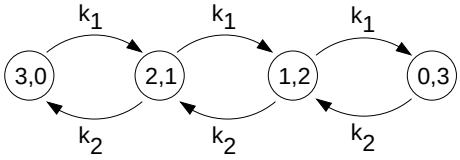


Abbildung 1.2.3. Zustandssystem der Warteschlange

Abbildung 1.2.3 zeigt das Zustandssystem mit den Übergängen. Die Mittelwerte der exponentiell verteilten Bedienzeiten seien $\overline{t_1} = \frac{1}{k_1}$ und $\overline{t_2} = \frac{1}{k_2}$. Damit sind $p(3,0) * k_1 = p(2,1) * k_2$, $p(2,1)(k_1 + k_2) = p(3,0) * k_1 + p(1,2) * k_2$, $p(1,2)(k_1 + k_2) = p(2,1) * k_1 + p(0,3) * k_2$ und $p(0,3)k_2 = p(1,2)k_2$. Die Generator-Matrix ergibt sich damit zu

$$Q = \begin{pmatrix} -k_1 & k_1 & 0 & 0 \\ k_2 & -(k_1 + k_2) & k_1 & 0 \\ 0 & k_2 & -(k_1 + k_2) & k_1 \\ 0 & 0 & k_2 & -k_2 \end{pmatrix}.$$

Kritik. Bereits an den Elementen einer Warteschlange läßt sich anmerken, daß nötige Annahmen und geforderte Restriktionen die Modellierung einschränken.

(1) Aufträge müssen gleichförmig sein.
(2) Größe und Anzahl der Aufträge haben nur Einfluß auf die Bedienzeit, wenn alle Bedieneinheiten besetzt sind.
(3) Die Disziplin der Warteschlange ist nicht flexibel.
(4) Aufträge sind nicht unterscheidbar.

Wegen (1) können keine Aufträge verschiedener Größe modelliert werden. Aus (2) resultiert eine sehr differenzierte, vorgegebene Modellvorstellung für die Realisierung von Nebenläufigkeit. Ein einfaches Ethernet-Netzwerk, beispielsweise, kann so nur durch große Umwege nachgebildet werden. Ressourcen sind durch (3) nur eingeschränkt modellierbar. Aufträge können keine Zwischenzustände ihrer Bearbeitung annehmen und wegen (4) nur anonym in die Warteschlange zurückgestellt werden (*Round-Robin*).

Für Modelle, die alle Einschränkungen erfüllen, sodaß die stationäre Verteilung berechnet werden kann, lassen sich interessante Kennzahlen ableiten. Diese sind beispielsweise die Auslastung, der Durchsatz, die mittlere Anzahl von Aufträgen und mittlere Antwortzeiten. Auf die Möglichkeiten wird hier nur auf die Literatur [**21, 19**] verwiesen. Auch und gerade für diese Technik wurden Übersetzungen aus UML-Notationen vorgeschlagen [**11**]. Kernpunkt ist wieder der Hinweis auf die Beschränkungen. Prozesse müssen einförmig sein, das Modell einer Bedienstation ist uniform vorgegeben, die Prozesse sind unabhängig und die Funktionen für die Bedienwahrscheinlichkeiten sind zustandslos. Außerdem können manche Ergebnisse nur bei gewissen Netzen erzielt werden. In unserem Beispiel war es ein geschlossenes Netz, in dem Aufträge kreisen. Alle Ansätze dieser Art haben ähnliche Einschränkungen. Die Anwendungsbeispiele zeichnen sich meist dadurch aus, daß durch eine implizite Einschränkung der Auswahl an Beispielen erst die wesentlichen Probleme vermieden wurden. Um die Eignung der Technik zu beurteilen, sind Vorbedingungen zu prüfen, die mehrere Aspekte gleichzeitig betreffen und eine tiefe Kenntnis der Quell- und der Zieldomäne erfordern.

Schließlich setzt eine Extraktion von praktischen Ergebnissen wieder eine Rücktransformation in die Begriffe der Quelldomäne voraus. Die ermittelten Ergebnisse sind nur dann sicher bewertbar, falls überprüft wurde, daß keine der zahlreichen Voraussetzungen verletzt wurden. Die Überprüfung der Plausibilität der Werte ist notwendig und hoch kompliziert. Werden einzelne Voraussetzungen nicht erfüllt, zB.

eine offenes anstatt ein geschlossenes System, oder ein inhärent periodisches Modell, muß der Formalismus gewechselt und eine Menge an anderen Voraussetzungen erörtert werden. Diese setzen wiederum andere Fachkenntnisse voraus.

Bildhaft gesprochen, ist in der Technik der Performanz-Modellierung durch Markov-Prozesse, der Formalismus in das Netz und die Matrix zerlegbar. Parameter und Funktionen, welche den zeitlichen Einfluß einer beschriebenen Ressource kennzeichnen, sind aber über beide Teile verstreut. Die Modellierung des dynamischen Systems ist ebenso in beiden Teilen zu finden. Sie spiegelt sich in dem Netzgraphen ebenso wider, wie in der Matrix des Markov-Prozesses. Der Grund ist einfach. Ausgangspunkt der Entwicklung dieser Technik war ein existierender Kalkül, der auf eine Problemstellung angewendet wird. Die Zerlegung bringt also keinen analytischen Gewinn, sondern zusätzliche Fehlerquellen durch erhöhte Redundanz.

Performance Evaluation Process Algebra - PEPA. Basierend auf einer klassischen stochastischen Prozessalgebra bietet *PEPA*, **[33, 34, 35]** die Möglichkeit Ausführungszeiten zu spezifizieren. Es werden Komponenten (*components*) und Aktivitäten (*activities*) betrachtet und man versteht ein System als Menge von interagierenden Komponenten, die Aktivitäten ausführen. Eine Aktivität $a \in \mathcal{A}$ ist dabei ein Tupel (α, r) aus einem Aktionstyp α und einer Aktionsdauer r, die als exponentiell verteilte Zufallsvariable gesehen wird. Dieser Wert kann auch unbestimmt sein.

Es wird die Vorstellung eingeführt, daß eine Aktivität *aktiviert* wird. Die Dauer der Ausführung ist bestimmt durch eine zugeordnete Funktion $F_a(t) = (1 - e^{rt})$. Ist diese Zeit verstrichen, betrachtet man die Aktivität als *beendet*.

Die PEPA-Sprache besteht aus zwei Primitiven (Komponenten und Aktivitäten) und mehreren Operatoren mit folgender Syntax:

$$P ::= (\alpha, r).P | P \bowtie_L Q | P + Q | P/L | X | A$$

Die Operatoren bedeuten im Einzelnen:

Prefix: $(\alpha, r).P$ bildet das Grundmuster der Verhaltensbeschreibung. Die Aktivität (α, r) beschreibt das zeitliche Verhalten der Komponente P. Dabei wird vorausgesetzt, daß Ressourcen immer zur Verfügung stehen, unbegrenzt sind und nicht explizit modelliert werden.

Choice: Die Ausführung einer aus P und Q zusammengesetzte Komponente $P + Q$ wird so verstanden, daß beide Aktivitäten gleichzeitig gestartet werden. In der Semantik ist definiert, daß sich beispielsweise $P + Q \rightarrow Q'$ ableiten läßt, falls die Aktivität, die P beschreibt, früher terminiert.

Cooperation: Mit Kooperation wird eine Synchronisation mehrerer Aktivitäten der kooperierenden Komponenten $P \bowtie_L Q$ ausgedrückt. Kooperierende Aktivitäten besitzen den gleichen Aktivitätstyp L und erhalten die jeweils langsamste Funktionsvariable der jeweiligen Aktivität. Auch hier wird für beide Komponenten die *uneingeschränkte* Verfügbarkeit jeweils *eigener* Ressourcen vorausgesetzt.

Hiding: Der Ausdruck P/L zeigt an, daß die Aktivitäten der Komponente L in der Komponente P verborgen.

Weiterhin werden Variablen X und Konstante A eingeführt. Abhängigkeiten verschachtelter Komponenten sind durch einen gerichteten Graphen darstellbar. Durch die Definition einer Semantik lassen sich Ausdrücke als Ganzes Auswerten und berechnen. Interessant ist an diesem Ansatz die leichte Übertragbarkeit von Spezifikationen aus Netzgraphen von QN-Modellen und einfachen UML-Diagrammen in PEPA. Dies wird unter anderem durch ein Werkzeug (PEPA-Workbench) [**34**] genutzt.

Kritik: Der bestechenden Eleganz und guten Handhabbarkeit der Methode stehen große Einschränkungen gegenüber. Die Berechnung der wahrscheinlichsten Raten eines beschriebenen Gesamtsystems ist einfach, zuverlässig, schnell und mit hohem Grad an Automatisierbarkeit möglich. Es ist ein großer Vorteil, daß Aktivitäten nicht (wie bei Markov-Prozessen) unabhängig und gleichförmig sind.

Jedoch stellen die Annahmen und Voraussetzungen erhebliche Einschränkungen dar. Ressourcen können nicht explizit modelliert werden. Daraus folgt, daß Nebenläufigkeit in Aktivitätsgraphen zwar ausgedrückt werden kann. Die Auswirkung einer geteilten Ressource auf die Bearbeitungszeitfunktion ist darstellbar, da Ressourcen immer und uneingeschränkt verfügbar sind. Eine Fragestellung für welches Modell sich eine schlechte Performanz durch strukturell bedingte, übermäßige Belastung einer Ressource ergibt, läßt sich nur schwer ausdrücken.

Stochastische Petrinetze - SPN. Ähnlich wie bei PEPA finden exponentiell verteilte Zeitvariablen hier als Annotationen an den Transitionen eines Petrinetzes Verwendung. Die eigentliche Ausführung der Aktion, repräsentiert durch das Feuern der Transition, wird als zeitlos betrachtet. Die Konzessionierung ist durch die Anschrift aber zeitabhängig.

Einige Ansätze (Balsamo, Mirandola, Mersenguer et. al.) [**9, 47**] verwenden diesen Formalismus unmittelbar, oder übersetzen beispielweise auch UML-Diagrammen in diesen Formalismus [**6, 7, 10**].

Kritik: Auch bei diesen Ansätzen sind keine Möglichkeiten gegeben, Abhängigkeiten der Wahrscheinlichkeitsfunktionen von dem Belastungszustand gemeinsam genutzter Ressourcen zu betrachten. Auch bleibt die Gewinnung der Zufallsvariablen ein ungelöstes Problem.

Palladio-Component-Model. Der neueste Ansatz geht den Weg der Simulation zeitlichen Verhaltens. *Reussner et al.* [**18, 41**] führen ein Komponentenmodell (*Palladio*) ein, welches es erlaubt, Verhalten einer Komponenten-basierten Architektur in einem Simulator zu bewerten. Dieser interessante Ansatz enstand parallel bzw. zeitlich nach den Grundlagen zu dieser Arbeit und ist wohl durch ähnliche Kritik an anderen Techniken motiviert. Die Ausdrucksfähigkeit des Modells ist auf genau definierte Archtitektur-Elemente beschränkt und eignet sich nach eigenen Beispielen besonders zur Simulation und Bewertung von Software, die auf passenden Komponentenarchitekturen basieren, wie beispielsweise EJB- und Webservice-Anwendungen.

Methodik

Da alle vorgestellten Techniken den Weg der Modellbildung beschreiten, ist eine Integration dieser Techniken in die Praxis in einem Entwicklungsprozess zu erwarten, der mindestens Phasen von *Analyse*, *Entwurf*, *Implementierung* und *Test* unterscheidet.

Eine zusammenfassende Übersicht liefert der technische Bericht *Software Performance: state of the art and perspectives* von S. Balsamo, et. al. [**4**] und führt eine verfeinerte Sicht auf die Phasen ein.

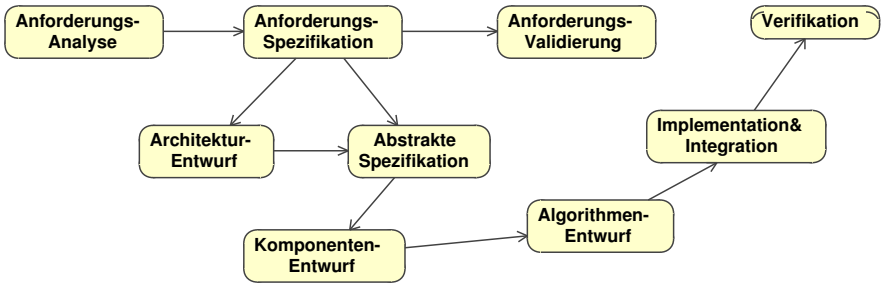


ABBILDUNG 1.2.4. Projektphasen

In dieser Übersicht ist zu sehen, daß unterschiedliche Techniken in einzelnen Phasen gut zu verwenden sind. Keine Technik eignet sich zur

Begleitung der Entwicklung durch den gesamten Prozeß. Das PEPA-Modell und die verfügbaren Werkzeuge wie etwa die PEPA-Workbench zeigen noch eine gute Kontinuität von den ersten Studien bis zum Komponenten-Entwurf. Eine weitreichende Methodik, die jedoch auch auf die frühen Projektphasen beschränkt ist, liefern *Cortellessa* und *Miradola* mit PRIMA-UML [**25**].

Für den Entwurf von Datenstrukturen und Algorithmen, der Implementierung, dem Deployment, dem Test und der Verifikation bieten die gängigen Techniken nicht durchgängig ausreichende Ausdruckskraft. Aus den bereits vorgebrachten Kritikpunkten war zu entnehmen, daß eine Beschränkung beispielsweise auf die Darstellung durch azyklische Graphen, die Vereinheitlichung der Aktivitäten, die Voraussetzung deren Atomizität und Unabhängigkeit und schließlich den Ausschluß der Modellierung von Ressourcen, deren Verhalten von ihrem Lastzustand abhängig und variabel ist, in Kauf genommen werden muß.

Diese Beschränkungen machen eine Darstellung der Entwurfsentscheidungen bereits für einen Implementations-Entwurf problematisch. Spätestens bei Fragen des Deployments ist ein weiterer Einsatz ausgeschlossen.

Zusammenfassung. Aus einem pragmatischen Blickwinkel sieht man eine Weiterentwicklung der Ausdruckskraft der Modelle, von den schon lange bekannten und verwendeten Markov-Ketten, die azyklische Graphen durch Warteschlange mit unabhängigen gleichförmigen Prozessen untersuchen, über PEPA, in dem Komponenten verschiedenartige Aktivitäten aufweisen können, die über eine Art Typsystem in der Nebenläufigkeit aufeinander reagieren, bis hin zu stochastischen Petrinetzen, in denen Zyklen nicht ausgeschlossen sind und auch die Aktivitäten durch verschiedene Funktionen spezifiziert werden können.

Viele Techniken zeichnen sich durch mathematische Eleganz und eine mit kalkulierbarem Aufwand erbringbaren Entwurfsleistung aus. Sie sind leistungsfähig bezüglich ihrer Ausdruckskraft zu den Gesamtmodellen. Die Vorhersagequalität der Laufzeiten einzelner Abläufe sind nicht beliebig skalierbar. Auch ist keine Technik in Sicht, die über in allen Phasen eines Entwicklungsprozesses benutzt werden kann.

Will man komplexe Funktionen zulassen, um beispielsweise die Abhängigkeit der Antwortzeiten von Ressourcen von dem Gesamtzustand des Systems auszudrücken, werden mathematische Modelle, die alle Fragestellungen in einem Ansatz lösen schnell komplex.

1.3. Unser neuer Ansatz

Eine simulationsbasierte Technik

In der Anforderungsanalyse werden die Performanz-Eigenschaften informell eingefordert und dokumentiert. Bereits im ersten Entwurf werden Eigenschaften spezifiziert, die das zeitliche Verhalten entscheidend beeinflussen können. Die technische Umsetzung durch die Implementierung löst die Aufgabenstellung des Entwurfes und in der Testphase kann am laufenden Programm zum ersten Mal das zeitliche Verhalten authentisch beobachtet und gemessen werden.

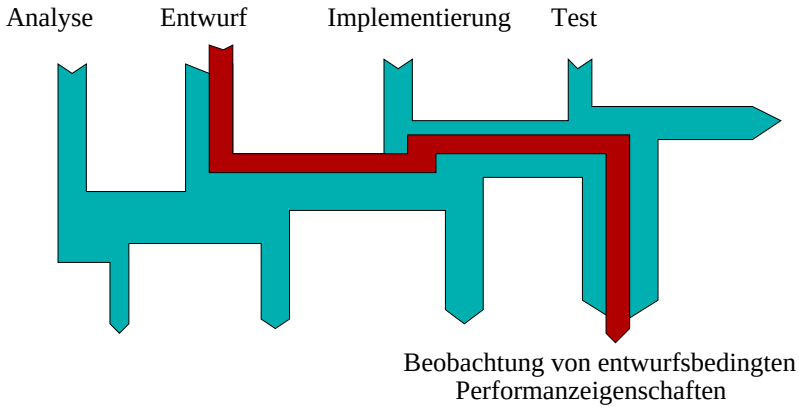


ABBILDUNG 1.3.1. Entwurfsabhängige Performanzeigenschaften

Frühe Bewertung durch Mikrozyklen. In Abbildung 1.3.1 ist ein Lastenfluß-Schema im Projektablauf dargestellt. Die Breite der einfließenden Ströme soll dabei die Einbringen von zu bewältigenden Arbeitslasten schematisch quantifizieren. Beispielsweise sind dies Anforderungen in die Analyse, die zu erfüllen sind. Der den Fluß verlassende Strom soll diejenigen Lasten symbolisieren, die in der Phase abschließend bewältigt wurden. In der Analysephase kann dies beispielsweise in der Identifikation von redundanter Funktionalität bestehen, die auf sprachlicher Basis bereits in den Interviews geleistet wird. In der Entwurfsphase können durch Spezifikation geforderter, struktureller Eigenschaften, oder der Nachweis eines konsistenten Entwurfes bereits Ergebnisse erreicht werden.

Der in Abbildung 1.3.1 hervorgehobene Strom an Arbeitslast hebt eine Performanz-Eigenschaft hervor, die durch Bestandteile des Entwurfes in das Projekt eingebracht wurde. Hier wird sichtbar gemacht, daß die Eingenschaft weder an den Entwürfen sichtbar wird, noch in

der Implementierung entdeckt, sondern erst in der Testphase beobachtet werden kann. Erst nach der Implementierung des Entwurfes und der Durchführung der Tests ist es möglich, die Performanz-Eigenschaften zu beobachten und zu messen. Eine abschließende Beurteilung vermag sie dann, im Erfolgsfall, wieder aus dem Arbeitsfluß entfernen. Gelingt dies nicht müssen unzureichende Eigenschaften auch über mehrere Iterationen mitgeführt werden. Könnten bereits Modelldokumente zur Bewertung der Performanz-Eigenschaften herangezogen werden, wäre es möglich, viele Eigenschaften innerhalb einer Entwurfsphase einer Iteration durch Microzyklen sicherzustellen.

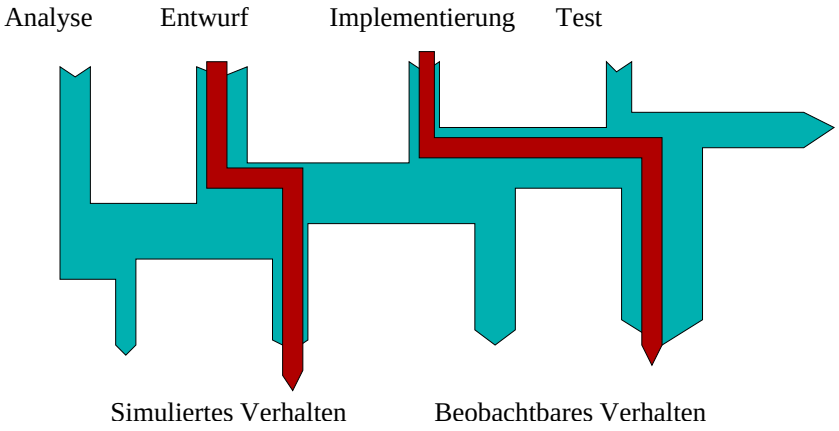


ABBILDUNG 1.3.2. Modellsimulation

Abbildung 1.3.2 zeigt die Auswirkung der möglichen Bewertung von Modellen auf den Lastfluß im Idealfall. Die performanzrelevanten Eigenschaften, die durch den Entwurf bestimmt werden, können bereits während des Entwurfes quantitativ abgeschätzt und optimiert werden. Die Performanz-Eigenschaften, die durch technische Umsetzung bestimmt sind, werden separat im Test identifiziert. Auswirkungen der Implementierung auf die Performanz könnten zusätzlich durch ein gesondertes Implementierungsmodell erfasst werden, was in der Graphik nicht dargestellt ist.

In der Praxis wird sich ein solcher Mikrozyklus nur leicht etablieren lassen, falls genau die Modelle, die funktionale Eigenschaften ausdrücken, bewertet werden können. In den Projekten, auf die diese Arbeit zielt, sind dies meist UML-Dokumente, die dynamische Modelle spezifizieren, also Aktivitätsdiagramme oder Zustandsautomaten. Die Verwendung von Kalkülen, welche die Überführung in einen anderen Formalismus notwendig machen, setzen diesem Mikrozyklus einen oft entscheidenden, kostenintensiven Widerstand entgegen.

Ein Ansatz zur Modellbewertung. Aktivitätsdiagramme der UML 1.X Versionen erfreuen sich in der Praxis großer Beliebtheit, weil die ausgedrückte Abstraktion dem algorithmischen Denken einer späteren Implementierung am ähnlichsten ist. In der gewünschten Werkzeugimplementierung wird versucht, Aktivitätsdiagramme möglichst unmittelbar zu bewerten. In diesem Ansatz wird die Beurteilung durch Simulation gewählt. Es werden lediglich einzelne Abläufe eines Modells, als eine Art von Testdurchführung, bewertet. Dies erlaubt ausdrucksstärkere Modelle zuzulassen. Aussagekräftige Testfälle können aus den in der Anforderungsanalyse dokumentierten Geschäftsfällen entnommen werden. Somit stellt die Beschränkung der Bewertungsmöglichkeit auf Testfälle meist keinen wesentlichen praktischen Nachteil dar.

Wohlgeformte, dynamische Modelle. Ausführliche Untersuchungen [**13**] haben gezeigt, daß die unmittelbare Verwendung eines implementierten UML-Modells aufgrund der ungenauen Semantik problematisch ist. Beispielsweise ist eine Verwendung der *Fork*- und *Join*-Knoten derart erlaubt, daß Modelle möglich sind, bei denen die Multiplizität einzelner Aktivitätsinstanzen vom jeweiligen Ablauf abhängig ist. Differenzierter Umgang mit den einzelnen Instanzen aber ist nicht vorgesehen. Ein Modell wird eingeführt, welches einen Wohlgeformtheitsbegriff definiert, der genau die in diesem dynamischen Sinne eindeutig interpretierbaren Modelle zuläßt. Dies wird unter anderem durch Einführung nebenläufiger und sequentieller Regionen erreicht. In Kapitel 2 wird ausführlich auf die Menge der wohlgeformten, dynamischen Modelle und ihre strukturellen Eigenschaften eingegangen. Dann wird eine formale Semantik eingeführt, welche das dynamische Verhalten durch Abbildung auf ein Petrinetz [**17, 51, 58**] exakt definiert. Die Wahl dieser semantischen Domäne bringt den Vorteil, daß Petrinetze unmittelbar implementiert werden können [**52, 20, 63, 64**] und dies die korrekte Umsetzung in ein Werkzeug später erleichtert. Die Ausdrucksmächtigkeit der Modelle muß dabei einer späteren, praktischen Umsetzung Rechnung tragen. Beispielsweise wird die Endlichkeit des Erreichbarkeitsgraphen für eine Werkzeugiplementierung eine wichtige technische Eigenschaft sein.

Ablaufspezifikation. Die Menge möglicher Testfällen wird vom jeweiligen dynamischen Modell vorgegeben. Der hier entwickelte Entwurf für dynamische Modelle bereitet bereits eine gute Handhabung der Theorie dieser Testfälle (hier *Szenarien* genannt) vor. In Kapitel 3 wird ein Begriff für Szenarien definiert und in die vorhandene Technik eingebettet. Einzelne Abläufe stellen sich hier als konkrete Schaltseqenzen, bzw. Feuersequenzen (je nach Sprachgebrauch der Literatur) eines

Petrinetzes dar. Die in den 80er und 90er Jahren ausgearbeiteten Spurtheorie [**39, 43, 1, 44, 45, 2, 26, 27**] und die Theorie der endlichen asynchronen Automaten [**53, 65, 66**] bieten eine gute Basis für eine theoretische Einbettung. Nach einer kurzen Einführung in die Begriffe dieser nicht mehr so gebräuchlichen Theorie wird ein exakter Begriffe einer Spurgrammatik definiert, die eine Feuersequenz eines Modells genau soweit spezifiziert, daß eine Simulation automatisch durchgeführt werden kann. Auch diese theoretische Einbettung leistet neben einer exakten Definition der verwendeten Begriffe die nötigen Vorarbeiten für eine effiziente Umsetzung in ein Werkzeug.

Virtuelle Umgebung. Zeitliches Verhalten zeigt jede Software natürlich nur in einer ausführenden Laufzeitumgebung. Demgemäß ist die Beurteilung eines dynamischen Modells nur in der Gegenüberstellung mit einer virtuellen Umgebung möglich. In einem, im Jahre 2003 von der OMG verabschiedeten Profil *Schedulability, Performance and Time* [**60**] wird eine sehr gegenständliche Beschreibungsmöglichkeit für solche Umgebungen geschaffen. Diese bietet auch eine abweichende Verwendung der Aktivitätsdiagramme der UML [**62**] und ein konkretes Zeitmodell. Diese Definition stellt aber auch eine Festlegung dieser Begriffe dar. In der vorliegenden Arbeit sollte diese Festlegung aber durch den Modellierer frei bleiben. Soweit frei, daß der Begriff Ressource beispielsweise sogar auf einen Arbeiter im Prozeß selbst angewendet werden kann und somit das Bewertungsverfahren sogar eine Reflektion über die Performanz des Software-Entwicklungsprozesses erlaubt. Die Begriffe des Profils für *Schedulability, Performance and Time* wurden deshalb bewußt nicht angewendet.

Der Ressourcen- und des Zeitbegriff soll in den Modellen gestaltet werden. Als *Ressource* wird vorab nur eine abstrakte Einheit aufgefaßt, die mehrere *Lasten* gleichzeitig bearbeiten kann. Die konkrete Beschreibung erfolgt für jeden einzelnen Last- und Ressourcentyp getrennt. Die Beschaffenheit des Modells und dessen Abstraktionsgrad soll frei gewählt werden.

Eine solche Ressource muß nur durch die Geschwindigkeit beschrieben werden, in der diese Bearbeitung fortschreitet. Das reduziert die Beschreibung auf zwei Funktionen. Die Eine beschreibt die benötigte Zeit für die Bearbeitung einer Last und die Andere schreibt den Zustand der Ressource mitsamt ihrer Lastmenge fort. In Kapitel 4 werden diese Funktionen exakt definiert und am Beispiel verdeutlicht. Die konkrete Angabe solcher Funktionen kann auf Schätzungen, Messungen oder Ableitung von Modellen beruhen und begründet somit die Anwendbarkeit des Schemas in allen Phasen eines Software-Entwicklungsprozesses. Die Bedeutung der Modellelemente, zeitlichen

Funktionen und Maßeinheiten werden also ausserhalb des Werkzeugs definiert und nicht in die Definition des Metamodells hineingetragen. Sie müssen also auch bei der Auswertung der Ergebnisse adäquat angewendet werden.

Laufzeitsemantik. Mit jeweils einem wohlgeformtem, dynamischen Modell, einer Szenariospezifikation und einer virtuellen Umgebung lassen sich die Dauer dieses Ablaufes simulieren. In Kapitel 5 wird eine Laufzeitsemantik [**16**] zu dieser zeitlichen Interpretation definiert, die eine eindeutige Auswertung eines solchen Tripels erlaubt.

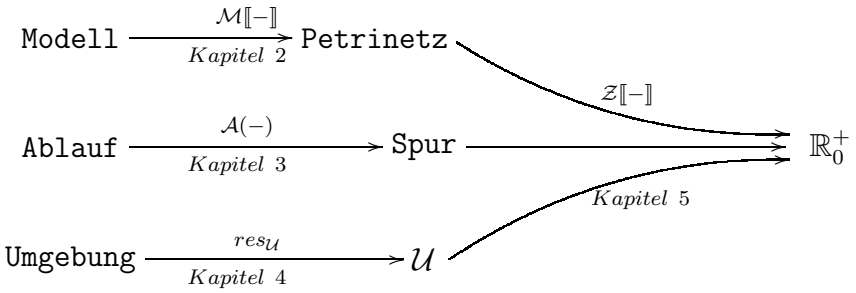


ABBILDUNG 1.3.3. Schematischer Simulationsablauf als Inhalts- und Themenaufteilung des ersten Teils

Jedes Tripel führt demgemäß zu einem Wert für die benötigte Bearbeitungszeit eines Ablaufs. Abbildung 1.3.3 zeigt im Überblick, wie die im ersten Teil der Arbeit definierten Funktionen zu einer zeitlichen Bewertung eines Ablaufes führen.

Methodik.

Eine Fallstudie, welche die Vorteile dieses Ansatzes belegt, kann sich nicht auf ein Rechenbeispiel einer Modellbewertung beschränken. Der Ansatz strebt eine in den gesamten Software-Entwicklungsprozeß integrierbare Technik an, sowie eine Integrierbarkeit in die aktuell in der Praxis verwendeten Modellierungsmethoden und einen Entwurf, der die Entwicklung eines Werkzeuges begünstigt.

Die versprochenen Vorteile werden im zweiten Teil dieser Arbeit untersucht. Einige Grundbegriffe des Software-Entwicklungsprozesses, so wie sie im *Unified Software Development Process* [**40**] eine große Verbreitung gefunden haben, werden in Kapitel 6 aufgegriffen. Aus der Umsetzung des Schemas in Abbildung 1.3.3 in der Praxis ergeben sich neue Artefakte. Es handelt sich dabei um dynamische Modelle mit speziellen Performanz-Annotationen. Diese sollen von Modellen

abgeleitet werden, die auch die funktionalen Eigenschaften spezifizieren. Weiterhin werden Abläufe, also die Szenario-Spezifikationen, eingeführt. Eine Ableitung von Sequenzdiagrammen böte sich hier an. Aufgrund deren Detailreichtums wären diese aber oft zu umfangreich. Abläufe, die typische Fälle funktionaler Aspekte kennzeichnen sind nicht notwendig identisch mit den Abläufen, die kritische Eckpunkte zeitlichen Verhaltens markieren. Durch eine Anpassung der Granularität der Sequenzdiagramme wäre dieser Diagrammtyp zwar verwendbar, aber die visualisierten Eigenschaften beinhalten keinen Erkenntnisgewinn. Deshalb haben wir uns entschieden eine eigenständige Notation für diesen Artefakttyp vorzusehen. Die Umgebungen schließlich entsprechen den Implementationsdiagrammen der klassischen Modelle, die um Ressourcenmodell-Annotationen erweitert werden. Diese müssen mit den adäquaten Meßwerten und Performanzfunktionen korrespondieren. Dieser Zusammenhang und die Performanzfunktionen selbst müssen in bislang nicht verwendeten eigenen Artefakten verwaltet und dokumentiert werden.

Diese neuen Artefakte werden den klassischen Artefakten gegenübergestellt und deren Zusammenhänge und Abhängigkeiten identifiziert. Eine problemlose Anwendbarkeit der im ersten Teil entwickelten Begriffe auf die Notationen der UML zeigt in Kapitel 7 unter Abschnitt 7.1 die Übersetzung der Aktivitätsdiagramme und Implementationsdiagramme in den im ersten Teil vorgestellten Formalismus.

Die neuen Aktivitäten, die sich aus dem Ablauf des Prozesses ergeben, werden skizziert und zu neuen Rollenbeschreibungen zusammengefaßt. Dies sind beispielsweise der Performanz-Analytiker, der Ressouren-Designer, der Performanz-Modellierer, der Szenario-Entwickler oder der Simulations-Ingenieur. Die notwendigen Fachkenntnisse und Tätigkeiten der Akteure, die diese Rollen besetzen sollen, werden herausgearbeitet. Die neuen Rollen werden den klassischen Rollen in einem Prozeß, der Analyse, Entwurf, Implementierung und Test unterscheidet, gegenübergestellt und eine nahtlose Integration gezeigt [**14, 15**].

Die erwünschte Eignung des Ansatzes zur effizienten Entwicklung eines Werkzeuges wird im Kapitel 7 im Abschnitt 7.2 demonstriert. Hier wird ein Architektur-Vorschlag aus den Definitionen abgeleitet, aus dem sich ein Komponentenmodell für ein Werkzeug ableiten läßt. Der dort vorgestellte Entwurf erlaubt nicht nur eine Implementierung einer Simulations- und Auswertungssoftware, sondern auch eine Umsetzung derart, daß die in Kapitel 6 erarbeiteten Rollen durch die Schnittstelle

des Werkzeuges gestützt werden. In den Jahren 2005 bis 2007 wurde am Lehrstuhl für Programmierung und Softwaretechnik der LMU München die Software |SEMPER|, als Prototyp eines Werkzeuges, entwickelt **[13, 48, 49, 29, 30, 12, 56, 46, 55]**. Dieser Prototyp belegt anschaulich den Erfolg dieses Ansatzes und demonstriert die problemlose Umsetzbarkeit in einem Entwicklungswerkzeug.

Trotz der großen Komplexität der Zusammenhänge zwischen den drei Simulationselementen erlaubt das Werkzeug eine intuitive Bedienung. In Kapitel 7, Abschnitt 7.3 ist dies an einem einfachen Modellierungsbeispiel, das durch das Werkzeug bewertet wurde, leicht nachvollziehbar. Eine Bewertung eines Modells wird besonders anschaulich, wenn mehrere Simulations-Ergebnisse zu Gruppen zusammengestellt und diese graphisch aufbereitet werden können. Das führt schließlich noch zu einer Diskussion um die Anwendungsmöglichkeiten dieser Technik. Grundlegende Klassen von Fragestellungen in Bezug auf den Entwurf und ihre methodische Behandlung werden herausgearbeitet.

Performanz ist oft auch ein wirtschaftlicher Risikofaktor. Ebenso wie Funktionalität, die nicht erreicht wird, kann eine unzureichende Performanz einen Projektfehlschlag bedeuten. Damit ergeben sich für den Einsatz dieser Technik und Methodik nicht nur inhaltliche Fragen. Die Kosten der Modellierung und Bewertung selbst werden zum Thema und müssen bei wichtigen Management-Entscheidungen in die Bewertung miteinbezogen werden. Eine Diskussion des Managements risikobehafteter Prozesse **[14, 15]** in Kapitel 8 schließt diese Arbeit ab. Diese Diskussion arbeitet Begriffe und Maßzahlen heraus, die nur in einem vollständig gemanagten Prozeß des höchsten Reifegrades **[28]** zugänglich sind. Da sich nur wenige Unternehmen weltweit dieses Reifegrades rühmen können, öffnet diese Diskussion ein Themengebiet, das den Blick des Lesers mit Spannung in die Zukunft entläßt.

Zusammenfassung. Ausgangspunkt für den Ansatz dieser Arbeit, und damit ein zentrales Alleinstellungsmerkmal, bildet die gegenständliche Vorstellung von den zu modellierenden Bestandteilen: Dynamische Modelle und deren Abläufe, sowie Ressourcen und deren Lasten. Dafür werden jeweils getrennte Modellierungstechniken entwickelt, welche die interessanten Aspekte auch wirklich separat behandeln. Die Modellvorstellungen sind praxisnah und intuitiv leicht verständlich. Somit erlaubt die Technik eine leichtere Plausibilitätsprüfung, bessere Anwendbarkeit und Interpretation der Ergebnisse. Eine benutzerfreundliche Integration in ein Werkzeug, das den Benutzer von der Beschäftigung mit fundamentalen theoretischen Wissen um die verwendeten Kalküle befreit, rundet diesen Ansatz ab.

KAPITEL 2

Dynamische Modelle

2.1. Grundlagen

Da die Notation der UML weitläufig bekannt ist, motivieren wir das im Anschluß eingeführte dynamische Modell durch eine Reihe von Mustern, die wir als Aktivitätsdiagramme darstellen. Es werden Begriffe berührt wie die des letzten gemeinsamen Vorgängers (*least common ancestor/LCA*), der Wohleinbettung (*well nestedness*), der Bereiche (*orthogonal regions*) und der korrespondierenden Synchronisationsbalken (*corresponding join*). Die Grenze zwischen problematischen und unproblematischen Mustern soll dabei verdeutlicht werden.

Muster werden genau dann problematisch, wenn einer Spaltung des Kontrollflusses nicht eindeutig eine korrespondierende Verschmelzung zugeordnet werden kann. Wir werden zeigen, daß eine explizite Unterscheidung von nebenläufigen und sequentiellen Regionen, welche die UML nicht vorsieht, hilfreich ist.

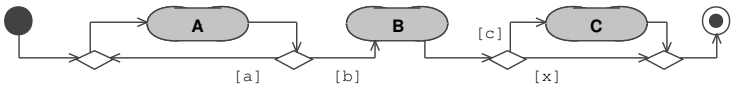


ABBILDUNG 2.1.1. Kontrollfluß ohne Nebenläufigkeit

2.1.1. Einfache Muster.

2.1.1.1. *Unverzweigter Kontrollfluß.* Im einfachsten Fall ist ein Kontrollfluß, wie in Abb. 2.1.1 dargestellt, außer den Aktivitätszuständen (*activity states*) nur aus Knoten aufgebaut, an denen sich der Aktivitätsgraph verzweigt (*decisions*). Nach der UML Spezifikation werden die durch Rauten dargestellten Verzweigungspunkte auf Pseudozustände (*junction states*) abgebildet. In Graphen dieser Art werden also nur Sprünge und Schleifen dargestellt. Während der Ausführung ist genau ein Element aktiviert. Sind an allen Verzweigungen die Bedingungen eindeutig auswertbar, oder werden dort entsprechende Entscheidungen getroffen, kann der Graph zu einer Sequenz von Aktivitätszuständen entfaltet werden. Einschränkung: Dies kann technisch nur umgesetzt

werden, falls keine Bedingungen (potentiell mögliche) endlose Schleifen formulieren.

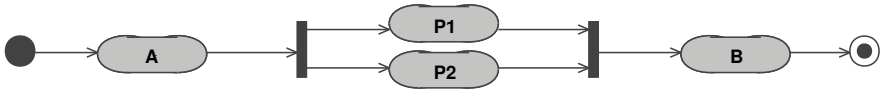


ABBILDUNG 2.1.2. Nebenläufigkeit

2.1.1.2. *Verzweigter Kontrollfluß.* Nebenläufigkeit wird in Aktivitätsgraphen durch Synchronisationsbalken ausgedrückt, die in Pseudozustände zur Verzweigung (*fork vertex*) und zur Verschmelzung (*join vertex*) abgebildet werden. Wenn auch nur ein graphisches Symbol verwendet wird, ist im Metamodell [**62**] die Eindeutigkeit von *fork* und *join* gefordert. Weiterhin wird dort festgelegt, daß die aus einem *fork* ausgehende Transitionen in voneinander verschiedene Regionen weisen müssen und Transitionen, die in einen *join* münden, aus verschiedenen nebenläufigen Regionen entspringen müssen. Solche Regionen werden graphisch nicht dargesellt und bleiben auch weitgehend unklar.

Für jeden *fork* wird ein korrespondierender *join* und die Wohleinbettung dieser Paare gefordert, ohne daß diese Begriffe genau definiert werden. Legen wir diese Forderungen streng aus, erhalten wir eine Struktur, die einer mathematischen Klammerung entspricht. Das einfache Beispiel in Abb. 2.1.2 stellt zwei Aktivitätszustände P1 und P2 dar, die in nebenläufige Bereiche eingebettet nach der Aktivität A ausgeführt werden und beide vor Beginn der Aktivität B terminieren müssen. Durch die Einführung eines Konkatenationsoperators $\circ$ und eines Parralleloperators $\|$ kann dieser Sachverhalt durch den Ausdruck $\mathsf{A} \circ (\mathsf{P1} \parallel \mathsf{P2}) \circ \mathsf{B}$ dargestellt werden. Graphen, die durch solche Terme dargestellt werden können, sind gut handhabbar und würden zur Verwendung anderer Methoden zu einer Performanz-Bewertung einladen, die auch allgemeinere Aussagen zuließen. Oft sollen in Modellen aber beliebig verzweigte Netze dargestellt werden. Dies wäre durch diese Einschränkung ausgeschlossen.

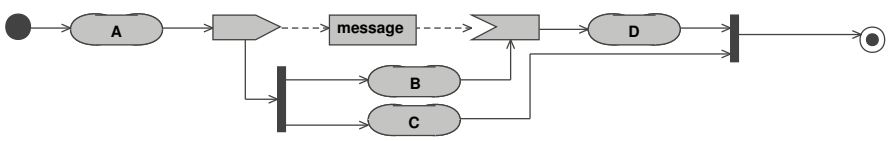


ABBILDUNG 2.1.3. Asynchrone Kommunikation und Events

2.1.1.3. *Asynchrone Kommunikation und Events.* Kommunikation über das Erzeugen und Empfangen von *Events* wird gewöhnlich als Annotation an Transitionen spezifiziert. Für Aktivitätsgraphen werden konkave und konvexe Pentagone als graphische Repräsentation für das Senden und Empfangen von Nachrichten oder Ereignissen eingeführt.

In Abb. 2.1.3 wird nach der Terminierung des Aktivitätszustands A ein Objekt (*message*) erzeugt und vor der Aktivierung des Zustandes D wieder empfangen. Wir thematisieren dieses Musterbeispiel, da eine Transition (hier ist eine gestrichelte Linie vorgeschrieben) offensichtlich zwei unterschiedliche nebenläufige Bereiche verbindet. Wenn wir diesen Fall ausschließlich als Datenfluß spezifizieren, betrachten wir ihn nicht als Verletzung der Wohleinbettung. Nur die durchgezogene Linie repräsentiert in diesem Fall den Kontrollfluß. (In Abschnitt 2.1.1.4 repräsentiert die gleiche gestrichelte Linie unglücklicherweise sowohl Datenfluß als auch Kontrollfluß und würde die Trennung der Regionen verletzen.)

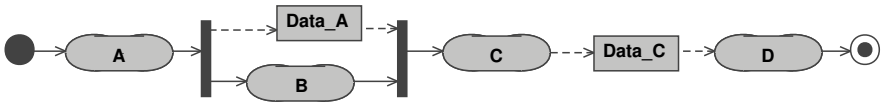


ABBILDUNG 2.1.4. Datenfluß

2.1.1.4. *Datenfluß.* Werden die in einem Aktivitätsgraphen durch eine Aktivität erzeugten Daten, die von einer folgenden Aktivität verwendet werden, unmittelbar dargestellt, so muß nach der Spezifikation (UML 1.X)[1] der durchgängige Kontrollflußpfeil unterdrückt werden. Abb. 2.1.4 zeigt Varianten dieses Datenflusses. Wir betrachten Daten im Sinne der Performanz-Bewertung ebenso als Last und machen keinen wesentlichen Unterscheidung zu den Aktivitätszuständen. Wir behandeln in diesem Beispiel den Aspekt des Kontrollflusses und die Transitionen gleich den durchgehenden Pfeilen, die sonst den Kontrollfluß darstellen.

2.1.2. Erweiterte Muster.

2.1.2.1. *Synchronisation.* Bei vielen Beispielen in der Literatur (zum Beispiel auch bei Arbeitsprozeßbeschreibungen) finden wir eine Synchronisation, wie in Abb. 2.1.5, die nicht durch die *SynchStates* der UML [**62**] darstellbar sind.

[1]Die Arbeiten zu diesem Thema fanden im Jahre 2004 statt. Die UML 2.0 lag zu dieser Zeit immer noch nicht in offiziell verabschiedeter Version vollständig vor. Somit werden in Folge nur die Diagramme nach der UML 1.X betrachtet.

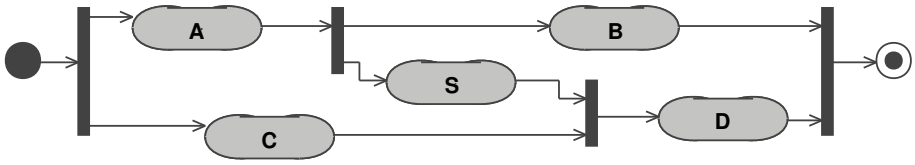


ABBILDUNG 2.1.5. Teilweise Verschmelzung nebenläufiger Kontrollflüsse

Die Aktivität D kann erst aktiviert werden, wenn die Aktivitäten A, C und S terminiert sind[2]. B kann dagegen direkt nach dem Terminieren von A aktiviert werden. Auch hier synchronisiert die Region, zu der A und B gehören, die Region von C und D, umgekehrt aber nicht. Die Aktivität S ist in diesen Synchronisationsvorgang irgendwie integriert. Dieses Beispiel ähnelt bis auf den Aktivitätszustand S den *sync-states*-Beispiel der UML-Spezifikation. Auch behaupten wir an dieser Stelle, daß die Interpretation dieses Graphen intuitiv unproblematisch ist. Bezüglich der Begriffe der korrespondierenden Synchronisationselemente und der Wohleinbettung ist jedoch etwas Aufwand zu betreiben.

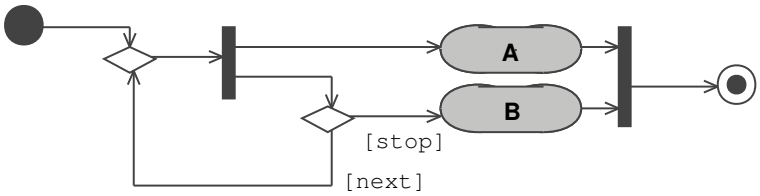


ABBILDUNG 2.1.6. Problematische Schleifen

2.1.3. Poblematische Muster. Einige Modelle sollen nicht akzeptiert werden. Im Folgenden geben wir Beispiele typischer Muster, die verschiedene Probleme bei der Interpretation verursachen.

2.1.3.1. *Schleifengebundene fork-Knoten.* In Abb. 2.1.6 wird ein Graph thematisiert, der nebenläufige Regionen unterschiedlicher Identität verbindet. Als typisches Beispiel dient hier die Einbindung eines *fork*-Knotens in eine Schleife. Derselbe *fork*-Knoten kann aus einer der nebenläufigen Regionen mehrfach erreicht werden. Eine durch die Schleife bestimmte Zahl an Instanzen des Aktivitätszustands A kann erzeugt werden und es bleibt unbestimmt, welche terminierten Instanzen der Aktivität A mit welcher Instanz der Aktivität B synchronisiert werden.

[2]Dabei werden fehlerbehaftete und fehlerlose Ausführung nicht unterschieden

Das Muster in Abb. 2.1.7 wird in der Spezifikation der UML 2.0 diskutiert.

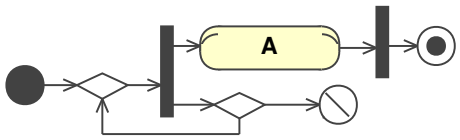


ABBILDUNG 2.1.7. Aktivitätsgraph-Beispiel aus der UML 2.0 Spezifikation

Zusätzlich wird ein *consume*-Zustand eingeführt, der nur mit dem neuen Begriff des *Tokens*, welches den gerade in Ausführung befindlichen Zustand markiert, verstanden werden kann. Tokens entstehen in einer solchen Schleife nicht zwangsläufig gleichzeitig. Einige können am *join* warten, während das Flußende in der Schleife noch nicht erreicht ist. Dann setzt eine eindeutige Interpretation des Verhaltens ein sehr komplexes Identitätsmodell eines solchen Tokens voraus. Ein Tokenbegriff analog zu Petrinetzen löst das Problem nicht. Die Begriffsbildung in der UML 2.0 Spezifikation bleibt zumindest für den Aspekt der Performanzeigenschaften unzureichend. Deshalb werden Graphen, die solche Strukturen enthalten, zurückgewiesen.

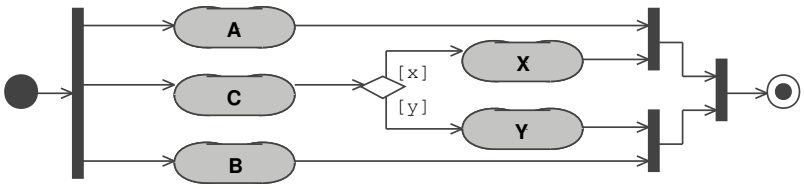


ABBILDUNG 2.1.8. Alternative Join-Knoten

2.1.3.2. *Alternative Join-Knoten.* Abb. 2.1.8 zeigt eine Möglichkeit wie ein Graph eine Auswahlmöglichkeit zwischen zwei Synchronisationsbalken spezifiziert, die zu einer Verklemmung führt. Auch wenn nicht alle Graphen dieser Art notwendig zu einer Verklemmung führen, weisen wir Graphen, die Strukturen der Art von Abb. 2.1.6 enthalten, zurück.

2.2. Dynamisches Kontrollflußmodell

2.2.1. Einführung der Modellelemente. Das dynamisches Modell einer Implementierung nach unserer Lesart benennt die Aktivitäten, die im Verlauf des Programmes ausgeführt werden, die Daten, die erzeugt und verbraucht werden, und spezifiziert deren Reihenfolge, Abhängigkeiten und Nebenläufigkeiten durch einen geeigneten Graphen. Die Aktivitäten, Daten und andere Elemente selbst müssen als *Lasten* nur insofern spezifiziert werden, als dies für die Beurteilung des Zeitverbrauchs während der Bearbeitung nötig ist. Wir fassen diese zunächst vereinfachend als Lastelemente zusammen und betrachten allein die strukturellen Eigenschaften des Graphen. Dieser Graph dient zunächst ausschließlich als Modell für den Kontrollfluß. Dem reinen Datenfluß und der Kommunikation zwischen nebenläufigen Prozessen widmen wir uns in Kapitel 3.

Wir streben die Definition eines Kontrollflußmodells an, daß sequentielle und nebenläufige Bereiche sichtbar unterscheidet, wichtige technische Eigenschaften für die spätere Werkzeugimplementierung garantiert, nur eindeutige Interpretation in allen möglichen Anwendungen zuläßt und eine große Nähe zum intuitiven Verständnis des Modells bewahrt. Wir werden dazu Elemente einführen, die der graphischen Darstellung eine strenge Einteilung in sequentielle und parallele Regionen ermöglichen. Strukturelle Modellelemente werden als zeitlos interpretiert. Als zeitbehaftete Elemente werden wir Lasten einführen, deren Bewertung wir später erörtern.

Sequentielle und parallele Regionen werden wir durch eine Übersetzung in ein Petrinetz zusammenführen. Dieses Petrinetz und sein Erreichbarkeitsgraph werden dann unmittelbar der Simulation und der Bewertung zugrunde liegen. Eine Marke an einer Stelle wird den den Zustand des Kontrollflusses anzeigen, weshalb wir *1-beschränkte* Petrinetze verwenden. Einige Stellen des Petrinetzes werden aus der Übersetzung von Lastelementen stammen. Diesen Stellen fügen wir Anschriften bei, die später vom Simulator verwendet werden, um das zeitliche Verhalten zu bewerten. Bei diesen Elementen abstrahieren wir vom qualitativen Einfluß auf einen Programmzustand. In einer Simulation nehmen die zeitlichen Effekte, die aus den Zuständen der Lastmodelle durch das Schalten spezieller Transitionen ermittelt werden, Einfluß auf die Markierung des Petrinetzes. Die neue Markierung dient dann wieder zur Ermittlung des nächsten Zustandes der Lastmodelle. Die Wechselwirkung beider Systeme modelliert das zeitliche Verhalten des dynamischen Modells in Bearbeitung. Diese Wechselwirkung spezifizieren wir schließlich durch eine zeitliche Semantik. Beide Spezifikationen

werden wir später zur Implementierung einer Simulationssoftware heranziehen.

Zunächst wollen wir die Elemente einzeln herleiten, aus denen ein dynamisches Kontrollflußmodell aufgebaut sein soll.

2.2.2. Unverzweigter Kontrollfluß. Zunächst wird ein Kontrollfluß ohne Nebenläufigkeit betrachtet.

2.2.2.1. *Sequentielle Abfolge von Lasten.* Ein Lastelement, dessen Bearbeitung durch eine Ressource Zeit beansprucht, wird durch ein Rechteck dargestellt. Dieses Rechteck muß vom Graphen durch mindestens eine eingehende Kante erreichbar sein und mindestens eine Kante muß ihn verlassen. Die Last wird gekennzeichnet durch einen Namen, der in das Rechteck geschrieben wird (in unserem Beispiel in Abbildung 2.2.1: A, bzw. B), durch die Spezifikation eines Lastmodells und einer Ressource, durch die diese Last bearbeitet werden soll. Auf die beiden letzten Attribute werden wir in späteren Kapiteln eingehen.

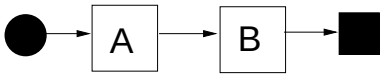


ABBILDUNG 2.2.1. Sequentielle Ausführung

Ein Startpunkt wird durch einen kleinen gefüllten Kreis symbolisiert, von dem mindestens eine Kante ausgeht. Der Startpunkt des gesamten Systems hat keine eingehende Kante. Zum Startzeitpunkt aktiviert er das nächste, durch die von ihm ausgehende Kante erreichbare Element unmittelbar (In unserem Fall das Lastelement A). Dieser Vorgang wird als zeitlos betrachtet. Ist eine Last aktiviert, wird dadurch eine Aktivität modelliert, die von einer Ressource ausgeführt wird und erst nach Ablauf einer gewissen Zeit $t(\mathsf{A})$ terminiert[3]. Erst nach dem Verstreichen dieser Zeit wechselt die Last in einen inaktiven Zustand. Zeitgleich aktiviert sie das Element, welches durch ihre ausgehende Kante erreichbar ist (im Beispiel ist dies Last B), da dieser Vorgang wiederum als zeitlos betrachtet wird.

Ein Endpunkt unseres Modells wird durch ein gefülltes kleines Quadrat dargestellt, welches mindestens eine eingehende Kante aufweist. Der Endpunkt des gesamten Systems weist keine ausgehende Kante auf.

2.2.2.2. *Bedingte Ausführung und Schleifen.* Der Entwickler eines Modells erwartet Ausdrucksmöglichkeiten, um Sprünge, Alternativen und Schleifen zu formulieren. Dies wird möglich, wenn mehrere Kanten

[3]In unserem Beispiel ergibt sich die Bearbeitungszeit zu $t_{ges} = t(\mathsf{A}) + t(\mathsf{B})$. Die Ermittlung und Berechnung der Funktion t werden wir in Kapitel 4 erörtern.

auf ein Element zeigen und mehrere Kanten von einem Element ausgehen können. Um eine Auswahl eindeutig festlegen und eine Schleife ausführen und abbrechen zu können, fordern wir eine eindeutige Kennzeichnung bei mehr als einer ausgehender Kante. Ein aus diesen Darstellungsmitteln aufgebauter Graph erlaubt noch keine Verzweigung des Kontrollflusses und verkörpert somit gut den Begriff einer sequentiellen Region. Die Elemente eines Teilgraphen, der einen unverzweigten Kontrollfluß modelliert, fassen wir zu einer Region zusammen und stellen sie in einem grau unterlegten, durch eine gestrichelte Linie begrenzten Rechteck mit abgerundeten Kanten dar.

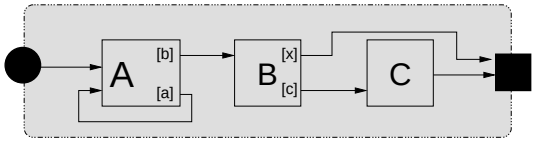


ABBILDUNG 2.2.2. Bedingte und wiederholte Ausführung

Dieses Element bezeichnen wir als *Sequenzkartusche*. Sie hat genau einen Eintritts- und genau einen Austrittspunkt. Start- und Endpunkt des eingebetteten Graphen übernehmen diese Funktion.

In Abbildung 2.2.2 sind eine Schleife und eine Auswahl beispielhaft realisiert. Dieses Beispiel stellt die Übertragung des Beispiels aus Abbildung 2.1.1 dar. Aktivität A hat hier mehr als eine ausgehende Kante. Die Auswahl der Kante a realisiert hier eine Schleife. Aktivität B hat zwei ausgehende Kanten. Die Wahl der Kante x überspringt die Aktivität C , wogegen die Auswahl der Kante c zur Bearbeitung von Aktivität C führt. Mehrfach ausgehende Kanten ermöglichen also keine Nebenläufigkeit. In Teilmodellen, die zu solch einer Kartusche zusammengefaßt sind, ist stets nur ein Element aktiviert. Damit sind mehrere eingehende Kanten unproblematisch, da nur über jeweils eine Kante das nächste Element aktiviert werden kann.

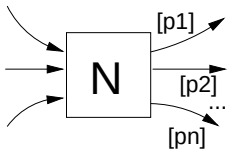


ABBILDUNG 2.2.3. Last

2.2.2.3. *Lastelement.* Wie schon in den vorhergehenden Abbildungen eingeführt, werden Lastelemente durch ein Quadrat dargestellt. Lastelemente allein sind zeitbehaftet. Dieses Element kann eine beliebige Anzahl von eingehenden Kanten aufweisen. Es kann eine beliebige Anzahl von ausgehenden Kanten aufweisen, die aber jeweils unterscheidbare Anschriften tragen müssen. Lastelemente sind immer in einer Sequenzkartusche untergebracht. War eine Last aktiviert

und terminiert, kann nur eine ausgehende Kante entsprechend einer bestimmten Anschrift eine Folgelast aktivieren. Im Gegenzug kann eine Last nur von einer eingehenden Kante aktiviert werden. Lastelemente innerhalb einer Sequenzkartusche können ausschließlich jeweils nacheinander aktiviert sein. Es ist also möglich, eine bestimmte Folge[4] von aktivierten Elementen einer Sequenzkartusche abzuleiten.

2.2.2.4. *Sequenzen.* Eine sequentiell ausführbare Region, die Sequenzkartusche, setzt sich zunächst aus der Menge an Lastelementen zusammen. Weiterhin fügen wir als weiteren Elementtyp noch nebenläufige Regionen ein, so genannte Parallelkartuschen, deren Struktur und Interpretation wir später definieren.

Jedes Element kann beliebig viele ausgehende und eingehende Kanten haben. War ein Element aktiviert, so war gefordert, daß es höchstens ein Folgeelement aktivieren kann, das von ihm durch eine eindeutig bestimmbare Transition erreicht wird. Die erwünschte Intuition, daß die Elemente einer solchen Sequenzkartusche sich analog zu einem deterministischen, endlichen Automaten verhalten, festigen wir durch folgende Definition.

DEFINITION 1 (SK-Automat). *Ein SK-Automat $A_{\mathcal{SK}}$ ist ein Quintupel $\left(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\right)$, mit*

$$\begin{aligned} \mathcal{Z}, &\quad \textit{der endlichen Zustandsmenge,} \\ z_{start} \in \mathcal{Z}, &\quad \textit{dem Startzustand,} \\ z_{end} \in \mathcal{Z}, &\quad \textit{dem Endzustand,} \\ \Sigma, &\quad \textit{dem SK-Alphabet,} \\ \delta : \mathcal{Z} \times \Sigma \rightarrow \mathcal{Z}, &\quad \textit{die Übergangsfunktion.} \end{aligned}$$

Dabei betrachten wir im Folgenden gesondert die Eigenschaften von

$$\begin{aligned} l \in \Sigma, &\quad \text{jeweils einem Bezeichner aus dem SK-Alphabet,} \\ \mathcal{Z}_{PK} \subset \mathcal{Z}, &\quad \text{der Menge der eingebetteten Parallelkartuschen und} \\ \mathcal{Z}_{LE} \subset \mathcal{Z}, &\quad \text{der Menge der eingebetteten Lastelemente,} \\ &\quad \text{für die außerdem gilt:} \mathcal{Z} = \mathcal{Z}_{PK} \uplus \mathcal{Z}_{LE} \uplus \{z_{start}, z_{end}\} \end{aligned}$$

An den gewählten Bezeichnungen erkennt man die Ähnlichkeit zu einem deterministischen endlichen Automaten. Seine Zustandsmenge setzt sich aus zwei Teilmengen zusammen, erstens aus der Menge an Lastelementen, die bei der Performanz-Bewertung eine Rolle spielen werden, zweitens aus einer Menge an eingebetteten Parallelkartuschen[5].

[4] Aus technischen Gründen müssen wir die Endlichkeit dieser Folge fordern.

[5] Die Struktur dieser Teilmodelle werden wir später noch einführen

Eine Parallelkartusche wird aktiviert, wenn das Modell eine nebenläufige Aktivität beschreibt. Die innere Struktur dieser Elemente beschreiben wir im nächsten Abschnitt.

Die Start- und Endzustände decken sich mit den Start- und Endpunkten einer Sequenzkartusche des graphischen Modells. Die Ausdrücke l des Kartuschenalphabets Σ decken sich mit den Transitionsanschriften, die im graphischen Modell zur Unterscheidbarkeit mehrfacher ausgehender Transitionen gefordert waren. Die Transitionen wiederum werden im formalen Modell durch die Übergangsfunktion δ dargestellt.

2.2.2.5. *Sequenzkartuschen.* Die $l \in \Sigma$ dienen später zur Spezifikation einer bestimmten Zustandsfolge dieses Automaten. Um dies zu gewährleisten, fordern wir die zweifelsfreie Unterscheidbarkeit mehrerer Transitionen, die einen Zustand verlassen, anhand der Bezeichner. Eine endliche Folge ω über dem Kartuschenalphabet Σ, das von diesem Automaten akzeptiert wird, kann somit während der Simulation die Interaktion des Systems mit der Außenwelt[6] modellieren. Eine eindeutige Spezifikation dieser Zustandsfolge muß möglich sein.

DEFINITION 2 (Eindeutige Übergangsfunktionen). *Eine Übergangsfunktion δ heißt* eindeutig, *falls*

$$\forall z, z', z'' \in \mathcal{Z}. \forall\ l \in \Sigma\ .\ \delta(z,l) \rightarrow z' \in \delta \wedge \delta(z,l) \rightarrow z'' \in \delta\ \Rightarrow\ z' = z''$$

Sei $(z_1, z_2, \ldots, z_{n+1})$ eine Zustandsfolge (und damit eine Lastfolge), die sich aus einer Folge von Übergängen (oder Transitonen) $(\delta_1, \delta_2, \ldots, \delta_n)$ derart ableiten läßt, daß $\forall(1 \leq i < n)\ .\ \delta_i = (z_i, l_i, z_{i+1}) \Rightarrow \delta_{i+1} = (z_{i+1}, l_{i+1}, z_{i+2})$, so kann diese durch ihre Bezeichnerfolge $\omega = (l_1, l_2, \ldots, l_n)$ über dem Kartuschenalphabet Σ eindeutig beschrieben werden.

DEFINITION 3 (Vollständiger Lauf). *Eine Bezeichnerfolge $\omega = (l_1, l_2, \ldots, l_n)$ erzeugt einen* vollständigen Lauf, *wenn es eine Folge $\delta_1, \delta_2, \ldots, \delta_n$ von Zustandsübergängen gibt mit:*

$$\begin{aligned} \delta_1 = (z_1, l_1) \rightarrow z_2\ .\quad & z_1 = z_{start}\ \textit{und} \\ \delta_n = (z_n, l_n) \rightarrow z_{n+1}\ .\quad & z_{n+1} = z_{end}. \end{aligned}$$

Wir fordern dabei, daß für eine Sequenzkartusche stets mindestens ein ω existieren muß, mit dem der Endzustand z_{end} erreicht werden kann.[7]

[6] Ein Wort stellt genau einen Testfall dar. Wir behandeln die Bedeutung der Testfälle (*Szenarien*) in Kapitel 3

[7] In der Einführung der graphischen Elemente hatte wir diese Forderung schon eingebracht

DEFINITION 4 (Sequenzkartusche). *Eine* Sequenzkartusche K_{SK} *ist ein SK-Automat*[8] *und für den mindestens ein vollständiger Lauf* ω *über seinem Alphabet existiert.*

2.2.3. Verzweigter Kontrollfluß. Da innerhalb einer Sequenzkartusche zu jeden Zeitpunkt nur ein einziges Element aktiv sein darf, führen wir zur Modellierung nebenläufiger Kontrollflüsse eine eigene Region ein.

In einem nebenläufigen Bereich können mehrere Elemente gleichzeitig aktiviert sein. Dies wird modelliert, indem diese Elemente durch gerichtete Kanten gleichzeitig erreichbar sind. Wir führen die Darstellung einer *Parallelkartusche* ein. Die Parallelkartusche selbst stellen wir in weißem Hintergrund mit einer gestrichelten Begrenzungslinie dar. Sie hat genau einen Startpunkt und genau einen Endpunkt.

2.2.3.1. *Nebenläufige Region.*

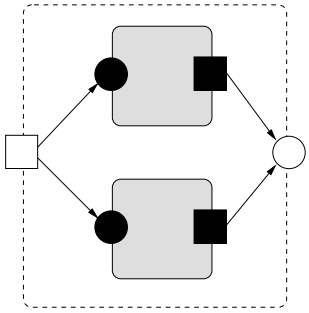


ABBILDUNG 2.2.4. Schema einer Parallelkartusche

Die Parallelkartusche selbst ist wiederum in eine Sequenzkartusche eingebettet. Der Eintritt in eine solche nebenläufige Region ist also der *vorübergehende Austritt* aus einer sequentiellen Region. Der Austritt aus der nebenläufigen Region ist dann der *Wiedereintritt* in die vormals verlassene, sequentielle Region. Deshalb wählen wir die (scheinbar) gegensätzlichen Symbole von Quadrat und Kreis, die hier Austritt und Eintritt aus Sicht der übergeordneten Sequenzkartusche symbolisieren. Wir legen fest, daß jeweils ein verübergehender Austrittspunkt und ein Wiedereintrittspunkt paarweise zusammengehören und die Sequenzkartusche mit der Parallelkartusche verbinden, in der wir Nebenläufigkeit modellieren. Eine Sequenzkartusche kann mehrere, ihr zugeordnete Parallelkartuschen[9] besitzen. Eine Parallelkartusche gehört aber stets nur zu einer Sequenzkartusche.

In Abbildung 2.2.5 ist ein Graph dargestellt, der spezifiziert, daß nach der Ausführung der Aktivität A die Aktivitäten M und N parallel ausgeführt werden. Sobald beide Aktivitäten terminieren, beginnt die Aktivität B und schließt die Bearbeitung des Modells ab. Elemente, die in einer nebenläufigen Region aktiviert werden sollen, müssen

[8]Mit diesem beiden Definitionen handelt es sich bei einer Sequenzkartusche um einen deterministischen endlichen Automaten

[9]Dies sind eben die Elemente der Teilmenge $\mathcal{PK}$

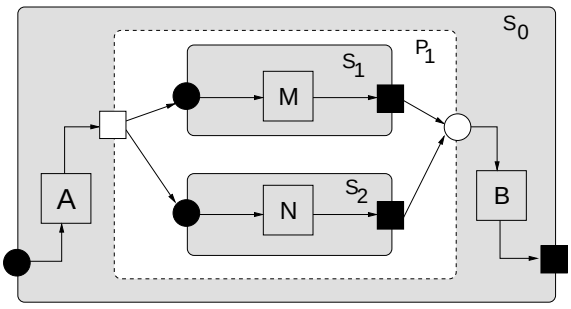


ABBILDUNG 2.2.5. Kartusche mit nebenläufiger Region

also ihrerseits wieder in eine Sequenzkartusche eingebettet werden, deren Startpunkt von einer Kante des Graphen in der Parallelkartusche erreicht wird. Der Graph in der Parallelkartusche besteht nur aus Kanten und eingebetteten Sequenzkartuschen. Daß Sequenzkartuschen ihrerseits wieder Parallelkartuschen aufweisen können, ist in Abb. 2.2.5 angedeutet. In dieser Darstellung wird auch der Grund für die Wahl eines weißen Quadrates als temporärer Austrittspunkt und der eines weißen Kreises als Wiedereintrittspunkt augenfällig.

Diese Abbildung realisiert genau das in Abschnitt 2.1.1.2 eingeführte Beispiel einer wohleingebetteten Nebenläufigkeit aus Abbildung 2.1.2.

2.2.3.2. *Unterliegender Netzgraph.* In einer Parallelkartusche hatten wir ein anderes Verhalten der Kanten gefordert als das der Sequenzkartuschen. Alle von einem aktivierten Knoten ausgehenden Kanten sollten den folgenden Knoten gleichzeitig aktivieren und so Nebenläufigkeit modellieren. Außerdem sollte Synchronisation ausgedrückt werden, indem ein Knoten mit mehreren eingehenden Kanten nur aktiviert werden kann, wenn alle eingehenden Kanten aktiviert sind. Die Möglichkeit mehrerer gleichzeitig aktivierter Knoten macht eine andere formale Beschreibung notwendig. Intuitiv handelt es sich zwar strukturell um einen gerichteten Netzgraphen, aber für die zusätzlichen Verhaltenseigenschaften sind weitere Definitionen nötig.

DEFINITION 5 (PK-Graph). *Ein* PK-Graph $G_{\mathcal{PK}}$ *ist ein Quadrupel* $\left(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus}\right)$, *mit* $\mathcal{K} \subseteq \mathcal{E} \times \mathcal{E}$, *wobei*

$\mathcal{E}$ *eine endliche Menge an Elementen*
$\mathcal{K}$ *eine endliche Menge an gerichteten Kanten,*
$e_{ein} \in \mathcal{E}$ *die Quelle und*
$e_{aus} \in \mathcal{E}$ *die Senke ist.*

Dabei betrachten wir im Folgenden gesondert die Eigenschaften von

$\mathcal{E}_{SK} \subset \mathcal{E}$ der Teilmenge eingebetteter Sequenzkartuschen, wobei $\mathcal{E}_{SK} = \mathcal{E} \setminus \{e_{ein}, e_{aus}\}$.

Lastelemente kommen in der Parallelkartusche nicht vor. Selbst falls dort nur ein einziges Lastelement zu bewerten ist, müßte dieses in die Sequenzkartusche eingebettet sein. Alle in ein beliebiges $e \in \mathcal{E}_{SK}$ eingehenden Kanten werden mit dem Startpunkt z_{start} der Sequenzkartusche verbunden. Analog werden alle von $e \in \mathcal{E}_{SK}$ ausgehenden Kanten als vom Endpunkt bzw. z_{end} der Kartusche ausgehend interpretiert. Mit ausgezeichneten Start- und Endknoten, e_{ein} und e_{aus}, die oft auch als *Quelle* und *Senke* bezeichnet werden, beschreiben wir die geforderten, eindeutigen Start- und Endpunkte der Parallelkartusche.

2.2.3.3. *Strukturelle Eigenschaften des Netzgraphen.* Für die in Abschnitt 5 definierten Netzgraphen führen wir weitere Einschränkungen ein, um wichtige Eigenschaften zu erreichen. Jeder Knoten der Menge $\mathcal{E}_{SK}$ modelliert eine eingebettete Sequenzkartusche. Die ausgezeichneten Elemente e_{ein} und e_{aus} repräsentieren die *Austritts-* und *Wiedereintritts*-Knoten. Zur Vollständigkeit führen wir einige aus der Graphentheorie bekannten Begriffe explizit ein.

DEFINITION 6 (Kopf- und Fußknoten). *Für $e_i, e_j \in \mathcal{E}$ und $k \in \mathcal{K}$ mit $k = (e_i, e_j)$ bezeichnen wir mit*

$k^- = e_i$ *den Anfangsknoten und*
$k^+ = e_j$ *den Endknoten der Kante k.*

Die Anzahl von eingehenden bzw. ausgehenden Kanten aus einem Knoten ist ein interessantes Maß bei der Betrachtung des Graphen.

DEFINITION 7 (Grade von Kanten). *Sei $(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus})$ ein PK-Graph und $e \in \mathcal{E}$ ein Knoten, dann bezeichnen wir die Anzahl der eingehenden Kanten (In-Grad) mit*

$$d^+(e) = |\{k \in \mathcal{K} \,.\, k^+ = e\}|$$

und die Anzahl der ausgehenden Kanten (Aus-Grad) mit

$$d^-(e) = |\{k \in \mathcal{K} \,.\, k^- = e\}|$$

Ein Fluß in diesen Graphen muß an Knoten (Quellen) beginnen, die keine eingehenden Kanten haben, und an denen (Senken) enden, die keine ausgehenden Kanten haben.

FORDERUNG 1 (Abgrenzung). *Ein PK-Graph heißt* abgegrenzt, *falls er keine eingehenden Kanten an der Quelle und keine ausgehenden*

Kanten an der Senke aufweist, also

$$d^+(e_{ein}) = 0 \text{ und } d^-(e_{aus}) = 0 \text{ gilt.}$$

Diese Start- und Endknoten, e_{ein} und e_{aus}, sind gemäß Definition 5 eindeutig. Ein Weg, den man über die gerichteten Kanten auch über mehrere Knoten hinweg zurücklegen kann, bezeichnen wir als Pfad.

DEFINITION 8 (Pfad). *Eine endliche Folge von Kanten e_n heißt* Pfad P_n *der Länge $n \geq 0$ im PK-Graphen, falls zwei aufeinanderfolgende Knoten jeweils mit einer Kante verbunden sind:*

$$\forall\ 1 \leq i < n\ .\ (e_i, e_{i+1}) \in \mathcal{K}$$

Als vollständigen Pfad $P_n^{\leftrightsquigarrow}$ der Länge $n \geq 0$ bezeichnen wir einen Pfad P_n, der Quelle und Senke verbindet.

DEFINITION 9 (Vollständiger Pfad). *Ein Pfad P_n heißt* vollständiger *Pfad $P_n^{\leftrightsquigarrow}$, falls gilt:*

$$(e_1 = e_{ein}) \wedge (e_n = e_{aus})$$

Würde eine Schleife modelliert werden, könnte diese durch keine Angabe in einem Szenario abgebrochen werden. Die mehrfache Ausführung einer Sequenzkartusche innerhalb einer Parallelkartusche muß in der übergeordneten Sequenzkartusche formuliert werden, zu der diese Parallelkartusche gehört. Jeder PK-Graph selbst muß also azyklisch sein.

FORDERUNG 2 (Zyklenfreiheit). *Ein PK-Graph heißt* zyklenfrei, *falls für alle Pfade $P_n = (e_1, e_2, \ldots, e_n)$ in diesem Graphen gilt: $(e_n, e_1) \notin \mathcal{K}$*

Zusätzlich zu der Forderung 1, daß nur die Senke keine ausgehenden Kanten aufweist, müssen wir noch $G_{\mathcal{PK}}$ als zusammenhängend fordern und jeder Knoten sollte Element mindestens eines Pfades sein, der die Quelle e_{ein} mit der Senke e_{aus} verbindet.

FORDERUNG 3 (Vollzusammenhang). *Ein PK-Graph heißt* vollzusammenhängend, *falls für alle Kanten $e \in \mathcal{E}$ ein vollständiger Pfad $P^{\leftrightsquigarrow}$ im PK-Graphen existiert, sodaß e im Pfad enthalten ist.*

FOLGERUNG 1 (Sackgassenfreiheit des Graphen). *Ein vollzusammenhängender, azyklischer, abgegrenzter PK-Graph ist* sackgassenfrei*:*

$$\forall e \in \mathcal{E}_{SK}\ .\ d^-(e) \geq 1$$

Beweis: Dies folgt unmittelbar aus der Annahme: $\exists e \in \mathcal{E}_{SK} \mid d^-(e) = 0$ durch Widerspruch. Denn existierte ein Pfad P_n, mit der Eigenschaft, daß $(e_x \in P_n) \wedge (e_1 = e_{ein}) \wedge (e_n = e_{aus})$ nach Forderung 3. Dann $e_x \in P_n \Rightarrow \exists (e_x, e_{x+1}) \in \mathcal{K}$ nach Definition 8. Aus der Annahme $d^-(e_x) = 0$ folgt aber nach Definition 7: $\{k \in \mathcal{K} \mid (k^- = e_x)\} = \emptyset$, was ein Widerspruch ist. Nur falls $e_x = e_n$ wäre die Annahme erfüllt, da damit $e_x = e_{aus}$. Damit ist nach Definition 5 zwar $e_x \in \mathcal{E}$, aber durch $e_x \notin \mathcal{E}_{SK}$ ein weiterer Widerspruch entstanden. □

Durch die gerichteten Kanten des Graphen $G_{\mathcal{KP}}$ wird die Aufspaltung des Kontrollflusses in nebenläufige Bereiche dort formuliert, wo ein Knoten mehrere ausgehenden Kanten trägt. Alle Zweige des Kontrollflusses müssen schließlich wieder zusammengeführt werden. Um den Anteil einer Kante am aufgespaltenen Kontrollfluß anschaulicher zu machen, definieren wir einen Flußanteil.

DEFINITION 10 (Flußanteil). *Der Flußanteil ist der Anteil einer Kante am aufgespaltenen Kontrollfluß, gegeben zu*

$$\phi(k) = \begin{cases} \dfrac{1}{d^-(e_{ein})}, & \textit{falls } k^- = e_{ein} \\ \dfrac{1}{d^-(k^-)} \cdot \sum \left\{ \phi(k') \;.\; k^{+\prime} = k^- \right\}, & \textit{sonst.} \end{cases}$$

Dadurch ist es in einer Werkzeugimplementierung möglich, eine weitere nötige Einschränkung zu überprüfen. Nach unserem Modellverständnis muß der Kontrollfluß genau dann wieder in den Bereich der Sequenzkartusche wechseln, wenn alle nebenläufigen Stränge beendet sind. Dies wäre genau dann erreicht, wenn eine Kante erneut den Flußanteil 1 trägt, also einen einzigen Übergang zwischen zwei Bereichen des Graphen $G_{\mathcal{PK}}$, der eine sogenannte *Brücke* bildet.

FORDERUNG 4 (Brückenfreiheit). *Ein PK-Graph ist* brückenfrei, *falls*

$$\forall k \in \mathcal{K} \;.\; \exists n \in \mathbb{N}, \exists P_n^{\leftrightsquigarrow} \;.\; k \notin P_n^{\leftrightsquigarrow}.$$

Die Anforderung gemäß dieser Beschreibung liegt nahe an dem Verständnis unserer Modellelemente. Die Überprüfung dieser Eigenschaft kann aber in einer Implementierung wieder besser durch leichter verfügbare Parameter ermittelt werden [**29, 30**].

BEMERKUNG 1. *Interessant ist weiterhin folgender Zusammenhang. Wir betrachten eine Adjazenzmatrix $A(G_{\mathcal{PK}})$. Ein PK-Graph weist hier in der Diagonalen nur Nullen und im Übrigen nur die Werte*

1 oder 0 auf.

$$A(G_{\mathcal{PK}}) = \begin{pmatrix} 0 & a_{1,2} & \dots & a_{1,x} \\ \dots & 0 & \dots & \dots \\ \dots & \dots & 0 & \dots \\ a_{y,1} & \dots & a_{y,x-1} & 0 \end{pmatrix}, \text{ mit } a_{i,j} \in \{0,1\} \wedge a_{i,i} = 0.$$

Da ein gerichteter Graph zusammenhängend ist, wenn sein unterliegender, ungerichteter Graph zusammenhängend ist, können wir auf diesem die folgenden Betrachtungen durchführen. Wir führen das Gerüst als Hilfskonstruktion ein. Unter einem Gerüst versteht man einen zusammenhängenden, zyklenfreien Untergraph, der alle Knoten enthält und gleichzeitig mit einer minimalen Kantenteilmenge auskommt. Damit hat ein nicht zusammenhängender Graph genau kein Gerüst (und ein Baum genau ein Gerüst). Führen wir nun eine Gradmatrix D ein, die auf der Diagonalen die gesamte Anzahl der Kanten eines Knoten aufweist, wie folgt:

$$D = \begin{pmatrix} a_{1,1} & 0 & 0 \\ 0 & \dots & 0 \\ 0 & 0 & a_{n,n} \end{pmatrix} \text{ mit } a_{x,x} = d^+(e_x) + d^-(e_x).$$

Dann können wir, unter Verwendung der vorher definierten Adjazenzmatrix A, die Admittanzmatrix (oder Laplace-Matrix) L berechnen durch $L = D - A$. Nach dem Satz von Kirchhoff berechnet sich die Anzahl der Gerüste des Graphen aus der Determinante von L_x nach $t(G_{PK}) = detL_x$, wobei man L_x durch Streichung einer beliebigen Zeile und Spalte x aus L erhält [**59**]*. Existiert mindestens ein Gerüst, ist der Graph der Parallelkartusche zusammenhängend.*

Somit kann auf diesem Wege der Zusammenhang und die Brückenfreiheit gleichzeitig ermittelt werden, indem man t für jeden Graphen bestimmt, den man durch Weglassen einer jeweils anderen Kante erhält. Der Graph G_{PK} ist also zusammenhängend und brückenfrei, falls $\forall k_i \in \mathcal{K} \; . \; t(G_{PK} \setminus k_i) \geq 1$. [10]

Man bemerke noch einige zusätzliche statische Eigenschaften des Graphen. Den Eingangsknoten e_{ein} verlassen stets mehr als eine Kante, was unmittelbar aus Definition 10 und Forderung 4 ablesbar ist (Ohne Beweis).

[10] Möglicherweise ist die fortschreitende Berechnung eines Flußanteil performanter, worauf wir in den Vorschlägen zu einer Werkzeugarchitektur noch näher eingehen.

FOLGERUNG 2 (Initiale Spaltung des Kontrollflusses). *Für jeden vollzusammenhängenden brückenfreien PK-Graphen gilt:*

$$d^-(e_{ein}) > 1$$

Ebenso kann nicht eine einzelne Kante in den Ausgangsknoten führen.

FOLGERUNG 3 (Finale Vereinigung des Kontrollflusses). *Für jeden vollzusammenhängenden brückenfreien PK-Graphen gilt:*

$$d^+(e_{aus}) > 1$$

Durch die Kanten in $G_{\mathcal{PK}}$ werden Aufspaltung und Synchronisation des Kontrollflusses ausgedrückt. Ein Teilkontrollfluß, der aber nicht weiter aufgespalten wird, soll durch genau eine Sequenzkartusche modelliert werden. Daraus folgt, daß Kanten nie als einzige ausgehende und einzige eingehende Kanten zwei Knoten verbinden (Zwei so verbundene Kartuschen müßten zwangsläufig die gleiche Identität haben[11]).

FORDERUNG 5 (Eindeutige Modellierung eines Kontrollflusses).

$$\nexists k \in \mathcal{K} \; . \; (d^+(k^+) = 1) \wedge (d^-(k^-) = 1)$$

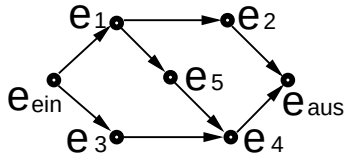


ABBILDUNG 2.2.6. Beispiel mit Synchronisation

BEISPIEL 1. *Abbildung 2.2.6 zeigt einen wohlgeformten Graphen einer Parallelkartusche. Er ist zyklenfrei, brückenfrei, sackgassenfrei und zusammenhängend. Kein Knoten mit nur einer ausgehenden Kante ist mit einem Knoten mit nur einer eingehenden Kante verbunden. Dieses Beispiel zeigt außerdem, daß das erweiterte Modell aus Abbildung 2.1.5 in einem PK-Graphen ausgedrückt werden kann. Hier wäre die Last A im Knoten e_1 zu finden, die Last B im Knoten e_2, die Last C im Knoten e_3, die Last D im Knoten e_4 und die Last S, die zwischen den nebenläufigen Strängen kreuzt, wäre in der Kartusche zu finden, die durch Knoten e_5 repräsentiert wird.*

Das einfachere Beispiel in Abbildung 2.2.7 repräsentiert die Parallelkartusche aus dem einfachen Beispiel mit Nebenläufigkeit in Abbildung 2.2.5. Der Knoten e_1 verweist dann auf die Kartusche, welche die

[11]Die Brückenfreiheit allein ist bezüglich dieser Modelleigenschaft zu schwach. Diese zusätzliche Forderung dient nur der Gestaltung der Modellelemente und ist keine formale Notwendigkeit.

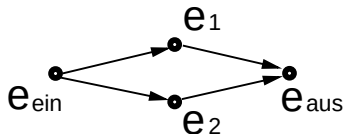


ABBILDUNG 2.2.7. Beispiel zweier nebenläufiger Kartuschen

Last M enthält. Der Knoten e_2 referenziert die Kartusche mit der Last N.

BEMERKUNG 2 (Eindeutigkeit von Fork und Join). *Erzeugen wir später dynamische Modelle aus UML-Diagrammen, verlassen Kanten nie Knoten mit mehreren ausgehenden Kanten und gehen ein in Knoten mit mehreren eingehenden Kanten.*

$$\forall k \in \mathcal{K} : (d^-(k^+) = 1 \wedge d^+(k^-) > 1) \vee (d^-(k^+) > 1 \wedge d^+(k^-) = 1)$$

Dies würde gleichzeitig eine Spaltung und Synchronisation des Kontrollflusses bedeuten und die Eindeutigkeit von Fork und Join verletzen. Für die weitere Betrachtung ist dies zwar unerheblich, soll hier aber als Beobachtung erwähnt werden.

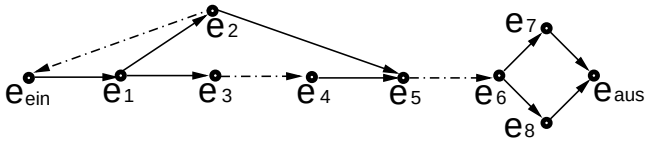


ABBILDUNG 2.2.8. Kein wohlgeformter PK-Graph

BEISPIEL 2 (Fehlerhafter Graph). *Der Graph in Abbildung 2.2.8 ist aus mehreren Gründen nicht wohlgeformt. Die gestrichelt dargestellte Kante (e_2, e_{ein}) bildet einen Zyklus und wäre eine nicht erlaubte eingehende Kante in die Quelle e_{ein}. Die Kante (e_3, e_4) bildet die einzige Verbindung zwischen den beiden Knoten. e_3 und e_4 müßten deshalb ein einziger Knoten sein. Und schließlich wird durch die Kante (e_5, e_6) eine Brücke gebildet.*

Der beschriebene Graph definiert ausreichend die statische Struktur einer wohlgeformten Parallelkartusche.

DEFINITION 11 (Parallelkartusche). *Eine* Parallelkartusche K_{PK} *ist ein abgegrenzter, zyklenfreier, vollzusammenhängender und brückenfreier PK-Graph.*

Da die Elemente des PK-Graphen nebenläufige Prozesse modellieren und die Knoten jeweils synchronisierende Wirkung haben, müssen

wir unsere Modellvorstellungen noch um dynamischen Aspekte erweitern.

2.2.3.4. *Dynamische Eigenschaften.* Eine Bearbeitung eines PK-Graphen startet an der Quelle. Es ist zu zeigen, daß jeder Knoten genau einmal besucht wird, bevor auf allen vollständigen Pfaden die Senke erreicht wird. Synchronisation drückt die Einschränkung aus, daß ein Knoten nur besucht wird, nachdem alle unmittelbaren Vogänger besucht worden sind. Deshalb muß zusätzlich gezeigt werden, daß die Senke auch mit dieser Einschränkung immer erreichbar ist.

Zur Untersuchung der dynamischen Eigenschaften führen wir eine Partition π der Knotenmenge $\mathcal{E}$, in dem Paar $(B, \overline{B})$ ein. Dabei sind $B \subseteq \mathcal{E}$ und $\overline{B} \subseteq \mathcal{E}$ disjunkt, sodaß $B \cap \overline{B} = \emptyset$. Außerdem gilt $B \cup \overline{B} = \mathcal{E}$. Ein Knoten gilt als besucht, wenn er in B enthalten ist, sonst gilt $e \in \overline{B}$. Daraus ergeben sich Start- und Endpartitionen.

DEFINITION 12 (Start- und Endpartition). *Sei* $K_{PK} = (\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus})$ *eine Parallelkartusche mit einer Teilmenge* $\mathcal{E}_{SK} \subset \mathcal{E}$ *an Knoten, die Sequenzkartuschen modellieren. Dann definieren wir als*

$$\begin{aligned} \textit{Startpartition:} \quad & \pi_{Start} = \big(\{e_{ein}\},\ \mathcal{E}_{SK} \cup \{e_{aus}\}\big) \\ \textit{und als Endpartition:} \quad & \pi_{End} = \big(\{e_{ein}\} \cup \mathcal{E}_{SK}\ ,\ \{e_{aus}\}\big). \end{aligned}$$

Wir betrachten eine Kante genau dann als aktiviert, wenn sie einen besuchten Knoten als Fußpunkt und einen nicht besuchten Knoten als Kopfpunkt hat. Damit ergibt sich die Menge der aktivierten Kanten für jede Partition unmittelbar aus der Menge, die man in der Graphentheorie als Schnitt $S(B, \overline{B})$ bezeichnet.

DEFINITION 13 (Menge der aktivierten Kanten (Schnitt)). *Sei* $K_{PK} = (\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus})$ *eine Parallelkartusche. Die Menge aller Kanten mit* $\{k \in \mathcal{K}\ .\ (k^- \in B) \wedge (k^+ \in \overline{B})\}$ *heißt* Schnitt *von* B *und* $\overline{B}$. *Die Menge der Schnitte wird bezeichnet mit*

$$S(B, \overline{B}).$$

Die aktivierten Kanten der Startpartition sind also genau die von der Quelle ausgehenden Kanten. Um die Synchronisation auszudrücken, fordern wir, daß ein Knoten aktiviert ist und besucht werden kann, wenn alle eingehenden Kanten aktiviert also im Schnitt sind.

DEFINITION 14 (Aktivierte Knoten). *Sei* $K_{PK} = (\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus})$ *eine Parallelkartusche und* $\pi = (B, \overline{B})$ *eine Partition. Ein Knoten* e *heißt* aktiviert *in* π, *falls für alle Kanten* $k\ .\ (k^+ = e)$ *gilt:*

$$(k^- \in B) \wedge (k^+ \in \overline{B})$$

Wird der Knoten besucht, so fordern wir dessen Wechsel von der Menge $\overline{B}$ in die Menge B und wir erhalten eine neue Partition.

DEFINITION 15 (Folgepartition). *Sei $\pi_i = (B_i, \overline{B}_i)$ eine Partition einer Parallelkartusche $K_{PK} = \big(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus}\big)$ und D_i die Menge der aktivierten Knoten in π_i. Die Partition $\pi_{i+1} = (B_{i+1}, \overline{B}_{i+1})$ heißt* Folgepartition *von π_i, wenn gilt:*

$$(B_{i+1} = B_i \cup D_i) \wedge (\overline{B}_{i+1} = \overline{B}_i \setminus D_i),$$

Wenn für jede Partition π_i gilt $D_i \neq \emptyset$ und somit eine von π_i verschiedene Folgepartition π_{i+1} erzeugt werden kann, ist eine Partitionsfolge $P_k^\pi = \pi_{Start}, \pi_1, \pi_2, \ldots, \pi_k, \pi_{End}, (k \geq 0)$ ableitbar.

LEMMA 1 (Erreichbarkeit des Endzustandes). *In jeder Parallelkartusche K_{PK} ist die Endpartition von der Startpartition durch eine Partitionsfolge P_k^π erreichbar.*

$$\pi_{Start} \rightsquigarrow \pi_{End}$$

Beweis: Diese Eigenschaft läßt sich für gewöhnliche, gerichtete Graphen aus der zwingenden Existenz einer Quelle und einer Senke ableiten. Durch die Synchronisationseigenschaften, die wir in unserem Modell den Knoten zuschreiben, können wir die Eigenschaft durch Widerspruchsbeweis zeigen. Eine Menge D der aktivierten Knoten einer Partition π wäre leer und damit keine weitere Folgepartition erzeugbar, falls jeder Knoten in $\overline{B}$ mindestens eine Kante besitzt, die nicht aktiviert ist. Es müßte also gelten: $\forall e_i \in \overline{B} \;.\; \exists (e_j, e_i) \in \mathcal{K} \;.\; e_j \in \overline{B}$. Bei der Betrachtung eines längsten Pfades in der Teilmenge $\overline{B}$ führt dies zum Widerspruch. Sei ein Pfad $P_n = (e_1, e_2, \ldots, e_n)$ der längste Pfad in $\overline{B}$, müßte nach unserer Forderung eine Kante $(e_0, e_1) \in \mathcal{K}$ existieren, für die gilt: $e_0 \in \overline{B}$. Damit wäre dieser Pfad aber nicht der längste. Auch kann der Knoten nicht bereits im Pfad vorkommen, sodaß ein $1 \leq i \leq n$ existiert mit $e_0 = e_i$. Das wäre ein Widerspruch zur Forderung 2. Somit muß stets mindestens ein Knoten e existieren, für den gilt: $\forall (e_k, e) \in \mathcal{K}.\; e_k \in B$. Die Menge D ist also nie leer. (Ist $D = \overline{B} = \{e_{aus}\}$, ist die Endpartition π_{End} erreicht.)$\square$

LEMMA 2 (Summe der Flußanteile). *Die Summe der Flußanteile der aktiven Kanten, d.h. die Summe der Flußanteile der Kanten im Schnitt jeder Partition einer Partitionsfolge ist stets 1.*

$$\forall (B, \overline{B}) \;.\; \Big(\sum \{\phi(k) \;.\; k \in S(B, \overline{B})\} = 1 \Big)$$

Beweis: Für die Anfangspartition sind nur die Kanten aktiviert, welche die Quelle verlassen. Wie man leicht nachprüfen kann, läßt sich zu allen ausgehenden Kanten ein und desselben Knotens der gleichen Flußanteil berechnen. Das Summenzeichen der Flußanteils-Definition kann eliminiert werden. Die Summe der Flußanteile der von der Quelle ausgehenden Kanten ergibt sich nach Definition 10 zu: $x = \sum\{\phi(k).k^- = e_{ein}\} = d^-(e_{ein}) \cdot \phi(k) = d^-(e_{ein}) \cdot \frac{1}{d^-(e_{ein})} = 1$. Wir nehmen an, die Summe der Kanten im Schnitt jeder gültigen Partition n wäre $x_n = 1$. Wir betrachten nun jeden Knoten e_n, der aktiviert ist, d.h. dessen eingehende Kanten im Schnitt sind einzeln. Besuchen wir sie, liegt sie in Folge in B. Seine eingehenden Kanten sind dann nicht mehr aktiviert, aber dafür sind es alle seine ausgehenden Kanten. Nun läßt sich die Summe der Flußanteile der Kanten im Schnitt der Folgepartition berechnen, indem wir von einer gegebenen Anfangssumme die Summe der Flußanteile der eingehenden Kanten subtrahieren und die der ausgehenden Kanten addieren. Wenden wir Definition 10 an, sieht man, daß die Summe der Flußanteile der eingehenden Kanten so in die Definition der ausgehenden Kanten eingearbeitet ist, daß sie eliminiert werden kann. Damit ist:

$$x_{n+1} = x_n - \Big(\sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) + \Big(\sum_{\{k_{n+1,i}.k^-_{n+1,i}=e_n\}} \phi(k_{n+1,i})\Big)$$

$$\begin{aligned} x_{n+1} &= x_n - \Big(\sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) \\ &\quad + \Big(\sum_{\{k_{n+1,i}.k^-_{n+1,i}=e_n\}} \big(\frac{1}{d^-(k^-_{n,i})} \cdot \sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\big)\Big) \\ &= x_n - \Big(\sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) \\ &\quad + \Big(\big(d^-(k^-_{n,i}) \cdot \frac{1}{d^-(k^-_{n,i})}\big) \cdot \sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) \\ &= x_n - \Big(\sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) + \Big(\sum_{\{k_{n,i}.k^+_{n,i}=e_n\}} \phi(k_{n,i})\Big) \\ &= x_n \end{aligned}$$

□

BEISPIEL 3 (Partitionsfolge). *Das bereits eingeführte Beispiel aus Abbildung 2.2.9 ist hier mit einer hervorgehobenen Partition dargestellt*

(durch die ausgezeichnete Teilmenge B) und der zugehörigen Menge D der aktivierten Knoten. In diesem Fall ist die Partitionsfolge π_2 *dargestellt.*

Die gesamte Partitionsfolge $P_4^\pi = (\pi_{Start}, \pi_2, \pi_3, \pi_{End})$ *dieser Parallelkartusche und die zugehörigen Mengen aktivierter Knoten resultieren in:*

$$\begin{aligned}
\pi_{Start} &= \big(\{e_{ein}\}, \{e_1, e_2, e_3, e_4, e_5, e_{aus}\}\big) \text{ und } D_1 = \{e_1, e_3\} \\
\pi_2 &= \big(\{e_{ein}, e_1, e_3\}, \{e_2, e_4, e_5, e_{aus}\}\big) \text{ und } D_2 = \{e_2, e_5\} \\
\pi_3 &= \big(\{e_{ein}, e_1, e_2, e_3, e_5\}, \{e_4, e_{aus}\}\big) \text{ und } D_3 = \{e_4\} \\
\pi_{End} &= \big(\{e_{ein}, e_1, e_2, e_3, e_4, e_5\}, \{e_{aus}\}\big) \text{ und } D_4 = \{e_{aus}\}
\end{aligned}$$

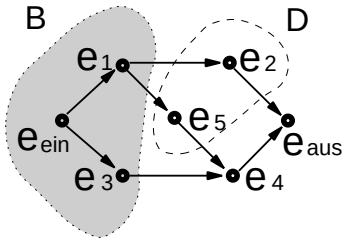


ABBILDUNG 2.2.9. Einzelne Partition in einer Folge

Es gilt zu bedenken, daß die jeweilige Menge an aktivierten Knoten nicht gleichzeitig in den bearbeiteten Zustand wechseln muß. Dieses Modell dient nur dem Beweis, daß mindestens eine *mögliche Partitionsfolge existiert. Die Knoten* $e \in D$ *einer Menge sind unabhängig voneinander. Würde z.B. in* π_2 *der Knoten* e_2 *alleine in B transportiert, ergäbe sich ein* $D_2' = \{e_4, e_5\}$ *und damit ein Element, das nicht in dieser Partitionsfolge enthalten ist. Die Mengenfolge dieser Partitionsfolge beschreibt nur einen Aspekt.*

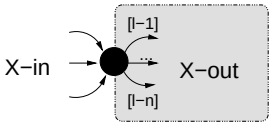


ABBILDUNG 2.2.10. Startpunkte

2.2.4. Verbindungselemente der Kartuschen. Das dynamische Gesamtmodell entsteht, wie schon der Beschreibung der einzelnen Elemente zu entnehmen war, aus der wechselweisen Einbettung von Sequenz- und Parallelkartuschen. Die Einbettung erfolgt über die jeweils auf den Grenzlinien liegenden Start- und Endpunkten bzw. den temporären Austritts- und Wiedereintrittspunkten. An diesen graphischen Elementen verschmelzen jeweils Endzustände s_{end} und Quellen e_{ein}, bzw. Senken e_{aus} und Startzustände s_{start}. Wir betrachten zunächst nochmals die graphischen Verbindungselemente und danach formal das Gesamtmodell.

2.2.4.1. *Startpunkte. Startpunkte* sind zeitlos. Eine Sequenzkartusche besitzt genau einen Startpunkt, der auf der Grenzlinie der Kartusche angeordnet ist und den Eintritt in die Kartusche von außen erlaubt. Kanten gehen in den Startpunkt nur ein, wenn sie von außerhalb der Kartusche kommen. Eine Kartusche ist als Wurzelelement ausgezeichnet. Diese selbst ist in keiner Parallelkartusche eingebettet und hat keine eingehenden Kanten. Ausgehende Kanten führen nur zu Elementen innerhalb der Kartusche. Jeder Startzustand hat mindestens eine bis beliebig viele ausgehende Kanten, die dann aber durch eindeutige Anschriften unterscheidbar sein müssen.

Das Modell wird aktiviert, indem der Startpunkt der Wurzelkartusche aktiviert wird. Jeder weitere Startpunkt wird aktiviert, wenn die eingehende Kante[12] aktiviert ist. Gehen mehrere Kanten ein, wird der Startpunkt erst dann aktiviert, wenn alle eingehenden Kanten aktiviert[13] sind. Ein aktivierter Startpunkt aktiviert eine einzige ausgehende Kante.

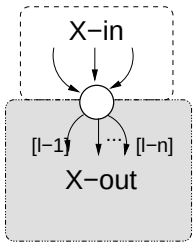


ABBILDUNG 2.2.11. Wiedereintrittspunkte

2.2.4.2. *Wiedereintrittspunkte.* Durch einen leeren Kreis werden *Wiedereintrittspunkte* dargestellt. Für eine Sequenzkartusche kann es mehrere Wiedereintrittspunkte geben, jeweils einer, für jede eingebettete Parallelkartusche. Aus dieser Parallelkartusche führen mehrere Kanten in den Wiedereintrittspunkt. Analog zu der Definition des Startpunktes aktivieren diese, falls es mehrere sind, den Wiedereintrittspunkt erst, wenn alle eingehenden Kanten aktiviert sind.

Ausgehende Kanten verbinden den Wiedereintrittspunkt ausschließlich mit Elementen, die sich innerhalb der Sequenzkartusche befinden. Gibt es mehr als eine ausgehende Kante, muß jede eine eindeutig unterscheidbare Anschrift tragen, da ein aktivierter Wiedereintrittspunkt nur jeweils eine ausgehende Kante aktiviert. Das einer von einem *Startzustand* oder einem *Wiedereintrittszustand* ausgehenden Kante folgende Element $X-out$, kann ein beliebiges Element einer Sequenzkartusche sein, also entweder ein Last-Element oder ein *End-* oder *Austritts-Zustand* oder eine Parallelkartusche.

[12]Auch Kanten werden grundsätzlich als zeitlos interpretiert.

[13]Zur Vollständigkeit sei erwähnt, daß die Kanten, die zur Aktivierung beigetragen haben, danach deaktiviert werden.

Wiedereintrittspunkte, wie in Abbildung 2.2.11 dargestellt, sind bezüglich der ausgehenden Kanten mit den *Startzuständen* identisch. Ihre eingehenden Kanten kommen aus eingebetteten Parallelkartuschen[14].

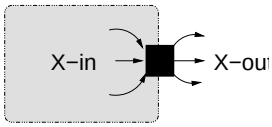


ABBILDUNG 2.2.12. Endpunkte

2.2.4.3. *Endpunkte. Endpunkte* sind zeitlos. Eine Sequenzkartusche besitzt genau einen Endpunkt, der auf der die Kartusche begrenzenden Linie, durch ein ausgefülltes Quadrat dargestellt wird. Der Endpunkt symbolisiert den Austritt aus der Kartusche. Kanten gehen in den Endpunkt nur von Elementen innerhalb der Kartusche ein. In der ausgezeichneten Wurzelkartusche, die selbst in keiner Parallelkartusche eingebettet ist, hat der Endpunkt keine ausgehenden Kanten. Jeder Endzustand hat eine bis beliebig viele eingehende Kanten.

Jeder weitere Endpunkt wird aktiviert, wenn eine eingehende Kante aktiviert ist. Von mehreren eingehenden Kanten wird stets nur eine einzige aktiviert. Ein solcher aktivierter Endpunkt aktiviert die ausgehende Kante, oder, falls es mehrere gibt, alle zugleich. Das Modell ist vollständig bearbeitet, wenn der Endpunkt der Wurzelkartusche aktiviert wird. Dies modelliert genau die Endpartition.

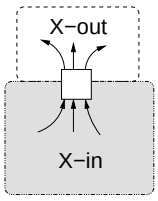


ABBILDUNG 2.2.13. Vorübergehende Austrittspunkte

2.2.4.4. *Vorübergehende Austrittspunkte.* Durch ein leeres Quadrat werden Zustände dargestellt, die einen vorübergehenden Austritt aus der serialisierbaren Region in eine zur Kartusche gehörigen, nebenläufigen Region symbolisieren. Es handelt sich dann um vorübergehende Austrittspunkte. Davon kann es in einer Sequenzkartusche mehrere geben, jedoch nur eine je Parallelkartusche. Zu jedem vorübergehenden Austrittspunkt gehört genau ein Wiedereintrittspunkt, wie er in Abschnitt 2.2.4.2 beschrieben wurde.

Das einer in den *Endzustand* oder den *Austrittszustand* eingehenden Kante vorausgehende Element $X - in$ kann ein Last-Element aus der serialisierbaren Region oder unmittelbar ein *Start-* oder *Wiedereintrittszustand* sein. Vorübergehende Austrittspunkte, wie in Abbildung 2.2.13 dargestellt, sind bezüglich der eingehenden Kanten mit

[14]Der *Startzustand* und der *Wiedereintrittszustand* entsprechen somit einem *UML-Join* mit anschließendem optionalem *Decision*-Knoten.

dem Verhalten der Endpunkte identisch. Ihre ausgehenden Kanten gehen jedoch in die nebenläufigen Regionen einer eingebetteten Parallelkartusche. Alle ausgehenden Kanten werden gleichzeitig aktiviert und nachfolgend alle Startpunkte.

Der End- und der vorübergehende Austrittspunkt entsprechen mit den folgenden Startpunkten somit einem *UML-Fork* mit vorgehendem, optionalem *Merge*-Knoten.

2.2.5. Modellstruktur. Wir führen eine formalere, zunächst nur statische Sicht auf das Gesamtmodell ein. In Beispiel 2.2.6.1 wird die Interpretation der statischen Struktur erläutert. Abschließend findet man die Struktur nochmals in der rekursiv formulierten formalen Kartuschensemantik in Abschnitt 2.3.5 wieder.

2.2.5.1. *Modellbaum.* Auf oberster Ebene betrachtet, ist ein dynamisches Modell M ein bipartiter Graph in Baumform. Die Knoten sind also in zwei disjunkte Mengen unterteilt. Es handelt sich dabei um Υ, die Menge der Parallelkartuschen und Θ, die Menge der Sequenzkartuschen. M ist dabei ein Baum mit einem ausgezeichneten Wurzelelement $K^0_{SK} \in \Theta$, in der Abbildung entspricht S_0 der Wurzelkartusche K^0_{SK}.

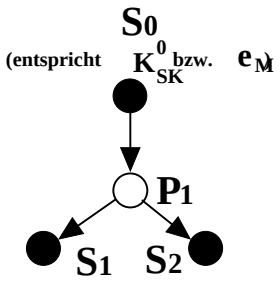


ABBILDUNG 2.2.14. Modell-Baum

In Abbildung 2.2.14 findet sich der einfache Baum des Beispiels aus Abbildung 2.2.5 mit einer parallelen Region. Der Knoten, welcher die Parallelkartusche verkörpert, ist durch einen leeren Kreis dargestellt. Die Repräsentanten der Sequenzkartuschen sind ausgefüllte Kreise. Die Blätter sind stets Sequenzkartuschen. Auch Parallelkartuschen, die nur einen Sohn haben, sind nicht sinnvoll. Viele weitere Eigenschaften ergeben sich aus der Definition der Einzelkomponenten.

DEFINITION 16 (Statische Struktur eines dynamischen Modells). *Die Struktur des Ausführungsmodells M ist ein Quintupel $(\Upsilon, \Theta, \mathcal{R}, \rho, e_M)$, mit*

> *Υ, einer endliche Menge von Parallelkartuschen,*
> *Θ, einer endliche Menge von Sequenzkartuschen,*
> *$\mathcal{R} \subseteq (\Upsilon \times \Theta) \cup (\Theta \times \Upsilon)$, der Kartuschenrelation,*
> *$\rho = \rho_{\mathcal{SK}} \cup \rho_{\mathcal{PK}}$, der Kartuschenzuordnungsfunktion und*
> *$e_M \in \Theta$, dem Startknoten des Modells.*
> *Ausserdem gilt: $|\Upsilon| + |\Theta| - 1 = |\mathcal{R}|$*
> *und der Graph $(\Upsilon \cup \Theta, \mathcal{R})$ ist zusammenhängend.*

Die Interpretation des Modells beginnt an der K^0_{SK}-Kartusche. Konkrete, ausführbare Lastmodelle, welche in dieser Kartusche enthalten sind, werden hier nicht behandelt. Das Hauptaugenmerk liegt auf nebenläufigen Bereichen, die abstrakt als Parallelkartuschen eingebunden werden und über eine entsprechende Relation $r \in \mathcal{R}$ erreichbar sind. Jede dieser Parallelkartuschen ist wiederum die Wurzel eines Teilbaumes, der diese nebenläufige Region modelliert. Auf der nächst tieferen Ebene ist eine nebenläufige Region erneut durch eine Menge an sequentialisierbaren Regionen modelliert, die jeweils durch die Relation zu einer Sequenzkartusche eingebunden werden. Jede so erreichbare Sequenzkartusche enthält entweder ausschließlich konkrete Lastmodelle, oder ist die Wurzel eines weiteren Teilbaumes, der in diesem Sinne rekursiv als Modell interpretiert wird. Es sind also p-Teilbäume mit Parallelkartuschen als Wurzeln von s-Teilbäume mit Sequenzkartuschen als Wurzeln unterscheidbar. Dies wird später bei der Übersetzung in ein Petrinetz benötigt.

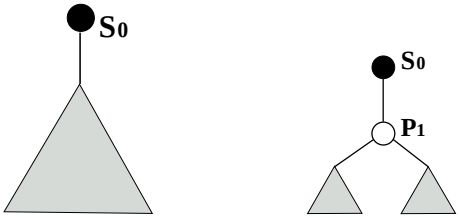


ABBILDUNG 2.2.15. Zwei Arten von Teilbäumen

Im einfachsten Fall besteht der Baum nur aus dem K^0_{SK}-Element, also dem im Beispiel mit S_0 bezeichneten Knoten. In den anderen Fällen lassen sich aus Sicht jedes Knotens, der kein Blatt ist weitere Teilbäume betrachten.

Für die im nächsten Kapitel definierte formale Semantik führen wir noch, aus syntaktischen Gründen, die Funktionen $\rho_{\mathcal{SK}}$ und $\rho_{\mathcal{PK}}$ ein. Sie ermitteln die jeweils zu den Zuständen, bzw. Knoten, gehörenden Kartuschen, die sie repräsentieren.

DEFINITION 17 (Ermittlung der eingebetteten Parallelkartuschen). *Die jedem Zustand $z \in \mathcal{Z}_{PK}$ einer Sequenzkartusche K_{SK} zugeordnete Parallelkartusche K_{PK} wird durch eine injektive Funktion*

$$\rho_{\mathcal{PK}} \ : \ \mathcal{Z}_{PK} \rightarrow \Upsilon$$

für eine Sequenzkartusche $K_{SK} = \left(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\right)$ *und* $z \in \mathcal{Z}_{PK} \subset \mathcal{Z}$ *ermittelt, setzen wir*

$$\rho_{\mathcal{PK}}(z) = K_{PK} \,.\, \left(K_{SK}, K_{PK}\right) \in \mathcal{R}.$$

Analog definieren wird die Funktion, welche die in Parallelkartuschen eingebetteten Sequenzkartuschen ermittelt.

DEFINITION 18 (Ermittlung der eingebetteten Sequenzkartuschen). *Die jedem Zustand* $e \in \mathcal{E}^+_{SK}$ *eines Modells oder einer Parallelkartusche* K_{PK} *des Modells zugeordnete Sequenzkartusche* K_{SK} *wird durch die Funktion*

$$\rho_{\mathcal{SK}} \;:\; \mathcal{E}^+_{SK} \to \Theta$$

mit Definitionsbereich $\mathcal{E}^+_{SK} = \mathcal{E}_{SK} \cup \{e_M\}$ *ermittelt.*

Durch die Erweiterung des Definitionsbereichs ist die Wurzelkartusche im Wertebereich von $\rho_{\mathcal{PK}}$ *und die Funktion kann wie folgt definiert werden:*

$$\rho_{\mathcal{SK}}(e) = \begin{cases} K_{SK} \,.\, \left(K_{PK}, K_{SK}\right) \in \mathcal{R}, \textit{ falls } e \in \mathcal{E}_{SK} \\ K^0_{SK}, \textit{ falls } e = e_M. \end{cases}$$

Diese Funktionen verwenden wir im nächsten Kapitel zu einer induktiv definierten, semantischen Funktion, die bei der Modellsemantik eingesetzt wird.

2.2.6. Beispielmodelle. Jeweils ein Beispiel für nebenläufige und sequentielle Modelle werden nun das statische Modell veranschaulichen.

2.2.6.1. *Nebenläufigkeit.* Das bereits vorgestellte Beispiel, welches in Abbildung 2.2.16 bereits mit den Annotationen aus dem Modell dargestellt ist, wird nun in das formale Modell übersetzt.

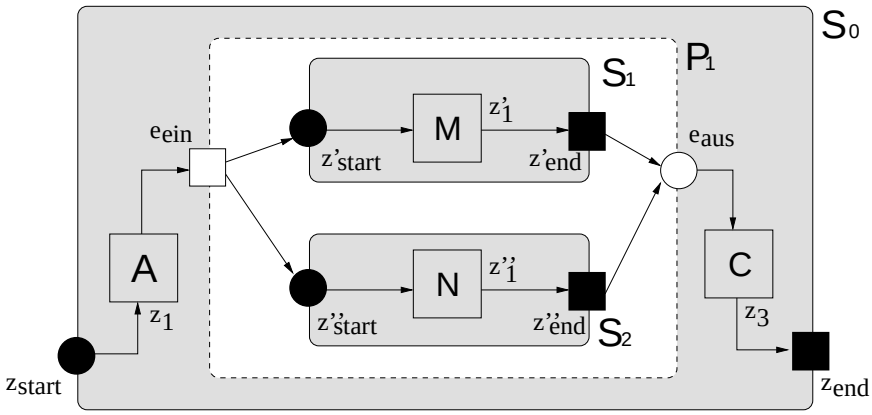


ABBILDUNG 2.2.16. Beispiel-Kartusche mit paralleler Region

BEISPIEL 4 (Beispielmodell mit Nebenläufigkeit).

$$\textit{Modell: } M \;=\; \big(\Upsilon, \Theta, \mathcal{R}, \rho, e_M\big)$$

Teilmengen:

$$\begin{aligned}
\textit{Parallelkartuschen: } \Upsilon &= \{P_1\} \\
\textit{Sequenzkartuschen: } \Theta &= \{S_0, S_1, S_2\} \\
\textit{Modellrelation: } \mathcal{R} &= \{(S_0, P_1), (P_1, S_1), (P_1, S_2)\} \\
\textit{Modellknoten mit:} &\quad e_M.(\rho_{\mathcal{SK}}(e_M) = S_0)
\end{aligned}$$

Kartuschen:

in Θ*:*

$$\begin{aligned}
S_0 &= \big(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\big) \textit{ mit:} \\
\textit{Lasten:} &\quad \mathcal{Z}_{LE} = \{(z_1 \;.\; \beta(z_1) = \mathsf{A}), (z_3 \;.\; \beta(z_3) = \mathsf{C})\} \\
\textit{Parallelkartuschen:} &\quad \mathcal{Z}_{PK} = \{(z_2 \;.\; \rho_{\mathcal{PK}}(z_2) = P_1)\} \\
\textit{Kartuschenalphabet:} &\quad \Sigma = \{\epsilon\} \\
\textit{Übergangsfunktion:} &\quad \delta = \{(z_{start}, \epsilon, z_1), (z_1, \epsilon, z_2), (z_2, \epsilon, z_3), (z_3, \epsilon, z_{end})\}
\end{aligned}$$

$$\begin{aligned}
S_1 &= \big(\mathcal{Z}', \Sigma', \delta', z'_{start}, z'_{end}\big) \textit{ mit:} \\
\textit{Lasten:} &\quad \mathcal{Z}'_{LE} = \{(z'_1 \;.\; \beta(z'_1) = \mathsf{M})\} \\
\textit{Parallelkartuschen:} &\quad \mathcal{Z}'_{PK} = \emptyset \\
\textit{Kartuschenalphabet:} &\quad \Sigma' = \{\epsilon\} \\
\textit{Übergangsfunktion:} &\quad \delta' = \{(z'_{start}, \epsilon, z'_1), (z'_1, \epsilon, z'_{end})\}
\end{aligned}$$

$$\begin{aligned}
S_2 &= \big(\mathcal{Z}'', \Sigma'', \delta'', z''_{start}, z''_{end}\big) \textit{ mit:} \\
\textit{Lasten:} &\quad \mathcal{Z}''_{LE} = \{(z''_1 \;.\; \beta(z''_1) = \mathsf{N})\} \\
\textit{Parallelkartuschen:} &\quad \mathcal{Z}''_{PK} = \emptyset \\
\textit{Kartuschenalphabet:} &\quad \Sigma'' = \{\epsilon\} \\
\textit{Übergangsfunktion:} &\quad \delta'' = \{(z''_{start}, \epsilon, z''_1), (z''_1, \epsilon, z''_{end})\}
\end{aligned}$$

in Υ*:*

$$\begin{aligned}
P_1 &= \big(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus}\big) \textit{ mit:} \\
\textit{Sequenzkartuschen:} &\quad \mathcal{E}_{SK} = \{(e_1 \;.\; \rho_{\mathcal{SK}}(e_1) = S_1), (e_2 \;.\; \rho_{\mathcal{SK}}(e_2) = S_2)\} \\
\textit{Kanten:} &\quad \mathcal{K} = \{(e_{ein}, e_1), (e_{ein}, e_2), (e_1, e_{aus}), (e_2, e_{aus})\}
\end{aligned}$$

Für dieses Beispiel wird am Ende des nächsten Abschnitts formal eine Semantik abgeleitet.

2.2.6.2. *Unverzweigter Kontrollfluß.* Für das Beispiel ohne nebenläufige Aspekte in Abbildung 2.2.17 lautet das formale Modell:

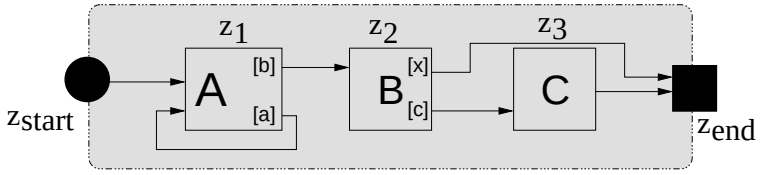


ABBILDUNG 2.2.17. Beispiel einer Sequenzkartusche

BEISPIEL 5 (Beispielmodell mit Bedingung und Schleife).

$$\textit{Modell: } M \;=\; \big(\Upsilon, \Theta, \mathcal{R}, \rho, e_M\big)$$

Teilmengen:

$$\begin{aligned} \textit{Parallelkartuschen:} &\quad \Upsilon = \emptyset \\ \textit{Sequenzkartuschen:} &\quad \Theta = \{S_0\} \\ \textit{Modellrelation:} &\quad \mathcal{R} = \emptyset \\ \textit{Modellknoten:} &\quad e_M.(\rho_{\mathcal{SK}}(e_M) = S_0) \end{aligned}$$

Kartuschen:

in Θ*:*

$S_0 = \big(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\big)$ *mit:*

$\mathcal{Z}_{LE} = \{\ (z_1.\beta(z_1) = \mathsf{A}), (z_2.\beta(z_2) = \mathsf{B}), (z_3.\beta(z_3) = \mathsf{C})\ \}$

$\mathcal{Z}_{PK} = \emptyset$

$\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{x}, \epsilon\}$

$\delta = \big\{(z_{start}, \epsilon, z_1), (z_1, \mathsf{a}, z_1), (z_1, \mathsf{b}, z_2), (z_2, \mathsf{c}, z_3),$
$(z_2, \mathsf{x}, z_{end}), (z_3, \epsilon, z_{end})\big\}$

Diese Beispiele sind Ausdrücke aus dem syntaktischen Bereich der dynamischen Kontrollflußmodelle, der im nächsten Abschnitt mit einer Semantik ausgestattet wird.

2.3. Kontrollflußsemantik

Motivation. Durch das formale Modell, welches wir im Abschnitt 2.2 eingeführt haben, wurden verschiedene strukturelle Aspekte getrennt behandelt, Begriffe definiert und Einschränkungen gefordert, die wichtige Eigenschaften begründen. Diese Begriffe und Einschränkungen sollen vor allem die Entwicklung eines Testwerkzeugs erleichtern. Die vorrangige Frage bei der Entwicklung solcher Software ist immer ein algorithmisch leicht zugänglicher Wohlgeformtheitsbegriff, der es erlaubt, ein zu bewertendes Modell elementweise zu behandeln. Mit dem Modell aus Abschnitt 2.2 gelingt es uns, nicht wohlgeformte dynamische Modelle zurückzuweisen. Für eine Simulation des Verhaltens sind aber noch keine ausreichenden dynamischen Eigenschaften spezifiziert.

Hierzu unterlegen wir die eingeführten Elemente erst jetzt mit einer eigenen Semantik. Dieses Vorgehen bringt zwar den Nachteil einer zusätzlichen Indirektion. Jedoch wird so die Gestaltung einer intuitiv leicht verständlichen graphischen Notation von der Definition einer exakten Semantik entflochten und voneinander unabhängig.

Als Formalismus für die Kontrollflußsemantik wählen wir Petrinetze. Wir ordnen jedem Element genau das dynamische Verhalten zu, das dem Schalten des Petrinetzes entspricht. Petrinetze sind ein mittlerweile weithin akzeptierter, leicht verständlicher Formalismus, der später einen formalen Zugang zur Interpretation des Performanz-Aspektes erlauben wird. Es existieren zahlreiche Standards und Implementierungen, die eine unmittelbare Speicherung und Verarbeitung der hier beschriebenen Modelle erlaubt [**52, 63, 64**]. Ein weiterer Vorteil dieser Indirektion ist die Möglichkeit die Semantik unmittelbar zu implementieren und eine Instanz eines so implementierten Netzes in einem Simulationwerkzeug zu untersuchen. So wird bei der praktischen Umsetzung in der Werkzeugentwicklung wieder eine Indirektionsstufe eingespart. Eben diese Indirektionsstufe ist in der Softwareentwicklung oft sehr fehlerträchtig, weil die Modelle dieser Phase weniger präzise ausfallen.

2.3.1. Informelle Einführung.

2.3.1.1. *Sequenzkartuschenübersetzung.* In Tabelle 1 sind die Übersetzungsmuster für die graphischen Elemente der Sequenzkartuschen schematisch aufgeführt. In der zweiten Zeile findet sich die Übersetzung als Petrinetz-Fragment in graphischer Darstellung. Sowohl Lasten als auch einzubindende Parallelkartuschen werden hier durch eine Stelle (als Kreis dargestellt) repräsentiert. Diese Vereinfachung finden wir in

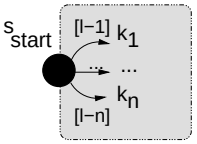
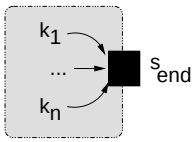
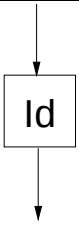
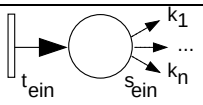
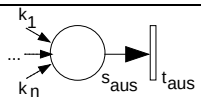
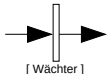
Startpunkt	Endpunkt	Kanten	Lasten	Parallel K.
				
				

TABELLE 1. Übersetzungstabelle der Sequenzkartuschen

der formalen Semantik nicht mehr. Beide Stellen müssen weiter verfeinert werden, was durch die Substitution durch ein Subnetz erreicht wird, welches die Semantik der Elemente repräsentiert. Dann werden immer zwei Stellen erzeugt, zwischen die später das Subnetz eingefügt wird. Transitionen (als Balken dargestellt) mit Ausnahme der einmaligen Übersetzung eines Start- und eines Endpunktes resultieren aus der Übersetzung von Modellkanten. Kanten des Petrinetzes haben immer die Vielfachheit 1 und werden immer in Verbindung mit einer Transition übersetzt. Deshalb besitzt eine Transition in einer Übersetzung einer Sequenzkartusche stets eine eingehende und eine ausgehende Kante. Eine Vermehrung oder Verminderung der Marken beim Schalten ist also ausgeschlossen.

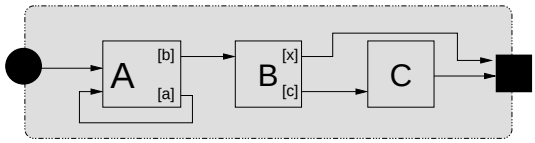


ABBILDUNG 2.3.1. Beispiel einer Sequenzkartusche

Abbildung 2.3.1 zeigt das schon in Abbildung 2.2.2 eingeführte Beispiel einer einfachen Sequenzkartusche. Zunächst betrachten wir das Petrinetz, welches sich aus der vereinfachten schematischen Übersetzung nach Tabelle 1 ergäbe.

In Abbildung 2.3.2 ist das Petrinetz dargestellt, welches sich aus der Übersetzung des Beispiels nach der Tabelle ergibt. Die mit `Start` und `Stop` bezeichneten, in grau dargestellten Stellen stammen aus der

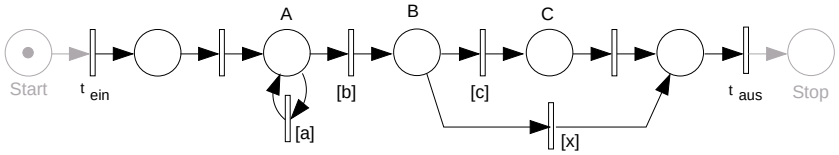


ABBILDUNG 2.3.2. Beispielübersetzung einer Sequenzkartusche (1.Schritt)

Übersetzung des Grundmodells. Dieses wird am Ende dieses Abschnittes eingeführt. Das Netz ist mit der Startmarkierung dargestellt.

Lastelemente und Parallelkartuschen haben verschiedene Semantiken. Die Übersetzung muß also durch eine Ersetzung der jeweiligen Stellen fortgeführt werden. Eine Last oder eine Parallelkartusche wird ihrerseits in ein stellenberandetes Subnetz transformiert, welches dann die Stelle aus der Kartuschenübersetzung ersetzt.

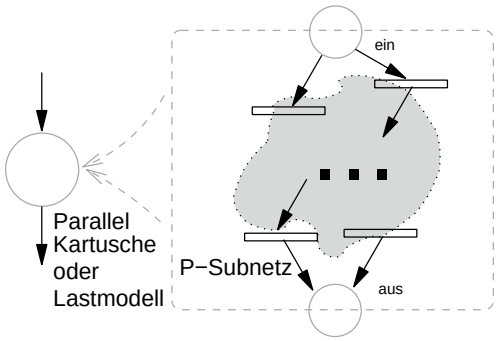


ABBILDUNG 2.3.3. Stellen-Substitution

Die Stellen eines Subnetzes aus einer Sequenzkartusche stammen nach Definition 1 entweder aus der Menge der Lastelemente, oder sie stammen aus der Menge der eingebetteten Sequenzkartuschen. In letzterem Fall repräsentieren diese Stellen Parallelkartuschen und damit p-Teilbäume. Die Lastelemente oder Parallelkartuschen werden jeweils in Subnetze übersetzt. Diese Stellen werden dann durch diese Subnetze substituiert.

2.3.1.2. *Lastzustände.* Jede Last wird im Modell durch ein einzelnes Symbol dargestellt. Zur Bewertung ihres Zeitverbrauchs muß eine Mindestzahl von inneren Bearbeitungszuständen unterschieden werden. Wird ein Lastelement durch eine eingehende Kante aktiviert, kann es gestartet werden. Es ist also *bereit*. Nachdem es gestartet wurde, ist sein Zustand *aktiv*, es belegt eine Ressource und verändert dadurch deren zeitliches Verhalten und die Ressource bestimmt den Zeitpunkt, zudem diese Last vollständig bearbeitet ist. Nach diesem Zeitpunkt ist

das Lastelement *fertig*. Erst dann kann eine vom Lastelement ausgehende Kante aktiviert werden. Abbildung 2.3.4 zeigt das Teilnetz, welches die Lastsemantik definiert.

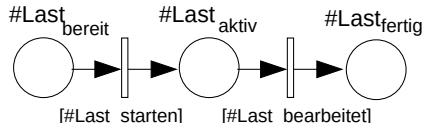


ABBILDUNG 2.3.4. Lastzustandssemantik (verfeinert)

Grundsätzlich ist davon auszugehen, daß die Bearbeitung einer Last unmittelbar nachdem sie bereit ist gestartet wird,. Sollte eine zusätzliche Synchronisation nötig sein, müßte dies im dynamischen Modell explizit spezifiziert werde. Der *bereit*-Zustand und die erste Transition können eliminiert werden. Der sofortige Start der Lastbearbeitung muß in der Implementierung der Simulationsumgebung realisiert werden. Somit gelangt man zu einem einfacheren semantischen Modell für die Lastelemente, wie in Abbildung 2.3.5 dargestellt.

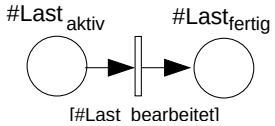


ABBILDUNG 2.3.5. Lastzustandssemantik (vereinfacht)

Die beiden Stellen für die Zustände *aktiv* und *fertig* sind selbst die randständigen Stellen, die in der Semantik zur Ersetzung der Stelle verwendet werden, welche bis dahin die Last repräsentierte. Bei der Substitution der Lastsubnetze wird in der vereinfachten Version lediglich eine Transition eingefügt, welche den Namen der Last trägt. Später kann sie vom Simulator, wenn sie konzessioniert ist, nach Ablauf der berechneten Bearbeitungszeit geschaltet werden. Ebenso ist der Einfluß dieser zusätzlichen Transition auf die Struktur des Erreichbarkeitsgraphen interessant.

2.3.1.3. *Parallelkartuschenübersetzung.* In Tabelle 2 sind die Übersetzungsmuster für die graphischen Elemente der Parallelkartuschen schematisch aufgeführt. Außer der Quelle e_{ein} und der Senke e_{aus} weist eine Parallelkartusche nur einen interessanten Knotentyp auf, der die eingebettete Sequenzkartusche repräsentiert. Jede Kante des Netzgraphen wird in eine Kombination aus einem eingehenden Bogen, einem ausgehenden Bogen und einer Stelle übersetzt. Dadurch wird sichergestellt, daß jede Stelle nur einen eingehenden und einen ausgehenden

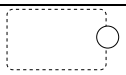
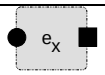
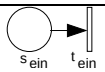
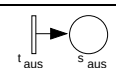
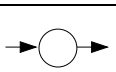
Startpunkt	Endpunkt	Kanten	Sequenzkartuschen
			
			

TABELLE 2. Übersetzungstabelle der Parallelkartuschen

Bogen besitzt. In Parallelkartuschen können mehrere Kanten aktiviert sein. Da aber keine Zyklen existieren, bleibt die 1-Sicherheit des Netzes gewahrt. Jeder Knoten, der eine Sequenzkartusche repräsentiert, wird in eine Transitionen übersetzt. Analog zur Semantik der Sequenzkartuschen werden diese Transitionen später substituiert, wie schematisch in Abbildung 2.3.6 dargestellt.

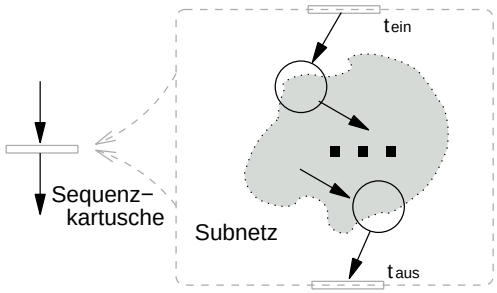


ABBILDUNG 2.3.6. Transitions-Substitution

Aus der Übersetzungstabelle der Sequenzkartuschen war bereits zu ersehen, daß die resultierenden Netze genau die geforderten randständigen Stellen aufweisen.

2.3.1.4. *Gesamtmodell.* Die Interpretation eines Modells beginnt an der Wurzelkartusche, der sogenannten K^0_{SK}-Kartusche. Wir betrachten zunächst informell ein Grundmodell M_0, welches die Semantik eines Modells auf der abstraktesten Ebene repräsentiert.

Startmarkierung Endmarkierung

ABBILDUNG 2.3.7. Petrinetz des Grundmodells

Das Startmuster zur Übersetzung des Petrinetzes besteht aus dem Netz nach dem Schema in Abbildung 2.3.7. Die Modell-Transition ist in

der Startmarkierung $\mathtt{m}_0$ konzessioniert. Die Abarbeitung eines konkreten Szenarios eines dynamischen Modells wird durch das Schalten der einzigen Transition $\mathtt{t}_{Modell}$ verkörpert. Die Nachmarkierung $\mathtt{m}_{stop}$ modelliert das bearbeitete Szenario. Die Modell-Transition ist hier nicht mehr konzessioniert. [15]

Ohne (den trivialen) Beweis halten wir die wichtigen Eigenschaften dieses Netzes fest: M_0 ist 1-beschränkt und die Endmarkierung ist erreichbar. Später ist noch für jedes Subnetz zu zeigen, daß diese Eigenschaften dort ebenso nicht verletzt werden.

Die initiale Interpretation der K^0_{SK}-Kartusche ist die erste Transitionssubstitution. Ähnlich wie in der Deklaration der Teilfunktionen erzeugen wir unmittelbar die Start- und Stop-Stellen, eine Eingangstransition ${}_{\mathrm{I}}K^0_{SK}$, eine Ausgangstransition $K^0_{SK\mathrm{I}}$ und die beide Elemente der Flußrelation. Dann substituiert man die K^0_{SK}-Kartusche. Die dort möglicherweise eingebetteten Parallelkartuschen werden rekursiv durch die bekannten Substitutionsfunktionen eingebettet. Die Rekursion bricht an den Sequenzkartuschen K_{SK} ab, deren Teilmenge $\mathcal{Z}_{PK}$ leer ist.

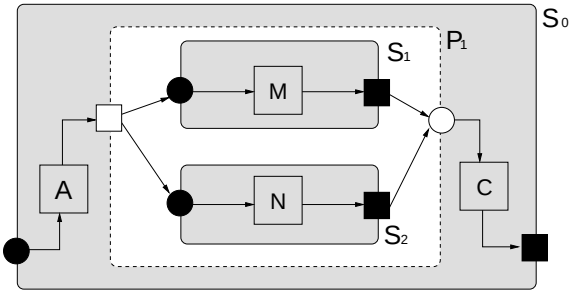


ABBILDUNG 2.3.8. Kartusche mit paralleler Region

2.3.1.5. *Beispiel.* In Abbildung 2.3.8 sieht man nochmals das Beispiel aus Abbildung 2.2.5, erweitert um die Bezeichnung der Kartuschen, damit im Folgenden die Übersetzung schrittweise nachvollzogen werden kann.

Zunächst wird die S_0-Kartusche übersetzt. Das Netz in dieser Zwischensituation ist in Abbildung 2.3.9 dargestellt. [16] Hier sind die Elemente aus der Übersetzung des Grundmodells und der S_0-Kartusche zu sehen. Die Stelle P_1, welche aus der Parallelkartusche P_1 entstanden ist, wird in diesem Netz noch nicht substituiert.

[15] Hier findet sich auch die Erklärung der im Beispiel 2.3.2 eingeführten Start- und Stop-Stellen.

[16] Wobei zu bedenken ist, daß die später zu substituierenden Stellen nicht wirklich erzeugt werden.

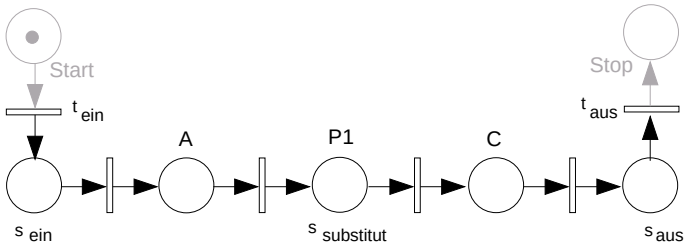


ABBILDUNG 2.3.9. Übersetzung der S_0-Kartusche (1.Schritt)

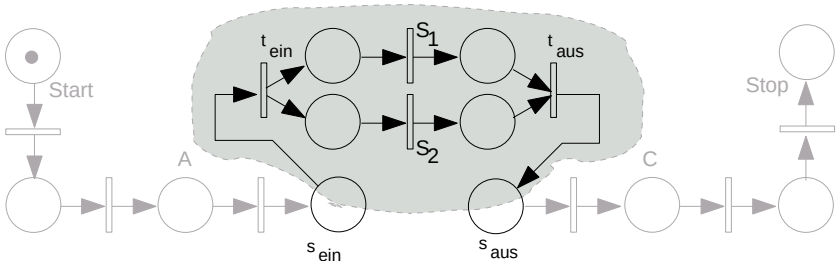


ABBILDUNG 2.3.10. Übersetzung der Parallelkartusche (2.Schritt)

In Abbildung 2.3.10 zeigen wir das Subnetz aus der zu substituierenden Parallelkartusche an ihrem Platz. Die Elemente des vorherigen Modells sind grau dargestellt. Ebenso sind die Bezeichner weggelassen, zB. $\mathtt{t}_{ein}$ oder $\mathtt{s}_{ein}$, die nach der letzten Substitution nicht mehr notwendig sind. Wir sehen im Subnetz die mit $\mathtt{S}_1$ und $\mathtt{S}_2$ bezeichneten Transitionen, die für die eingebetteten Sequenzkartuschen eingefügt wurden. Diese Transitionen müssen nun wiederum durch je ein Subnetz substituiert werden. Die beiden Subnetze sind denkbar einfach aufgebaut und bestehen nur aus je einem Lastelement M und N. Dem bekannten Mechanismus folgend, werden Ein- und Ausgangstransitionen und Stellen beigefügt und mit den Subnetzen die Transitionen substituiert.

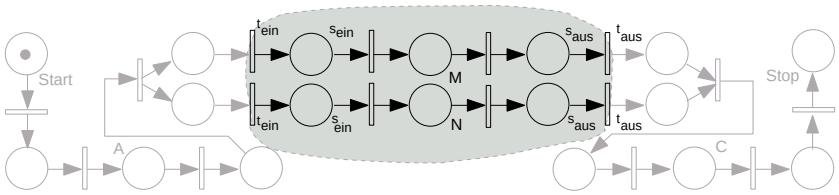


ABBILDUNG 2.3.11. Übersetzung der eingebetteten Sequenzkartuschen (3.Schritt)

In Abbildung 2.3.11 ist das resultierende Gesamtnetz zu sehen. Zwar ist die Herkunft jeder Transition und Stelle nachvollziehbar, aber

das Netz wurde durch Einfügung vieler Leerstellen und redundanter Transitionen unübersichtlich. Für die folgenden Beispiele werden wir uns oft auf die Elemente des Netzes beschränken, die wichtige Annotationen tragen oder zu dynamischen Eigenschaften entscheidend beitragen.

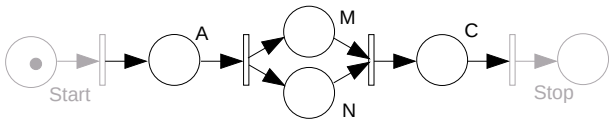


ABBILDUNG 2.3.12. Reduktion unbeschrifteter Stellen und spontaner Transitionen (Zum leichteren Verständnis)

Wie bei dieser Betrachtung interpretieren wir jede Stelle, die mit einer Last assoziiert ist, als deren Zustand. Die Last sehen wir als in Bearbeitung befindlich, falls sich eine Marke an dieser Stelle befindet. So bleibt vom Gesamtnetz von Abbildung 2.3.11 nach Reduktion nicht bedeutender Elemente nach dieser Lesart nur noch das in Abbildung 2.3.12 dargestellte Netz übrig. Dieser Vorgang wird in der formalen Semantik im Folgenden nicht beschrieben und dient für dieses Beispiel der leichteren Lesbarkeit der Diagramme.

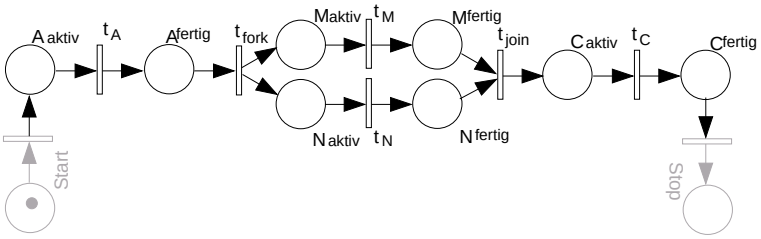


ABBILDUNG 2.3.13. Unterscheidung laufender und bearbeiteter Lasten

Nach der Substitution der Lastmodelle (hier in der reduzierten Form) ergibt sich das Gesamtbild, wie in Abbildung 2.3.13 dargestellt. Die Transitionen, welche zwischen den mit *aktiv* und *fertig* gekennzeichneten Lastzuständen liegen, werden bei der Simulation noch einer weiteren Diskussion unterliegen. Nach dieser informellen Einführung folgt die Definition einer formalen Semantik für das dynamische Kontrollflußmodell.

2.3.2. Syntaktische Bereiche. Wir verwenden die in Abschnitt 2.2 formal eingeführten Elemente als syntaktische Bereiche. Zur

leichteren Lesbarkeit der in diesem Abschnitt eingeführten Definitionen verwenden wir folgende Konvention.

$$\begin{array}{ll} K_{SK} = & \big(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\big), \text{ eine Sequenzkartusche aus} \\ \mathbb{K}_{SK}, & \text{der Menge der betrachteten Sequenzkartuschen, es wird} \\ \mathcal{Z}_{LE}, & \text{die Menge der Lastelemente und} \\ \mathcal{Z}_{PK}, & \text{die Menge der Parallelkartuschen unterschieden und} \\ \mathcal{Z} = & \mathcal{Z}_{LE} \cup \mathcal{Z}_{PK} \cup \{z_{start}\} \cup \{z_{end}\}. \end{array}$$

Für Parallelkartuschen bezeichnen wir nach folgender Konvention:

$$\begin{array}{ll} K_{PK} = & \big(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus}\big), \text{ eine Parallelkartusche aus} \\ \mathbb{K}_{PK}, & \text{der Menge der betrachteten Parallelkartuschen, dabei ist} \\ \mathcal{E}_{SK}, & \text{die Teilmenge der Sequenzkartuschen, für die gilt} \\ \mathcal{E} = & \mathcal{E}_{SK} \cup \{e_{ein}\} \cup \{e_{aus}\}. \end{array}$$

Dynamische Modelle bezeichnen wir im Folgenden nach dieser Konvention

$$\begin{array}{ll} M = & \big(\Upsilon, \Theta, \mathcal{R}, K^0_{SK}\big), \text{ ein dynamisches Modell aus} \\ \mathbb{M}, & \text{der Menge der betrachteten dynamischen Modelle.} \end{array}$$

Damit ist die endliche Menge der Parallelkartuschen eines dynamischen Modells eine Teilmenge des Definitionsbereichs $\Upsilon \subseteq \mathbb{K}_{PK}$ und die endliche Teilmenge der Sequenzkartuschen eine Teilmenge des Definitionsbereichs $\Theta \subseteq \mathbb{K}_{SK}$.

Für die Definition der folgenden Funktionen verwenden wir stets die entsprechenden Kleinbuchstaben wie für die Zustände der Sequenzkartusche $z \in \mathcal{Z}$, die Bezeichner eines Zustandsübergangs $l \in \Sigma$, einen Knoten im Graphen der Parallelkartusche $e \in \mathcal{E}$ und eine Kante im Graphen der Parallelkartusche $k \in \mathcal{K}$.

Wo in Definitionen Kartuschen als Parameter verwendet werden, bezeichnen wir sie mit K_{SK} im Falle einer Sequenzkartusche und mit K_{PK} im Falle einer Parallelkartusche, die gegebenenfalls zusätzlich indiziert werden können.

2.3.3. Semantische Bereiche. Struktur und Schaltregeln von Petrinetzen definieren das genaue Verhalten der Modelle. In der Literatur der letzten Jahre wurden Petrinetze verschiedenster Komplexität beschrieben. Um die Lesart der hier verwendeten einfachen Petrinetze eindeutig festzulegen, führen wir die nötigen Definitionen kurz ein.

2.3.3.1. *Einführung in einfache Petrinetze.*

DEFINITION 19 (Petrinetz). *Ein Quintupel* $N = (S, T, F, V, m_0)$ *wird Petrinetz (oder Stellen/Transitionsnetz)* N *genannt, wenn*

S, T	*jeweils zwei Mengen sind, für die gilt:*
$S \cap T = \emptyset$	*, und*
$F \subseteq (S \times T) \cup (T \times S)$	*eine Relation (Flußrelation) ist, und*

V, *die sogenannte* Vielfachheit, *eine Funktion ist, die jedem Element* f *der Relation* F *eine natürliche Zahl zuordnet:*

$$V : F \to \mathbb{N}_0^+$$

und jede Abbildung der Stellenmenge S *in die natürlichen Zahlen* $\mathbb{N}_0^+$ *ist eine sogenannte Anfangsmarkierung* $m_0 \in \mathbb{N}^S$.

Wir verwenden als semantischen Bereich sogenannte gewöhnliche Petrinetze.

DEFINITION 20 (Gewöhnliches Petrinetz). *Ein Petrinetz, für das gilt* $\forall f \in F \; . \; V(f) = 1$, *heißt* gewöhnliches *Petrinetz.*

Da im Folgenden nur gewöhnliche Petrinetze betrachtet werden, kann die Vielfachheit der Kanten ignoriert werden. Ein Petrinetz N wird hinreichend durch (S, T, F, m) beschrieben. Alle Definitionen werden ebenso nur für gewöhnliche Netze eingeführt.

DEFINITION 21 (Konzession). *Sei* $N = (S, T, F, m_0)$ *ein Petrinetz,* m *eine Markierung (oder Belegung) von* N *und* $t \in T$ *eine Transition des Netzes. Dann hat* t *Konzession, falls gilt* $(\forall s \in S \; . \; (s, t) \in F) \Rightarrow m(s) \geq 1$

DEFINITION 22 (Hilfsfunktionen). *Wir führen drei Hilfsfunktionen ein. Sei* $s \in S$ *eine beliebige Stelle und* $t \in T$ *eine beliebige Transition, dann definieren wir als:*

$$t^+(s) = \begin{cases} 1, & \textit{falls } (t, s) \in F \\ 0, & \textit{sonst.} \end{cases}$$

$$t^-(s) = \begin{cases} 1, & \textit{falls } (s, t) \in F \\ 0, & \textit{sonst.} \end{cases}$$

$$\textit{und } \Delta t(s) = \; t^+(s) - t^-(s).$$

Das Schalten einer Transition und damit die Ableitung einer Folgemarkierung m' aus einer Markierung m wird mit diesen Funktionen wie gewohnt definiert.

Das Schalten der Transitionen ist nicht alleine abhängig von deren Konzessionierung. Eingabe und Interaktion mit der Umgebung (realisiert durch das Szenario) und zeitliche Abhängigkeiten, die durch den Simulator aufgelöst werden, legen später die Schaltreihenfolgen fest. Darauf wird im Folgenden detailiert eingegangen.

2.3.3.2. *Notation der semantischen Kategorien.* Als semantischer Bereich finden gewöhnliche Petrinetze Verwendung. Die Kontrollflußsemantik konzentriert sich zunächst hauptsächlich auf die Mengen der Stellen und Transitionen und der Flußrelation.

DEFINITION 23 (Semantische Kategorien).

$N = (S, T, F)$ *das einem Petrinetz zugrundeliegende Netz mit*
$\mathbb{S}$ *der Menge der betrachteten Teilmengen der Stellen,*
$\mathbb{T}$ *der Menge der betrachteten Teilmengen der Transitionen,*
$\mathbb{F}$ *der Menge der betrachteten Teilmengen der Flußrelationen,*
$\mathbb{P}$ *der Menge aller betrachteten Netze und im Einzelnen, für deren Elemente* $N, N' \in \mathbb{P}$ *gelte* $(S \cup S') \cap (N \cup N') = \emptyset$.

2.3.3.3. *Besondere Operatoren.* Zur Vereinigung zweier Netze mittels der Vereinigung ihrer Teilmengen benutzen wir den Vereinigungsoperator:

DEFINITION 24 (Netzvereinigung). *Es seien zwei Petrinetze* $N = (S, T, F)$,$N' = (S', T', F') \in \mathbb{P}$, *dann ist*

$$N \cup N' := (S \cup S', T \cup T', F \cup F')$$

Es wird ein Netzsummen-Operator für die Vereinigung aller Netze, die durch die Funktion f aus den Elementen einer Menge erzeugt werden können, eingeführt.

DEFINITION 25 (Netzsumme). *Gegeben eine beliebige Funktion* $f : \mathbb{X} \longrightarrow \mathbb{P}$, *die aus beliebigen Parametern* $x \in \mathcal{X} \subseteq \mathbb{X}$ *ein Netz erzeugt. Gegeben die Menge* $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, *die n Elemente enthält, auf welche durch einen Index* $i = 1 \leq i \leq n$ *als* x_i *zugegriffen werden kann. Dann definieren wir*

$$\bigcup_{x \in \mathcal{X}} f(x) := f(x_1) \cup f(x_2) \cup \ldots \cup f(x_n)$$

2.3.3.4. *Semantische Hilfsfunktionen.* Die Übersetzung der Elemente in ein Petrinetz führt, wie in der informellen Einführung gezeigt,

manchmal zu einem einzelnen Element manchmal aber auch zu mehreren Elementen des semantischen Bereiches. Um die korrekte Verbindung der Elemente im semantischen Bereich untereinander zu gewährleisten, aber auch eine Zuordnung zum Element aus dem ursprünglichen Definitionsbereich zu ermöglichen, werden diese Elemente des Wertebereichs von S und T durch Paare $(elem, type)$ dargestellt. Dabei ist $elem$ das Element des Wertebereiches und $type$ ein Element aus:

$$type ::= \texttt{Spre} \mid \texttt{Spost} \mid \texttt{Sitem} \mid \texttt{Tpre} \mid \texttt{Tpost} \mid \texttt{Titem}$$

Um die textlich umfangreichen Beispiele der Petrinetzsemantik im Folgenden übersichtlicher zu halten, werden folgende Operatoren als Abkürzung verwendet:

DEFINITION 26 (Semantische Operatoren). *Sei $x \in \mathbb{X}$ ein Element eines beliebigen Definitionsbereichs und $\mathcal{F}[\![-]\!] : \mathbb{X} \to \mathbb{P}$ eine beliebige Funktion, die das Element des Definitionsbereichs in ein Petrinetz übersetzt, mit $\mathcal{F}[\![x]\!] = (S, T, F)$, dann ist*

$$\begin{aligned}
\circ x &= (elem, type).(elem = x) \wedge (type = \texttt{Spre}) \in S \\
x \circ &= (elem, type).(elem = x) \wedge (type = \texttt{Spost}) \in S \\
\circ x \circ &= (elem, type).(elem = x) \wedge (type = \texttt{Sitem}) \in S \\
{}_{\mid} x &= (elem, type).(elem = x) \wedge (type = \texttt{Tpre}) \in T \\
x_{\mid} &= (elem, type).(elem = x) \wedge (type = \texttt{Spost}) \in T \\
{}_{\mid} x_{\mid} &= (elem, type).(elem = x) \wedge (type = \texttt{Sitem}) \in T
\end{aligned}$$

Analog dazu wird eine Umkehrfunktion definiert, die erst in Kapitel 3 benötigt wird. Dort ermöglicht es bei der Definition einer Szenariogrammatik auf die Elemente des dynamischen Modell zuzugreifen, die für entsprechende Petrinetzelemente ursächlich waren.

DEFINITION 27 (Semantische Umkehrfunktion). *Sei für ein e, als ein Element einer beliebigen Menge eines Petrinetzes mit $(e \in S) \vee (e \in T) \vee (e \in F)$, dessen Domäne bezeichnet mit $\mathbb{E}$. Damit sei eine Umkehrfunktion*

$$elem : \mathbb{E} \to \mathbb{X}$$

gegeben derart, daß

$$elem(\mathcal{F}[\![x]\!]) = x.$$

Auf die Elemente einzelner Übersetzungsfunktionen muß in anderen Funktionen zugegriffen werden können, damit eine Verbindung (namentlich durch die Elemente der Flußrelation) mit den richtigen

Elementen erfolgen kann. Die interessanten Elemente sind die Stellen $(elem, \texttt{Spre})$ und $(elem, \texttt{Spost})$ und Transitionen $(elem, \texttt{Tpre})$ und $(elem, \texttt{Tpost})$, welche die randständigen Netzelement der Subnetzübersetzungen sind. Außerdem ist ein Zugriff auf die Elemente nötig, die wichtige Funktionen tragen oder weiter substituiert werden müssen.

2.3.4. Lastsemantik. Wir definieren nun exakt die verfeinerte Lastsemantik aus Abbildung 2.3.4.

DEFINITION 28 (Lastsemantik). *Die Semantik der Lastelemente wird durch folgende Funktion definiert:*

$$\begin{aligned} \mathcal{LE}[\![-]\!] &: \mathcal{Z}_{LE} \longrightarrow \mathbb{P} \\ \textit{mit } \mathcal{LE}[\![z]\!] &= \big(\{\circ z, z \circ\}, \\ &\qquad \{\shortmid z \shortmid\}, \\ &\qquad \{(\circ z, \shortmid z \shortmid), (\shortmid z \shortmid, z \circ)\}\big) \end{aligned}$$

2.3.5. Kartuschensemantik. Wir definieren nun die Semantik der beiden Kartuschenarten. Zur Definition der Semantik einer Kartuschenart wird die Semantik der jeweils anderen Kartuschenart benötigt.

2.3.5.1. *Signaturen.* Die Paralellkartuschen-Semantik übersetzt die Kartusche selbst in ein Subnetz und verbindet sie mit dem Netz der Kartusche, in die sie eingebettet war.

DEFINITION 29 (Parallelkartuschensemantik). *Die semantische Funktion der Parallelkartusche hat die Form:*

$$\mathcal{PK}[\![-]\!](-,-) \;:\; \Upsilon \to (\mathbb{M} \times \mathcal{Z}_{PK} \longrightarrow \mathbb{P})$$

Analog dazu benötigt die semantische Funktion der Sequenzkartusche noch einmal das Knotenelement der Parallelkartusche, in die sie eingebettet ist, um ihre Einbettung in das Netz zu erreichen.

DEFINITION 30 (Sequenzkartuschensemantik). *Die semantische Funktion der Sequenzkartusche hat die Form:*

$$\mathcal{SK}[\![-]\!](-,-) \;:\; \Theta \to (\mathbb{M} \times \mathcal{E}_{SK} \longrightarrow \mathbb{P})$$

2.3.5.2. *Formale Sequenzkartuschensemantik.* Lasten und Parallelkartuschen sind definiert durch stellenberandete Subnetze. Die Semantik der Zustandsübergangsfunktion δ muß diese Subnetze verbinden.

DEFINITION 31 (Übergangsfunktionssemantik). *Die Übergangsfunktion δ ist definiert durch*

$$\begin{aligned}
&\mathcal{T}[\![-]\!] : \delta \longrightarrow \mathbb{P} \\
&\quad mit\ \mathcal{T}[\![\delta_i = (z, l, z')]\!] = \big(\{z\circ, \circ z'\} \\
&\qquad\qquad\qquad\qquad \{{}_{\shortmid}\delta_{i\shortmid}\}, \\
&\qquad\qquad\qquad\qquad \{(z\circ, {}_{\shortmid}\delta_{i\shortmid}), ({}_{\shortmid}\delta_{i\shortmid}, \circ z')\}\big)
\end{aligned}$$

Die semantische Gesamtfunktion der Sequenzkartusche lautet so:

DEFINITION 32 (Formale Semantik der Sequenzkartuschen).

$$\begin{aligned}
&\mathcal{SK}[\![-]\!](-,-) : \Theta \to (\mathbb{M} \times \mathcal{E}_{SK} \longrightarrow \mathbb{P}) \\
&\qquad mit\ \mathcal{SK}[\![\big(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\big)]\!](M, e) \ = \\
&\qquad\quad \big(\{z_{start}\circ, \circ z_{end}\}, \\
&\qquad\qquad \{{}_{\shortmid}e\ , e_{\shortmid}\}, \\
&\qquad\qquad \{({}_{\shortmid}e\ , z_{start}\circ), (\circ z_{end}\ , e_{\shortmid})\}\big) \\
&\qquad \cup \bigcup_{z \in \mathcal{Z}_{LE}} \mathcal{LE}[\![z]\!] \\
&\qquad \cup \bigcup_{z \in \mathcal{Z}_{PK}} \mathcal{PK}[\![\rho_{\mathcal{PK}}(z)]\!](M, z) \\
&\qquad \cup \bigcup_{(z,l,z') \in \delta} \mathcal{T}[\![(z, l, z')]\!]
\end{aligned}$$

2.3.5.3. *Formale Parallelkartuschensemantik.*

DEFINITION 33 (Kantensemantik). *Die Semantik der Kantenelemente der Parallelkartuschen ist formal definiert mit*

$$\begin{aligned}
&\mathcal{K}[\![-]\!] \ : \ \mathcal{K} \longrightarrow \mathbb{P} \\
&\qquad\qquad mit\ \mathcal{K}[\![k]\!] = \big(\{\circ k\circ\}, \\
&\qquad\qquad\qquad\qquad \{k^{-}{}_{\shortmid}\ ,\ {}_{\shortmid}k^{+}\}, \\
&\qquad\qquad\qquad\qquad \{(k^{-}{}_{\shortmid}, \circ k\circ), (\circ k\circ, {}_{\shortmid}k^{+})\}\big)
\end{aligned}$$

Damit läßt sich die Parallelkartusche durch folgende Gesamtfunktion darstellen:

DEFINITION 34 (Formale Semantik der Parallelkartuschen).

$$\begin{aligned}
\mathcal{PK}[\![-]\!](-,-) :\ & \Upsilon \to (\mathbb{M} \times \mathcal{Z}_{PK} \longrightarrow \mathbb{P}) \\
& \textit{mit } \mathcal{PK}[\![(\mathcal{E}, \mathcal{K}, e_{ein}, e_{aus})]\!](M, z.z \in \mathcal{Z}_{PK}) \ = \\
& \quad \big(\{\circ z, z\circ\}, \\
& \quad\ \{e_{ein}{}_{\shortmid}, {}_{\shortmid}e_{aus}\}, \\
& \quad\ \{(\circ z, e_{ein}{}_{\shortmid}), ({}_{\shortmid}e_{aus}, z\circ)\}\big) \\
& \cup \bigcup_{e \in \mathcal{E}_{SK}} \mathcal{SK}[\![\rho_{\mathcal{SK}}(e)]\!](M, e) \\
& \cup \bigcup_{k \in \mathcal{K}} \mathcal{K}[\![k]\!]
\end{aligned}$$

In der Parallelkartusche sind alle Knoten Repräsentanten von Sequenzkartuschen. Die semantischen Funktionen der Sequenzkartuschen erzeugen transitionsberandete Subnetze. Die Kanten des Kartuschengraphen müssen die richtigen randständigen Ein- und Ausgangstransitionen der Subnetze verbinden. Dazu führen wir zunächst die semantische Funktion für die Kanten ein.

2.3.6. Modellsemantik. Die informelle Einführung zeigte, daß die Interpretation eines Modells an der K^0_{SK}-Kartusche beginnt.

Die initiale Interpretation der K^0_{SK}-Kartusche ist die erste Transitionssubstitution. Ähnlich wie in der Deklaration der Teilfunktionen erzeugen wir unmittelbar die Start- und Stop-Stellen, eine Eingangstransition, eine Ausgangstransition und die beiden Elemente der Flußrelation. Dann substituiert man die K^0_{SK}-Kartusche. Die dort möglicherweise einzufügenden Parallelkartuschen werden rekursiv durch die bekannten Substitutionsfunktionen eingebettet. Die Rekursion bricht an den Sequenzkartuschen K_{SK} ab, deren Teilmenge $\mathcal{Z}_{PK}$ leer ist.

DEFINITION 35 (Modellsemantik). *Formal ist die Semantik des Kontrollflußmodells gegeben mit:*

$$\mathcal{M}[\![-]\!] \ :\ \mathbb{M} \longrightarrow \mathbb{P}$$

Gegeben ein Modell $M = (\Upsilon, \Theta, \mathcal{R}, \rho, e_M)$, *dann ist*

$$\begin{aligned}
\mathcal{M}[\![(\Upsilon, \Theta, \mathcal{R}, \rho, e_M)]\!] \ = & \\
& \big(\{Start, Stop\}, \\
& \ \{{}_{\shortmid}e_M, e_M{}_{\shortmid}\}, \\
& \ \{(Start, {}_{\shortmid}e_M), (e_M{}_{\shortmid}, Stop)\}\big) \\
& \cup \mathcal{SK}[\![\rho_{\mathcal{SK}}(e_M)]\!](M, e_M)
\end{aligned}$$

2.3.7. Beispiele. Als Beispiel wird nun das in Abschnitt 2.2.6.1 in seiner Syntax vollständig dargestellte Modell eines nebenläufigen Ablaufes interpretiert und seine Semantik durch die vorgestellten Funktionen abgeleitet.

BEISPIEL 6 (Beispielsemantik mit Nebenläufigkeit). *Die Modellsemantik ergibt sich zu:*

$$\begin{aligned}\mathcal{M}[\![(\Upsilon,\Theta,\mathcal{R},\rho,e_M)]\!] =& \big(\{Start,Stop\},\{{\shortmid}e_M,e_M{\shortmid}\},\\ &\{(Start,{\shortmid}e_M),(e_M{\shortmid},Stop)\}\big)\\ &\cup\,\mathcal{SK}[\![\rho_{\mathcal{SK}}(e_M)]\!](M,e_M)\end{aligned}$$

mit $\rho_{\mathcal{SK}}(e_M)=S_0$ *ergibt sich der zweite Summand zu:*

$$\begin{aligned}&\mathcal{SK}[\![S_0=(\mathcal{Z},\Sigma,\delta,z_{start},z_{end})]\!](M,e_M)=\\ &\quad\big(\{z_{start}\circ,\circ z_{end}\},\{{\shortmid}e_M\ ,e_M{\shortmid}\},\{({\shortmid}e_M\ ,z_{start}\circ),(\circ z_{end}\ ,e_M{\shortmid})\}\big)\\ &\cup\,\mathcal{LE}[\![z_1]\!]\cup\mathcal{LE}[\![z_3]\!]\\ &\cup\,\mathcal{PK}[\![\rho_{\mathcal{PK}}(z_2)]\!](M,z_2)\\ &\cup\,\mathcal{T}[\![\delta_1=(z_{start},\epsilon,z_1)]\!]\cup\mathcal{T}[\![\delta_2=(z_1,\epsilon,z_2)]\!]\\ &\quad\cup\,\mathcal{T}[\![\delta_3=(z_2,\epsilon,z_3)]\!]\cup\mathcal{T}[\![\delta_4=(z_3,\epsilon,z_{end})]\!]\end{aligned}$$

Die Semantik der Kanten löst sich auf zu:

$$\begin{aligned}\mathcal{T}[\![\delta_1=(z_{start},\epsilon,z_1)]\!]&=(\{z_{start}\circ,\circ z_1\},\{{\shortmid}\delta_1{\shortmid}\},\{(z_{start}\circ,{\shortmid}\delta_1{\shortmid}),({\shortmid}\delta_1{\shortmid},\circ z_1)\})\\ \mathcal{T}[\![\delta_2=(z_1,\epsilon,z_2)]\!]&=(\{z_1\circ,\circ z_2\},\{{\shortmid}\delta_2{\shortmid}\},\{(z_1\circ,{\shortmid}\delta_2{\shortmid}),({\shortmid}\delta_2{\shortmid},\circ z_2)\})\\ \mathcal{T}[\![\delta_3=(z_2,\epsilon,z_3)]\!]&=(\{z_2\circ,\circ z_3\},\{{\shortmid}\delta_3{\shortmid}\},\{(z_2\circ,{\shortmid}\delta_3{\shortmid}),({\shortmid}\delta_3{\shortmid},\circ z_3)\})\\ \mathcal{T}[\![\delta_4=(z_3,\epsilon,z_{end})]\!]&=(\{z_3\circ,\circ z_{end}\},\{{\shortmid}\delta_4{\shortmid}\},\{(z_3\circ,{\shortmid}\delta_4{\shortmid}),({\shortmid}\delta_4{\shortmid},\circ z_{end})\})\end{aligned}$$

Die Semantik der Lastelemente ergibt im Einzelnen:

$$\mathcal{LE}[\![z_1]\!]=\big(\{\circ z_1,z_1\circ\},\{{\shortmid}z_1{\shortmid}\},\{(\circ z_1,{\shortmid}z_1{\shortmid}),({\shortmid}z_1{\shortmid},z_1\circ)\}\big)$$

und

$$\mathcal{LE}[\![z_3]\!]=\big(\{\circ z_3,z_3\circ\},\{{\shortmid}z_3{\shortmid}\},\{(\circ z_3,{\shortmid}z_3{\shortmid}),({\shortmid}z_3{\shortmid},z_3\circ)\}\big)$$

Die Semantik der eingebetteten Parallelkartusche mit $\rho_{\mathcal{PK}}(z_2)=P_1$ *ergibt sich zu:*

$$\begin{aligned}&\mathcal{PK}[\![P_1=(\mathcal{E},\mathcal{K},e_{ein},e_{aus})]\!](M,z_2)=\\ &\quad\big(\{\circ z_2,z_2\circ\},\{{\shortmid}e_{ein},e_{aus}{\shortmid}\},\{(\circ z_2,{\shortmid}e_{ein}),(e_{aus}{\shortmid},z_2\circ)\}\big)\\ &\cup\mathcal{SK}[\![\rho_{\mathcal{SK}}(e_1)]\!](M,e_1)\ \cup\ \mathcal{SK}[\![\rho_{\mathcal{SK}}(e_2)]\!](M,e_2)\\ &\cup\mathcal{K}[\![(e_{ein},e_1)]\!]\cup\mathcal{K}[\![(e_{ein},e_2)]\!]\cup\mathcal{K}[\![(e_1,e_{aus})]\!]\cup\mathcal{K}[\![(e_2,e_{aus})]\!]\end{aligned}$$

In der Parallelkartusche P_1 ergeben sich die Kanten zu:

$$\mathcal{K}[\![k_1 = (e_{ein}, e_1)]\!] = \big(\{\circ k_1 \circ\}, \{\mid e_{ein}, e_1 \mid\}, \{(\mid e_{ein}, \circ k_1 \circ), (\circ k_1 \circ, e_1 \mid)\}\big)$$
$$\mathcal{K}[\![k_2 = (e_{ein}, e_2)]\!] = \big(\{\circ k_2 \circ\}, \{\mid e_{ein}, e_2 \mid\}, \{(\mid e_{ein}, \circ k_2 \circ), (\circ k_2 \circ, e_2 \mid)\}\big)$$
$$\mathcal{K}[\![k_3 = (e_1, e_{aus})]\!] = \big(\{\circ k_3 \circ\}, \{\mid e_1, e_{aus} \mid\}, \{(\mid e_1, \circ k_3 \circ), (\circ k_3 \circ, e_{aus} \mid)\}\big)$$
$$\mathcal{K}[\![k_4 = (e_2, e_{aus})]\!] = \big(\{\circ k_4 \circ\}, \{\mid e_2, e_{aus} \mid\}, \{(\mid e_2, \circ k_4 \circ), (\circ k_4 \circ, e_{aus} \mid)\}\big)$$

Schließlich ist die Semantik der beiden eingebetteten Sequenzkartuschen S_1 und S_2 zu bestimmen. Die Menge der Knoten der eingebetteten Parallelkartuschen $\mathcal{Z}_{PK}$ ist bei beiden Kartuschen leer. Die Semantik der Kartusche S_1 ist damit:

$$\begin{aligned}\mathcal{SK}[\![S_1 = \big(\mathcal{Z}', \Sigma', \delta', z'_{start}, z'_{end}\big)]\!](M, e_1) = \\ &\big(\{\circ z'_{start}, z'_{end} \circ\}, \{\mid e_1 \ , e_1 \mid\}, \{(\mid e_1 \ , \circ z'_{start}), (z'_{end} \circ \ , e_1 \mid)\}\big) \\ &\cup \mathcal{LE}[\![z'_1]\!] \\ &\cup \emptyset \\ &\cup \mathcal{T}[\![\delta'_1 = (z'_{start}, \epsilon, z'_1)]\!] \cup \mathcal{T}[\![\delta'_2 = (z'_1, \epsilon, z'_{end})]\!]\end{aligned}$$

Die Übergangsfunktionen für S_1 ergeben sich zu:

$$\mathcal{T}[\![\delta'_1(z'_{start}, \epsilon, z'_1)]\!] = \big(\{z'_{start} \circ, \circ z'_1\}, \{\mid \delta'_1 \mid\}, \{(z'_{start} \circ, \mid \delta'_1 \mid), (\mid \delta'_1 \mid, \circ z'_1)\}\big)$$
$$\mathcal{T}[\![\delta'_2 = (z'_1, \epsilon, z'_{end})]\!] = \big(\{z'_1 \circ, \circ z'_{end}\}, \{\mid \delta'_2 \mid\}, \{(z'_1 \circ, \mid \delta'_2 \mid), (\mid \delta'_2 \mid, \circ z'_{end})\}\big)$$

und die eingebettete Last zu:

$$\mathcal{LE}[\![z'_1]\!] = \big(\{\circ z'_1, z'_1 \circ\}, \{\mid z'_1 \mid\}, \{(\circ z'_1, \mid z'_1 \mid), (\mid z'_1 \mid, z'_1 \circ)\}\ \big)$$

Die Semantik der zweiten eingebetteten Sequenzkartusche S_2 ergibt sich zu:

$$\begin{aligned}\mathcal{SK}[\![S_2 = \big(\mathcal{Z}'', \Sigma'', \delta'', z''_{start}, z''_{end}\big)]\!](M, e_2) = \\ &\big(\{z''_{start} \circ, \circ z''_{end}\}, \{\mid e_2 \ , e_2 \mid\}, \{(\mid e_2 \ , z''_{start} \circ), (\circ z''_{end} \ , e_2 \mid)\}\big) \\ &\cup \mathcal{LE}[\![z''_1]\!] \\ &\cup \emptyset \\ &\cup \mathcal{T}[\![\delta''_1 = (z''_{start}, \epsilon, z''_1)]\!] \cup \mathcal{T}[\![\delta''_2 = (z''_1, \epsilon, z''_{end})]\!]\end{aligned}$$

Die Übergangsfunktionen für S_2 ergeben sich zu:

$$\mathcal{T}[\![\delta''_1 = (z''_{start}, \epsilon, z''_1)]\!] = \big(\{z''_{start} \circ, \circ z''_1\}, \{\mid \delta''_1 \mid\}, \{(z''_{start} \circ, \mid \delta''_1 \mid), (\mid \delta''_1 \mid, \circ z''_1)\}\big)$$
$$\mathcal{T}[\![\delta''_2 = (z''_1, \epsilon, z''_{end})]\!] = \big(\{z''_1 \circ, \circ z''_{end}\}, \{\mid \delta''_2 \mid\}, \{(z''_1 \circ, \mid \delta''_2 \mid), (\mid \delta''_2 \mid, \circ z''_{end})\}\big)$$

und die eingebettete Last zu:

$$\mathcal{LE}[\![z_1'']\!] = \big(\{\circ z_1'', z_1''\circ\}, \{\shortmid z_1''\shortmid\}, \{(\circ z_1'', \shortmid z_1''\shortmid), (\shortmid z_1''\shortmid, z_1''\circ)\}\ \big)$$

Damit ist die Übersetzung abgeschlossen, und wir können die Teilnetze vereinigen. In diesem Beispiel wurde auf eine eindeutige Benennung der Übergangsfunktionen der drei jeweiligen Sequenzkartuschen verzichtet. Es existieren lediglich ϵ-Übergänge.

Werden die einzelnen Funktionen ineinander eingesetzt, ergibt sich das resultierende Netz zu:

$$\mathcal{M}[\![M]\!] = \big(S, T, F\big)$$

mit den jeweiligen Teilmengen, die im Einzelnen wie folgt zusammengesetzt sind:

$$\begin{aligned}
S = \{&Start, Stop, z_{start}\circ, \circ z_{end}, \circ z_1, z_1\circ, \circ z_2, z_2\circ, \circ z_3, z_3\circ, \circ k_1\circ, \circ k_2\circ,\\
&\circ k_3\circ, \circ k_4\circ, z'_{start}\circ, \circ z'_{end}, \circ z'_1, z'_1\circ, z''_{start}\circ, \circ z''_{end}, \circ z''_1, z''_1\circ\}
\end{aligned}$$

$$\begin{aligned}
T = \{&\shortmid e_M, e_M\shortmid, \shortmid\delta_1\shortmid, \shortmid\delta_2\shortmid, \shortmid\delta_3\shortmid, \shortmid\delta_4\shortmid, \shortmid z_1\shortmid, \shortmid z_3\shortmid, \shortmid e_{ein}, e_{aus}\shortmid, e_1\shortmid, e_2\shortmid, \shortmid e_1, \shortmid e_2,\\
&\shortmid\delta'_1\shortmid, \shortmid\delta'_2\shortmid, \shortmid z'_1\shortmid, \shortmid\delta''_1\shortmid, \shortmid\delta''_2\shortmid, \shortmid z''_1\shortmid\}
\end{aligned}$$

$$\begin{aligned}
F = \{&(Start, \shortmid e_M), (e_M\shortmid, Stop), (\shortmid e_M, z_{start}\circ), (\circ z_{end}, e_M\shortmid), (z_{start}\circ, \shortmid\delta_1\shortmid),\\
&(\shortmid\delta_1\shortmid, \circ z_1), (z_1\circ, \shortmid\delta_2\shortmid), (\shortmid\delta_2\shortmid, \circ z_2), (z_2\circ, \shortmid\delta_3\shortmid), (\shortmid\delta_3\shortmid, \circ z_3), (z_3\circ, \shortmid\delta_4\shortmid),\\
&(\shortmid\delta_4\shortmid, \circ z_{end}), (\circ z_1, \shortmid z_1\shortmid), (\shortmid z_1\shortmid, z_1\circ), (\circ z_3, \shortmid z_3\shortmid), (\shortmid z_3\shortmid, z_3\circ), (\circ z_2, \shortmid e_{ein}),\\
&(e_{aus}\shortmid, z_2\circ), (\shortmid e_{ein}, \circ k_1\circ), (\circ k_1\circ, e_1\shortmid), (\shortmid e_{ein}, \circ k_2\circ), (\circ k_2\circ, e_2\shortmid),\\
&(\shortmid e_1, \circ k_3\circ), (\circ k_3\circ, e_{aus}\shortmid), (\shortmid e_2, \circ k_4\circ), (\circ k_4\circ, e_{aus}\shortmid), (\shortmid e_1, z'_{start}\circ),\\
&(\circ z'_{end}, e_1\shortmid), (z'_{start}\circ, \shortmid\delta'_1\shortmid), (\shortmid\delta'_1\shortmid, \circ z'_1), (z'_1\circ, \shortmid\delta'_2\shortmid), (\shortmid\delta'_2\shortmid, \circ z'_{end}),\\
&(\circ z'_1, \shortmid z'_1\shortmid), (\shortmid z'_1\shortmid, z'_1\circ), (\shortmid e_2, z''_{start}\circ), (\circ z''_{end}, e_2\shortmid), (z''_{start}\circ, \shortmid\delta''_1\shortmid),\\
&(\shortmid\delta''_1\shortmid, \circ z''_1), (z''_1\circ, \shortmid\delta''_2\shortmid), (\shortmid\delta''_2\shortmid, \circ z''_{end}), (\circ z''_1, \shortmid z''_1\shortmid), (\shortmid z''_1\shortmid, z''_1\circ)\}
\end{aligned}$$

Besondere Aufmerksamkeit verdienen hier die Transitionen $\shortmid z_1\shortmid$ und $\shortmid z_3\shortmid$, die schalten sollen, wenn die Aktivitäten A und C terminieren und die Transitionen $\shortmid z'_1\shortmid$ und $\shortmid z''_1\shortmid$ für die Aktivitäten M und N. Die Mengendarstellung entspricht dem Netz in Abbildung 2.3.11. Die Zuordnung aller syntaktischen Elemente zu der Graphik an dieser Stelle verbleibt dem interessierten Leser.

Beispiel 7 (Unverzweigter Kontrollfluß). *Folgend wird die Semantik des Beispiels eines unverzweigten Kontrollflusses, wie es in Abschnitt 2.2.6.2 eingeführt wurde, abgeleitet. In diesem Beispiel, dessen graphische Darstellung in Abbildung 2.2.17 zu sehen ist, existiert nur die Wurzelkartusche S_0. Die Modellsemantik ergibt sich wie im vorigen Beispiel zu:*

$$\mathcal{M}[\![(\Upsilon, \Theta, \mathcal{R}, \rho, e_M)]\!] = \\ \big(\{Start, Stop\}, \{\imath e_M, e_{M}\imath\}, \{(Start, \imath e_M), (e_{M}\imath, Stop)\}\big) \\ \cup\ \mathcal{SK}[\![\rho_{\mathcal{SK}}(e_M)]\!](M, e_M)$$

mit $\rho_{\mathcal{SK}}(e_M) = S_0$ ergibt sich die Semantik der einzigen Kartusche des Modells zu:

$$\begin{aligned}\mathcal{SK}[\![S_0 = \big(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\big)]\!](M, e_M) = \\ &\big(\{z_{start}\circ, \circ z_{end}\}, \{\imath e_M\ , e_{M}\imath\}, \{(\imath e_M\ , z_{start}\circ), (\circ z_{end}\ , e_{M}\imath)\}\big) \\ &\cup\ \mathcal{LE}[\![z_1]\!] \cup \mathcal{LE}[\![z_2]\!] \cup \mathcal{LE}[\![z_3]\!] \\ &\cup\ \emptyset \\ &\cup\ \mathcal{T}[\![\delta_1 = (z_{start}, \epsilon, z_1)]\!] \cup \mathcal{T}[\![\delta_2 = (z_1, \mathtt{b}, z_2)]\!] \\ &\quad\cup\ \mathcal{T}[\![\delta_3 = (z_2, \mathtt{c}, z_3)]\!] \cup \mathcal{T}[\![\delta_4 = (z_3, \epsilon, z_{end})]\!] \\ &\quad\cup\ \mathcal{T}[\![\delta_5 = (z_1, \mathtt{a}, z_1)]\!] \cup \mathcal{T}[\![\delta_6 = (z_2, \mathtt{x}, z_{end})]\!]\end{aligned}$$

Die Semantik der Kanten läßt sich unmittelbar berechnen zu:

$$\begin{aligned}\mathcal{T}[\![\delta_1 = (z_{start}, \epsilon, z_1)]\!] &= (\{z_{start}\circ, \circ z_1\}, \{\imath\delta_1\imath\}, \{(z_{start}\circ, \imath\delta_1\imath), (\imath\delta_1\imath, \circ z_1)\}) \\ \mathcal{T}[\![\delta_2 = (z_1, \mathtt{b}, z_2)]\!] &= (\{z_1\circ, \circ z_2\}, \{\imath\delta_2\imath\}, \{(z_1\circ, \imath\delta_2\imath), (\imath\delta_2\imath, \circ z_2)\}) \\ \mathcal{T}[\![\delta_3 = (z_2, \mathtt{c}, z_3)]\!] &= (\{z_2\circ, \circ z_3\}, \{\imath\delta_3\imath\}, \{(z_2\circ, \imath\delta_3\imath), (\imath\delta_3\imath, \circ z_3)\}) \\ \mathcal{T}[\![\delta_4 = (z_3, \epsilon, z_{end})]\!] &= (\{z_3\circ, \circ z_{end}\}, \{\imath\delta_4\imath\}, \{(z_3\circ, \imath\delta_4\imath), (\imath\delta_4\imath, \circ z_{end})\}) \\ \mathcal{T}[\![\delta_5 = (z_1, \mathtt{a}, z_1)]\!] &= (\{z_1\circ, \circ z_1\}, \{\imath\delta_5\imath\}, \{(z_1\circ, \imath\delta_5\imath), (\imath\delta_5\imath, \circ z_1)\}) \\ \mathcal{T}[\![\delta_6 = (z_2, \mathtt{x}, z_{end})]\!] &= (\{z_2\circ, \circ z_{end}\}, \{\imath\delta_6\imath\}, \{(z_2\circ, \imath\delta_6\imath), (\imath\delta_6\imath, \circ z_{end})\})\end{aligned}$$

Die Semantik der drei Lastelemente ergibt:

$$\begin{aligned}\mathcal{LE}[\![z_1]\!] &= \big(\{\circ z_1, z_1\circ\}, \{\imath z_1\imath\}, \{(\circ z_1, \imath z_1\imath), (\imath z_1\imath, z_1\circ)\}\big) \\ \mathcal{LE}[\![z_2]\!] &= \big(\{\circ z_2, z_2\circ\}, \{\imath z_2\imath\}, \{(\circ z_2, \imath z_2\imath), (\imath z_2\imath, z_2\circ)\}\big) \\ \mathcal{LE}[\![z_3]\!] &= \big(\{\circ z_3, z_3\circ\}, \{\imath z_3\imath\}, \{(\circ z_3, \imath z_3\imath), (\imath z_3\imath, z_3\circ)\}\big)\end{aligned}$$

Die Teilmengen des resultierenden Netzes $\mathcal{M}[\![M]\!] = \big(S, T, F\big)$ *sind wie folgt zusammengesetzt:*

$$
\begin{aligned}
S &= \{Start, Stop, z_{start}\circ, \circ z_{end}, \circ z_1, z_1\circ, \circ z_2, z_2\circ, \circ z_3, z_3\circ\} \\
T &= \{\shortmid e_M, e_M\shortmid, \shortmid\delta_1\shortmid, \shortmid\delta_2\shortmid, \shortmid\delta_3\shortmid, \shortmid\delta_4\shortmid, \shortmid\delta_5\shortmid, \shortmid\delta_6\shortmid, \shortmid z_1\shortmid, \shortmid z_2\shortmid, \shortmid z_3\shortmid\} \\
F &= \{(Start, \shortmid e_M), (e_M\shortmid, Stop), (\shortmid e_M, z_{start}\circ), (\circ z_{end}, e_M\shortmid), (z_{start}\circ, \shortmid\delta_1\shortmid), \\
&\quad (\shortmid\delta_1\shortmid, \circ z_1), (z_1\circ, \shortmid\delta_2\shortmid), (\shortmid\delta_2\shortmid, \circ z_2), (z_2\circ, \shortmid\delta_3\shortmid), (\shortmid\delta_3\shortmid, \circ z_3), (z_3\circ, \shortmid\delta_4\shortmid), \\
&\quad (\shortmid\delta_4\shortmid, \circ z_{end}), (z_1\circ, \shortmid\delta_5\shortmid), (\shortmid\delta_5\shortmid, \circ z_1), (z_2\circ, \shortmid\delta_6\shortmid), (\shortmid\delta_6\shortmid, \circ z_{end}), (\circ z_1, \shortmid z_1\shortmid), \\
&\quad (\shortmid z_1\shortmid, z_1\circ), (\circ z_2, \shortmid z_2\shortmid), (\shortmid z_2\shortmid, z_2\circ), (\circ z_3, \shortmid z_3\shortmid), (\shortmid z_3\shortmid, z_3\circ)\}
\end{aligned}
$$

Die Abbildung 2.3.2 aus der informellen Einführung zeigte nur den ersten Übersetzungsschritt, ohne Übersetzung der Lastelemente. In Abbildung 2.3.14 ist nur das ganze Netz gemäß der vorherigen Übersetzung dargestellt.

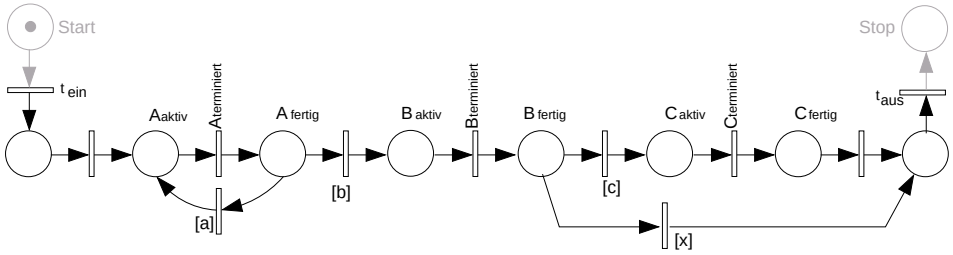


Abbildung 2.3.14. Vollständige Übersetzung der Sequenzkartusche

In dieser Darstellung sind die Bezeichnungen der Kartuschendarstellung entnommen. Die Stellen im Einzelnen sind: Start und Stop wie bezeichnet. Die unbenannten Stellen an der zweiten und vorletzten Postion entsprechen den Elementen $z_{start}\circ$ *und* $\circ z_{end}$*.* A_{bereit} *entspricht* $\circ z_1$ *und* A_{fertig} *dem Element* $z_1\circ$*. Die Stellen, die B repräsentieren, sind* $\circ z_2$ *und* $z_2\circ$ *und C wird verkörpert durch* $\circ z_3$ *und* $z_3\circ$*.*

Die randständigen Grundmodell-Transitionen t_{ein} *und* t_{aus} *entsprechen* $\shortmid e_M$ *und* $e_M\shortmid$*, die aus der Übersetzung des* Modellknotens *resultieren. Dann sind weiterhin zwei unbeschriftete Transitionen zu sehen (die zweite und die vorletzte), welche aus den beiden unbeschrifteten Übergängen der Sequenzkartusche vom Startzustand zur Last A und von Last C zum Endzustand stammen. Diese entsprechen den Transitionen* $\shortmid\delta_1\shortmid$ *und* $\shortmid\delta_6\shortmid$*. Dann sind drei Transitionen zu erkennen, die das Terminieren der Lastbearbeitung der Lasten A, B und C modellieren. Diese werden durch* $\shortmid z_1\shortmid$*,* $\shortmid z_2\shortmid$ *und* $\shortmid z_3\shortmid$ *modelliert. Die Elemente* $\shortmid\delta_2\shortmid$*,* $\shortmid\delta_3\shortmid$*,* $\shortmid\delta_4\shortmid$ *und* $\shortmid\delta_5\shortmid$

werden in die Transitionen übersetzt, welche die Wächterannotationen a, b, c und x tragen und zur Interpretation der Szenarien dienen, die im nächsten Kapitel beschrieben werden.

2.4. Erreichbarkeitsgraph

Das Petrinetz modelliert durch die Stellen unter anderem die Lasten, deren Zeitverbrauch wir abschätzen wollen. Jede Stelle repräsentiert die Instanz einer Last. Diese Instanz kann im Verlauf eines Ablaufszenarios mehrfach nacheinander, aber nie mehrfach gleichzeitig abgearbeitet werden. Nebenläufig bearbeitete Lasten müssen explizit modelliert werden und mehrfach vorkommen. Transitionen verwenden wir, um Ablaufszenarios zu realisieren. An den Stellen, an denen durch eine vorhandene Marke ein Konflikt entsteht, war es nach unserer Forderung möglich, durch unterscheidbare Anschriften genau eine aus mehreren gleichzeitig konzessionierten Transitionen auszuwählen. Eine Marke an einer Stelle zeigt an, daß die dadurch repräsentierte Last sich in Bearbeitung befindet. Durch diese Interpretation forderten wir ein 1-beschränktes Netz. Später werden wir Informationen über die Verteilung von Lasten auf Ressourcen hinzufügen. Aus Teilmengen der Markierung läßt sich dann auf die jeweiligen zeitbestimmenden Faktoren schließen. Verfolgen wir also die Markierung und ihre Änderungen, ist der Erreichbarkeitsgraph des Modellnetzes besonders interessant. Dieser drückt genau die Übergänge zwischen möglichen Markierungen aus.

SATZ 3 (Erreichbarkeitsgraph). *Ist ein Petrinetz N k-beschränkt, ist sein Erreichbarkeitsgraph $\mathcal{EG}(N)$ erzeugbar und endlich.*

Beweis: siehe [**57**].

2.4.1. Performanzdeterminierte Verzweigungen. Betrachtet man nochmals das vereinfachte semantische Modell aus Abbildung 2.3.12, das wir hier in Abbildung 2.4.1 wiederholen, ergibt sich ein sehr einfacher Erreichbarkeitsgraph.

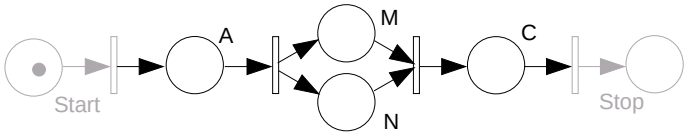


ABBILDUNG 2.4.1. Reduktion unbeschrifteter Stellen und spontaner Transitionen (Zum leichteren Verständnis)

In Abbildung 2.4.2 ist der Erreichbarkeitsgraph dieser Semantik graphisch dargestellt. Der dargestellte Vektor der Markierung ordnet den einzelnen Stellen nach dem Schema $\langle m(\texttt{Start}), m(A), m(M), m(N), m(C), m(\texttt{Stop})\rangle$ der Reihe nach einen Wert zu.

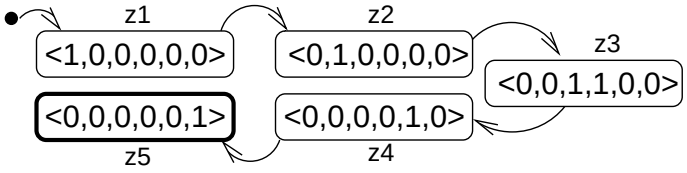


ABBILDUNG 2.4.2. Erreichbarkeitsgraph des Beispielmodells

Wir sehen an `z3` unmittelbar die nebenläufige Bearbeitung der beiden Lasten `M` und `N`. Da sie nur in einem Zustand markiert sind, wird ihre Nebenläufigkeit augenfällig. Beide Lasten benötigen aber nicht zwingend die gleiche Bearbeitungszeit. Da der Folgezustand `z4` beide Lasten als terminiert modelliert und im Erreichbarkeitsgraphen unmittelbar folgt, kann das zeitlich frühere Terminieren der einen oder anderen Last nicht ausgedrückt werden. Die Zustände *laufend* und *terminiert* müssen daher unterschieden werden, wie es durch die Lastsemantik eingeführt wird.

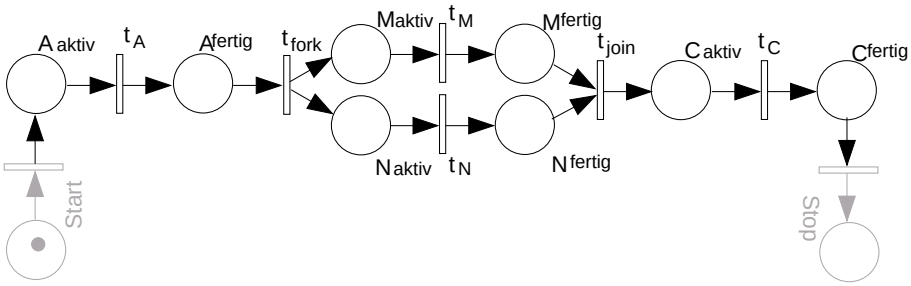


ABBILDUNG 2.4.3. Unterscheidung laufender und bearbeiteter Lasten

Abbildung 2.4.3 wiederholt das vollständige Modell mit jeweils einer zusätzlichen Stelle, welche die abgeschlossene Bearbeitung einer Last modelliert. An diesem Netz kann nun anhand der Markierung unterschieden werden, ob die Last M oder N terminiert ist oder beide terminiert sind. In Abbildung 2.4.4 wird der Erreichbarkeitsgraph dieser Semantik dargestellt.

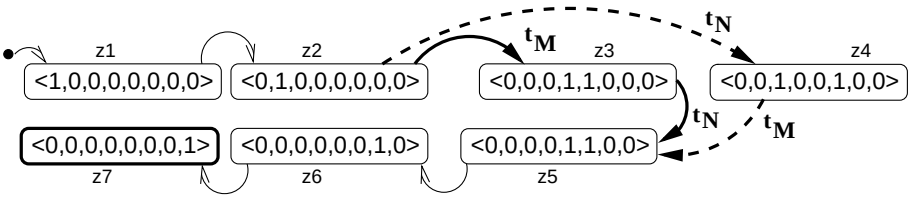


ABBILDUNG 2.4.4. Erreichbarkeitsgraph des Beispielmodells

Der Erreichbarkeitsgraphe ist in zwei Pfade aufgespalten. Der Eine modelliert das Terminieren der Last M zeitlich nach der Last N (gestrichelt dargestellt), der Andere die umgekehrte Reihenfolge (durchgezogen dargestellt). Betrachtet wird der Vektor der Markierung nach:

$$\langle m(\texttt{Start}), m(\texttt{A}), m(\texttt{M}_{aktiv}), m(\texttt{N}_{aktiv}), \\ m(\texttt{M}_{fertig}), m(\texttt{N}_{fertig}), m(\texttt{B}), m(\texttt{Stop})\rangle$$

Dargestellt ist, der Übersichtlichkeit halber, nur der Teil der Markierung, der für das Vorhandensein mehrerer Pfade im Erreichbarkeitsgraphen verantwortlich ist.

Der Pfad über $[\ldots t_M, t_N \ldots]$ stellt die Ablaufvariante vor, in der die Bearbeitung von `M` als erstes terminiert, und der Pfad über $[\ldots t_N, t_M \ldots]$, in der die Bearbeitung von `N` als erstes terminiert.

In einem späteren Kapitel wird eine zeitliche Semantik als Bewertungsfunktion für einen Pfad des Erreichbarkeitsgraphen eingeführt und als zentrales Mittel zur Bewertung des zeitlichen Verhaltens diskutiert. Im hier gezeigten Beispiel haben wir keine Interaktion mit der Außenwelt. Es ist kein anderer Ablauf durch bedingte Ausführung von Aktivitäten möglich. Die Wahl eines bestimmten Pfades im Erreichbarkeitsgraphen hängt nur von der Reihenfolge des Terminierens von Lasten ab. Damit ist der Pfad nur durch die Leistungsfunktionen der Ressourcen determiniert. Sind in den Sequenzkartuschen bedingte Ausführungen spezifiziert, entstehen im Erreichbarkeitsgraphen neue Gruppen von Pfaden. Ein Testfall (dh. ein Szenario) stellt genau eine vollständige Auflösung dieser Konflikte dar. Die formale Struktur dieser Szenarien wird im nächsten Kapitel eingeführt.

2.4.2. Kommunikationsdeterminierte Verzweigungen. Das zweite Beispiel des unverzweigten Kontrollflußes zeigt keine performanzdeterminierten Verzweigungen. Die Verzweigungen am Erreichbarkeitsgraphen, der sich aus diesem Modell ermitteln läßt, sind determiniert durch die entsprechenden Wächter, die von den Anschriften an den konfliktbehafteten Transitionen des Petrinetzes stammen.

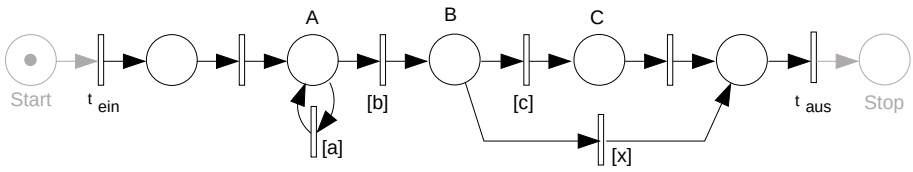


ABBILDUNG 2.4.5. Beispielübersetzung der Sequenzkartusche (1.Schritt)

Betrachtet man an dem bereits eingeführten Beispiel, das hier in Abbildung 2.4.5 wiederholt wird, die Markierung der relevanten Stellen für die Lasten und den Start- und Endzustand nach: $\langle m(Start), m(A), m(B), m(C), m(Stop)\rangle$, ergibt sich der Erreichbarkeitsgraph nach Abbildung 2.4.6.

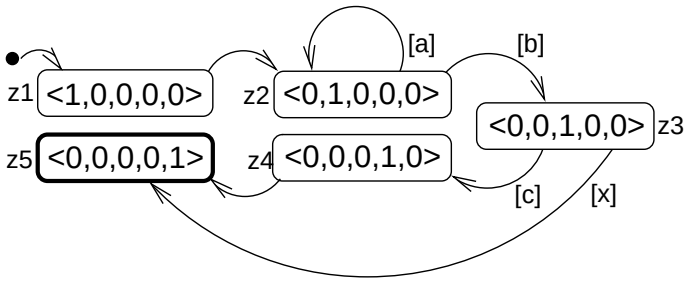


ABBILDUNG 2.4.6. Erreichbarkeitsgraph des Beispielmodells

Die Bestimmung eines Pfades in diesem Erreichbarkeitsgraphen ist das Ergebnis einer Kommunikation mit der Außenwelt, der in Form der Szenariospezifikation in die Interpretation des Modells einfließt. Die beiden in diesem Kapitel bemühten Beispiele weisen jeweils nur einen Typ an den Verzweigungen auf. Modelle in der Praxis stellen Mischformen dar. Im nächsten Kapitel werden die verschiedenen Aspekte der Kommunikation eingeführt.

2.5. Zusammenfassung

In diesem Kapitel wurde ein graphisches Modell zur Beschreibung des Kontrollflusses eingeführt, das mit einer formalen Spezifikation seiner strukturellen und dynamischen Eigenschaften ausgestattet ist. Es soll ausdrucksstark genug sein, um gängige Modelle unmittelbar beschreiben zu können. Es ist darauf ausgelegt, in seiner graphischen Notation intuitiv leicht verständlich zu sein. Dabei wurde darauf Wert gelegt, Doppeldeutigkeiten bei der Interpretation weitgehend zu vermeiden und Intuition und Eigenschaftsdefinition kohärent zu halten.

Vor allem aber formuliert das hier eingeführte Modell in einem klaren Wohlgeformtheitsbegriff die Trennung von nebenläufigen und sequentiellen Regionen und garantiert technische Eigenschaften, die bei der späteren Interpretation als Performanzmodell eine eindeutige und reproduzierbare Bewertung durch eine Simulationsumgebung ermöglichen und eindeutige Interpretierbarkeit und das Terminieren einer Simulation garantieren. Die formale Petrinetzsemantik gibt nicht nur eine exakte Definition der Bedeutung jedes Modells, sondern bietet durch den gewählten Formalismus eine Grundlage für die unmittelbare Verwendung der Semantik in der Implementierung eines Testwerkzeugs.

KAPITEL 3

Szenarien

In Kapitel 2 wurde ausführlich die Struktur und die Semantik von dynamischen Modellen (Kontrollflußmodellen) diskutiert. Um Performanzeigenschaften, also das zeitliche Verhalten eines im Lauf befindlichen Modells zu betrachten, müßen einzelne zu testende Abläufe, sogenannte Szenarien spezifiziert werden. Dies ist eine der Aufgaben bei der Modellbewertung. Der Entwickler möchte Szenarien auf eine bequeme und intuitive Art erstellen können und dabei von einem leistungfähigen Werkzeug unterstützt werden.

In diesem Kapitel werden Szenarien in theoretische Kategorien eingeordnet. Aus dieser Einordnung ergeben sich wichtige Erkenntnisse für die spätere Implementierung eines Szenario-Editor in einem Werkzeug und ein größeres Verständnis deren grundlegender Struktur.

3.1. Grundlagen

Die Ausführungszeit für einen vollständigen Lauf steht im Mittelpunkt der Betrachtung. Als vollständiger Lauf wurde dabei eine Folge aktivierter Lastzustände bezeichnet, die vom Startzustand der K^0_{SK}-Kartusche zu deren Stoppzustand führt. Jedes dynamische Modell verfügt über mindestens einen vollständigen Lauf. Dazu wurden die im letzten Kapitel definierten Einschränkungen für Parallelkartuschen auf syntaktischer Ebene gemacht, die Erreichbarkeit des Endzustandes in Lemma 1 bewiesen und die Existenz eines Laufes für Sequenzkartuschen gefordert. Diese Eigenschaft muß später durch die syntaktische Analyse eines Editors eines Werkzeugs sichergestellt werden.

Die meisten Modelle in der Praxis werden aber eine Vielzahl möglicher Läufen spezifizieren. Existiert in einer Sequenzkartusche eine Schleife, so ist diese Menge bereits abzählbar unendlich und enthält prinzipiell auch Läufe unbeschränkter Länge. In der Praxis beschäftigen wir uns aber nur mit endlichen Verläufen. Die Auswahl geeigneter Läufe ist ein Willensakt des Entwicklers, der durch sein Wissen über diese Domäne begründet ist.

In der Spurtheorie [**43, 1, 66, 44, 45, 2, 26, 27, 53**] und bei asynchronen Automaten [**65, 66, 53**] findet man passende sprachtheoretischen Modelle zur Beschreibungen einzelner Abläufe von Petrinetzen.

3.1.1. Läufe als Schaltsequenz. Das Beispiel einer Sequenzkartuschen-Übersetzung aus Abbildung 2.3.14, hier in Abbildung 3.1.1 wiederholt, zeigt drei Lasten A, B und C. Eine Marke an der jeweiligen Stelle bezeichnet die Aktivierung oder Terminierung der jeweiligen Last.

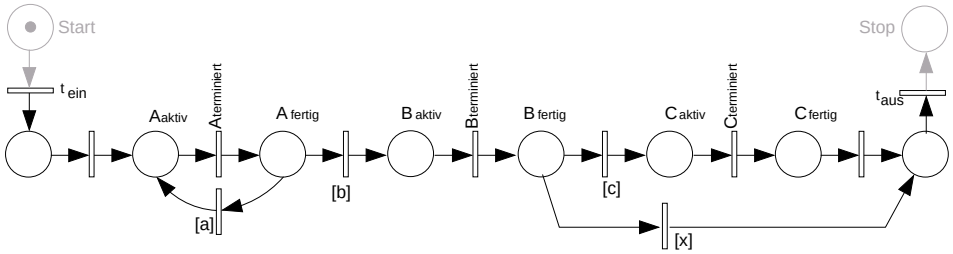


ABBILDUNG 3.1.1. Beispiel 1 (Sequenzkartusche)

Die Folge der aktivierten Lasten kann somit der Folge der Markierungen entnommen werden. Die Last A ist aktiv, falls $m(\mathsf{A}_{\mathsf{aktiv}}) = 1$.

DEFINITION 36 (Vorbereich, Schalten, Schaltsequenz). *Sei* $\mathcal{M}[\![M]\!] = (S, T, F)$ *das sichere Netz eines dynamischen Performanzmodells,* $\bullet t$ *der Vorbereich der Transition* t *mit* $\bullet t = \{s.(s,t) \in F\}$, $t\bullet$ *der Nachbereich der Transition* t *mit* $t\bullet = \{s.(t,s) \in F\}$, *so bezeichnet man die Überführung der Markierung* m *in die Folgemarkierung* m' *durch eine Transition* t *als* Schalten *einer Transition* $m[t\rangle m'$, *wobei gilt:*

$$m'(s) = \begin{cases} m(s), & \textit{falls } s \notin \bullet t \cup t\bullet \\ 1, & \textit{falls } s \in t\bullet \\ 0, & \textit{falls } s \in \bullet t \end{cases}$$

Eine Folge von Transitionen heißt Schaltsequenz *von* m_0 *nach* m_n *falls:*

$$t_{seq} = (t_1, \ldots, t_n) \Leftrightarrow \exists m_0, \ldots, m_n . m_0[t_1\rangle m_1[t_2\rangle \ldots [t_n\rangle m_n$$

Kurz: $m_0[t_{seq}\rangle m_n$ *für* $n \geq 0$.

Da es für jedes Modell mindestens einen vollständigen Lauf gibt, existiert mindestens eine Schaltsequenz t_{seq} mit $m[t_{seq}\rangle m'$, sodaß $m(\mathsf{Start}) = 1$ und $m'(\mathsf{Stop}) = 1$.

In Abbildung 3.1.2 ist der Erreichbarkeitsgraph des Beispiels erneut dargestellt. Bei der Markierung wurde der Vektor auf die interessanten Stellen nach $\langle m(\mathsf{Start}), m(\mathsf{A}), m(\mathsf{B}), m(\mathsf{C}), m(\mathsf{Stop})\rangle$ reduziert.

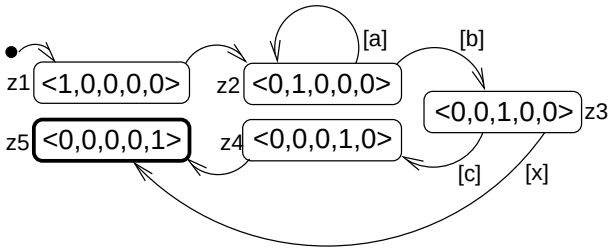


ABBILDUNG 3.1.2. Erreichbarkeitsgraph des Beispielmodells

Anschriften an Transitionen sind in diesem Beispiel, wenn vorhanden, in eckigen Klammern (als Wächter) dargestellt. Diese Art von Transitionen findet man an einer Stelle s genau dann, wenn sich nach dieser Stelle Transitionen im Petrinetz in Konflikt befinden, d.h. für diese Transitionen gilt $\mid \{t.(s,t) \in F\} \mid > 1$. Durch Interaktion mit der Außenwelt können diese Konflikte aufgelöst werden. In der Praxis wird ein konfliktfreier Lauf genau durch Vorgabe einer Schaltsequenz t_{seq} determiniert. Dazu genügt es hier die Transitionen darzustellen, die solche Annotationen tragen. Repräsentieren wir eine solche Transition durch das den Wächter repräsentierende Zeichen, sind Beispiele für Schaltsequenzen $t_{seq} = (\mathsf{a},\mathsf{a},\mathsf{b},\mathsf{x})$, $t_{seq} = (\mathsf{b},\mathsf{c})$ oder $t_{seq} = (\mathsf{a},\mathsf{a},\mathsf{a},\mathsf{b},\mathsf{c})$.

Am Beispiel mit einfacher Nebenläufigkeit in Abbildung 3.1.3 findet sich ein anderer beschrifteter Transitionstyp.

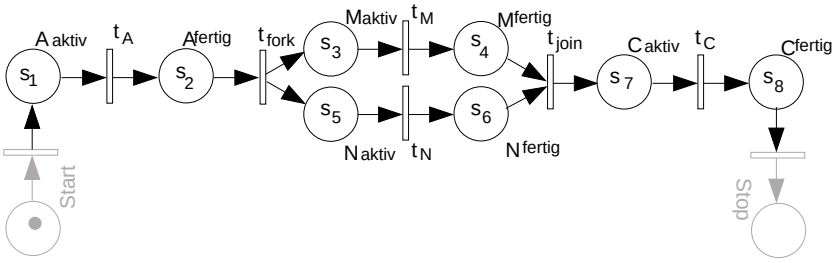


ABBILDUNG 3.1.3. Beispiel 2 (Nebenläufigkeit)

Diese Transitionen feuern erst, wenn die Last, die sie repräsentieren, vollständig bearbeitet wurde. Der Wächter beschreibt also das Beendigen einer Last und damit einen Vorgang, der im Simulationslauf ermittelt wird.

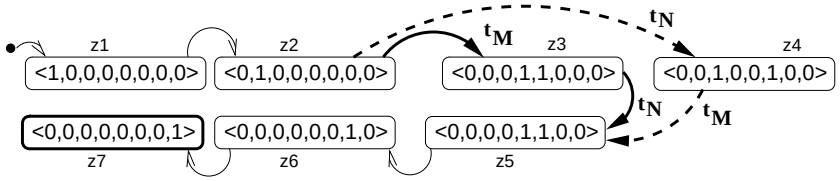


ABBILDUNG 3.1.4. Erreichbarkeitsgraph bei Nebenläufigkeit

Dieses Beispiel enthält keine Transitionen im Konflikt. Es existieren zwei Schaltsequenzen $t_{seq} = (\mathsf{t_M}, \mathsf{t_N})$ und $t_{seq} = (\mathsf{t_N}, \mathsf{t_M})$. Anders als beim vorherigen Beispiel ist das Vorkommen der Transitionen in der Reihenfolge unabhängig. Betrachtet man nur die Transitionen im Konflikt, kann nur die leere Sequenz $t_{seq} = \omega$ vorgegeben werden. Die beiden möglichen Sequenzen entsprechen den beiden möglichen Pfaden im Erreichbarkeitsgraphen, der in Abbildung 3.1.4 dargestellt ist.

3.1.2. Äquivalenzklassen von Läufen. Betrachtet man die Menge an Transitionen als Alphabet Σ, so spezifiziert das Petrinetz eine Menge an möglichen Schaltsequenzen, die man als Sprache L über dem Alphabet Σ auffassen kann. Eine Schaltsequenz kann durch eine Transitionssequenz, also als ein Wort $w \in L$ repräsentiert werden. Für Nebenläufigkeit reicht der Begriff der Wortsprache zur Beschreibung nicht aus, und es werden Spurtheorie und endliche asynchrone Automaten eingeführt.

3.1.2.1. *Spuren und Spursprachen.* Die Klasse an Wörtern, die sich innerhalb einer Sprache nur durch die Vertauschung der Zeichen aus einer Teilmenge jeweils unabhängiger Elemente unterscheidet, bezeichnet man als *Spur*. Die Bezeichnung geht auf Mazurkiewicz zurück. Eine umfassende Darstellung ist unter anderem bei [**44**, Mazurk.], [**45**, Mazurk.], [**2**, Rozenb.], [**26**, Diek.] und [**27**, Diek.] zu finden. Hier wird die Menge der unabhängigen Elemente explizit definiert.

DEFINITION 37 (Alphabet mit Unabhängigkeitsrelation). *Ein Alphabet mit Unabhängigkeitsrelation ist ein Paar* (Σ, I) *bestehend aus einem Alphabet* Σ *und einer irreflexiven, symmetrischen zweistelligen Relation, der sogenannten* Unabhängigkeitsrelation $I \subseteq \Sigma^2$.

Später wird die Quasitransitivität als Eigenschaft der Unabhängigkeitsrelation eine wichtige Grundlage wünschenswerter technischer Eigenschaften begründen.

DEFINITION 38 (Quasitransitivität). *Sei* (Σ, I) *ein Alphabet mit Unabhängigkeitsrelation, dann heißt* I quasitransitiv*, falls gilt:*

$$\forall a, b, c \in I, a \neq b, b \neq c \; . \; ((a, b) \in I \wedge (b, c) \in I) \Rightarrow (a, c) \in I$$

Zwei Ausdrücke, die sich nur um Elemente aus der Unabhängigkeitsrelation unterscheiden, sind bezüglich dieser Relation äquivalent.

DEFINITION 39 (Äquivalenz von Ausdrücken). *Sei (Σ, I) ein Alphabet mit Unabhängigkeitsrelation. Zwei Ausdrücke $v, w \in \Sigma^*$ heissen äquivalent ($v \cong_I w$) bezüglich der Unabhängigkeitsrelation I, wenn*

$$v \cong_I w \Leftrightarrow (\exists u, u' \in \Sigma^*, \exists a, b \in \Sigma)\ .v = uabu' \wedge w = ubau' \wedge (a, b) \in I$$

Die Teilmenge von äquivalenten Ausdrücken bildet eine Äquivalenzklasse, die man als Spur bezeichnet.

DEFINITION 40 (Spuren und Spursprache). *Es sei (Σ, I) ein Alphabet mit einer Unabhängigkeitsrelation. Eine Menge von Ausdrücken $t \subseteq \Sigma^*$ für die gilt: $v, w \in t \Leftrightarrow v \cong_I w$, heißt* Spur.

Sei $E(\Sigma, I)$ die Menge der Äquivalenzklassen (Spuren) über dem Alphabet Σ bezüglich der Unabhängigkeitsrelation I. Dann heißt jede Teilmenge $T \subseteq E(\Sigma, I)$ Spursprache.

Für jedes $w \in \Sigma^*$ bezeichnet $[w]_I$ die Äquivalenzklasse aller zu w bzgl. I äquivalenten Wörter und ist damit eine Spur $t \in E(\Sigma, I)$.

Um die Begriffe auf ein Petrinetz anzuwenden, muß nur die Transitionsmenge T_{Trans} als Alphabet interpretiert werden ($\Sigma = T_{Trans}$) und die Unabhängigkeitsrelation aus dem Netz abgeleitet werden.

DEFINITION 41 (Relation unabhängiger Transitionen). *Es sei T die Menge der Transitionen eines Petrinetzes N. Die Unabhängigkeitsrelation des Petrinetzes ist wie folgt erklärt*

$$t_1, t_2 \in \Sigma, (t_1, t_2) \in I_N \Leftrightarrow$$
$$(\bullet t_1 \cap \bullet t_2) = \emptyset \wedge (\bullet t_1 \cap t_2 \bullet) = \emptyset \wedge (t_1 \bullet \cap \bullet\, t_2) = \emptyset \wedge (t_1 \bullet \cap t_2 \bullet) = \emptyset$$

DEFINITION 42 (Spursprache über einem Petrinetz). *Sei $T_{Trans} = \Sigma$ und I_N die Unabhängigkeitsrelation über dem Petrinetz N, so nennt man $PT \subseteq E(T_{Trans}, I_N)$ eine* Spursprache *über dem Petrinetz N.*

3.1.2.2. *Endliche asynchrone Automaten.* In den Formalismen wurden bisher Strukturierungsmöglichkeiten für Σ^*-Mengen eingeführt. Zur Beschreibung des Szenarios wäre dies ausreichend. Wenn die Beschreibung von Performanzmodellen angestrebt wird, muß eine Teilmenge $L \subseteq \Sigma^*$ betrachtet werden. Mehr noch ist uns diese Teilmenge als Sprache über einem Petrinetz $L = \mathcal{L}(\mathcal{M}[\![M]\!])$ vorgegeben. Fragen, wie Worte dieser Sprache erzeugt und erkannt werden, spielen aber bei der praktischen Umsetzung eine große Rolle. Eine sehr anschauliche Theorie, welche die praktische Umsetzung unterstützt, findet sich bei [**65, 66**, Zielonka], in Form der endlichen, asynchronen Automaten.

DEFINITION 43 (Endlicher, asynchroner Automat). *Ein endlicher, asynchroner Automat EAA, bestehend aus n-Teilautomaten, mit $n \geq 1$, ist ein $(n+3)$-Tupel $\mathcal{A} = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{I}, \mathcal{D}, \mathcal{F})$. Dabei sind*

- *$\mathcal{P}_i = (\Sigma_i, S_i)$ ist dabei der i-te Teilautomat, mit Σ_i dem Alphabet und S_i der endlichen nichtleeren endlichen Zustandsmenge des Teilautomaten,*
- *$\mathcal{I} \subseteq S$ der endlichen Menge der Anfangszustände,*
- *$\mathcal{D}$ einer Familie von Übergangsrelationen und*
- *$\mathcal{F} \subseteq S$ der endlichen Menge der Endzustände.*

Dabei ist das Gesamtalphabet des Automaten Σ_A gleich der Vereinigung der Zeichenmengen der Teilautomaten $\Sigma_A = \cup_{i \in \{1..n\}} \Sigma_i$.
Unter einem globalen Zustand von $\mathcal{A}$ versteht man dann ein Tupel S aus $S_1 \times \cdots \times S_n$. Für jedes Zeichen $a \in \Sigma_A$ läßt sich die Heimat *über die Menge $Dom(a) = \{i \in \{1..n\}.a \in \Sigma_i\}$ ermitteln.*
$\mathcal{D} = \{\delta_a \ . \ a \in \Sigma_A\}$ ist eine Schar von Übergangsrelationen $\delta_a : \Pi_{i \in Dom(a)} S_i \to \mathfrak{P}(\Pi_{i \in Dom(a)} S_i)$. Ein a-Übergang $(s_1, \ldots, s_n) \Rightarrow_a (s'_1, \ldots, s'_n)$ von einem globalen Zustand $(s_1, \ldots, s_n)$ in einen Nachfolgezustand $(s'_1, \ldots, s'_n) \in S$ wird definiert als $(s_1, \ldots, s_n) \Rightarrow_a (s'_1, \ldots, s'_n)$

$$\Leftrightarrow \begin{cases} s'_i = s_i & , \ i \notin Dom(a) \\ (s'_{i_1}, \ldots, s'_{i_k}) \in \delta_a(s_{i_1}, \ldots, s_{i_k}) & , \ i \in Dom(a). \end{cases}$$

Ein a-Übergang bewirkt einen Zustandsübergang in den Heimatautomaten des Terminals und läßt die übrigen Teilautomaten unberührt.

Daraus ergibt sich die Wortsprache eines EAA aus der Menge an Terminalfolgen, die einen Anfangszustand in einen Endzustand überführen.

DEFINITION 44 (Wortsprache eines EAA). *Es seien $\mathcal{A}$ ein EAA, $w \in \Sigma_A^*$ ein Wort über dem Alphabet von A und $s,s' \in S$ globale Zustände von $\mathcal{A}$. Dann wird die Überführungsfunktion induktiv wie folgt definiert:*

$$\delta_w(s) = \begin{cases} \{s\} & , \ falls \ w = \epsilon \\ \bigcup\limits_{\delta_w \in \delta_a(s)} \delta_v(s') & , \ falls \ w = av, a \in \Sigma_{\mathcal{A}}, v \in \Sigma_{\mathcal{A}}^* \end{cases}$$

Somit ergibt sich die Wortsprache des EAA zu

$$L(\mathcal{A}) = \{w \in \Sigma_{\mathcal{A}}^* | \delta_w(\mathcal{I}) \cap \mathcal{F} \neq \emptyset\}$$

Das Vorkommen zweier Terminale mit disjunkter Heimat ist unabhängig von der Reihenfolge.

LEMMA 4 (Parallele Zeichenverarbeitung). *Sei $\mathcal{A}$ ein EAA, $a, b \in \Sigma_\mathcal{A}$ und $Dom(a) \cap Dom(b) = \emptyset$, dann gilt für alle $s, s' \in S$:*

$$\delta_{ab}(s) = s' \Leftrightarrow \delta_{ba}(s) = s'$$

Beweis siehe [**65**]. Um den EAA auf den Spurbegriff übertragen zu können, leitet man die Unabhängigkeitsrelation über das Alphabet des EAA ab und nutzt die parallele Zeichenverarbeitung.

LEMMA 5 (Erkennung von Spuren). *Sei $\mathcal{A}$ ein EAA. Mit*

$$I_\mathcal{A} = \{(a, b) \in \Sigma_\mathcal{A}^2 \;.\; Dom(a) \cap Dom(b) = \emptyset\},$$

gilt für zwei Wörter $v, w \in \Sigma_\mathcal{A}^*$, für die gilt $v \cong_{I_\mathcal{A}} w$ und beliebige Zustände $s, s' \in S$

$$\delta_v(s) = s' \;\Leftrightarrow\; \delta_w(s) = s'$$

Darauf läßt sich die von einem EAA erkennbare Spursprache definieren.

DEFINITION 45 (Spursprache eines EAA). *Sei $\mathcal{A}$ ein EAA, $(\Sigma_\mathcal{A}, I_\mathcal{A})$ das Alphabet mit Unabhängigkeitsrelation, $[w] \in E(\Sigma_\mathcal{A}, I_\mathcal{A})$ eine Spur, $s, s' \in S$ beliebige Zustände und $s_0 \in \mathcal{I}, s_f \in \mathcal{F}$, dann definieren wir*

$$\delta_{[w]}(s) = s' \;\Leftrightarrow\; \delta_w(s) = s'$$

und die Spursprache von $\mathcal{A}$ als

$$T(\mathcal{A}) = \{t \in E(\Sigma_\mathcal{A}, I_\mathcal{A}) \;.\; \delta_t(s_0) = s_f\}$$

Nach Lemma 5 akzeptiert ein EAA *alle* Wörter einer Spur $w \in \Sigma_\mathcal{A}^* \;.\; [w] = t$ oder *keines*. Also ist $T(\mathcal{A}) = [L(\mathcal{A})]$. Der EAA $\mathcal{A}$ über seine globale Zustandsmenge S betrachtet ist ein DEA. Daher ist die Sprache $L(\mathcal{A})$ regulär und damit gilt nach Korollar **??**:

LEMMA 6 (Erkennbarkeit der Spursprache). *Die Spursprache $T(\mathcal{A})$ über einem EAA $\mathcal{A}$ ist erkennbar.*

Durch Einführung zweier weiterer Begriffe in dieser Einordnung in die Theorie werden später weitere Vorteile der Definition des hier verwendeten dynamischen Performanzmodells offenbar.

DEFINITION 46 (Deterministischer und sicherer EAA). *Ein EAA $\mathcal{A} = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{I}, \mathcal{D}, \mathcal{F})$ heißt* deterministisch, *wenn*

- *es genau einen globalen Anfangszustand gibt:* $\mid \mathcal{I} \mid = 1$
- $\forall(\delta_a.a \in \Sigma_\mathcal{A}), s \in \Pi_{j \in Dom(a)} S_j \;.\; \mid \delta_a(s) \mid \leq 1.$

$\mathcal{A}$ heißt sicher, *wenn*

$$\forall s \in S, s_0 \in \mathcal{I}, s_f \in \mathcal{F}, v \in \Sigma^* \;.\; \exists w \in \Sigma^* \;.\; \delta_v(s_0) = s \Rightarrow \delta_w(s) = s_f.$$

Sichere EAAs können von jedem ihrer globalen Zustände zu ihrem Endzustand fortgesetzt werden und nur in diesem verklemmen. Intuitiv sieht man bereits jetzt, daß Szenarien als die Schaltsequenzen der Petrinetze durch Ausdrücke rationaler Spursprachen repräsentiert werden können. Sie können durch deterministische, sichere, endliche, asynchrone Automaten repräsentiert werden.

Abschlußeigenschaften und Entscheidbarkeitsfragen werden später wichtige technische Eigenschaften begründen. Deshalb folgen einige Sätze, zitiert aus [**65**, Ziel.], bzw. [**53**, Rein.] die später diskutiert werden.

SATZ 7 (Wortproblem). *Sei $T \subseteq E(\Sigma, I)$ eine beliebige rationale Spursprache über einem Alphabet und einer Unabhängigkeitsrelation und $t \in E(\Sigma, I)$, dann ist $t \in T$ entscheidbar.*

Entscheidbarkeit der Gleichheit für rationale Spursprachen.

SATZ 8 (Gleichheitsproblem bei rationalen Spursprachen). *Seien $T_1, T_2 \subseteq E(\Sigma, I)$ zwei beliebige rationale Spursprachen. Genau dann, wenn I quasi-transitiv ist, ist $T_1 = T_2$ entscheidbar.*

SATZ 9 (Gleichheitsproblem bei erkennbaren Spursprachen). *Seien $T_1, T_2 \subseteq E(\Sigma, I)$ zwei beliebige, erkennbare Spursprachen, definiert durch die endlichen Automaten der Wortsprachen $L(T_1), L(T_2)$, dann ist $T_1 = T_2$ entscheidbar.*

Die Untersuchung auch Gleichheit spielt in Werkzeugen eine Rolle, wenn die Kompatibilität von Szenario und Modell festgestellt werden muß. Die Beantwortung des Wortproblems geschieht während der Simulation selbst, was im Folgenden konkretisiert wird.

3.2. Dynamische Modelle während der Simulation

Nach Einführung grundlegender Begriffe und Vorstellungen aus der Theorie der Spuren und endlichen asynchronen Automaten soll die Beschreibung von einzelnen Abläufen dynamischer Modelle untersucht werden.

3.2.1. Grammatikalische Einordnung. Die Auswahl einer Kante durch Kommunikation ist für den Modellierer im dynamischen Performanzmodell sichtbar. Rein laufzeitbedingte Vorgänge sind im Modell unsichtbar. Die Bildung der Begriffe Ablauf und Szenario sind also für die Entwicklung von Testfällen sinnvoll. Folgend wird der Szenariobegriff anhand der eingeführten formalen Begriffe definiert.

3.2.1.1. *Endlicher, asynchroner Automat eines Modells.* Für ein Beispiel wird erneut das Modell aus Abschnitt 2.2.3.1 bemüht.

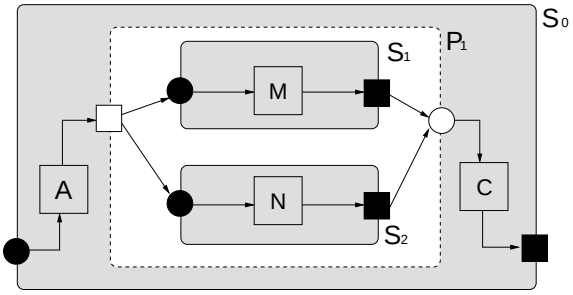


ABBILDUNG 3.2.1. Kartusche mit paralleler Region

BEISPIEL 8 (Beispielmodell als EAA).

Das Modell in Abbildung 3.2.1 weist eine einfache Nebenläufigkeit zweier Lasten auf. In der in Abbildung 3.2.2 dargestellten Petrinetzübersetzung wird aus Platzgründen wieder auf einige Modellelemente der vollständigen Übersetzung verzichtet.

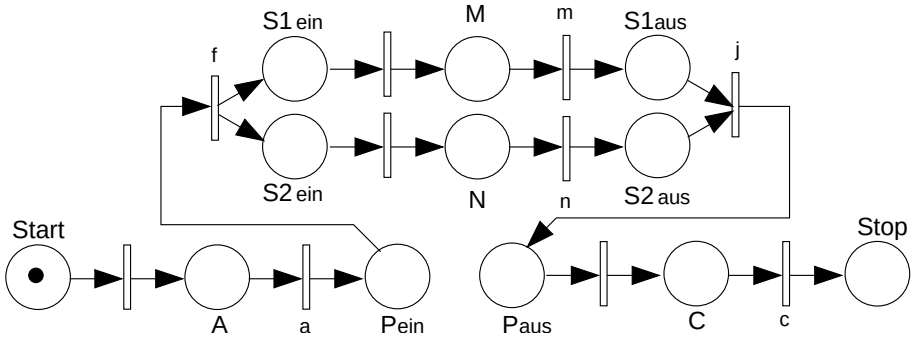


ABBILDUNG 3.2.2. Petrinetz des Modells

Anders als in Abschnitt 2.2.3.1 sind aber die Stellen $\mathsf{P_{ein}}$, $\mathsf{P_{aus}}$, $\mathsf{S1_{ein}}$, $\mathsf{S1_{aus}}$, $\mathsf{S2_{ein}}$ und $\mathsf{S2_{aus}}$ aufgeführt, die aus der Übersetzung der Strukturelemente des Modellbaumes stammen. Zur Erläuterung der Schaltsequenzen waren diese Stellen verzichtbar. In der Darstellung des asynchronen Automaten werden sie aber zur Synchronisation benötigt und haben eine wichtige Aufgabe. Im Beispiel sind die Lasten nur durch eine Stelle repräsentiert und das Augenmerk lastet auf der folgenden Transition, die das Terminieren dieser modelliert. Für die Last A ist diese folgende Transition in diesem Beispiel der einfacheren Lesbarkeit

halber mit a annotiert[1]. Die bislang unbeachteten Transitionen, die aus der Verzweigung und Vereinigung des Kontrollflußes in der Parallelkartusche stammen, bekommen jetzt eine zentrale Bedeutung. Die Verzweigung ist mit f (für *fork*) und die Vereinigung mit j (für *join*) annotiert.

Die Ereignisse m und n sind unabhängig, die Ereignisse a und f beispielsweise aber abhängig. Die Struktur eines Kartuschenmodells kann nun hilfreich sein einen EAA zu konstruieren, der diesen Eigenschaften Rechnung trägt. Die Synchronisation mit den Automaten der jeweils korrespondierenden nebenläufigen Region erfolgt durch gemeinsam verwendete Terminale. Diese Terminale ergeben sich genau aus den Transitionen der Parallelkartusche in welche die entsprechenden Sequenzkartuschen eingebettet sind. Im Beispiel sind dies die Terminale f und j der entsprechende Transitionen.

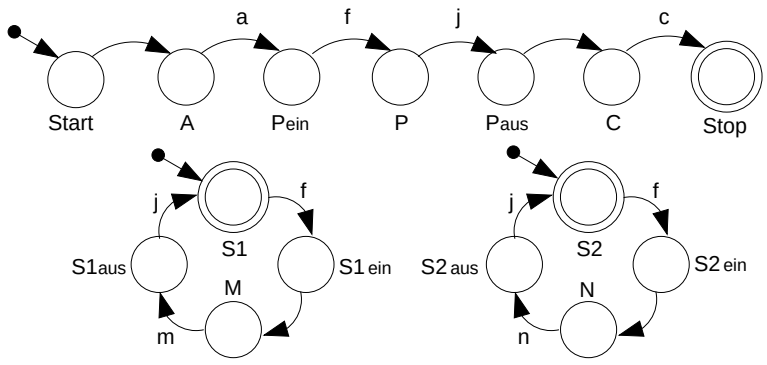


ABBILDUNG 3.2.3. Asynchroner Automat des Beispiels

Das Beispiel gibt unmittelbar folgende formale Darstellung:

$$\mathcal{A} = (\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3, \mathcal{I}, \mathcal{D}, \mathcal{F}) \quad mit:$$
$$\mathcal{P}_1 = (\Sigma_1, Q_1), \quad \Sigma_1 = \{\mathsf{a}, \mathsf{c}, \mathsf{f}, \mathsf{j}\},$$
$$Q_1 = \{\mathsf{Start}, \mathsf{A}, \mathsf{C}, \mathsf{P}_{\mathsf{ein}}, \mathsf{P}, \mathsf{P}_{\mathsf{aus}}, \mathsf{Stop}\},$$
$$\mathcal{P}_2 = (\Sigma_2, Q_2), \quad \Sigma_2 = \{\mathsf{m}, \mathsf{f}, \mathsf{j}\},$$
$$Q_2 = \{\mathsf{M}, \mathsf{S1}_{\mathsf{ein}}, \mathsf{S1}, \mathsf{S1}_{\mathsf{aus}}\},$$
$$\mathcal{P}_3 = (\Sigma_3, Q_3), \quad \Sigma_3 = \{\mathsf{n}, \mathsf{f}, \mathsf{j}\},$$
$$Q_3 = \{\mathsf{N}, \mathsf{S2}_{\mathsf{ein}}, \mathsf{S2}, \mathsf{S2}_{\mathsf{aus}}\},$$
$$\mathcal{I} = \{(\mathsf{Start}, \mathsf{S1}, \mathsf{S2})\},$$
$$\mathcal{F} = \{(\mathsf{Stop}, \mathsf{S1}, \mathsf{S2})\},$$

[1]Kleine Buchstaben finden im Folgenden wieder Verwendung, um Wächter, also Interaktion mit der Außenwelt, anzuzeigen. In diesem Beispiel 8 soll nur die ansonsten verwendete lange Schreibweise $\mathsf{a}_{\mathsf{term}}$ vermieden werden.

Und die partiell definierte globale Übergangsfunktion ergibt sich zu:

$$
\begin{aligned}
\mathcal{D} = \{ \quad \delta_a(\mathsf{A}) &\rightarrow (\mathsf{P_{ein}}),\\
\delta_c(\mathsf{C}) &\rightarrow (\mathsf{Stop}),\\
\delta_f(\mathsf{P_{ein}}, \mathsf{S1}, \mathsf{S2}) &\rightarrow (\mathsf{P}, \mathsf{S1_{ein}}, \mathsf{S2_{ein}}),\\
\delta_j(\mathsf{P}, \mathsf{S1_{aus}}, \mathsf{S2_{aus}}) &\rightarrow (\mathsf{P_{aus}}, \mathsf{S1}, \mathsf{S2}),\\
\delta_m(\mathsf{M}) &\rightarrow (\mathsf{S1_{aus}}),\\
\delta_n(\mathsf{N}) &\rightarrow (\mathsf{S2_{aus}}),\\
\delta_\epsilon(\mathsf{Start}) &\rightarrow (\mathsf{A})\\
\delta_\epsilon(\mathsf{P_{aus}}) &\rightarrow (\mathsf{C})\\
\delta_\epsilon(\mathsf{S1_{ein}}) &\rightarrow (\mathsf{M})\\
\delta_\epsilon(\mathsf{S2_{ein}}) &\rightarrow (\mathsf{N}) \quad \}
\end{aligned}
$$

Hinweis: ϵ-Transitionen des EAA existieren nur im Beispiel durch Weglassung uninteressanter Netzelemente. Eine Werkzeugimplementierung wird die Lasten kontrolliert aktivieren. Jede Transition eines Modells hat eine eindeutige Identität, die zum Gesamtalphabet gerechnet wird. Damit ist $\forall(\delta_a.a \in \Sigma_{\mathcal{A}}), s \in \Pi_{j \in Dom(a)} S_j. \mid \delta_a(s) \mid \leq 1$, d.h. daß keine zwei Übergangsrelationen für das gleiche Terminal (bzw. die gleiche Transition) in einem Teilautomaten existieren, und $\mid \mathcal{I} \mid = 1$ folgt nach Definition 46, daß $\mathcal{A}$ sicher und deterministisch ist. Aus den Definitionen des dynamischen Performanzmodells läßt sich dies generell für alle Petrinetz-Übersetzungen sagen. Weiterhin ist die getrennte Darstellung des *Start-* und *Stop*zustands optional. Ebenso ist die Aufnahme der Endzustände der Teilautomaten in den globalen Endzustand überflüßig, da dies aus der Modellstruktur folgt.

Diese Erzeugung eines EAA aus dem Modell soll als Grundlage für eine Werkzeugimplementierung formal skizziert werden.

3.2.1.2. *Erzeugung des endlichen, asynchronen Automaten aus dem Modell.* Jede Transition des Netzes wird zu einem Terminal in einem der Kartusche zugeordneten Teilalphabet. Jede Stelle wird zu einem Zustand und aus der Flußrelation läßt sich die Schar der Übergangsrelationen ableiten. Die Zugehörigkeit zur richtigen Teilmenge läßt sich dabei ausdrücken durch eine Hilfsfunktion, die eine Umkehrung der semantischen Funktionen darstellt.

Mit der semantischen Umkehrfunktion aus Definition 27 läßt sich die Spursprache des dynamischen Modells in Form eines endlichen, asynchronen Automaten unmittelbar darstellen.

DEFINITION 47 (Spursprache eines Modells). *Sei N_M das Petrinetz eines Modells $M = \left(\Upsilon, \Theta, \mathcal{R}, \rho, e_M\right)$ mit $\mathcal{M}[\![M]\!] = (S, T, F)$. Die Menge aller Kartuschen wird zusammengefaßt als:* $\Xi = \Upsilon \cup \Theta$.

Die Spursprache des Netzes wird definiert durch den endlichen, asynchronen Automaten $\mathcal{A} = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{I}, \mathcal{D}, \mathcal{F})$. *Dabei ist* $n = |\Xi|$. *Die Teilalphabete* $\mathcal{P}_1, \ldots, \mathcal{P}_n$ *leiten sich ab von den Kartuschen* $X_1, \ldots, X_n \in \Xi$. *Dabei erzeuge eine Funktion A den EAA nach* $\mathcal{A} = A(N_M)$ *wie folgt:*

Jeder Teilautomat $P_k = (\Sigma_k, , Z_k).1 \leq k \leq n$ *setzt sich aus dem Teilalphabet* Σ_k *und der Zustandsmenge* Z_k *wie folgt zusammen:*

$$\begin{aligned} \Sigma_k = & \quad \{t \in T \ . \ elem(t) \in \Xi_k\} \\ Z_k = & \quad \{s \in S \ . \ elem(s) \in \Xi_k\} \end{aligned}$$

Die Schar der Übergangsfunktionen ergibt sich zu:

$$\mathcal{D} = \left\{\delta_{t \in T}(\{s \in S.(s,t) \in F\}) = (\{s \in S.(t,s) \in F\})\right.$$

Schließlich sind:

$$\begin{aligned} \mathcal{I} = & \quad \{Start \in S\} \\ \mathcal{F} = & \quad \{Stop \in S\} \end{aligned}$$

Diese Definition sieht eine vollständige Übersetzung aller Kartuschen vor. Parallelkartuschen $X \in \Upsilon \subseteq \Xi$ fügen stets spontane Transitionen in die Teilalphabete ein. Diese entsprechen ϵ-Übergängen. In den folgenden Beispielen wird das Netz der Übersichtlichkeit halber um die spontanen Transitionen reduziert und eine Zerlegung nur in die Teilalphabete dargestellt, die aus den Sequenzkartuschen $X \in \Theta \subseteq \Xi$ entstanden sind.

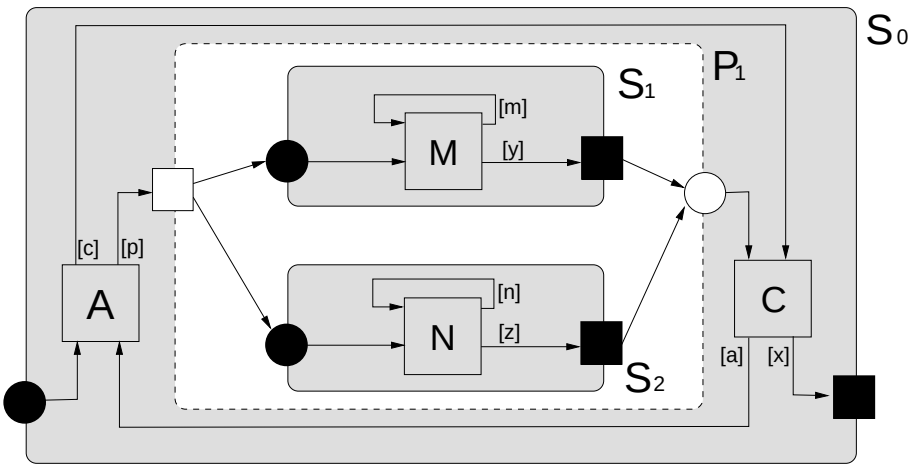


ABBILDUNG 3.2.4. Beispiel für Spursprachen

3.2.1.3. *Modelle mit Ablaufsteuerung und Nebenläufigkeit.* In Abbildung 3.2.4 wird ein Modell vorgestellt, welches Nebenläufigkeit aufweist, Transitionen mit Auswahlmöglichkeit besitzt und mehrere Zyklen besitzt. Hier sind unterschiedliche Transitionstypen kombiniert und sollen zur informellen Einführung des Problems dienen.

Auf die Darstellung der Petrinetze wird aus Platzgründen verzichtet. Wir betrachten unmittelbar den Erreichbarkeitsgraphen mit dem interessanten Teil der Markierung, die neun Stellen, welche die Bereitschaft der Lasten A, M, N, C und deren vollständige Bearbeitung bezeichnen, und den Stop-Zustand.

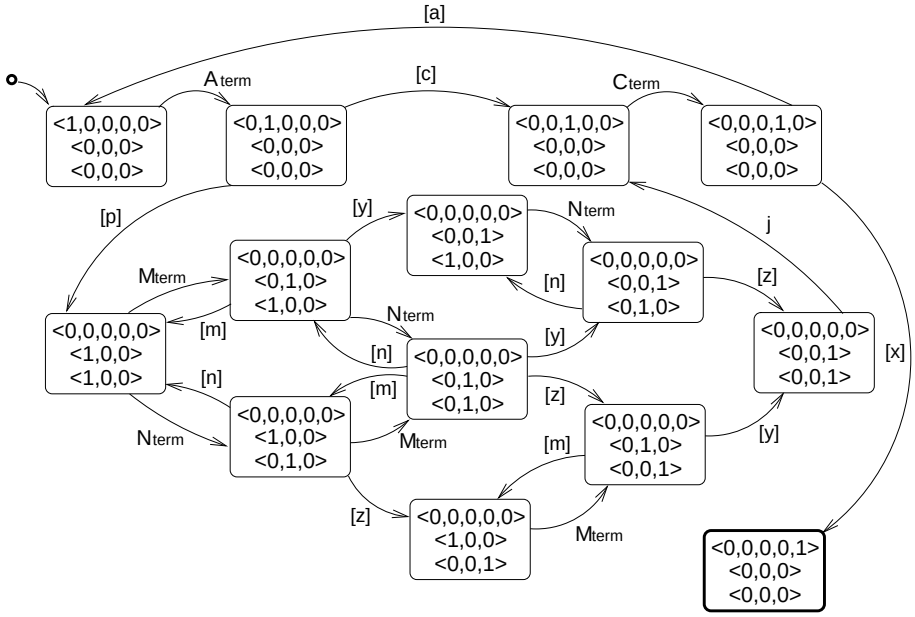


ABBILDUNG 3.2.5. Erreichbarkeitsgraph, Beispiel

BEISPIEL 9 (Komplexes Beispielmodell).

Sind die Lasten der eingebetteten Sequenzkartuschen fertig bearbeitet, muß in beiden Kartuschen jeweils zunächst die Entscheidung getroffen werden, ob die Last M, bzw N wiederholt bearbeitet wird, oder die Parallelkartusche verlassen werden soll. Der leichteren Lesbarkeit halber teilen wir die Markierung in drei Vektoren auf. Der erste enthält die Elemente der Wurzelkartusche nach: $\langle m(\mathsf{A}_{\mathsf{aktiv}}), m(\mathsf{A}_{\mathsf{fertig}}), m(\mathsf{C}_{\mathsf{aktiv}}), m(\mathsf{C}_{\mathsf{fertig}}), m(\mathsf{Stop})\rangle$, der Zweite die der S_1-Kartusche: $\langle m(\mathsf{M}_{\mathsf{aktiv}}), m(\mathsf{M}_{\mathsf{fertig}}), m(\mathsf{z}'_{\mathsf{End}})\rangle$ und der Dritte die der S_2-Kartusche: $\langle m(\mathsf{N}_{\mathsf{aktiv}}), m(\mathsf{N}_{\mathsf{fertig}}), m(\mathsf{z}''_{\mathsf{End}})\rangle$. In Abbildung 3.2.5 ist der resultierende Erreichbarkeitsgraph dargestellt.

In Abschnitt 2.4 wurden *performanzdeterminierte* Verzweigungen und *kommunikationsdeterminierte* Verzweigungen am Erreichbarkeitsgraphen vorgestellt. An Verzweigungen, die mit performanzbezogenen Annotationen bezeichnet sind, kann durch den Simulator eine Auswahl getroffen werden. Dies waren die alleinigen Transitionen des Beispiels 8.

Kommunikationsbezogene Verzweigungen resultieren aus Transitionen des Petrinetzes, die im Konflikt stehen. Diese bilden Schleifen oder Sprünge und tragen jeweils an der Verzweigung unterscheidbare Kantenanschriften. Dies war eine Anforderung für Transitionen im Konflikt. Der durch diesen Erreichbarkeitsgraphen gebildete Automat ist also deterministisch. Die Wahl einer solchen Kante entspricht dem Schalten der entsprechenden Transition, welche diese Anschrift trägt. Im Erreichbarkeitsgraphen sind nur die Anschriften abgetragen. Da aber die Anschriften im Zusammenhang mit den Transitionen interpretiert werden, sind auch zwei Transitionen mit dem gleichen Label modellweit eindeutig unterscheidbar.

Im Erreichbarkeitsgraphen sind ebenso alle Kombinationen an Annotationstypen zu finden. Beispielsweise Markierungen, deren Folgemarkierung nur anhand kommunikationsbezogener Anschriften zu ermitteln ist, wie z.B. die Markierung $\langle 0,1,0,0,0\rangle\langle 0,0,0\rangle\langle 0,0,0\rangle$, welche die Auswahl oder die Umgehung der Parallelkartusche realisiert. An der Markierung $\langle 0,0,0,0,0\rangle\langle 1,0,0\rangle\langle 1,0,0\rangle$ erkennt man ausschließlich performanzbezogene Annotationen. In dieser Markierung entscheidet das frühere Terminieren einer Lastbearbeitung über die Wahl des Folgezustandes.

Ein besonderer Fall ist die Markierung $\langle 0,0,0,0,0\rangle\langle 0,1,0\rangle\langle 1,0,0\rangle$. Diese Markierung mit ihren ausgehenden Transitionen ist in Abbildung 3.2.6 dargestellt. In dieser Abbildung ist die Kartuschenzugehörigkeit mit angegeben.

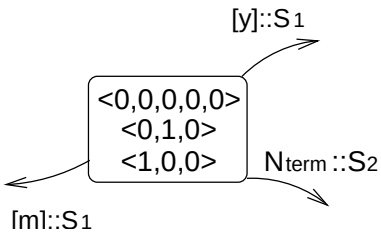


ABBILDUNG 3.2.6. Einzelzustand des Erreichbarkeitsgraphen

Auf den ersten Blick sieht man, daß von den Zuständen des Erreichbarkeitsgraphen ausgehende Kanten zu beiden Typen gehören können.

Später führen wir ein, daß das Schalten einer performanzdeterminierten Transition nur erfolgt, falls keine anderen Transitionen Konzession haben, und man auf diese Transition gewissermaßen warten muss. Diese Anforderung, die sich aus dem willentlich festgelegten Zeitbegriff ergibt, wird in den nächsten Kapiteln formal definiert.

Zur Spezifikation eines Testfalles müssen nur für alle Transitionen an denen eine Auswahl stattfinden muß eine Entscheidung getroffen und geeignet gespeichert werden. Die Ableitung des gesammten EAAs für dieses Beispiel würde deutlich zu umfangreich sein. Hauptsächlich würden auch Zustände und Transitionen spezifiziert, die strukturellen Ursprung haben oder performanzbezogen sind. Das Modell von Beispiel 9 eignet sich gut um interessante Szenarien zu formulieren. Die im folgenden Kapitel herausgearbeiteten Definitionen dienen einer effizienten Implementierung und einer inuitiven Darstellung, die Szenarien auf die wesentlichen Elemente reduziert. Beipiel 9 wird deshalb in den nächsten Abschnitten wiederholt zitiert.

3.2.2. Abläufe. Die Sprache $T(\mathcal{A})$ des Modells aus Beispiel 9 ist mächtiger als die des vorherigen Beispiels. Zunächst notieren wir einen Ablauf durch eine Kette von Zeichen, die einen konkreten Ablauf angeben. Großbuchstaben symbolisieren zunächst das Terminieren einer Lastbearbeitung. Kleinbuchstaben symbolisieren eine Auswahl einer Kante. Zum Beispiel AcCx heißt nach beendigter Bearbeitung der Last A wird die Kante c gewählt, dann beginnt die Bearbeitung der Last C. Terminiert diese, wird Kante x gewählt und man erreicht den Endzustand. Tritt das Modell in eine nebenläufige Region ein, ist die Reihenfolge der Lasttermination nicht mehr eindeutig zu beantworten, da der die Reihenfolge der Termination nebenläufiger Lasten erst in der sequentiellen Simulation entschieden werden kann. In diesem Fall gehört ein Ablauf zu einer Äquivalenzklasse. Jedes Element dieser Klasse unterscheidet sich dann nur durch eine Vertauschung von Elementen, die unabhängig voneinander sind.

Die Größe der Äquivalenzklassen ist unterschiedlich. Für die Beispiele $w_1 =$ AcCx und $w_2 =$ ApMNyzjCx, die zum Erreichbarkeitsgraphen in Abbildung 3.2.5 passen, ergeben sich $|[w_1]| = 1$ und $|[w_2]| = 6$. Die Größe von w_2 verwundert, da erkennbar ist, daß nur die zwei Fälle, nämlich das frühe Terminieren von M oder N für die Simulation interessant sind. Am Erreichbarkeitsgraphen läßt sich die Größe an der Anzahl an Pfaden ablesen, auf denen der Endzustand zu erreichen ist. Als Einzelszenarien geschrieben sind dies sechs Einzelabläufe ApMNyzjCx, ApNMyzjCx, ApMNzyjCx, ApNMzyjCx, ApMyNzjCx und ApNzMyjCx, die zu einer Äquivalenzklassen gehören. Das Terminal j rührt hier von der

Transition her, welche die Kontrollflüsse der Parallelkartusche wieder vereint. Im Erreichbarkeitsgraphen in Abbildung 3.2.5 ist die entsprechende Kante so beschriftet. Da diese Transition die zur Synchronisation tragende Rolle spielt wird sie hier mitgeführt und der Umgang mit ihr später erläutert.

3.2.2.1. *Performanz-Szenario.* Jeder bislang betrachtete Ablauf modellierte eine eindeutige Reihenfolge *aller* Ereignisse. Es wird festgelegt, daß alle Transitionen, die nicht zeitbehaftet sind und Konzession haben, unmittelbar feuern können. Dies sind Transitionen, die aus Strukturelementen stammen und die Auswahl bei Transitionen im Konflikt. Erst an Zuständen des Erreichbarkeitsgraphen, deren ausgehende Kanten nur performanzabhängige Annotationen tragen sollen, diese aufgelöst werden.

In den Symbolsequenzen galt die Aufmerksamkeit Transitionen, die das Terminieren von Lasten modellieren für alle $a \in \Sigma_{\mathcal{A}}.(elem(a) \in \mathcal{Z}_{LE})$, sowie den Transistionen, die Interaktion mit der Außenwelt modellieren für alle $a \in \Sigma_{\mathcal{A}}.(elem(a) \in \delta)$. Beispiele wie $w_1 =$ AcCx, $w_2 =$ ApMNyzjCx oder $w_3 =$ ApMNynNzjCx.

Der Szenarioentwickler möchte nur die Reihenfolge der Interaktionen mit der Außenwelt spezifizieren. Diese Auswahl an Transitionen im Konflikt können als Modelle für Benutzerauswahl, Ereignisse oder Bedingungen verstanden werden. Wie sie im Softwaremodell zu verstehen sind, muß nicht angegeben werden, da nur die Auswirkung durch die Auswahl einer Transition für die Simulation entscheidend ist. Eine Szenariodarstellung, die Elemente, die Lastterminierung modellieren reduziert ist, bietet eine bessere Übersicht. In den vorherigen Beispielen würde also $w_1 =$ cx, $w_2 =$ pyzx oder $w_3 =$ pynzx zum Verständnis eines Ablaufes genügen.

DEFINITION 48 (Spurreduktion). *Sei $\mathcal{A}_M = A(N_M)$ der aus dem Netz N_M erzeugte EAA, $a \in \Sigma_{\mathcal{A}}$ und $v, w \in \Sigma_{\mathcal{A}}*$, dann ist die Reduktion $red_\delta : \Sigma^* \to \Sigma^*$ um Terminierungstransitionen definiert als:*

$$red_\delta(w) = \begin{cases} a \circ red_\delta(v), & \texttt{falls } w = av,\ a \in \Sigma_{\mathcal{A}}\ .\ (elem(a) \in \delta) \\ red_\delta(v), & \texttt{falls } w = av,\ a \in \Sigma_{\mathcal{A}}\ .\ (elem(a) \in \mathcal{Z}_{LE}) \\ \epsilon, & \texttt{falls } w = \epsilon \end{cases}$$

Das Beispiel $w_3 =$ pynzx drückt aus, daß die Parallelkartusche einmal bearbeitet, die Last M in der Sequenzkartusche S_1 einmal und die Last N in der Sequenzkartusche S_2 zweimal abgearbeitet werden sollen.

[1]Die Darstellung zur Vereinfachung um weniger bedeutende Terminale verkürzt.

Da die Reihenfolge der Bearbeitung der Auswahl von y, n und z in Bezug auf das Szenario unabhängig ist, ist der Ausdruck $w_3' = \mathsf{pnyzx}$ äquivalent. Was ausgedrückt werden soll, ist die Spur $[w_3] \in T(\mathcal{A})$. Diese Spur spezifiziert eine eindeutige Auswahl einer Reihe von Entscheidungen, aber nicht die Reihenfolge, die nach der Unabhängigkeitsrelation äquivalent ist und nicht die Reihenfolge des Terminierens von Lasten.

3.2.2.2. *Spurzerlegung.* Ein Szenario der nach der eingeführten Definition enthält alle nötigen Informationen zur Ablaufsteuerung. Die Darstellung ist bislang aber nicht sehr anschaulich. In der Theorie der Spursprachen gibt es zahlreiche Zerlegungstechniken. Hier soll eine Zerlegung definiert werden, die einen Ausdruck in Teilausdrücke zerlegt, welche die Abläufe einer bestimmten Sequenzkartusche beschreiben und reduziert sind auf die Terminale, welche die Auswahl einer bestimmten Übergangsfunktion bewirken. Ein so zerlegtes Szenario ist im Hinblick auf ein dynamisches Modell leicht lesbar und hält sowohl für den Szenarienentwickler, als auch bei der Werkzeugimplementierung benötigte Informationen in geeigneter Form bereit.

Bei w_3, das noch aus einem recht einfachen Ablauf stammt, ist schwer zu erkennen, durch welchen Weg der Verlauf bestimmt ist. Damit ist gemeint, daß Begriffe wie die Anzahl der Ausführung von Schleifen nicht offensichtlich sind. Die Verteilung der Terminale auf die Teilautomaten ist eine wichtige strukturelle Information, die erhalten werden soll.

DEFINITION 49 (Spur-Reduktion auf Teilautomaten). *Sei $\mathcal{A} = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{I}, \mathcal{D}, \mathcal{F})$ ein aus einem Modell erzeugter EAA, $\Sigma_{\mathcal{A}}$ das globale Alphabet, $\mathcal{P}_i = (\Sigma_i, S_i)$ ein Teilalphabet von $\mathcal{A}$, $a \in \Sigma_i$ ein beliebiges Terminal aus dem Alphabet des Teilautomaten $\mathcal{P}_i$, $v, w \in \Sigma_{\mathcal{A}}^*$, Teilausdrücke über dem globalen Alphabet, dann ist die Reduktion $red_i : \Sigma_{\mathcal{A}}^* \to \Sigma_i^*$ definiert als:*

$$red_i(w) = \begin{cases} a \circ red_i(v), & \texttt{falls } w = av \;.\; a \in \Sigma_i \\ red_i(v), & \texttt{falls } w = av \;.\; a \notin \Sigma_i \\ \epsilon, & \texttt{falls } w = \epsilon \end{cases}$$

DEFINITION 50 (Spurzerlegung). *Sei $\mathcal{A} = (\mathcal{P}_1, \ldots, \mathcal{P}_n, \mathcal{I}, \mathcal{D}, \mathcal{F})$ ein aus dem Modell M erzeugter EAA mit n-Teilautomaten, $w \in L(\mathcal{A})$ ein Ausdruck über der Sprache von $\mathcal{A}$ und $[w]_{\mathcal{A}} \in T(\mathcal{A})$ eine Spur über der durch den Automaten definierten Spursprache, dann ist die* Zerlegung *einer Spur die Menge $\Delta([w]_{\mathcal{A}})$ mit den Eigenschaften:*

$$\Delta([w]_{\mathcal{A}}) = \{[v_1], \ldots, [v_n]\}, \quad \texttt{mit}$$
$$\forall v_i, 1 \leq i \leq n \;.\; v_i = \; red_i(w)$$

BEISPIEL 10 (Verschiedene Szenario-Darstellungen). *Folgendes bereits in Abb 3.2.4 eingeführte Beispiel soll die Vorteile der Spurreduktion und Spurzerlegung zeigen.*

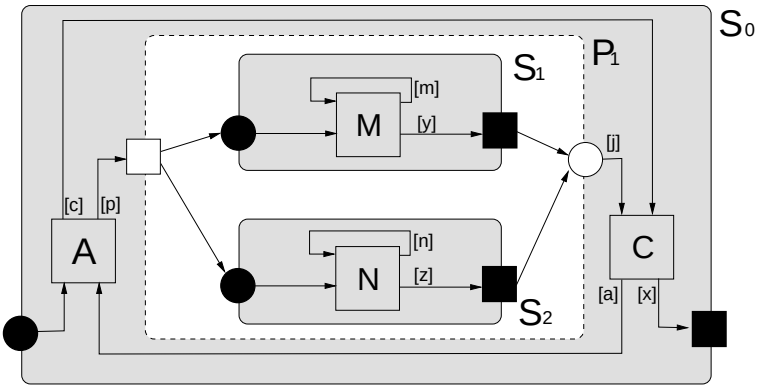


ABBILDUNG 3.2.7. Beispiel zur Reduktion

Ein Ablauf der durch die Auswahl p *die Parallelkartusche betritt, dann beide Sequenzkartuschen* S_1 *und* S_2 *startet. Die Last* M *wird in der Kartusche* S_1 *nur einmal bearbeitet. Die Kartusche wird durch die Auswahl* y *sofort verlassen. Die Last* N *soll hingegen zweimal bearbeitet werden. Dazu muß einmal die Auswahl* n *getroffen werden um die Bearbeitung zu wiederholen. Dann wird die Kartusche* S_2 *durch die Wahl von* z *verlassen. Die Parallelkartusche kann nun verlassen werden und die Bearbeitung der Last* C *beginnt. Nach der Termination von* C *wird das Modell durch Auswahl von* x *beendet. Dieser Ablauf wurde bereits als Beispiel* $w_3 =$ ApMyNnNzjCx *eingeführt.* w_3 *ist dabei ein typisches Element, welches aber die gesamte Äquivalenzklasse* $[w_3]$ *repräsentieren soll. Dabei haben wir die spontane Transition, welche aus der Vereinigung der Kontrollflüße der Parallelkartusche stammt wieder mit dem Terminal* j *mitgeführt.*

Die Spurreduktion nach Definition 48 entfernt dabei die bei der Spezifikation der Abläufe unwichtigen Lastterminationen und ergibt damit $red_\delta(w_3) =$ pynzx.

Bei der Spurzerlegung bleibt die Information über die Teilalphabete erhalten und man kommt zu einer intuitiveren Darstellung. $w_3 =$ ApMyNnNzjCx *wird somit festgelegt durch* $\Delta(w_3) =$ {ApjCx,Myj,NnNzj}. *Die sequentiellen Teilabläufe der einzelnen Kartuschen sind gesondert dargestellt. In der zerlegten vollständigen Darstellung wird die Synchronisation über das Terminal* j *augenfällig. Dies wird bei der Simulation äußerst wichtig sein, für*

das Verständnis der Szenarien ist es aber überflüssig. Die lokalen Entscheidungen an Verzweigungen in den einzelnen Kartuschen werden in der Darstellung eines zerlegten Szenarios mit $\Delta(red_\delta(w_3)) = \{\mathsf{px,y,nz}\}$ *ausreichend dargestellt. Jedes Element der zerlegten Spur kann damit als Ausdruck eines Teilablaufs einer einzelnen Sequenzkartusche verstanden werden.*

Die reduzierte, zerlegte Spur ist günstig für effiziente Speicherung eines Testfalles und für die Visualisierung für den Szenarienentwickler, da diesem vor allem die Annotationen an den auswählbaren Kanten geläufig sind. Für die spätere Simulation findet die vollständige Spur Verwendung.

DEFINITION 51 (Performanz-Szenario). *Sei* $N_M = \mathcal{M}[\![M]\!]$ *das Petrinetz zum Modell* M, $\mathcal{A}_M = A(N_M)$ *der aus dem Modell erzeugte EAA,* $L_M = L(\mathcal{A}_M)$ *die Sprache des Modells,* $w \in L_M$ *ein beliebiger gültiger Ausdruck, dann ist die zerlegte, reduzierte Spur*

$$szen_{w,M} = \Delta\big(red_\delta([w]_M \in T(\mathcal{A}_M))\big)$$

ein Performanzszenario *für das Modell* M. *Die zerlegte, aber unreduzierte Spur*

$$in = \Delta\big([w]_M \in T(\mathcal{A}_M)\big)$$

ist ein Szenario *oder* vollständiges Szenario. *Die Menge aller Szenarien eines dynamischen Modells bezeichnen wir folgend mit* $\mathcal{I}_{Szenario}$.

3.2.2.3. *Ablaufsteuerung.* Zweck eines Szenarios ist die Steuerung eines Modells während der Simulation bezüglich der Aspekte des Ablaufes, die nicht performanzabhängig sind. Basierend auf dem Szenario läßt sich ein Prädikat über alle Transitionen des Petrinetzes und damit über alle Kanten des Erreichbarkeitsgraphen definieren, die Wächterannotationen tragen.

DEFINITION 52 (Ablaufsteuerung). *Sei* $in = \langle u_1, \ldots, u_n \rangle \in \mathcal{I}_{Szenario}$ *ein Szenario für das Modell* M, $\mathcal{A}$ *der zugrundeliegende endliche, asynchrone Automat. Dann ist:*

$$\eta_{in} : \quad T \to \mathbb{B}$$

$$\eta_{in}(t) = \begin{cases} \texttt{true}, & \texttt{falls es ein } \delta_t \in \mathcal{D} \texttt{ des zugrunde} \\ & \texttt{liegenden Automaten } \mathcal{A} \texttt{ gibt,} \\ \texttt{false}, & \texttt{sonst.} \end{cases}$$

Es wird festgelegt, daß ein Szenario während der Simulation seinem Modell folgt, d.h. von ihm gelesen wird. Schaltet eine Transition des Modells oder wird nach Auswahl einer wächterbewehrten Kante,

im Erreichbarkeitsgraphen ein Folgezustand eingenommen, so wird das Szenario *gelesen.*

DEFINITION 53 (Lesen des Szenarios). *Sei* $in = \langle u_1, \ldots, u_n \rangle \in \mathcal{I}_{Szenario}$ *ein Szenario für das Modell* M *mit dem zugrundeliegenden endlichen asynchronen Automaten* $\mathcal{A}$. *Sei* $\alpha : T \to \Sigma$ *eine Funktion, die das Terminal liefert, welches durch die Transition gebildet wurde.*

Für das Schalten einer Transition t, *das die Markierung in eine Folgemarkierung gemäß* $m[t > m'$ *überführt,* $a = \alpha(t)$ *das Terminal welches der Transition entspricht und* δ_a *die* a*-Übergangsfunktion des Automaten* $\mathcal{A}$ *und* S *ein globaler Zustand aus* $\Pi_{i \in \{1..n\}} S_i$ *gemäß Definition 43. Dann wird festgelegt: Eine feuernde Transition überführt auch das angewendete Szenario mit seinem zugrundeliegenden endlichen asynchronen Automaten* $\mathcal{A}$ *in einen Folgezustand gemäß:*

$$S' = \delta_a(S)$$

Sei weiterhin jeder Ausdruck u_i *der Form* $u_i = a_i v_i$, *mit* $(a \in \Sigma \wedge v \in \Sigma^*)$, *oder* $u_i = \epsilon$, *so wird für das Schalten einer Transition* t, *das die Markierung in eine Folgemarkierung gemäß* $m[t > m'$ *überführt, wird festgelegt: Eine feuernde Transition überführt auch das angewendete Szenario in einen Lesezustand gemäß der Funktion:*

$$st_{in} \ : \ \mathcal{I}_{Szenario} \times T \to \mathcal{I}_{Szenario}$$

$$\texttt{mit:} \quad st_{in}(\langle u_1, \ldots, u_n \rangle, t) = \langle u'_1, \ldots, u'_n \rangle$$

$$\texttt{wobei:} \ u'_i = \begin{cases} u_i, & \texttt{falls } (u_i = a_i v_i) \wedge (a_i \neq \alpha(t)) \\ v_i, & \texttt{falls } (u_i = a_i v_i) \wedge (a_i = \alpha(t)) \\ \epsilon, & \texttt{falls } u_i = \epsilon \end{cases}$$

Die technische Vereinbarung sorgt dafür, daß das Szenario mit dem Modell gekoppelt ist und bei einer Zustandsänderung selbsttätig das Szenario gelesen wird. Man kann sich dies als eine Art Cursor vorstellen, der für jeden Ausdruck der zerlegten Menge existiert und beim Schalten einer Transition inkrementiert wird. Die Definition ist in der formalen Darstellung intuitiv verständlich und gleichzeitig direkt implementierbar.

Die Kopplung des Modells, repräsentiert durch den Erreichbarkeitsgraphen, und des Szenarios ist so eng, daß später bei der Definition der Simulation der Erreichbarkeitsgraph nicht mehr referenziert werden muß. Der Erreichbarkeitsgraph des Modells enthält Zyklen. Das Szenario beschreibt einen Pfad im Graphen und ist somit eine Abwicklung des azyklischen gerichteten Graphen. Jedem Lesezustand eines Szenarios kann also eindeutig ein Knoten des Erreichbarkeitsgraphen

zugeordnet werden. (Die Umkehrung gilt nicht). Diese Zuordnung ist durch iterative Anwendung der st_{in}-Funktion definierbar, kann aber auch unmittelbar aus dem Lesezustand abgeleitet werden.

DEFINITION 54 (Ermittlung des Knoten im Erreichbarkeitsgraphen). *Gegeben ein Modell M und $(S,T,F) = \mathcal{M}[\![M]\!]$ die zugehörige Netzsemantik, $(V,E) = G(\mathcal{M}[\![M]\!]$ der Erreichbarkeitsgraph. Sei $\alpha : T \to \Sigma$ eine Funktion, die das Terminal liefert, welches durch eine Transition gebildet wurde.*

Sei $in = \langle u_1, \ldots, u_n \rangle \in \mathcal{I}_{Szenario}$ ein Szenario für das Modell M. Sei weiterhin jeder Ausdruck $u_i \in in$ der Form $u_i = a_i v_i$, mit $(a \in \Sigma \wedge v \in \Sigma^*)$, oder $u_i = \epsilon$.

Dann ist eine Funktion $v : \mathcal{I}_{Szenario} \to V$, welche zu einem Lesezustand des Szenarios den zugehörigen Knoten des Erreichbarkeitsgraphen v liefert gegeben mit:

$$v(in) = v \;.\; \big(m(s) = 1 \Leftrightarrow (\exists t, (s,t) \in F) \wedge (\exists a_i, \alpha(t) = a_i)\big)$$

Dh. alle markierten Stellen $s \in S$ konzessionieren mindestens eine Transition $t \in T$, die im Lesezustand des Szenarios spezifiziert wird.

3.2.2.4. *Schleifen.* Im Beispiel 10 wurde die Last N zweimal ausgeführt. Eine wiederholte Ausführung spiegelt sich in der Iteration von Teilspuren wieder. Das Szenario einer 5-fachen Wiederholung etwa würde wie folgt aussehen: $w_4 = \{[cx], [y], [nnnnz]\}$. Da in der Testfallentwicklung vor allem die Kardinalität solcher Wiederholungen ausschlaggebend ist, empfiehlt sich für eine geschickte Implementierung der Iteration beispielsweise die Angabe eines Multiplikators.

Die Erstellung von Szenarien erfolgt sequentiell, und die Steuerung eines Werkzeugs kann unmittelbar durch das Petrinetz übernommen werden. Ein spezielles Problem ist die Identifikation von Schleifen.
Legt man folgende Notation fest:

$$t^-(s) = \begin{cases} 1, & \texttt{falls } (s,t) \in F \\ 0, & \texttt{sonst} \end{cases}$$

$$t^+(s) = \begin{cases} 1, & \texttt{falls } (t,s) \in F \\ 0, & \texttt{sonst,} \end{cases}$$

kann man das Netz $N = (S,T,F)$ als $|S| \times |T|$-Matrix betrachten, wobei $C_{i,j} = t_j^+(s_i) - t_j^-(s_i)$. Eine Schleife ist dadurch charakterisiert, daß sie eine Markierung m in einer endlichen Schaltsequenz in eine Markierung m' überführt, für die gilt $m = m'$. Man bezeichnet dies als eine Transitions- oder T-Invariante, siehe **[57, 30]**. Eine T-Invariante

hat eine ganzzahlige Lösung λ des Gleichungssystems $C_{i,j} \cdot \lambda = 0$. Die Menge der Zyklen im Modell entspricht also $\{\lambda.(C_{i,j} \cdot \lambda = 0) \wedge (\lambda > 0)\}$.

So identifizierte Schleifen können in einer Werkzeugimplementierung gesondert behandelt werden. Die Schaltfolge, welche diese Schleife bildet, kann als Ganzes zur Auswahl gestellt werden, und dem Testfall-Entwickler kann die Möglichkeit gegeben werden, die Kardinalität dieser Schleife als Zahl anzugeben.

3.3. Kommunikation

Das bislang behandelte Modell erlaubt die Spezifikation des Kontrollflusses. Zwar kann man Szenarien als Kommunikationsaspekt mit der Außenwelt betrachten. Das Problem des Datenflusses und der Kommunikation innerhalb des Modells wurde jedoch bislang nicht behandelt.

3.3.1. Kommunikation im Kartuschenmodell. Zeitrelevante Bearbeitung von Daten kann mit den besprochenen Techniken als Last modelliert werden, falls der Datenfluß deckungsgleich mit dem Kontrollfluß ist. Bislang nicht ausdrückbar ist das Abweichen des Datenflusses vom Kontrollfluß und die Synchronisation zwischen nebenläufigen Bereichen, die als wesentlicher Effekt der Kommunikation zwischen Prozessen betrachtet werden müssen.

3.3.1.1. *Notation.* Das Thema Kommunikation wird hier nur am Beispiel der asynchronen Kommunikation behandelt. Dazu wird das *Senden* und das *Empfangen* von Nachrichten eingeführt.

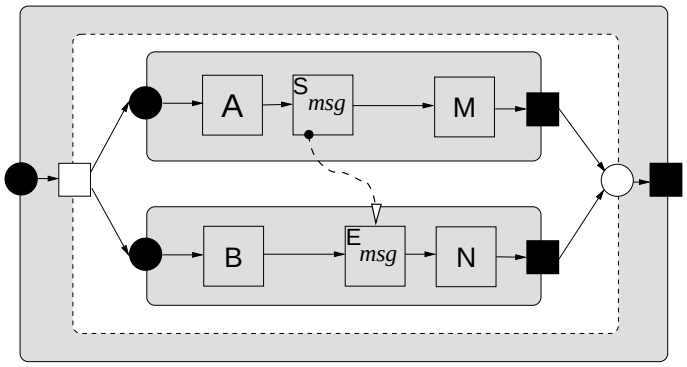


Abbildung 3.3.1. Parallelkartuschen mit Kommunikation

In Abbildung 3.3.1 ist eine Parallelkartusche dargestellt, die zwei kommunizierende, nebenläufige Bereiche darstellt. Es sind zwei Lastelemente dargestellt, die das Senden und Empfangen einer Nachricht

modellieren. Das Element, welches das Senden bezeichnet, ist mit einem S gekennzeichnet und trägt einen Identifikator. In Beispiel hier ist es die Zeichenkette msg. Das korrespondierende, empfangende Element trägt ein E und den gleichen Identifikator, den das korrespondierende Send-Element trägt. Die Visualisierung der Kommunikation durch den gestrichelten Pfeil ist nicht notwendig und kann unterdrückt werden. Dieser Aspekt unterstreicht die graphische Intuition, daß die Grenzen der in die Parallelkartusche eingebetteten Sequenzkartuschen und damit auch der Wohlgeformtheitsbegriff nicht verletzt werden, obwohl die Kante die Grenzen der Kartuschen überschreitet.

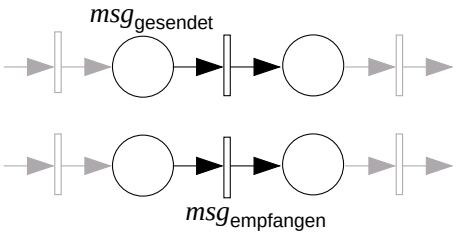


ABBILDUNG 3.3.2. Kommunikation im Petrinetz

3.3.1.2. *Informelle Semantik.* Abbildung 3.3.2 zeigt schematisch die Übersetzung in das Petrinetz. Strukturell ergeben sich keine Unterschiede zur Semantik einer einfachen Last. Lediglich die Annotationen werden anders interpretiert. Somit muß die Semantik nicht erweitert werden. Die in Abbildung 3.3.1 eingetragene gestrichelte Kante ist nicht in das Netz übertragen. Für konventionelle Lasten wurde eine Modellvorstellung angenommen in der ein Terminieren der Lastbearbeitung zwingend auf den Start folgt. In der Semantik folgte die annotierte Transition immer unmittelbar auf die Stelle, welche den Start der Last darstellte. Dieser kausale Zusammenhang wird für die Kommunikation genutzt, um einen Zusammenhang zwischen zwei nebenläufigen Bereichen herzustellen. Die Lastsimulation wirkt durch diesen technischen Kniff als Puffer. Zunächst beschränken wir uns auf die Betrachtung eines einelementigen Puffers. Durch diese Lösung ist zwar die Semantik weiterhin verwendbar, aber das Problem der Verklemmung zur Laufzeit muß behandelt werden.

3.3.2. Abläufe in Modellen mit Kommunikation. Kommunikation führt zu neuen sequentiellen Abhängigkeiten, gegebenenfalls sogar zu Zyklen, in dynamischen Modellen. Nach der hier verwendeten informellen Semantik wird ein Paar aus *Senden* und *Empfangen* durch

einen einelementigen Puffer, also eine asynchrone Kommunikation realisiert. In der Definition der Szenarien über endliche, asynchrone Automaten ist dies aber nicht leicht ausdrückbar. Es könnten mit größerem Aufwand Zwischenzustände in *Pufferteilautomaten* eingeführt werden, die sich den Zustand des Puffers *merken*. Wegen des unverhältnismäßig großen Aufwands und der Tatsache, daß wir in der Darstellung der EAAs die erreichte Anschaulichkeit verschlechtern würden, werden die Bewertung des performanzrelevanten Aspektes auf den Simulationsvorgang delegiert und nur technisch wichtig Probleme durch eine Analyse der einzelnen Szenarien behandelt.

Um die Ereignisse der unterschiedlichen Lasttypen zu unterscheiden und Gleichheit ihrer Namen feststellen zu können, gelten folgende Prädikate:

DEFINITION 55 (Lastprädikate). *Sei $w \in L(\mathcal{A})$ ein Ablauf eines Modells und $a, b \in \Sigma_{\mathcal{A}}$ Elemente, die im Ablauf vorkommen, so sind $s : \Sigma \to \mathbb{B}$ und $e : \Sigma \to \mathbb{B}$ wie folgt definiert:*

$$s(a) = \begin{cases} \texttt{true} & \text{, falls } a \text{ eine Sendeaktivität ist} \\ \texttt{false} & \text{, sonst} \end{cases}$$

$$e(a) = \begin{cases} \texttt{true} & \text{, falls } a \text{ eine Empfangsaktivität ist} \\ \texttt{false} & \text{, sonst} \end{cases}$$

und für Sende- und Empfangsaktivitäten gelte weiter, daß sie durch ein Variable ausgezeichent sind (Diese Variable kann als Bezeichner eines Nachrichtenkanals aufgefasst werden, für die es einen eigenen Namensraum $\mathcal{V}_{msg}$ gibt). Diese Variable sei ermittelbar durch eine Funktion:

$$msg : \Sigma \to \mathcal{V}_{msg}$$

und für die jeweils zusammengehörigen Sende- und Empfangselemente sei dann das Prädikat $p : \Sigma \times \Sigma \to \mathbb{B}$ definiert als:

$$p(a, b) = \begin{cases} \texttt{true} & \text{, falls } (s(a) \wedge e(b) \wedge (msg(a) = msg(b))) \\ \texttt{false} & \text{, sonst} \end{cases}$$

BEISPIEL 11 (Senden und Empfangen). *Für das bereits in Abbildung 3.3.2 eingeführte einfache Beispiel seien die Sendeaktivität im Szenario mit S^{msg} und die Empfangsaktivität mit E^{msg} bezeichnet. Damit ist $s(S^{msg}) =$ `true`, $e(S^{msg}) =$ `false`, $s(E^{msg}) =$ `false` und $e(E^{msg}) =$ `true`. Und nur für diese beiden Elemente ist das Prädikat $p(S^{msg}, E^{msg}) =$ `true`.*

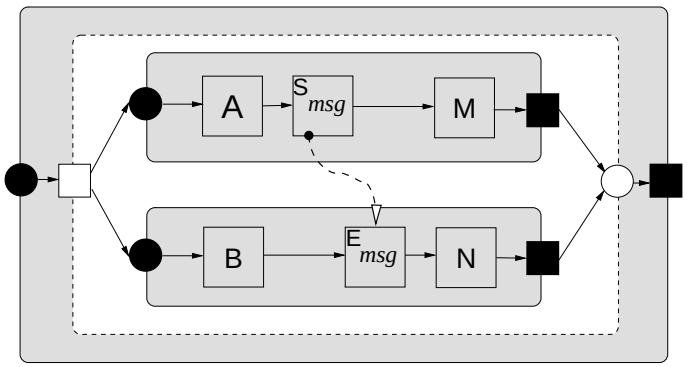


ABBILDUNG 3.3.3. Parallelkartuschen mit Kommunikation

3.3.2.1. *Verklemmung durch Kommunikation.* Die in Abschnitt 2.2 definierten, dynamischen Modelle hatten den, für den Werkzeugbau entscheidenden Vorteil, daß mindestens ein *vollständiger Pfad* durch das Modell führen mußte. Man konnte also die Erwartung haben, daß ein syntaktisch korrektes Modell nicht klemmt. Das Modell in Abbildung 3.3.4 hat offensichtlich zwei empfangende Elemente, die nicht terminieren können. Dieses Beispiel ist nicht interpretierbar, obwohl es wohlgeformt ist.

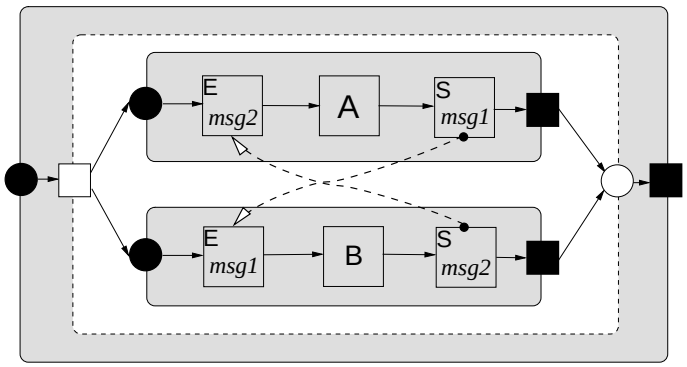


ABBILDUNG 3.3.4. Verklemmung, Beispiel 1

Modelle wie in Abbildung 3.3.5 führen nur in speziellen Fällen zu einer Verklemmung. Beispielsweise kann problemlos ein Szenario $\Delta[w_4] = \{[\mathsf{AS}^{msg}\mathsf{Mx}], [\mathsf{BE}^{msg}\mathsf{NbBE}^{msg}\mathsf{Ny}]\}$ simuliert werden. Dieses neue Problem muß also auf Ebene der Szenarien gelöst werden. Dies begründet die Entscheidung, das Problem der Verklemmung durch Kommunikation nicht schon auf Modellebene syntaktisch zu beheben.

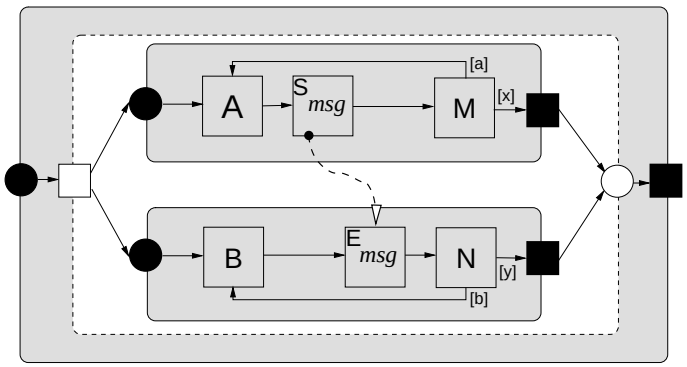


ABBILDUNG 3.3.5. Verklemmung, Beispiel 2

Sollte die Kardinalität der Schleife der oberen Kartusche in einem Szenario unter der der Unteren liegen, könnte diese nicht terminieren. Mit einem Szenario $\Delta[w_5] = \{\mathsf{AS}^{msg}\mathsf{Mx}, \mathsf{BE}^{msg}\mathsf{NbBE}^{msg}\mathsf{Ny}\}$, würde die Bearbeitung in der zweiten Wiederholung steckenbleiben. Bei diesen Arten von Verklemmung handelt es sich gewissermaßen um Laufzeitfehler. Deshalb wird eine Analyse der Szenarien Aufschluß geben müssen. Zwar ist die Identifikation von Verklemmungen nicht eine der Aufgaben der Performanzsimulation, aber eine notwendig insoweit sie dazu führen, daß die Bewertung der Modelle selbst nicht terminiert. Verklemmungsfreiheit ist eine wichtige technische Eigenschaft für die Simulation, die abschließend nur bei Betrachtung eines Paares von Modell und Szenario entschieden werden kann.

3.3.2.2. *Synchronisierte Szenarien.* In der Literatur zu Spursprachen wird Synchronisation behandelt. Hier wird das Vorkommen gleicher Terminale untersucht.

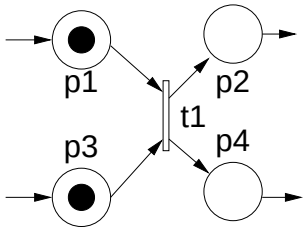


ABBILDUNG 3.3.6. Synchronisation in der Spurtheorie

Abbildung 3.3.6 zeigt ein Netzbeispiel, welches diese Art der Synchronisation informell verdeutlicht. Die hier eingeführte, gerichtete Kommunikation entspricht dem Modell des Erzeuger-Verbraucher-Problems, beziehungsweise der Anforderung von Ressourcen durch Prozesse. Das Problem der Verklemmung (*Deadlock*) wird für verteilte Algorithmen beispielsweise thematisiert in [**24**, Coffman] und [**22**, Chandy]. Deadlockerkennung basiert hier vorwiegend auf der Erkennung von Zyklen in Abhängigkeitsgraphen.

Eine Abhängigkeit eines Empfangsereignisses b, mit $e(b) = \texttt{true}$ und eines Sendeereignisses a, mit $s(a) = \texttt{true}$, tritt dann auf, wenn beide Ereignisse zusammengehören $p(a, b)$ und b. Im hier vorliegenden Fall muß zusätzlich bedacht werden, daß nur ein einelementiger Puffer zur Verfügung steht. Zur Beurteilung dieser Abhängigkeiten ist die Einführung eines Folgeoperators zweckmäßig:

DEFINITION 56 (Unmittelbare Folge). *Sei $|v|_a$ die Anzahl der Vorkommnisse eines Terminals a im Ausdruck v, $[w \in L(\mathcal{A})] \in T(\mathcal{A})$ eine Spur eines Ablaufes, und $a, b \in \Sigma_{\mathcal{A}}$ zwei beliebige Ereignisse, dann wird die* unmittelbare Folge *eines Paares aus Senden und Empfangen $a \rhd b$ definiert als:*

$$a \rhd b \Leftrightarrow p(a,b) \wedge (\exists s,t,u \in \Sigma_{\mathcal{A}}^{*}.((satbu) \wedge (|t|_a = 0) \wedge (|t|_b = 0)))$$

Jedes Szenario eines Modells ist genau dann simulierbar, wenn ein vollständiger Ablauf möglich ist. Das ist genau dann der Fall, wenn jedes Empfangsereignis unmittelbar einem zugehörigen Sendeereignis folgt.

DEFINITION 57 (Simulierbares Szenario). *Ein beliebiges Szenario $szen_{w,M}$ zum Modell M heißt* simulierbar*, wenn:*

$$\exists w \in L(\mathcal{A}(\mathcal{M}[\![M]\!])) \;.\; (szen_{w,M} = red_{\delta}([w]_M)) \;\Rightarrow \\ \forall (b \in w \mid e(b)) \;.\; \big(\exists a \in w \;\mid\; s(a) \wedge p(a,b) \wedge (a \rhd b)\big)$$

3.4. Zusammenfassung

Nach der Entwicklung dynamischer Modelle ist die Spezifikation von Testfällen, die auf die Modelle anwendbar sind, der nächste Schritt auf dem Weg zur Simulation eines zeitlichen Ablaufes. In einer Werkzeugimplementierung kann die Semantik des Kontrollflußmodells als Petrinetz unmittelbar implementiert werden. Das Auflösen der Schaltkonflikte des Netzes führte zur Auswahl einer Teilmenge an möglichen

Schaltsequenzen, deren einzelne Elemente nur noch durch performanzabhängige Kriterien ausgewählt werden können. Die Vorgabe einer solchen Auswahl, die mindestens einen Ablauf ermöglicht, der zu einem Endzustand führt spezifiziert ein simulierbares Szenario.

Im vergangenen Abschnitt wurde dieser Begriff präzise definiert und zum besseren Verständnis in die passende Theorie der Spursprachen eingeordnet. Es wurde eine Spurzerlegung vorgestellt, die eine anschaulich strukturierte Darstellung eines Szenarios erlaubt. Das Problem einer möglichen Verklemmung, die erst durch Einführung von Datenflussß und Kommunikation in Zusammenhang mit einem ungeeigneten Szenario entstehen kann, wurde diskutiert, aber nicht gelöst. Ähnlich einer Software, die neben statisch erkennbaren Syntaxfehlern auch in ungeeigneten Abläufen Laufzeitfehlern erliegen kann, müssen Verklemmungen, die sich in Szenarien durch Kommunikation ergeben, zur Laufzeit vom Werkzeug erkannt und behandelt werden.

KAPITEL 4

Umgebungen

In den letzten Kapiteln wurde eine Modellierungssprache beschrieben, mit der dynamische Modelle dargestellt werden können. Darunter ist eine Art Automat zu verstehen, der eine große Menge möglicher Abläufe aufspannt. Dann wurden Szenarien eingeführt und sprachlich eingeordnet. Szenarien sind Ausdrücke, also Einzelfallbeschreibungen, die eine Äquivalenzklasse von Abläufen bilden, die sich untereinander nur bezüglich zeitbezogener Aspekte unterscheiden lassen.

Nach der hier verwendeten Modellvorstellung entsteht ein Zeitverbrauch, erst durch den Ablauf eines Modells in einer Umgebung, auf atomarer Ebene der Bearbeitung einer einzelnen Last in dieser Umgebung. Im einfachsten Fall mag man sich unter einer Last ein Programmfragment vorstellen, als Binärdatei gespeichert, oder genauer noch, bereits als Abbild in den Arbeitsspeicher geladen und zur Ausführung bereit. Ein Computer sei ein Beispiel für eine Umgebung, in welcher diese Last bearbeitet werden kann. Ist das Programmfragment bearbeitet und bereit, aus dem Speicher entfernt zu werden, gilt es als terminiert. In diesem Kapitel wird diskutiert, wie man beschreiben kann, mit welchem Inkrement der Bearbeitungszeit dies zu der Bearbeitungszeit des gesamten Modells beiträgt. Dazu wird der Begriff des Lastmodells nur auf eine Quantität zu bewältigender Arbeit eines spezifischen Typs aufgefasst. Der Begriff der Umgebung wird als Quantität der Zeit pro bewältigter Arbeit eines spezifischen Typs aufgefasst.

Durch diese abstrakte Sicht wird keine Festlegung auf die Art von Lasten und Umgebungen getroffen und es sind spezifische Paarungen wie beispielweise: Programm/Computer, Datenübertragungsmenge/Netzwerk oder auch Arbeitslast/menschlicher Entwickler denkbar.

4.1. Lasten, Ressourcen und Umgebung

Einer Diskussion der Begriffe auf dieser abstrakten Ebene folgt die formale Definition der Modellelemente und schließlich wird mit einem einfachen Beispiel deren Anwendung erläutert.

4.1.1. Informelle Einführung. Bereits in Kapitel 2 wurde von Lastelementen gesprochen, die als eine Teilmenge der Knoten

$z \in \mathcal{Z}_{LE} \subseteq \mathcal{Z}$ einer Sequenzkartusche eingeführt wurden. Es wird als zusätzliche Struktur der Umgebung der Begriff der *Ressource* eingeführt. Eine Ressource ist in der Lage, spezifische Lasten, grundsätzlich auch mehrere nebenläufige, zu bearbeiten. Die Menge der Ressourcen, die ein dynamisches Modell benötigt, bildet die Umgebung.

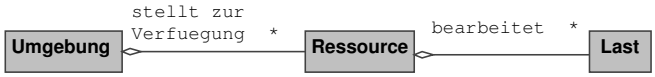


ABBILDUNG 4.1.1. Umgebung und Lasten

Umgekehrt ausgedrückt ist die Umgebung eine Menge an Ressourcen, und eine Ressource weist jeweils eine Menge an zugeordneten Lasten auf. Eine im dynamischen Modell spezifizierte Last steht für eine Lastinstanz, die einer Ressource zugeordnet wird. Eine Last ist also eindeutig in der Umgebung lokalisierbar. Während eines Ablaufes ist eine Last entweder in Bearbeitung (*aktiv*) oder nicht. Jede Last kann während eines Ablaufes mehrfach aktiviert sein oder auch nie. Jede Ressource hat ein typisches Laufzeitverhalten. Durch dieses unterschiedliche Verhalten kann die Feinstruktur einer Umgebung modelliert und durch die Lastlokalisation eine Lastverteilung (*Deployment*) und ansatzweise auch eine Hardware-Architektur modelliert werden.

Bei den Lasten selbst wurde bislang nur von deren Terminierung gesprochen. In Abschnitt 2.3.1.2 wurde mit Abbildung 2.3.4 das vollständige Netz eingeführt. Zwei Transition waren hier für jedes Lastelement vorgesehen: die erste beim Aktivieren der Last, die zweite beim Terminieren. Es wurde informell angegeben, daß jede Last unmittelbar dann aktiviert wird, wenn sie aktiviert werden kann, dh. eine Aktivierungstransition konzessioniert wird. Es kann weiterhin mit dieser Vereinfachung gearbeitet werden, wenn man im Gedächtnis behält, daß jede Implementation eines Werkzeugs bei der entsprechenden Markierung den Zustand der spezifizierten Last auf aktiv setzt, dh. initialisiert. Am zerlegten Ablaufausdruck heißt das: die Aktivierung erfolgt unmittelbar nach dem Lesen des letzten Terminals, bevor das entsprechende Lastterminal zum Lesen ansteht.

Nachfolgend werden die diskutierten Elemente formal eingeführt und dann am Beispiel deren Anwendung bei der Modellierung erläutert.

4.1.2. Formale Ressourcen- und Last-Modelle.

4.1.2.1. *Lasten.* Die Menge aller Lasten $\mathcal{L}$ ergibt sich aus der Vereinigung aller Lastelemente aller Sequenzkartuschen.

DEFINITION 58 (Gesamte Lastmenge). *Sei* $M = \left(\Upsilon, \Theta, \mathcal{R}, \rho, e_M\right)$ *ein Modell,* $\theta = \left(\mathcal{Z}, \Sigma, \delta, z_{start}, z_{end}\right) \in \Theta$ *eine Sequenzkartusche des Modells und* $\mathcal{Z}_{LE} \subseteq \mathcal{Z}$ *die Menge der Lastelemente dieser Kartusche, dann ist die gesamte Menge aller Lasten gegeben mit:*

$$\mathcal{L} = \bigcup_{\theta \in \Theta} \{\mathcal{Z}_{LE}(\theta)\}.$$

Bei der Erstellung des Modells muß der Entwickler spezifizieren, von welcher Ressource eine Last bearbeitet werden soll.

DEFINITION 59 (Lastlokalisation). *Es seien* $\mathcal{L}$ *eine Menge von Lasten, und* $\mathcal{R}$ *eine Menge von Ressourcen. Dann wird die ausführende Ressource einer Last bestimmt durch:*

$$loc_{\mathcal{L}} : \mathcal{L} \to \mathcal{R}$$

Technische Probleme, die entstehen, weil ein Entwickler in einem Modell eine unbekannte Ressource spezifiziert, werden hier nicht behandelt.

4.1.2.2. *Ressourcen und Umgebung.* Wir stellen uns vor, eine Umgebung $u \in \mathcal{U}$ besteht aus einer Menge an Ressourcen $\mathcal{R}$, denen jeweils eine Menge an Lasten zugeordnet sind. Der Einführung einer Menge von Ressourcen bewirkt, daß eine Umgebung in jeder Ressource ein unterschiedliches Verhalten besitzen kann. Dies wird durch unterschiedliche Funktionen modelliert. Weiterhin betrachten wir die Menge der Lasten $\mathcal{L}$ und den jeweiligen Bearbeitungszustand einer Ressource $\sigma \in \Sigma_T$ bezüglich der dort lokalisierten Lasten.

DEFINITION 60 (Umgebung).

$$\begin{aligned} \textit{Umgebung: } & u = \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle : \mathcal{U} \\ \textit{Lastlokalisation: } & loc_{\mathcal{U}} : \mathcal{L} \to \mathcal{R} \\ \textit{Ressourcen: } & res_{\mathcal{U}} : \mathcal{R} \to \left\langle \begin{array}{c} (\mathcal{L} \times \Sigma_T) \to \mathbb{R}_0^+, \\ (\Sigma_T \times \mathbb{R}_0^+) \to \Sigma_T \end{array} \right\rangle \end{aligned}$$

Dabei reicht die Umgebung die Lokalisation der Last weiter und ermittelt $loc_{\mathcal{U}}(l \in \mathcal{L}) = loc_{\mathcal{L}}(l)$.

Eine Ressource wird durch die Funktionen modelliert, die das zeitliche Verhalten der Umgebung bezüglich dieser Ressource spezifiziert. So können lokal (innerhalb der Ressource) nur unter Betrachtung des lokalen Zustandes Aussagen gemacht werden. Unter der Annahme, daß

sich dieser Zustand nicht ändert, wird nun eine Funktion (Bearbeitungszeitfunktion) verlangt, welche die Zeit berechnet, die für jede Last verstreichen muß, bis diese vollständig bearbeitet ist[1].

Weiter ist eine Funktion nötig, die einen Folgezustand einer Ressource aus einer zur Verfügung gestellten Zeit errechnet. Die Anwendung dieser sogenannten Einzelschrittfunktion f_s ist natürlich nur auf einen Zeitwert sinnvoll, der kleiner oder gleich dem kleinsten Zeitwert ist, der für die vollständige Bearbeitung einer auf dieser Ressource lokalisierten Last benötigt wird. Dieser Zeitwert muß durch eine Bearbeitungszeitfunktion f_t bestimmt werden können. Diese beiden Funktionen modellieren eine Ressource $r \in \mathcal{R}$ ausreichend und werden durch die Funktion $res_{\mathcal{U}}$ bestimmt.

DEFINITION 61 (Performanzfunktionen).

$$\textit{Bearbeitungszeit: } f_t(res_{\mathcal{U}}, r) = \pi_0\big(res_{\mathcal{U}}(r)\big)$$
$$\textit{Einzelschrittfunktion: } f_s(res_{\mathcal{U}}, r) = \pi_1\big(res_{\mathcal{U}}(r)\big)$$

Es bestehen kaum Einschränkungen, wie die Funktionen f_s und f_t gestaltet werden. Ergibt die Zeitfunktion f_t für eine Last einen Wert t, so fordern wir lediglich, daß der Wert von f_t für dieselbe Last 0 sei, im mit t erzeugten Folgezustand s_{next}. Wir stellen uns also vor, daß diese Last im Folgezustand als vollständig bearbeitet gilt.

AXIOM 10 (Korrelation von Schritt- und Zeitfunktion).

$$\forall s \in \Sigma_T, l \in \mathcal{L}, r \in \mathcal{R} \;.\; loc_{\mathcal{U}}(l) = r, \forall t \geq f_t(res_{\mathcal{U}}, r)\big(\; l, s \;\big) \Rightarrow f_t(res_{\mathcal{U}}, r)\big(\; l, f_s(res_{\mathcal{U}}, r)\big(\; s, t \;\big)\;\big) \;=\; 0$$

Außerdem wird festgelegt, daß sich der Zustand der Umgebung nicht ändert, wenn keine Zeit vergeht.

AXIOM 11 (Zeitliche Bestimmtheit).

$$\forall s', s'' \in \Sigma_T, (l \in \mathcal{L}, r \in \mathcal{R} \;.\; loc_{\mathcal{U}}(l) = r) \;.\; s'' = f_s(res_{\mathcal{U}}, r)\big(\; s', 0 \;\big) \Rightarrow s' = s''$$

Mit diesen Performanzfunktionen läßt sich eine Funktion δ definieren, welche für eine Last l die Zeit ermittelt, die in einem gegebenen Zustand s und einer Umgebung u, für die vollständige Bearbeitung dieser Last (bei unverändertem Zustand s) vergehen würde.

[1] Man kann nur erwarten, daß der Wert für die kürzeste Last sinnvoll ist, da jede vorher terminierende Last notwendig den Zustand der Ressource ändern muß und damit die Annahme verletzen würde.

DEFINITION 62 (Bearbeitungszeit).

$$\delta : \ \mathcal{L} \times \Sigma_T \times \mathcal{U} \to \mathbb{R}_0^+$$
$$\delta(l, s, \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle) \ = \ f_t(res_{\mathcal{U}}, loc_{\mathcal{U}}(l)) \ \left(\ l, s \ \right)$$

Diese Funktion betrachtet den lokalen Zustand. Da aber der Gesamtzustand s_{gesamt} nur für eine Bearbeitungszeit, nämlich die kürzeste der gesamten Umgebung, unverändert bleibt, ist für uns ausschließlich das Minimum dieser Funktionen über die Menge der aktiven Lasten, eine Teilmenge $l_{aktiv} \subseteq \mathcal{L}$ der gesamten Umgebung, interessant.

DEFINITION 63 (Kleinste Bearbeitungszeit).

$$\delta_{min} : \ \mathfrak{P}(\mathcal{L}) \times \Sigma_T \times \mathcal{U} \to \mathbb{R}_0^+$$
$$\delta_{min}(\mathcal{L}, s, u) \ = \ min \ \left\{ \delta(l, s, u) \ . \ l \in \mathcal{L} \right\}$$

Diese Funktionen können zur Bewertung von Abläufen herangezogen werden.

4.2. Anwendung

Die Anwendung der hier eingeführten Beschreibung einer Ressource durch eine Zeitfunktion und eine Zustandsübergangsfunktion bieten Spielraum zur Implementierung von Ressourcenmodellen. Eine Absicht dieser Arbeit ist es, keine Vorgaben für die Realisierung dieser Funktionen zu machen. Wenn nun in Folge einige Möglichkeiten skizziert werden, sollen diese nur Denkanstöße darstellen.

4.2.1. Eindimensionale Ressource. Eine sehr einfache Ressource soll nur die Anzahl an ausgeführten Operationen pro Zeiteinheit modellieren, also nur Lasten aus einem Skalar bearbeiten. Abbildung 4.2.1 zeigt die Bearbeitung zweier Lasten auf dieser Ressource. In diesem einfachen Beispiel ist die Übersetzung der UML auf dynamische Modelle vorweggenommen.

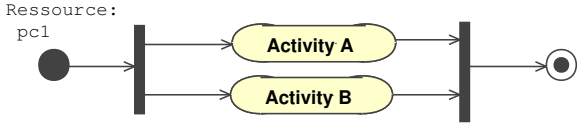


ABBILDUNG 4.2.1. UML Beispiel

Diese Arbeit macht keine Aussage über die Struktur von Lastmodellen, Ressourcenmodellen und der sie realisierenden Funktionen. Folgend sollen einfache Beispiele einzelne Möglichkeiten zeigen. Alle dafür

folgend gemachten Definitionen sind aber als willkürliche Festlegungen eines Modellierer, also als Teil der Anwendung dieser Methode zu betrachten.

4.2.1.1. *Lastmodell.* Eine Last ist beschrieben durch eine Quantität, anhand der mit einer bekannten Ressource die Bearbeitungszeit ermittelt werden kann.

Sei $j \in \mathcal{L}$ eine beliebige Last, so werde ihr *Initialwert* ermittelt durch eine Funktion:

$$q_{init} : \mathcal{L} \to \mathbb{R}^n_{0,+}$$

Das einfachste Lastmodell modelliert ein feste Größe, welche die durch die Ressource zu bewältigende Arbeit quantifiziert. Die Last wird dargestellt durch einen Skalar $l_N \in \mathbb{R}^1$. Sei für unser Beispiel $q_{init}(l_A) = 15$ und $q_{init}(l_B) = 33$ eine Modellierung der Lasten A und B.

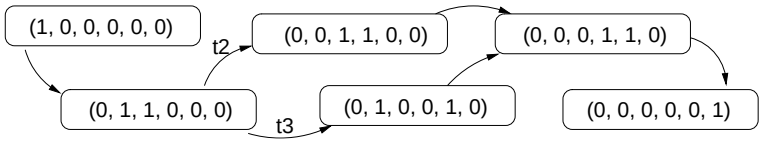


ABBILDUNG 4.2.2. Erreichbarkeitsgraph des Beispiels

Für dieses Beispiel existiert nur ein Szenario, aber zwei Ausdrücke. Der Weg über t_2, der das frühere Terminieren von A und der andere von B modelliert. Die beiden Initialwerte 15 und 33 werden in der Simulation, die hier auch vorweggenommen wird, schrittweise verringert.

4.2.1.2. *Einfache Ressource.* Bezeichnen wir folgend `Activity A` mit l_A und `Activity B` mit l_B. Der Zustand der Ressource ist damit in diesem Beispiel die Menge der aktuellen Werte der jeweiligen Lasten. Beschreibt man den Bearbeitungszustand einer einzelnen Last mit einer Funktion $s : \mathcal{L} \to \mathbb{R}^n_{0,+}$ mit $n = 1$ für dieses Beispiel, so kann man sich den Zustand vorstellen als Menge $s = \{s(l_A) \to x_A, s(l_B) \to x_B\}$.

Die Ressource `pc1` sei bezeichnet mit $r_{\texttt{pc1}}$. Die Umgebung $u \in \mathcal{U}$ für dieses Beispiel ist $u = \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle$, mit $loc_{\mathcal{U}}(l) = r_{\texttt{pc1}}$, $falls$ $(l = l_A) \vee (l = l_B)$ und $res_{\mathcal{U}}(r_{\texttt{pc1}}) = \langle f_t, f_s \rangle$.

Die Funktionen f_t und f_s für die Ressource $r_{\texttt{pc1}}$ müssen nun festgelegt werden. f_t modelliert die lastabhängige Antwortzeit. f_s modelliert das charakteristische Verhalten bei der nebenläufigen Bearbeitung.

Die Berechnung der Antwortzeit soll hier nach einer einfachen Regressionsgerade $y = ax + b$ erfolgen. Da wir von keiner Latenzzeit ausgehen, ist der Nullpunkt im Ursprung: $b = 0$. Die Einheit der Initialwerte legen wir für dieses Beispiel willkürlich mit *Operation* fest,

und die Zeiteinheit wird als *Tick* festgelegt. Durch Schätzung oder Messung wird nun die Funktion festgelegt. Es benötigen beispielsweise *eine Operation* eine Zeit von 20 *Ticks*. Schätzungen oder Messungen mögen beispielsweise zeigen, daß die Bearbeitungszeit durch die Anzahl der nebenläufigen Prozesse vervielfacht wird (d.h. 2 nebenläufige Prozesse bedeuten doppelte Anzahl an Ticks für die gleiche Last). Berechnen wir die Anzahl nebenläufiger Prozesse aus dem Zustand mit $n = |s|$, kann, mit der aktuellen Restlast $s(l)$, die Funktion f_t der Ressource definiert werden als:

$$f_t(l, s) = 20 * (|s|) * s(l).$$

Wäre l_A die einzige Last auf der Ressource, würde die Zeit bis zur vollständigen Bearbeitung, mit $s_0(l_A) = q_{init}(l_A)$, berechnet werden können zu:

$$f_t(l_A, s_0) = 20 * 1 * 15 = 300 \text{ Ticks}.$$

Nach unserem Beispiel ist die Last l_B auf der selben Ressource untergebracht. Daraus ergibt sich die Berechnung eines anderen Wertes. Die Anzahl aktiver Lasten ist somit $n = 2$. Ungeachtet des Einflußes der zweiten Last würde die Bearbeitungszeit für l_A berechnet werden nach:

$$f_t(l_A, s_0) = 20 * 2 * 15 = 600 \text{ Ticks}.$$

Die vollständige Bearbeitungszeit für l_B berechnete sich aber zu:

$$f_t(l_B, s_0) = 20 * 2 * 33 = 1320 \text{ Ticks}.$$

Der Zustand der Ressource ändert sich aber in dieser Zeit, da die Last A vollständig bearbeitet ist, bevor B terminiert ($f_t(l_A, s_0) < f_t(l_B, s_0)$). Dieser Zwischenzustand, unmittelbar nachdem A terminiert, muß nun berechnet werden. Dazu wird die Schrittfunktion f_s benötigt.

Die Ressource kann nur für die Periode $t' = f_t(l_A, s_0)$ in diesem Zustand berechnet werden. Danach ist das System im Folgezustand s_1. Der Anteil der bis dahin noch unbearbeiteten Quantität der Last B entspricht:

$$s_1(l_B) = (1 - \frac{t'}{f_t(l_B, s_0)}) \cdot s_0(l_B).$$

Mit dieser Gleichung kann offensichtlich, als Implementierung für f_s, der Restlast im Folgezustand für jede Last berechnet werden, deren Bearbeitung im Zeitintervall nicht abgeschlossen werden kann. Für Last B ist damit die noch verbleibende Anzahl an *Operationen*:

$$s_1(l_B) = (1 - \frac{600}{1320}) \cdot 33 = 18.$$

Weiterhin sieht man, daß das Axiom 10 bei der Bearbeitung von l_A nicht verletzt wird, da f_s sich berechnet zu:

$$s_1(l_A) = (1 - \frac{600}{600}) \cdot 15 = 0.$$

Mit einem Initialzustand, der sich unter der Verwendung der q_{init}-Definition ergibt zu:

$$s_0 = \langle s(l_A) = q_{init}(l_A), s(l_B) = q_{init}(l_B) \rangle,$$

kann die Schrittfunktion formal definiert werden als:

$$f_s(s,t) = \langle\ s(l)'\ .\ s(l)' = (1 - \frac{t}{f_t(l,s)}) \cdot s(l)\ ,\ \mathit{falls}\ loc_{\mathcal{U}}(l) = r_{\mathtt{pc1}} \rangle$$

In Vorwegnahme der Auswertungssemantik, die im nächsten Kapitel eingeführt wird, vermutet man richtig, daß die gesamte Bearbeitungszeit sich errechnet aus t' und der Zeit, die im Folgezustand noch für die vollständige Berechnung der Restlast von B in s_1 benötigt wird und sich damit berechnet zu:

$$t_{ges} = f_t(l_A, s_0) + f_t(l_B, s_1)$$

Mit $t'' = f_t(l_B, s_1) = 20 * 1 * 18 = 360\ Ticks$ ergibt sich t_{ges} zu $t' + t'' = 960\ Ticks$. Dieses triviale Beispiel sollte die Idee der Zeit- und Schrittfunktion verdeutlichen. Es wird nochmals darauf hingewiesen, daß viele anderen möglichen Realisierungen für Ressourcen und Lasten denkbar sind und die Technik nicht auf das gewählte Beispiel beschränkt ist.

Der zeitlichen Ablaufbewertung liegt ein intuitives Verständnis einer nebenläufigen Bearbeitung von Lasten zugrunde. Mit einer Auswertungssemantik, die im nächsten Kapitel vorgestellt wird, wird dieses Modellverständnis definiert und gezeigt, daß die gewählte Definition wiederholbare und eindeutige Ergebnisse erreicht.

4.2.2. Mehrdimensionale Lasten. Nach diesem Schema können auch komplexere Last- und Ressourcenmodelle entwickelt werden. Ein für die zeitliche Beurteilung wichtige Aspekt kann in die Last- und Ressourcenmodelle als eine weitere Dimension eingearbeitet werden. Dabei muß eine Funktion nicht unbedingt als Funktion über den ganzen Definitionsbereich konstruiert werden. Eine weitere Möglichkeit wäre es eine Tabelle mit Meßwerten aus Performanzmessungen aufzubauen, die

als Stützstellen für eine Interpolationsfunktion dienen. Die Antwortzeit kann durch die Interpolationsfunktion bestimmt werden, sofern der gesuchte Wert in einem Intervall zwischen den Stützstellen liegt.

Zur Durchführung einer Performanzmessung kann beispielsweise für jede Attributskombination ein Code-Beispiel implementiert und auf einer Ressource ausgeführt werden. Bei großen Abweichungen der Meßwerte können diese wiederholt ausgeführt und dann der Mittelwert berechnet werden. Das Mitführen statistischer Parameter wie etwa einer Standardabweichung wäre hier sinnvoll. Dies würde aber den Rahmen dieser Arbeit sprengen und muß deshalb auf zukünftige Erweiterungen delegiert werden.

Tabelle 1 zeigte ein Beispiel der Ergebnisse einer solchen Messung. Die Bedeutung des abstrakten Begriffs *Anzahl-Ops* und *Code-Größe* wird festgelegt durch Zuordnung des Code-Beispiels. Zur Durchführung einer solchen Messung ist eine geschickte Versuchsanordnung und zusätzliches Wissen über die wirklichen Ressourcen nötig, das hier nicht behandelt werden kann.

Anzahl-Ops n	**Code-Größe kB** g	**Meßwert: s**
10	1	0.002
100	1	0.019
500	1	0.107
10	50	0.006
100	50	0.065
500	50	0.234
10	250	0.024
100	250	0.312
500	250	1.287

TABELLE 1. Meßwerttabelle

Fehler bei der Bemaßung setzen sich natürlich in die Ergebnisse der Simulation fort. Solche Tabellen können mit Schätzwerten gefüllt werden, mit vorab entwickeltem Beispielcode oder vorhandenen authentischen Code-Fragmenten. Im zweiten Teil folgen grundsätzliche Überlegungen zur Fortentwicklung der Modelle im Verlauf eines iterativen Prozesses.

Im hier angedeuteten Beispiel können Simulationen für Lasten durchgeführt werden, deren Werte für den Initialvektor in den Intervallen $10 \leq n \leq 500$ und $1 \leq g \leq 250$ liegen. Mit einer lineare Interpolation würde f_t eine allein in Bearbeitung befindliche Last mit

einem Initialwert von $s(l) = \langle 250, 50 \rangle$ zu $f_t(l, s) = 0.1495$ ausgewertet werden.

Um die Abhängigkeit der Performanz-Eigenschaften von der Anzahl der nebenläufigen Prozesse zu bestimmen, sind weitere Meßreihen notwendig. Wählt man diesen Weg der Bestimmung der Zeitfunktion, muß der Kontext der zur Messung verwendeten Ressource standardisiert und genau dokumentiert werden. Eine Verwendung eines so kalibrierten Modells zur Beurteilung von Szenarien, die später in ganz anders konfigurierter Umgebung laufen würden, wäre sonst wertlos. Die hier vorgestellte Methode bietet die Möglichkeit, ein Modell als Referenz zu bestimmen und an diesem Urmaß das Verhalten von Modellen zu untersuchen. Wie bei jeder in der Realität eingesetzten Meßtechnik sind hier Geschick und Erfahrung für den Erfolg entscheidend.

4.3. Zusammenfassung

In diesem Abschnitt wurden die abstrakten Funktionen eingeführt, die eine Ressource aufweisen muß, um in einer Simulation verwendet werden zu können. Die Funktionen dienen zur quantitativen zeitlichen Bewertung des Bearbeitungszustandes einzelner Lasten und zur Fortschreibung der Folge an Zuständen der gesamten Umgebung längs des in der Simulation ermittelten Ablaufes. Diese Funktionen wurden aber auch als die Hilfsmittel eingeführt, die zur Modellierung einer Umgebung verfügbar sind.

An einem einfachen Beispiel wurden die Ideen für diese Funktionen verdeutlicht und Probleme und Möglichkeiten bei der Erstellung und Kalibrierung von Modellen diskutiert. Die Berechnung der Bearbeitungszeit des einfachen Beispiels wurde hier intuitiv beschrieben und von Hand durchgeführt. Im folgenden Kapitel wird eine Auswertungssemantik für Modelle, Szenarien und Umgebungen vorgestellt, welche den Bewertungsvorgang verallgemeinert und definiert.

KAPITEL 5

Simulation

5.1. Grundlegende Begriffe

In Kapitel 2 wurden dynamische Modelle definiert. In Kapitel 3 wurden Grundlagen zur Spezifikation von Szenarien als Äquivalenzklassen von Abläufen, die sich nur in zeitlichen Aspekten unterscheiden, ausgearbeitet. In Kapitel 4 werden auf hoher Abstraktionsebene die Anforderungen an die Beschreibung einer Umgebung definiert, die zeitliche Bewertung von Einzelschritten zwischen Zustandsänderungen erlauben. Mit einer adäquaten Umgebung kann aus einer Äquivalenzklasse, die durch ein Modell und ein Szenario definiert wurde, genau ein Lauf ausgewählt werden. Dieser Lauf soll mit einer Bearbeitungszeit bewertet werden.

In diesem Kapitel wird eine Auswertungssemantik vorgestellt, die zu einem Tripel aus Modell, Szenario und Umgebung, einen Pfad im Erreichbarkeitsgraphen des Modells auswählt, der Anfangs- und Endzustand verbindet. Diese semantische Funktion terminiert mit einer Bewertung des gewählten Pfads. So schätzt diese Funktion den Zeitaufwand zur Bearbeitung dieses spezifischen Ablaufs.

Lasten benötigen Zeit, um bearbeitet zu werden. Ihr Terminieren war deshalb an eine Zeitschrittfunktion gekoppelt. Im Gegensatz dazu werden spontane Übergänge, durch Szenarien determinierte Übergänge, Senden und Empfangen von Nachrichten und das Starten der Lasten als *zeitlos* betrachtet. Informell gesprochen erwartet man, daß alle Lasten, die gestartet werden können, auch gestartet werden. Erst wenn der danach ermittelte Gesamtzustand der Umgebung stabil ist, soll die Bewertung des nächsten Zeitschritts erfolgen. Aus dieser Vorstellung resultiert eine weitere spezifische Auswahl von Pfaden aus der szenario-determinierten Äquivalenzklasse. Es wurde bereits darauf hingewiesen, daß diese Auswahl außerdem die Regularität der Äquivalenzklasse zerstört. Folgende formale Semantik folgt dieser Vorstellung. Die Funktion wählt schrittweise Übergänge des Erreichbarkeitsgraphen aus und inkrementiert gegebenenfalls einen Zeitwert, der nach Erreichen des Endzustands zurückgegeben wird.

5.2. Simulationssteuerung

Die Simulation besteht in der wiederholten Fortsetzung der Ermittlung eines Folgezustandes (d.h. eines Folgeknotens), der Fortschreibung des Zustandes und dem Inkrement des Zeitverbrauchswertes.

5.2.1. Hilfsfunktionen. Zur Definition einer formalen Auswertungssemantik werden einige Hilfsfunktionen benötigt.

5.2.1.1. *Struktur der Lastmenge.* Die Semantik soll einen Ablauf des Modells M identifizieren und bewerten. Folgend wird mit dem Erreichbarkeitsgraphen $G(\mathcal{M}[\![M]\!]) = (V, K)$ und seiner Menge an Knoten V und Kanten E argumentiert. Dabei sei $m_V : V \to (S \to \mathbb{N}_0^+)$ verwendet um die Markierung für einen Knoten des Erreichbarkeitsgraphen zu darzustellen. Eine Umgebung $u \in \mathcal{U}$ wird es in der Simulation ermöglichen die Last zu identifizieren, die im aktuellen Zustand eines Modells als nächstes terminiert.

Definition 64 (Aktive und terminierende Lasten). *Sei M ein dynamisches Modell, $\mathcal{M}[\![M]\!] = N = (S, T, F)$ die Netzsemantik, $G(N) = (V, E)$ der zugehörige Erreichbarkeitsgraph, dann sind die Teilmenge der aktiven Lasten mit*

$$l_{act} : V \to \mathfrak{P}(\mathcal{Z}_{LE})$$
$$l_{act}(v) = \{z \in \mathcal{Z}_{LE},\ \exists s \in S\ .\ z = elem(s) \wedge m_V(v)\big(s\big) = 1\}$$

und die Teilmenge der in den möglichen Folgezuständen terminierten Lasten mit

$$l_{term} : E \to \mathfrak{P}(\mathcal{Z}_{LE})$$
$$l_{term}(v_i \to v_j) = l_{act}(v_i) \setminus l_{act}(v_j)$$

gegeben.

Jedes Modell spezifiziert eine endliche Menge an Lasten, die, abhängig vom Szenario, im aktuellen Zustand bearbeitet werden oder nicht. Eine einzelne Last mag in einem Szenario beliebig oft, möglicherweise auch nie bearbeitet werden.

Bemüht man wieder das bekannte Beispiel in Abbildung 5.2.1 und nimmt an, daß die Bearbeitung beider Lasten M und N gerade bekommen hat, so wäre $l_{act} = \{\mathsf{M},\mathsf{N}\}$. Beide Lasten wären aktiv, was an der Markierung des Petrinetzes, also am Erreichbarkeitsgraphen erkennbar ist (ohne Darstellung). Wählt man nun eine Kante und entfernt alle Elemente, die im Folgezustand immer noch aktiv sind, bleiben diejenigen übrig, die bei diesem Schritt terminieren. Sei N die Last, deren Bearbeitung weniger Aufwand erfordert, so wäre $l_{term} = \{\mathsf{N}\}$.

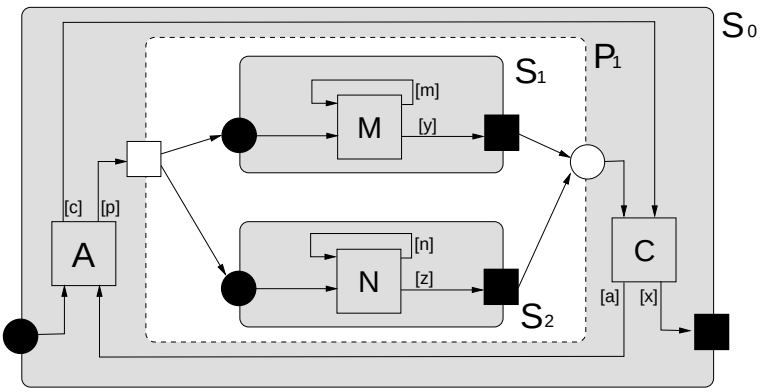


ABBILDUNG 5.2.1. Beispiel

5.2.1.2. *Struktur der Kantenmenge.* Wie informell bereits eingeführt, unterscheiden sich die Kanten, die vom aktuellen Zustand ausgehen. Zeitlos interpretiert sind Übergänge sind einerseits spontane und andererseits durch Szenarien bewachte Übergänge. Zeitbehaftet sind solche, die das Terminieren einer Last modellieren.

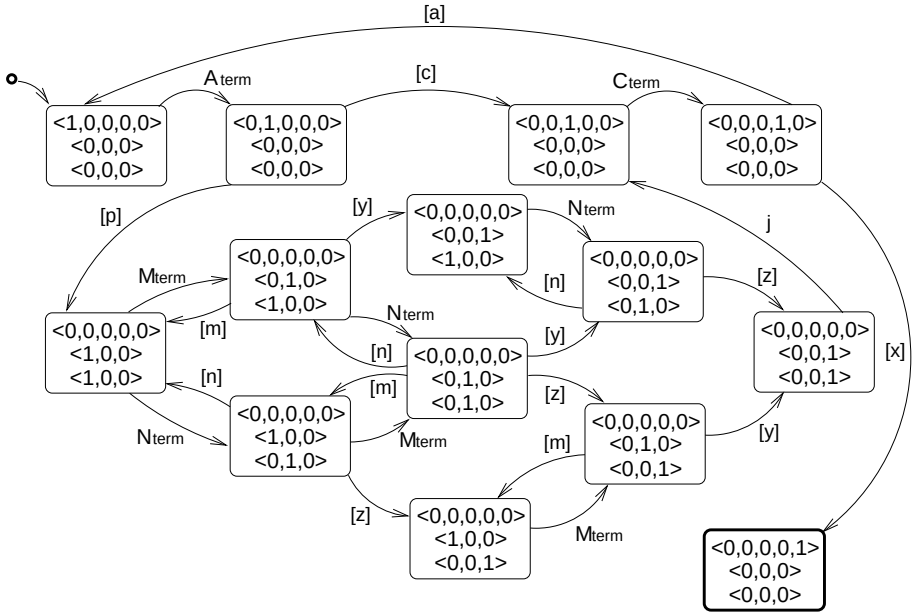


ABBILDUNG 5.2.2. Erreichbarkeitsgraph

Am Erreichbarkeitsgraphen, wie am Beispiel in Abbildung 5.2.2, die Kanten, die eine Annotation in eckigen Klammern tragen vom Szenario bewacht. Diejenigen, welche Lastnamen *Name* mit dem Index

$Name_{term}$ tragen, modellieren das Terminieren einer Last zu diesem Übergang und diejenigen, welche keine Annotationen besitzen, werden als zeitlos betrachtet erfolgen spontan. (Im Beispiel ist dies nur eine Kante, die die Rückkehr in die S_0-Sequenzkartusche modelliert.)[1]

DEFINITION 65 (Kantengruppen). *Sei M ein dynamisches Modell, $N = (S, T, F) = \mathcal{M}[\![M]\!]$ die Netzsemantik, $G(N) = (V, E)$ deren Erreichbarkeitsgraph, mit der Menge der Knoten V und der Menge der Kanten E. Dabei wird eine Kante $e_{i,j} \in E$, welche die Knoten $v_i, v_j \in V$ verbindet auch notiert als (v_i, v_j). Sei weiterhin $in = \langle u_1, \ldots, u_n \rangle \in \mathcal{I}_{Szenario}$ ein Szenario mit seiner Ablaufsteuerung η_{in} und $s \in \Sigma_T$ der aktuelle Lastzustand. Dann läßt sich die Menge $E(v) \subseteq E$ der vom Knoten $v \in V$ ausgehenden Kanten des Erreichbarkeitsgraphen wie folgt definieren und in Kantengruppen zerlegen:*

$$\begin{aligned} E(v) = &\ \{(v_i \to v_j) \in E \ . \ (v = v_i) \wedge \eta_{in}(t(v_i \to v_j)) \ \} \\ E_{Last}(v) = &\ \{(v_i \to v_j) \in E(v) \ . \ (elem(\alpha(t(e_{i,j}))) \in \mathcal{Z}_{LE}) \\ &\ \wedge(\delta_{min}(l_{Term}(v_i \to v_j), s, u) > 0) \ \} \\ E_{Spontan}(v) = &\ \{(v_i \to v_j) \in E(v)\} \setminus (E_{Last}(v) \end{aligned}$$

Die Einführung dieser Bezeichnungen erleichtert im Folgenden die Definition nötiger Funktionen.

Im aktuellen Zustand ist jeweils eine Teilmenge an Transitionen konzessioniert. Aus dieser Teilmenge spezifiziert das Szenario mittels seiner Ablaufsteuerung die Teilmenge der Transitionen, die feuern dürfen.

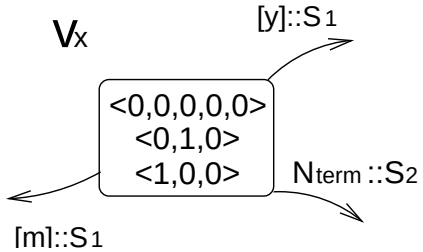


ABBILDUNG 5.2.3. Ein Zustand im Erreichbarkeitsgraphen

Betrachtet man am Beispiel in Abbildung 5.2.3 die ausgehenden Kanten des aktuellen Zustands, können nicht beide durch das Szenario bewachten Transitionen geschaltet werden, da sie zur selben Sequenzkartusche gehören und durch das Szenario eine Auswahl getroffen werden muß - hier y oder m. Da auch Kanten verschiedener Kartuschen

[1] Zustände, die von spontanen Transitionen verlassen werden, werden übrigens nie von Transitionen der anderen Arten verlassen. Dies folgt aus der Forderung, daß Transitionen im Konflikt mit unterscheidbaren Wächtern zu versehen sind.

existieren können, handelt es sich bei den auswählbaren Kanten um eine Menge.

In Abbildung 5.2.3 ergäbe sich $\eta(v_k, red_\delta(w)) = \{\mathsf{y}\}$, falls man wieder das Beispielszenario $w = \mathsf{ApNnNzMyjCx}$, mit $red_\delta(w) = \mathsf{pnzyx}$[2] ausgeht.

5.2.2. Schrittfunktionen. Vom Startknoten $v_0 \in V$ im Erreichbarkeitsgraphen $G(\mathcal{M}[\![M]\!])$ eines Modells M aus wird solange ein jeweils nächster Knoten ausgewählt, bis der Endknoten $v_{Stop} \in V$ erreicht ist. In der Markierung, die dieser Knoten modelliert, ist der Endzustand erreicht. Auf dem Weg werden bei jedem Übergang Funktionen benötigt, die den richtigen Übergang auswählen, einen Zeitwert für diesen Übergang ermitteln und den Zustand der Umgebung aktualisieren.

5.2.2.1. *Kleinster Zeitschritt.* Die willentliche Festlegung der spontanen und szenario-determinierten Übergänge als zeitlos führt folgende Funktion mit der durch δ_{min} berechenbaren, kleinsten Bearbeitungszeit aus der Teilmenge an Lasten aus, die im nächstmöglichen Zustand terminieren können.

Definition 66 (Kleinster Zeitschritt). *ES sei M ein dynamisches Modell, $N = (S, T, F) = \mathcal{M}[\![M]\!]$ die Netzsemantik, $G(N) = (V, E)$ deren Erreichbarkeitsgraph. Dann ist der kleinste mögliche Zeitschritt aus einem Knoten v gegeben durch*

$$\tau : \ V \to \mathbb{R}_0^+$$

mit

$$\tau(v) = \begin{cases} \delta_{min}\Big(\bigcup\limits_{e \in E_{Last}(v)} l_{term}(e) \Big), \\ \quad \textit{falls } E_{Spontan}(v) = \emptyset \\ \\ 0, \ \textit{sonst.} \end{cases}$$

Beispiel 12 (Kleinster Zeitschritt bei paralleler Lastbearbeitung). *Das Beispiel einfacher Nebenläufigkeit aus Abschnitt 4.2.1.2 ist in Abbildung 5.2.4 diesmal in Kartuschendarstellung abgebildet. Aus dem Modell ergibt sich der Erreichbarkeitsgraph in Abbildung 5.2.5. Es wer-*

[2] In diesem Szenario kommt ausschließlich die Annotation y vor. Somit ist die Auswahl trivial. Sollten alle Annotationen vorkommen wäre diejenige Annotation gewählt auf die der Cursor des entsprechenden Ausdrucks der zerlegten Spur zeigt, also diejenige, der als nächstes gelesen würde.

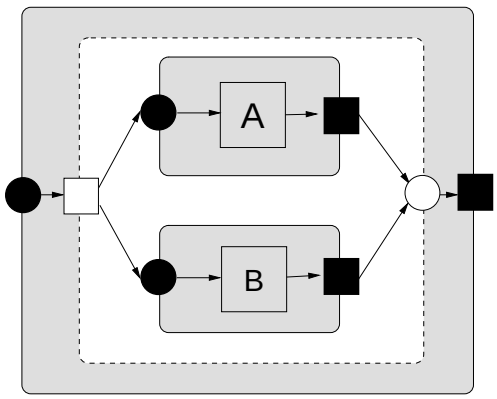


ABBILDUNG 5.2.4. Einfache Nebenläufigkeit

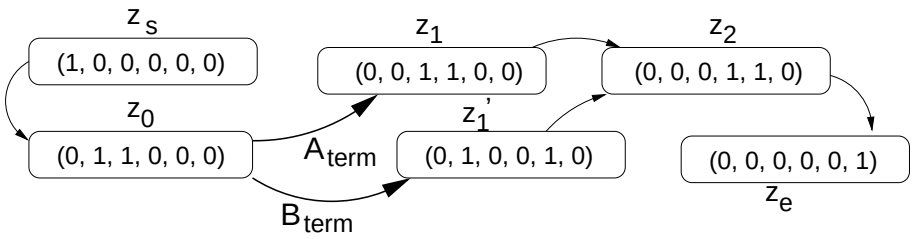


ABBILDUNG 5.2.5. Erreichbarkeitsgraph des Beispiels

den die gleichen Zeitfunktionen wie in Abschnitt 4.2.1.2 angewendet. Die Zeitfunktion wird durch eine Regressionsgerade nach $y = ax + b$ *mit* $b = 0$ *festgelegt. Für das Beispiel sollen* 10 *Ops eine Zeit von* 200 *Ticks benötigen und die Lasten seien quantifiziert mit* $q_{init}(A) = 15$ *und* $q_{init}(B) = 33$*. Wie in Abschnitt 4.2.1.2 nehme man an, daß eine Ressource die Antwortzeit um den Faktor der Anzahl an zu bearbeitenden Lasten* n *erhöht, womit* $f_t(l, s) = 20 * n * s(l)$ *die Zeitfunktion ergibt. Der Parameter zur Lastlokalisation wird der Anschaulichkeit halber wieder weggelassen.*

Zur Veranschaulichung des kleinsten Zeitschritts betrachtet man den Zustand z_0*. Hier ergibt sich* $\bigcup_{e \in E(v)} l_{term}(e) = \{A,B\}$*. Da* $\delta(A, s(z_0), u) = 20 * 2 * 15 = 600$ *und* $\delta(A, s(z_0), u) = 20 * 2 * 33 = 1320$*, berechnet sich* $\tau(z_0, s(z_0), u) = 600$*.*

5.2.2.2. *Folgeknoten.* Wie informell bereits eingeführt, sollen die Kanten, die vom aktuellen Zustand ausgehen, nicht gleich behandelt werden. Zeitlos interpretierte Übergänge müssen vorrangig behandelt

werden. Solche sind spontane und durch Szenario bewachte Übergänge. Die Übergänge, die das Terminieren einer Last modellieren, werden nachrangig behandelt.

DEFINITION 67 (Menge der Folgeknoten). *Sei M ein dynamisches Modell, $N = (S, T, F) = \mathcal{M}[\![M]\!]$ die Netzsemantik, $G(N) = (V, E)$ deren Erreichbarkeitsgraph, $in = \langle u_1, \ldots, u_n \rangle \in \mathcal{I}_{Szenario}$ ein Szenario mit seiner Ablaufsteuerung η_{in} und der dadurch gegebene Teilmenge der durch das Szenario erlaubten Übergänge η. Die Menge der Folgeknoten im Erreichbarkeitsgraphen der Netzsemantik wird durch folgende Funktion definiert:*

$$\nu_{Set}: \quad V \times \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \to \mathfrak{P}(\mathcal{V})$$

$$\nu_{Set}(v, in, s, u) = \begin{cases} \{v_j \ . \ \delta_{min}(l_{term}((v \to v_j)), s, u) = \tau(v)\}, \\ \qquad \textit{falls } E_{Spontan}(v) = \emptyset \textit{ (Zeitschritt)}, \\ \\ \{v_j \ . \ (v \to v_j) \in E_{Spontan}(v)\}, \\ \qquad \textit{sonst (Neutraler Schritt).} \end{cases}$$

Die Modellierung von Nebenläufigkeit bewirkt, daß anhand der ausgehenden Kanten des Erreichbarkeitsgraphen der Folgeknoten nicht deterministisch bestimmt werden kann. Sind in nebenläufigen Bereichen einer Parallelkartusche mehr als eine spontane Transition konzessioniert, so entsteht im Erreichbarkeitsgraphen ein Nichtdeterminismus. Ebenso kann gleichzeitig das Lesen eines Terminals aus verschiedenen Teilalphabeten des Szenarios anstehen. Und schließlich ist es nicht ausgeschlossen, daß zwei Lasten, die rechnerisch zur exakt gleichen Zeit terminieren, die minimalen Zeitschritte aller Lasten benötigen.

DEFINITION 68 (Simulationsteilmenge). *Sei M ein dynamisches Modell, $in \in \mathcal{I}_{Szenario}$ ein Szenario und $u \in \mathcal{U}$ eine Umgebung. Die Teilmenge der durch ein Szenario definierten Spur, welche unter Beachtung der Auswahl durch die umgebungsdefinierte Zustandsfolge, durch die nichtdeterministische Auswahl von Folgeknoten durch die Funktion ν_{Set} bestimmt ist, heißt* Simulationsteilmenge.

Zunächst wird ein beliebiger Knoten ausgewählt, um die Zeitsemantik für einen einzelnen Pfad zu definieren. Um eine Wahl von einer anderen unterscheiden zu können, wird die Funktion mit einem Index versehen.

DEFINITION 69 (Auswahl eines Folgeknoten). *Sei M ein dynamisches Modell, $N = (S, T, F) = \mathcal{M}[\![M]\!]$ die Netzsemantik, $G(N) =$*

(V, E) deren Erreichbarkeitsgraph, $in \in \mathcal{I}_{Szenario}$ ein Szenario. Sei $\nu_{Set}(v, in, s, u) = \{v_1, \ldots, v_n\}$, mit $n \geq 1$, die Menge der möglichen Folgeknoten und mit $vlin(\{v_1, \ldots, v_n\}) = \langle v_1, \ldots, v_n \rangle$ eine beliebige Linearisierung gegeben. Für ein $1 \leq i \leq n$ sei eine Funktion ν_i, die einen Knoten v_i bestimmt, folgendermaßen definiert:

$$\nu_i : \quad V \times \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \to \mathcal{V}$$

$$\begin{aligned} \nu_i(v, in, s, u) = \quad & v_i \ . \ v_i \in vlin(\nu_{Set}(v, in, s, u)) \\ sodaß \quad & i \neq j \Leftrightarrow \nu_i(v, in, s, u) \neq \nu_j(v, in, s, u) \end{aligned}$$

Später muß gezeigt werden, daß alle Pfade der Simulationsteilmenge bezüglich der Zeitsemantik invariant sind.

5.2.2.3. *Folgezustand.* Mit jedem besuchten Folgeknoten muß auch der Zustand der Umgebung fortgeschrieben werden. Für eine gegebene Zeit $t \geq 0$ läßt sich eine ressourcen-spezifische Schrittfunktion ableiten, die angewendet auf eine Ressource deren Zustand fortschreibt.

DEFINITION 70 (Ressourcen-Schrittfunktionen).

$$st_{Res} : \Sigma_T \times \mathcal{U} \times \mathbb{R}_0^+ \to (\mathcal{R} \to \Sigma_T)$$
$$st_{Res}\big(s, \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle, t\big) = \lambda \ r \ . \ f_s(res_{\mathcal{U}}, r)\big(\ s, t\ \big)$$

Es gilt: $\forall s, s' \in \Sigma_T, u \in \mathcal{U}, r \in \mathcal{R} \ . \ s' = st_{Res}(s, u, 0)(r) \Rightarrow s = s'$ nach Axiom 11.

Damit kann eine Funktion g definiert werden, die auf alle Lasten die von ihren Ressource abhängigen Funktionen anwendet.

DEFINITION 71 (Last-Schrittfunktion).

$$g : (\mathcal{L} \to \mathcal{R}) \times (\mathcal{R} \to \Sigma_T) \longrightarrow (\mathcal{L} \to \Sigma_T)$$
$$g(loc_{\mathcal{U}}, st_{Res}) = \lambda \ l \ . \ \Big(st_{Res}\big(loc_{\mathcal{U}}(l)\big)\Big)$$

Damit läßt sich die Funktion definieren, die abhängig von einem Zeitintervall t den Folgezustand für die Simulation berechnet:

DEFINITION 72 (Folgezustand).

$$state : \Sigma_T \times \mathcal{U} \times \mathbb{R}_0^+ \to \Sigma_T$$
$$state(s, \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle, t) = \bigcup_{l \in \mathcal{L}} g\big(loc_{\mathcal{U}}, st_{Res}(s, \langle loc_{\mathcal{U}}, res_{\mathcal{U}} \rangle, t)\big)(l)$$

5.3. Laufzeitsemantik

Zur Bewertung der Bearbeitungszeit eines Ablaufes wird einem Tripel aus dynamischem Modell, Szenario und Umgebung ein Zeitwert zugeordnet.

5.3.1. Semantik einzelner Abläufe. Für verschiedene Vorgänge bei der Simulation werden einzelne Funktionen definiert. Die Laufzeitsemantik wird dann aus diesen Elementen zusammengesetzt. Dies erlaubt einige wichtige Lemmata zu zeigen, aus welchen sich die Eindeutigkeit der Bewertung folgern läßt.

Grundsätzlich lassen sich Zustandsübergänge unterscheiden, in welche ein Zeitfortschritt stattfindet, von solchen, in denen die Zeit unverändert bleibt. Letztere, sogenannte neutrale Schritte, lassen auch den Lastzustand unverändert.

LEMMA 12 (Neutraler Schritt). *Neutrale Schritte lassen Lastzustand $s \in \Sigma_T$ und die Zeit t unverändert.*

Beweis. Nach Definition 67 gilt beim neutralen Schritt $\tau(v) = 0$. Dadurch ist $t' = t + \tau(v) = t$. Der Folgezustand ist $s' = state(s, u, 0)$ und damit nach Axiom 11: $s' = s$. □

Zur Auswahl einer Kante an einem Knoten mit mehreren ausgehenden Kanten wurde in der ν_i-Funktion der Index i eingeführt. Für mehrere aufeinander Folgende Schritte wird eine endliche Folge von Indizes $\overline{i}$ betrachtet.

DEFINITION 73 (Wohlgeformte Indexfolge). *Eine Folge $\overline{i} = \langle i_0, \ldots, i_n \rangle, (n \geq 1)$ von Indices heißt wohlgeformt bezüglich in, s und u, falls für jeden Index i_j in dem Lesezustand in des Szenarios, dem Lastzustand s und der Umgebung u, in dem er zur Anwendung kommt, gilt: $1 \leq i_j \leq |\nu_{Set}(v(in), in, s, u)|$ und erst dann $\overline{i} = \langle\rangle$ gilt, wenn $v(in) = v_{Stop}$.*

Weiterhin werden Hilfsfunktionen zur Ermittlung des ersten Index und der restlichen Indexfolge wie folgt eingeführt:

DEFINITION 74 (Kopf- und Restindexfolge). *Es sei eine nicht leere Indexfolge $\overline{i} = \langle i_0, i_1, \ldots, i_n \rangle$, dann sind:*

$$\begin{aligned} k(\overline{i}) &= i_0 \\ r(\overline{i}) &= \langle i_1, \ldots, i_n \rangle \end{aligned}$$

Zunächst führen wir eine Funktion ein, welche neutrale Schritte auf dem Pfad im Erreichbarkeitsgraphen abarbeitet.

DEFINITION 75 (Sem0-Funktion). *Es sei eine wohlgeformte Indexfolge $\bar{i}$ gegeben. Dann ist:*

$$sem0_{\bar{i}} \;:\; \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \to \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \cup \{\bot\}$$

mit:

$$sem0_{\bar{i}} \;\; (in, s, u) = \begin{cases} \langle in, s, u \rangle, \\ \quad \textit{falls } \tau(v(in)) > 0 \vee v(in) = v_{Stop} \\ \\ sem0_{r(\bar{i})}(st_{in}(in, t(v(in)) \to \nu_{k(\bar{i})}(v(in), in, s, u))), s, u), \\ \quad \textit{falls } r(\bar{i}) \neq \langle\rangle \textit{ und wohlgeformt,} \\ \\ \bot, \; \textit{sonst.} \end{cases}$$

Ausgehend von einem Tripel aus Lesezustand des Szenarios in, einem Lastzustand s und einer Umgebung u wird das Tripel ermittelt, beim dem der erste Zeitschritt erfolgt.

Eine zweite Funktion arbeitet Zeitschritte im Pfad des Erreichbarkeitsgraphen ab.

DEFINITION 76 (Sem1-Funktion).

$$sem1_i : \; \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+ \to \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+ \cup \{\bot\}$$

mit:

$$sem1_i(in, s, u, t) = \begin{cases} \langle st_{in}(in, t(v(in)) \to \nu_i(v(in), s, u))), \\ state(s, u, \tau(v(in))), \\ u, \\ t + \tau(v(in))\rangle, \; \textit{falls } \tau(v(in)) > 0, \\ \qquad \textit{und } \nu_i(v(in), s, u) \textit{ definiert.} \\ \\ \bot, \; \textit{sonst.} \end{cases}$$

Schließlich kombinieren wir beide Funktionen zu einem Gesamtschritt, indem jeweils ein Zeitinkrement erfolgt und alle folgenden neutralen Schritte abgearbeitet werden.

DEFINITION 77 (SemS-Funktion). *Sei eine wohlgeformte Indexfolge $\bar{i}$ gegeben. Dann ist:*

$$semS_{\bar{i}} : \; \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+ \to \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+ \cup \{\bot\}$$

$$
\begin{array}{rl}
& \textit{mit:} \\
semS_{\overline{i}} & (in, s, u, t) = \\
& \begin{cases}
(in'', s'', u, t'), & \textit{mit: } (in', s', u, t') = sem1_{k(\overline{i})}(in, s, u, t), \\
& \qquad \textit{falls } \overline{i} \textit{ wohlgeformt und} \\
& \qquad (in'', s'', u) = sem0_{r(\overline{i})}(in', s', u) \\
& \qquad \textit{falls } r(\overline{i}) \textit{ wohlgeformt,} \\
\bot & \textit{sonst.}
\end{cases}
\end{array}
$$

Gehen an Knoten mehrere gleichberechtigte Kanten aus, sind alle Funktionen von der Auswahl der Kante abhängig. Dies wurde durch eine spezifische Folge $\overline{i}$ an Indizes, angedeutet durch den Index i, aufgelöst. Die Folge zeitbewerteter Schritte ist für alle durch ein Szenario erlaubten Pfade gleich. Für zwei verschiedene, beliebige, wohlgeformte Indexfolgen $\overline{i}$ und $\overline{j}$ gilt somit folgende

PROPOSITION 1 (Unabhängigkeit von der Wahl des Pfades). *Für alle gültigen, nichtleeren in, s, u und t einer Simulation gilt, bezügliche zweier, verschiedener, wohlgeformter, beliebiger Indexfolgen $\overline{i}$ und $\overline{j}$:*

$$semS_{\overline{i}}(in, s, u, t) = semS_{\overline{j}}(in, s, u, t).$$

Die Richtigkeit der Proposition wird in Abschnitt 5.3.3 bewiesen. Da ein Performanzszenario stets einen Pfad zum Endknoten beschreibt, liefert ν_{Set} bis zum Erreichen des Endknotens v_{Stop} stets mindestens einen Folgeknoten. Daraus folgt ν_1 ist stets definiert, da der kleinste Index 1 nach unserem Verständnis bei einer möglicherweise einelementigen Menge mindestens dieses eine Element bezeichnet. Definiert man $\overline{1}$ als unendliche Folge von 1-Indices, mit $k(\overline{1}) = 1$ und $r(\overline{1}) = \overline{1}$, läßt sich damit eine allgemeingültige, willkürliche Definition einer Schrittfunktion angeben, die stets das sicher vorhandene Element mit dem kleinsten Index auswählt. Die Richtigkeit der Proposition vorausgesetzt, beeinflußt diese willkürliche Auswahl das Ergebnis der Bewertungsfunktion nicht.

DEFINITION 78 (Schritt-Funktion).

$$semS: \ \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+ \to \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_o^+$$

$$
\begin{array}{rl}
& \textit{mit:} \\
semS(in, s, u, t) = & \begin{cases}
sem0_{\overline{1}}(in, s, u, t) & , \textit{ falls } v(in) = v_{Start}, \\
semS_{\overline{1}}(in, s, u, t) & , \textit{ sonst.}
\end{cases}
\end{array}
$$

Damit kann schließlich die Laufzeit definiert werden, die einen Ablauf eines Modells in einer Umgebung in einem Schritt bewertet:

DEFINITION 79 (Laufzeitsemantik). *Gegeben seien:*

Modell	M
Netzsemantik	$N = (S, T, F) = \mathcal{M}[\![M]\!]$
Erreichbarkeitsgraph	$G(N) = (V, E)$
mit den Knoten	$v \in V$
und Kanten	$e \in E$
dabei Anfangsknoten	$v_0 \in V$
und Endknoten	$v_{Stop} \in V$
Umgebung	$u \in \mathcal{U}$
Szenario	$in \in \mathcal{I}_{Szenario}$
im Lesezustand	$in_0 \,.\, v_0 = v(in_0)$
Bearbeitungs-Zustand	$s \in \Sigma_T$
und dem Start-Zustand	$s_0 \in \Sigma_T$

Mit diesen Elementen erhalten wir die simulierte Laufzeit des Tripels von $G(\mathcal{M}[\![M]\!]), in$ und u zu:

$$\mathcal{Z}[\![G(\mathcal{M}[\![M]\!])]\!]_{u,in} = sem(in_0, s_0, u, 0) \;;$$

$$sem : \mathcal{I}_{Szenario} \times \Sigma_T \times \mathcal{U} \times \mathbb{R}_0^+ \longrightarrow \mathbb{R}_0^+ \text{ gegeben mit}$$

$$sem(in, s, u, t) =$$

$$= \begin{cases} sem(in', s', u, t'), & \text{falls} \\ & v(in) \neq v_{Stop} \\ & \wedge\ semS(in, s, u, t) = \langle in', s', u, t' \rangle \\ \\ t, & \text{sonst.} \end{cases}$$

Die Funktion $\mathcal{Z}[\![-]\!]_{u,in}$ bewertet das zeitliche Verhalten eines dynamischen Modells, welches als Parameter übergeben wird. Die Umgebung und die Spezifikation eines Ablaufs sind die bei der Auswertung betrachteten Kontextaspekte. Dies soll durch die Funktion, welche diese nach der Definition mitführt, ausgedrückt werden. Die Auswertung beginnt am Startknoten v_0 über einen Pfad aus Folgeknoten im Erreichbarkeitsgraphen und endet mit dem Endknoten v_{Stop}. Die Definition der Funktion als eine Rekursion über den ersten Parameter v und das Abbruchkriterium $v \neq v_{Stop}$ soll dies anschaulich machen. Das Szenario in wird während der Abarbeitung gelesen. *state* ermittelt den Folgezustand der Lasten in der Umgebung.

Der Haupteffekt der Funktion $\mathcal{Z}$ ist aber das Inkrement des Zeitwertes t, ausgehend von Anfangswert $t_0 = 0$. Dies modelliert das Fortschreiten der virtuellen Zeit. Ist der Endzustand v_{Stop} im Erreichbarkeitsgraphen erreicht, repräsentiert der Wert t die für diesen simulierten Ablauf nötige Zeit.

5.3.2. Eindeutigkeit der Laufzeitsemantik. Um die Eindeutigkeit der Laufzeitsemantik zu zeigen, führen wir zunächst einige Lemmata und Hilfsdefinitionen ein, die es erlauben das Problem weiter zu strukturieren.

Wir führen eine Funktion ein, die eine iterierte Anwendung der st_{in} ermöglicht.

DEFINITION 80 (Iterierte Lesefunktion). *Sei eine endliche Folge an Transitionen $\langle t_1, \ldots, t_n \rangle \in T^{<\mathbb{N}}$ so gegeben, daß sie von einem Szenario im gegebenen Lesezustand in eine gültige Folge an Übergängen im Erreichbarkeitsgraphen, des dem Szenario zugrundeliegenden Modells spezifiziert. Dann wird ein Folgezustand des Szenarios nach dem Schalten der Transitionsfolge definiert durch:*

$$st_{seq} \quad : \mathcal{I}_{Szenario} \times T^{<\mathbb{N}} \rightarrow \; I_{Szenario}$$

$$\begin{array}{rll} \textit{mit:} & st_{seq}(in, \langle\rangle) & = in \\ & st_{seq}(in, \langle t_1, \ldots, t_n \rangle) & = st_{seq}(st_{in}(in, t_1), \langle t_2, \ldots, t_n \rangle) \end{array}$$

In der Auswahl der Transitionsfolge fließt die Auswahl eines Folgeknoten ein, falls Nichtdeterminismus besteht. Wir führen nun eine Hilfsfunktion ein, die für Folgen neutraler Schritte eine Transitionsfolge in Abhängigkeit zu einem Index i erzeugt, so wie wir den Index in der Definition der $sem0$-Funktion verwendet haben mit:

DEFINITION 81 (Folge neutraler Schritte). *Eine Transitionsfolge ist gegeben mit:*

$$\overrightarrow{t}_{\overline{i}}(in, s, u) = \begin{cases} \langle\rangle, & \textit{falls } \tau(v(in)) > 0 \\ t(v(in) \rightarrow \nu_{k(\overline{i})}(v(in), in, s, u)) :: & \\ \quad \overrightarrow{t}_{r(\overline{i})}(st_{in}(in, t(v(in) \rightarrow \nu_{k(\overline{i})}(v(in), in, s, u))), s, u), & \\ \quad \textit{falls } \overline{i} \textit{ wohlgeformt}, & \\ \bot, & \textit{sonst.} \end{cases}$$

Die iterierte Lesefunktion erzeugt mit einer solchen Transitionsfolge genau den Folgelesezustand, der $sem0$-Funktion.

LEMMA 13. *Für alle wohlgeformten $\overline{i}$ und geeigneten in, s und u ist:*

$$\pi_0(sem0_{\overline{i}}(in, s, u)) = st_{seq}(in, \overrightarrow{t}_{\overline{i}}(in, s, u))$$

Beweis: Induktion nach der Länge l von $\overrightarrow{t}_{\overline{i}}$.
IA: $l = 0 \Rightarrow \tau(v(in)) > 0 \Rightarrow st_{seq}(in, \langle\rangle) = in$. Somit ist $\pi_0(sem0_{\overline{i}}(in, s, u)) = \pi_0(\langle in, s, u\rangle) = in$.
IV: Die Aussage gilt für $l = n - 1$.
IS: Für ein beliebiges $l = n$ und geeignetes $\overline{i}$, mit $n \geq 1$, gilt $\tau(v(in)) = 0$ nach Definition 81. Wir verwenden der leichteren Lesbarkeit halber die Abkürzung $t_1 = t(v(in) \rightarrow \nu_{k(\overline{i})}(v(in), in, s, u))$. Damit ist:

$$\begin{aligned} st_{seq}(in, \overrightarrow{t}_{\overline{i}}(in, s, u)) &= \\ \text{(nach Def 81)} &= st_{seq}(in, t_1 :: \overrightarrow{t}_{r(\overline{i})}(st_{in}(in, t_1))) \\ \text{(nach Def 80)} &= st_{seq}(st_{in}(in, t_1), \overrightarrow{t}_{r(\overline{i})}(st_{in}(in, t_1))) \end{aligned}$$

und $\pi_0(sem0_{\overline{i}}(in, s, u)) = \pi_0(sem0_{r(\overline{i})}(st_{in}(in, t_1), s, u))$. Da die Länge von $\overrightarrow{t}_{r(\overline{i})}(st_{in}(in, t_1))$ nach Konstruktion gleich $l - 1$ ist, folgt die Aussage aus der IV mit in'. □

Bereits ein möglicher zeitloser Übergang bedingt einen neutralen Schritt.

LEMMA 14. *Seien $in \in \mathcal{I}_{Szenario}$ und $u_i \in in$ mit $u_i = a_1 \ldots a_n$ und a_1 kennzeichnet einen zeitlosen Übergang, dann ist $\tau(v(in)) = 0$.*

Beweis: Da $u_i \in in$ gilt auch $\exists e \in E_{Spontan}(v(in)) \,.\, a_1 = \alpha(t(e)) \Rightarrow E_{Spontan} \neq \emptyset$. Nach Definition von τ gilt somit die Aussage. □

LEMMA 15. *Sei $u_j \in in$, $u_j = a_1 \ldots a_m a_{m+1} \ldots a_n$ und $a_1 \ldots a_m$ neutrale Schritte und a_{m+1} der erste Zeitschritt des Teilausdrucks. Sei $\overrightarrow{t}_{\overline{i}}(in, s, u) = \langle t_1, \ldots, t_k\rangle$, dann gilt:*

(1) *$a_1 \ldots a_m \in st_{in}(in, t_1)$ oder $a_2 \ldots a_m \in st_{in}(in, t_1)$*
(2) *Für alle l mit $1 \leq l \leq k$ gilt:*
    $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_l\rangle) \Rightarrow$
    $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_k\rangle)$
(3) *Sei $m \geq 1$. Dann ist $a_1 \ldots a_n \notin st_{seq}(in, \langle t_1, \ldots, t_k\rangle)$.*
(4) *$a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_k\rangle)$.*

Beweis zu (1): Die Richtigkeit der ersten Behauptung folgt unmittelbar aus Definition 53 der st_{in}-Funktion.

Beweis zu (2): Sei $in' = st_{seq}(in, \langle t_1, \ldots, t_l\rangle)$. Nach Definition von $\overrightarrow{t}_{\overline{i}}$ gilt $\tau(v(st_{seq}(in, \langle t_1, \ldots, t_p\rangle))) = 0$ (V1) für alle p mit $0 \leq p \leq k - 1$. Sei $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_l\rangle)$ (V2). Wir zeigen durch

Induktion nach q, daß gilt $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_l, \ldots, t_{l+q} \rangle)$, für alle $0 \leq q \leq k - l + 1$.
IA: Für $q = 0$ folgt die Aussage aus der Voraussetzung.
IV: Für alle $q \leq k - l$ und alle $t'_1, \ldots, t'_l$ mit $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t'_1, \ldots, t'_l \rangle)$ gilt $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t'_1, \ldots, t'_{l+q} \rangle)$.
IS: Für jeden einzelnen Schritt gilt $st_{Seq}(in, \langle t_1, \ldots, t_{l+q}, t_{l+q+1} \rangle) = st_{in}(st_{Seq}(in, \langle t_1, \ldots, t_{l+q} \rangle), t_{l+q+1})$. Sei ein $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_{l+q} \rangle)$ und a_{m+1} Terminal eines zeitbewerten Schritts, dann gilt $t(a_{m+1}) \neq t_{l+q+1}$. Denn ist a_{m+1} zeitbewehrt, so gilt für die Transition $t(a_{m+1}) = v \rightarrow v_j$ ein Zeitwert größer Null $\delta_{min}(l_{term}((v \rightarrow v_j)), s, u) \geq 0$ und damit gilt nach Definition 67: $v_j \notin \nu_{Set}(v, in, s, u)$. Da nach (V1) ein neutraler Schritt ausgeführt wird, kann ein Ausdruck $\langle a_{m+1}, \ldots, a_n \rangle$ nicht um a_{m+1} verkürzt werden, da Widerspruch zu Definition 81. Somit ist $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_{l+q+1} \rangle)$. Aus der Induktionsvoraussetzung folgt die Richtigkeit der Behauptung für $q = k$.

Beweis zu (3): Für $m \geq 1$ würde a_1 eines zeitlosen Schritt kennzeichnen. Wäre $a_1 \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_k \rangle)$, so wäre $\tau(v(st_{seq}(in, \langle t_1, \ldots, t_k \rangle))) = 0$, nach Lemma 14. Damit wäre aber $\overrightarrow{t}_{\bar{i}}(st_{seq}(in, \langle t_1, \ldots, t_k \rangle), s, u) \neq \langle\rangle$, in Widerspruch zu Definition 81. □

Beweis zu (4): Induktion nach m.
IA: $m = 0$, die Richtigkeit folgt nach Definition 80 und 81.
IV: $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_k \rangle)$.
IS: $m > 0$ Nach (3) gibt es l mit $a_1 \ldots a_n \notin st_{seq}(in, \langle t_1, \ldots, t_l \rangle)$. Aufgrund der Wohlordnung der natürlichen Zahlen gibt es ein kleinstes solches l. Wegen $a_1 \ldots a_n \in in$ gilt $l \geq 1$. Somit gilt $a_1 \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_{l-1} \rangle)$, $a_1 \ldots a_n \notin st_{seq}(in, \langle t_1, \ldots, t_l \rangle)$, dh. daß genau in diesem Schritt der Ausdruck um a_1 verkürzt wird. Aufgrund von Lemma 15, (1) folgt dann $a_2 \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_l \rangle)$. Somit folgt nach IV: $a_{m+1} \ldots a_n \in st_{seq}(in, \langle t_1, \ldots, t_k \rangle)$. □

Im Lesezustand in', der sich aus der Anwendung der $sem0$-Funktion ergibt, folgt zwingend ein Zeitschritt. Die Richtigkeit der Aussage folgt unmittelbar aus Definition 75.

Existieren in einem Lesezustand eines Szenarios Teilausdrücke, die neutrale Schritte konzessionieren, bleiben alle anderen Teilausdrücke, die Zeitschritte konzessionieren in Folgezustand unverändert.

LEMMA 16. *Seien $u_j \in in$, mit $u_j = a_1 \ldots a_n$ derart, daß a_1 einen Zeitschritt zu einem Folgeknoten kennzeichnet. Sei $\tau(v(in)) = 0$. Dann gilt für alle i, s und u: $u_j \in st_{in}(in, t(v(in) \rightarrow \nu_i(v(in), s, u)))$.*

Der besseren Lesbarkeit wegen führen wir folgende Notationen ein um die Folgezustände für Szenario, Last und Zeitwert aus der $sem1$-Funktion zu gewinnen.

DEFINITION 82 (Hilfsfunktionen).

$$\begin{aligned} sz_i(in,s,u,t) &= \pi_0(sem1_i(in,s,u,t)) \\ tstate_i(in,s,u,t) &= \pi_1(sem1_i(in,s,u,t)) \\ ttime_i(in,s,u,t) &= \pi_3(sem1_i(in,s,u,t)) \end{aligned}$$

Das Inkrement ist bei mehreren möglichen Folgeknoten für einen Zeitschritt gleich für jeden diesem Folgeknoten.

LEMMA 17. *Für alle möglichen i,j gilt: $ttime_i(in,s,u,t) = ttime_j(in,s,u,t)$*

Beweis: $ttime_i(in,s,u,t) = t - \tau(v(in)) = ttime_j(in,s,u,t)$ folgt unmittelbar aus Definition 76, falls $\tau(v(in)) > 0$. $ttime_i(in,s,u,t) = \perp = ttime_j(in,s,u,t)$, sonst.

Nun lassen sich verschiedene Eigenschaften des Lesezustandes nach einem Schritt durch eine $sem0$-Funktion zeigen.

LEMMA 18. *Sei ein Szenario $in \in \mathcal{I}_{Szenario}$ in einem Lesezustand, sodaß $\tau(v(in)) = 0$ und mit einer beliebigen, wohlgeformten Indexfolge $\bar{i}$: $(in', s', u) = sem0_{\bar{i}}(in,s,u)$. Dann gelten:*

1. *Für alle $u'_i \in in'$, seien entweder der Form $u'_i = a_1 \ldots a_n$, sodaß a_1 einen zeitbehafteten Übergang kennzeichnet, oder $u'_i = \epsilon$.*
2. *Seien von den $u_i \in$ in diejenigen der Form $u_i = a_1 \ldots a_m a_{m+1} \ldots a_n$ derart, daß alle $a_1 \ldots a_m$ neutrale Übergänge und a_{m+1} den ersten zeitbehafteten Übergang kennzeichnen. Dann gilt für diese $a_{m+1} \ldots a_n \in in'$.*
3. *Sei $u'_i \in in'$, dann gibt es ein Prefix aus Terminalen $a_1 \ldots a_m$, die neutrale Übergänge kennzeichnen, mit $a_1 \ldots a_n u'_i \in in$.*

Beweis zu (1): Sei $u'_j \in in'$, $u'_l \neq \epsilon$. Zu zeigen, daß $u_j = a_1 \ldots a_n$, sodaß a_1 einen Zeitschritt kennzeichnet, gelingt durch Widerspruch. Angenommen a_1 würde einen neutralen Übergang kennzeichnen, dann wäre $\tau(v(in)) = 0$, nach Lemma 14 und damit im Widerspruch zu Definition 75. □

Beweis zu (2): Nach Lemma 13 gilt $in' = st_{seq}(in, \overrightarrow{t}_{\bar{i}}(in,s,u))$. Nach Lemma 15 folgt die Aussage unmittelbar.

Beweis zu (3): Sei $u'_j \in in'$. Nach Lemma 13 gilt $in' = st_{seq}(in, \overrightarrow{t}_{\bar{i}}(in,s,u))$. Die Aussage folgt über einfache Induktion über die Länge von $\overrightarrow{t}_{\bar{i}}(in,s,u)$.

Folgende Hilfsfunktion soll anzeigen, ob ein Terminal eines Teilausdruckes einen zeitbewehrten möglichen Übergang kennzeichnet.

DEFINITION 83 (Hilfsfunktion). *Sei ein* $in \in \mathcal{I}_{Szenario}$ *und ein* $u_i \in in$ *mit* $u_i = au_i'$. *Dann sei*

$$zb(a, in) = \begin{cases} true, & falls\ \tau(v(in) > 0) \\ & \quad \wedge(a = \alpha(t(e))\ .\ e \in E_{Last}(v(in))) \\ & \quad \wedge\delta_{min}(l_{term}(e), s, u) = \tau(v(in)) \\ false, & sonst. \end{cases}$$

Für einen Zeitschritt lassen sich über die Folgezustände von Lastzustand und Lesezustand des Szenarios mit unterschiedlichen Indizes i, j nun folgende Aussagen machen.

LEMMA 19. *Sei* $in \in \mathcal{I}_{Szenario}$ *mit* $\tau(v(in)) > 0$. *Seien* i, j *gültige Indizes. So gilt:*

(1) $tstate_i(in, s, u, t) = tstate_j(in, s, u, t)$
(2) *Teilausdrücke im folgenden Lesezustand des Szenarios:*

$$\begin{array}{lll} \big(u_k \in & sz_i & (in, s, u, t) \Leftrightarrow u_k \in sz_j(in, s, u, t)\big) \\ \quad \vee & & \big(\exists u_k = au_k'\ .\ zb(a, in) \\ & & \wedge((u_k \in sz_j(in, s, u, t)) \vee (u_k' \in sz_j(in, s, u, t)))\big) \end{array}$$

Dh. die folgenden Lastzustände nicht deterministischer Schritte sind gleich. Für die Lesezustände folgt entweder, daß sie jeweils unverändert bleiben, oder ein führendes Terminal einen zeitbewehrten Schritt bezeichnet und möglicherweise im folgenden Lesezustand verkürzt ist.

Beweis zu (1): Folgt aus der Definition der *state*-Funktion.

Beweis zu (2): Folgt aus der Definition von st_{in}.

In der Umkehrung zu Lemma 19, (2) läßt sich sagen:

KOROLLAR 20. *Sei* $in \in \mathcal{I}_{Szenario}$ *mit* $\tau(v(in) > 0$ *und verschiedenen* i, j, *dann gilt:*

$$\begin{array}{lll} \big(u_k \in & sz_i & (in, s, u, t) \Leftrightarrow u_k \in sz_j(in, s, u, t)\big) \\ \quad \vee & & \big(\exists u_k = au_k'\ .\ \neg zb(a, st_{in}(in, t(v(in) \rightarrow \nu_j(in, s, u))))) \\ & & \wedge(\ (u_k \in sz_j(in, s, u, t) \vee u_k' \in sz_j(in, s, u, t))\ )\ \big) \end{array}$$

Dh. Teilausdrücke sind jeweils im folgenden Lesezustand unverändert, oder sind sie möglicherweise verändert, kennzeichnen sie im Folgezustand keine zeitbewehrten Lasten mehr.

Ein Zeitschritt terminiert minimale Lastelemente unabhängig von der gewählten Kante.

LEMMA 21. *Sei ein* $in \in \mathcal{I}_{Szenario}$ *und ein* $u_k = au'_k$ *so gegeben, daß* $zb(a, in)$ *gilt. Dann gilt für zwei Indices* i, j*:*

$$\neg zb(a, st_{in}(in, t(v(in) \rightarrow \nu_i(in, s, u)))) \Leftrightarrow$$
$$\neg zb(a, st_{in}(in, t(v(in) \rightarrow \nu_j(in, s, u))))$$

Sind die Indices i und j echt verschieden, bezeichnen sie das Terminieren jeweils verschiedener, aber mit gleicher minimaler Zeit bewehrter Lasten. Unabhängig von der Wahl der Kante sind in Folgezustand beide Lasten nicht mehr zeitbewehrt. Dies realisiert im Lastzustand echte Nebenläufigkeit.

Beweis: Die Aussage folgt unmittelbar aus Axiom 10.

5.3.3. Beweis der Richtigkeit von Proposition 1: Die Definition der Bewertungsfunktion basierend auf der willkürlichen Auswahl des kleinsten Index bei einer nichtdeterministischen Auswahl war nur sinnvoll, diese Auswahl ein etwaiges Ergebnis nicht quantitativ beeinflußt. In Proposition 1 wurde dies für jeden zeitbewehrten Einzelschritt behauptet. Die Richtigkeit der Proposition 1 ergibt nun durch die bis hier gezeigten Lemata wie folgt:

Seien in, s, u und t und zwei wohlgeformte Indexfolgen $\overline{i}$ und $\overline{j}$ derart, daß gilt $\tau(v(in)) > 0$ oder $v(in) = v_{Stop}$.

Somit ergeben sich aus in, s, u und t mit den Indexfolgen $\overline{i}$ und $\overline{j}$ die Funktionen zu $semS_{\overline{i}}(in, s, u, t) = (in', s', u, t')$ und $semS_{\overline{j}}(in, s, u, t) = (in'', s'', u, t'')$:

- $t' = t'' = t + \tau(v(in)$, nach Lemma 17
- Für $sem1_{k(\overline{i})}(in, s, u, t) = (in^*, s^*, u, t^*)$ und $sem1_{k(\overline{j})}(in, s, u, t) = (in^{**}, s^{**}, u, t^{**})$ ergibt sich nach Lemma 19-(1) $s^* = s^{**}$. Nach Lemma 12 bleibt der Zustand in einer Folge neutraler Schritte unverändert, sodaß mit $sem0_{r(\overline{i})}(in^*, s^*, u, t^*) = (in', s', u, t')$ und $sem0_{r(\overline{j})}(in^{**}, s^{**}, u, t^{**}) = (in'', s'', u, t'')$ folgt $s' = s''$.
- Für $sem1_{k(\overline{i})}(in, s, u, t) = (in^*, s^*, u, t^*)$ und $sem1_{k(\overline{j})}(in, s, u, t) = (in^{**}, s^{**}, u, t^{**})$ ergibt sich zwar $in^* \neq in^{**}$. Nach Lemma 21 gilt aber $\forall a \;.\; zb(a, in) \Rightarrow \neg zb(a, in^*)$, bzw.$\forall a \;.\; zb(a, in) \Rightarrow \neg zb(a, in^{**})$. Daraus folgt nach $sem0_{r(\overline{i})}(in^*, s^*, u, t^*) = (in', s', u, t')$ und $sem0_{r(\overline{j})}(in^{**}, s^{**}, u, t^{**}) = (in'', s'', u, t'')$ gilt $\forall a \;.\; zb(a, in) \Rightarrow \nexists u_k \in in' \;.\; u_k = au'_k$ oder $\forall a \;.\; zb(a, in) \Rightarrow \nexists u_k \in in'' \;.\; u_k = au'_k$, da sonst Widerspruch zu Lemma 18-(1). Daraus folgt $in' = in''$ nach der Entfernung aller nicht zeitbewehrten Elemente.

Daraus folgt, daß eine Rekursion über die $semS$-Funktion für alle übergebenen Parameter in, s, u und t unabhängig von einem ausgewählten Index, bei einer möglichen nicht-deterministischen Auswahl zu den selben Folgezuständen in', s', u und t' führt. Das Ergebnis der Bewertungsfunktion ist damit von der gewählten Indexfolge unabhängig. □

5.4. Zusammenfassung

In Kapitel 2 wurden dynamische Performanzmodelle mit einer Petrinetzsemantik so definiert, daß deren Erreichbarkeitsgraph stets endlich ist und vom Startknoten aus mindestens ein Pfad zum Endknoten führt. In Kapitel 3 wurde die Auswahl eines solchen Pfades so spezifiziert, daß genau die Klasse an möglichen Abläufen eines Modells eingegrenzt wird, die sich nur durch in unterschiedlicher Reihenfolge terminierende Lasten unterscheiden. Kapitel 4 führte einen abstrakten Umgebungsbegriff ein. Ressourcen dieser Umgebung werden mit den Funktionen beschrieben, die unter Beachtung eines Zustands gerade die Bestimmung der Reihenfolge der terminierenden Lasten und dabei verstreichenden Zeitintervalle erlauben.

In diesem Kapitel wurden diese drei Komponenten schließlich in einer sehr einfachen Funktion zusammengeführt, die für jedes Szenario eines Modells auf einer gegebenen Umgebung einen Zeitwert für die gesamte Bearbeitung aufsummiert. Mit jeder Berechnung dieser Funktion kann ein Testfall simuliert werden. Im Anschluß an ein kleines Beispiel folgt im zweiten Teil dieser Arbeit eine Auseinandersetzung mit der Methodik zu einer solchen Simulationstechnik für nicht-funktionale Eigenschaften. Die schon in der formalen Definition erahnbare Architektur eines Werkzeuges wird eingeführt und mit einer Anwendungsstudie |SEMPER| belegt. Außerdem wird die erfolgreiche Integration der Aspekte Technik und Methodik in diesem Werkzeug aufgezeigt.

Teil 2

Entwicklung zeitkritischer Software

Ein Entwurf zu performanzzentrierter Softwareentwicklung

Die bisher vorgestellte Technik der Entwicklung performanzkritischer Modelle erlaubt die Bewertung nicht-funktionaler, zeitabhängigen Eigenschaften in spezifischen Szenarien, vor ihrer vollständigen Implementierung. Wohlgeformte Performanzmodelle beschreiben aber nur strukturelle und dynamische Aspekte, soweit sie für die Bewertung relevant sind und fordern an einigen Stellen sogar eine weiterreichende Abstraktion, als sie für das Modell einer Software wünschenswert wäre. Den Szenarien, die für eine Simulation definiert werden, folgen Ergebnissen der Anforderungsanalyse oder müssen durch eigenständige Ergebnisse weiterer Analysen begründet werden. Für Umgebungsmodelle wurde ein abstraktes Konzept entwickelt und beispielhaft die Erstellung von Ressourcen aus Meßprotokollen erläutert. Für keines dieser zahlreichen Artefakte wurde aber bisher entwickelt, wie und wann sie im Verlauf eines Softwareentwicklungsprozesses entstehen können und welche Abhängigkeiten sich daraus ergeben. Weiterhin muß geklärt werden, wie die Erkenntnisse, die aus einer Performanz-Bewertung gezogen werden können, in qualitative, technische oder wirtschaftliche Verbesserungen umsetzbar sind. Eine Auseinandersetzung mit einer Methodik, welche die Integration dieser neuen Artefakte in den Software-Entwicklungs-Prozeß diskutiert, soll Grundlagen liefern, die auf viele Prozeßmodelle anwendbar sind, soweit sie sinngemäß Phasen wie Analyse, Modellierung, Implementierung und Test unterscheiden.

Zunächst begründet eine Klassifikation der neuen Artefakte und deren Abhängigkeiten zu klassischen Artefakten neue Rollen für Performanz-Modellierer, Szenario-Entwickler, Tester und Ressourcen-Ingenieure. Die Definition ihres Aufgabengebietes, der nötigen Ansprechpartner und ihrer Verantwortung im Ablauf des Prozesses erlaubt eine klar umrissene zeitliche Planung ihrer Tätigkeit und der Ansprüche an ihre Qualifikation. Dadurch wird die Entwicklung performanzkritischer Software in zeitlicher und damit auch in wirtschaftlicher Sicht besser kalkulierbar und gewinnt an Transparenz.

Weiterhin wird aber auch der Zusammenhang von qualitativen Modelleigenschaften, Design- und Architekturentscheidungen, zeitlich/wirtschaftlichen Vorgaben, besonderen Risikosituationen und personellen Restriktionen diskutiert. Die Bereitstellung von zuverlässigen Maßtabellen, die Management-Entscheidungen im Entwicklungsprozeß begründen, setzt umfangreiche Studien über eine Großzahl an personal- und kostenintensiven Softwareprojekten voraus, denen eine eingehende nachprüfbare Erfolgsanalyse folgt. Solche Feldstudien gingen weit über das Maß hinaus, welches diese Dissertation erreichen könnte. Außerdem würde die stets seitens der Industrie geforderte Anonymisierung der Datensätze aus empfindlichen Bereichen eine nützliche Interpretation weitgehend verhindern. Der Ansatz hier ist daher die Diskussion der Abhängigkeiten und der Entscheidungskriterien, sodaß innerhalb einer Firma auf dieser Grundlage ein Prozeß instantiiert werden kann, der es erlaubt, Projekterfahrung zu dokumentieren und in zukünftige Entscheidungsabläufe zu integrieren.

KAPITEL 6

Prozeßgrundlagen

Grundsätzlich lassen sich Artefakte im Softwareentwurfsprozeß nach ihrer Abstammung aus den verschiedenen Phasen des Prozesses gruppieren. Wir unterscheiden hier die bekannten Phasen der Analyse, die in Anforderungsanalyse und die Erstellung eines Analysemodells zerlegbar ist, den Entwurf, die Implementierung und den Test.

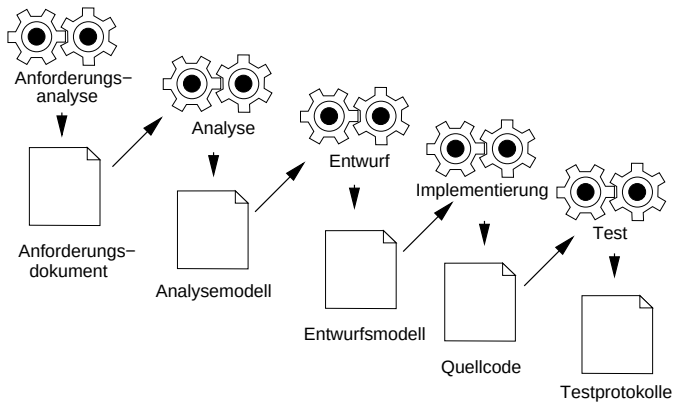


ABBILDUNG 6.0.1. Artefakte nach Prozeßphasen

Prinzipiell werden diese Phasen sequentiell durchlaufen und die Artefakte unterliegen entsprechenden Abhängigkeiten. Meist wird in der Literatur von Iterationen über diese Phasen gesprochen, wenngleich in der Praxis Mikrozyklen benachbarter Phasen[1] häufiger anzutreffen sind. Zunächst werden Beispiele einzelner Dokumenttypen der jeweiligen Phasen angesprochen. Da die verwendeten Formalismen projektabhängig sind, kann eine solche Klassifikation nur unvollständig sein und als offen verstanden werden. Hier werden beispielhaft unter anderem *UML*-Artefakte diskutiert, da sich so das Anwendungsbeispiel der

[1]Im *Unified Software Development Process* [**40**] werden diese Phasen als *Core-Workflows* bezeichnet. Als Phasen bezeichnet man dort größere Gruppen von Iterationen, in denen eine jeweils andere Zielrichtung und inhaltliche Gewichtung erfolgt.

Übersetzung aus Aktivitäts- und Implementationsdiagrammen fortsetzen läßt. In Projekten muß gegebenenfalls ein eigenes Artefaktmodell implementiert werden.

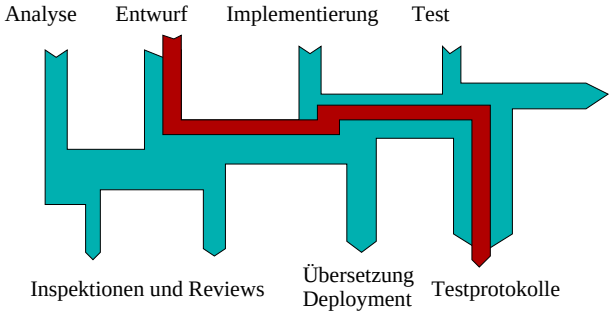


ABBILDUNG 6.0.2. Entwurfsabhängige Performanz-Eigenschaften

Die Bewertung von Performanz-Eigenschaften nach diesem Ansatz baut auf dem Entwurfsmodell auf, in welches Technologie-Entscheidungen eingeflossen sind und bis hinunter auf algorithmische Ebene die spätere Implementierung festgelegt wird. Das Analyse-

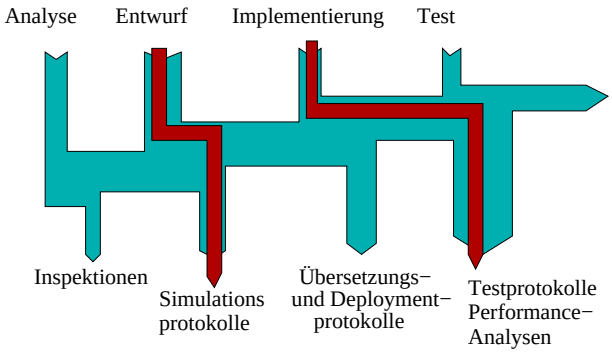


ABBILDUNG 6.0.3. Inspektion mit Simulationstechniken

Modell, das nur spezifiziert, *was* implementiert werden muß, aber nicht *wie* dies geschieht, dient vorwiegend zur Definition der nicht funktionalen Ziele. Viele Dokumenttypen, die statische Strukturen des Analysemodells betreffen, werden nicht behandelt.

Die in Abbildung 6.0.2 dargestellte Normalsituation zeigt, daß erst am Ende einer Iteration im Test zeitliche Eigenschaften beobachtet werden können. Wird sichergestellt, daß die Spezifikationen des Entwurfes tatsächlich in Code umgesetzt werden[2], kann davon ausgegangen werden, daß die wichtigsten leistungsbestimmenden dynamischen

[2]Dies ist eine wichtige Anforderung an die Einhaltung der Phasen

Strukturen bereits dem Implementierungsmodell entnommen werden können.

Werden außer Inspektionen und Reviews auf dem Modell eine Performanz-Simulation durchgeführt, können bereits hier Grundlagen für wichtige Entscheidungen vorliegen. So kann eine Modell-Inspektion negativ bewertet werden, falls die Simulation unzureichende Werte liefert. Die Modelle müssen dann nochmals verbessert werden. Aber auch ein Anhalten des ganzen Projektes wäre an dieser Stelle begründbar. Der Erfolg der Performanz-Bewertung wäre dann, Kosten für Implementierung und Test eingespart zu haben. Die schon aus der Einführung bekannte Graphik in Abbildung 6.0.3 zeigt schematisch die in einer früheren Prozeßphase gewonnenen Erkenntnisse.

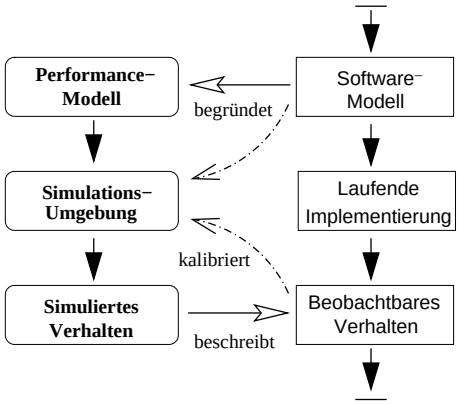


ABBILDUNG 6.0.4. Modellsimulation (schematisch)

Dabei nehmen wir an, daß die Simulation eine richtige Vorhersage des später beobachtbaren Verhaltens ist, wie es in Abbildung 6.0.4 schematisch dargestellt wird. Durch die gestrichelten Pfeile ist dargestellt, daß mit der hier vorgestellten Technik ein Umgebungsmodell erstellt werden muß. Dies geschieht etwa durch ein *UML-Deployment-Diagram* (oberer gestrichelter Pfeil). Um eine Simulation durchführen zu können, muß die Funktion, welche die Zeitschritte bewertet, mit zuverlässigen Schätzungen oder Meßwerten kalibriert werden. In der ersten Iteration müssen diese Werte geschätzt werden oder an eigens entwickelten Code-Proben gemessen werden. In weiteren Iterationen können Meßwerte aus früheren Iterationen verwendet werden, falls sichergestellt werden kann, daß der zugrunde liegende Code noch den aktuellen Modellen entspricht. Ziel ist es, den linken Strang der schematischen Darstellung zeitlich vor der Implementierung vollenden zu können.

6.1. Prozeßintegration

6.1.1. Klassische Artefakte.

6.1.1.1. *Interviewprotokolle.* Falls die Neuentwicklung von Software Ziel des Prozesses ist, stellen Interviewprotokolle die klassische primäre Informationsquelle der Anforderungsanalyse dar. Selbst umfangreiche Dokumente, die zum Beispiel für eine Ausschreibung vorab durch einen Auftraggeber erstellt worden sind, gehören, gleich welche Beschreibungstechniken sie verwenden, ausschließlich in diese Kategorie. Bei der Pflege, Erweiterung oder Verbesserung großer, bereits existierender Systeme stehen sie möglicherweise neben Artefakten, die das existierende System beschreiben oder aus dem Herstellungsprozeß dieses Systems erhalten sind.

Häufig findet man hier eine Zerlegung des Problems nach den funktionalen Anforderungen des Auftraggebers. In klassischen Geschäftsfeldern ist hier die Geschäftsprozeßanalyse zu nennen. Oft verwendet man neben der textuellen Aufbereitung auch UML-Anwendungsfalldiagramme.

6.1.1.2. *Anwendungsfälle und Subsystemzerlegung.* Eine gute Grundlage des Analysemodells ist die Entwicklung eines Systemsbegriffs, der die Subsysteme, oder Komponenten identifiziert. Dann werden die für die jeweiligen Subsysteme relevanten Anwendungsfälle[3] entwickelt.

6.1.1.3. *Domänenmodell, Datenmodell, Klassendiagramme.* Die Dokumente des Entwurfs, welche Datenstrukturen spezifizieren, sind insofern interessant, wie sie quantitative Aussagen über Datenstrukturen zulassen, die eine Ressource belasten, auf der ein implementiertes System laufen wird.

6.1.1.4. *Dynamisches Modell, Aktivitätsdiagramme, Zustandsdiagramme.* Die Dokumente, welche den Entwurf dynamischer Strukturen verkörpern, haben hier eine zentrale Stellung. Bezüglich dynamischer Aspekte muß ein Modell vollständig sein und der tatsächlichen späteren Implementierung entsprechen.

6.1.1.5. *Deployment- oder Implementationsmodell.* Der Entwurf der technischen Umgebung modelliert die Architektur der technischen Ressourcen, die später die implementierte Software ausführen soll. Ein wichtiger Aspekt ist die eindeutige Zuordnung jeder Funktionalität zu einer Ressource und die Spezifikation deren technischer Ausführung und Leistungfähigkeit.

[3]Diese liegen im Idealfall orthogonal zu den Anwendungsfällen, welche in den Geschäftprozessen identifiziert wurden und unterscheiden sich von diesen grundsätzlich.

6.1.1.6. *Quellcode, Implementierung, Konfigurationen.* Da funktionale Eigenschaften nicht Gegenstand der Betrachtung sind, muß kein Unterschied gemacht werden zwischen Quellcode und Kompilat. Von einem Quelldokument wird stets erwartet, daß es kompilierbar ist und fehlerfrei im Sinne der funktionalen Anforderungen läuft.

6.1.1.7. *Testdokumente.* Testprotokolle, Logfiles oder Debugging-Protokolle sind interessant, soweit sie es erlauben einzelne Abläufe konkret nachzuvollziehen. Hier werden ebenso nur die Arten von Dokumenten angesprochen, die Rückschlüsse auf performanzrelevante Eigenschaften zulassen.

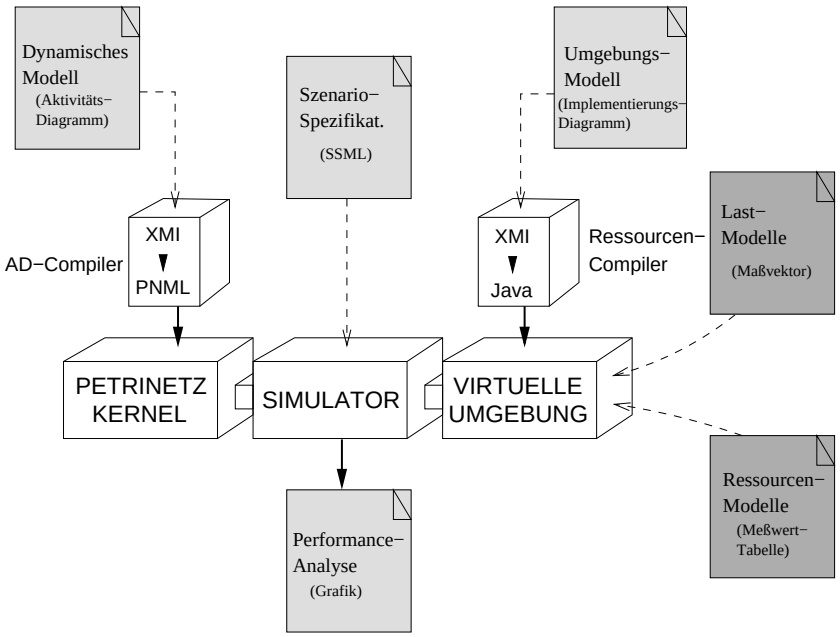


ABBILDUNG 6.1.1. Artefakte des Simulators

6.1.2. Neue Artefakte. Dynamische Modelle und Umgebungsmodelle bilden eine Gruppe neuer Artefakte, die unmittelbar mit den klassischen Artefakten verbunden sind. Dazu gehören die angesprochenen Szenarien, die jeweils einzelne Testfälle spezifizieren. Die zweite Gruppe neuer Artefakte sind Last- und Ressourcenmodelle. Diese nehmen eine Sonderstellung ein. Zwar werden sie meist spezifisch in einem Projekt meist sogar in einer spezifischen Iteration entwickelt. Sie sind aber nicht auf eine Verwendung allein in diesem Projekt beschränkt, sondern tragen zu einem Modellarchiv bei, von dem kommende Projekte profitieren können. Abbildung 6.1.1 bietet einen Überblick über diese Artefakte.

6.1.2.1. *Anforderungsanalyse.* Performanzeigenschaften benötigen eine separate Anforderungsanalyse. Oft bauen Performanz-Anforderungen bereits auf Entwurfsentscheidungen auf oder können erst formuliert werden, wenn die relevanten Elemente bekannt sind und für nicht funktionale Eigenschaften adressiert werden können. Die Anforderungsanalyse erstreckt sich von den ersten Interviews bis über das Ende der Erstellung des Analysemodells hinaus, dessen Fertigstellung erfolgt sein muß, bevor Performanzanforderungen integriert werden können. Dokumente der Anforderungsanalyse teilen sich auf in Spezifikation grundlegender Anforderungen, technische Anforderungen, die später die Umgebung beeinflussen werden und explizite Einzelanforderungen des Auftraggebers, die ausdrücklich erfüllt sein müssen. Letztere müssen unmittelbar in die Entwicklung von Szenarien münden und später durch Tests nachgewiesen werden. Aufbauend auf die in diesen Dokumenten spezifizierten Anforderungen werden die folgenden Modellartefakte entwickelt.

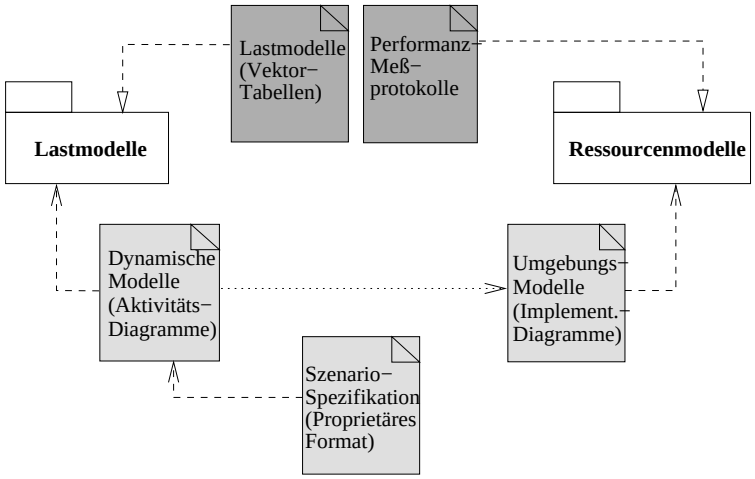


ABBILDUNG 6.1.2. Artefakteabhängigkeiten

6.1.2.2. *Last- und Ressourcenspezifikationen.* In Abbildung 6.1.2 werden die Abhängigkeiten der Artefakte untereinander schematisch dargestellt. Aufbauend auf den Vektortabellen der Lastmodelle, erzeugt der Lastmodellgenerator des Werkzeugs generische Klassen für die Simulationsumgebung. Die Performanz-Meßprotokolle, mit deren Hilfe Ressourcenmodelle erzeugt werden, liefern die Stützstellen der Funktion, mit der die Zeitabstände zwischen einzelnen Zustandsübergängen bewertet und der Simulationszustand fortgeschrieben wird. Die Artefakttypen sind die Grundlagen für die Last- und Ressourcenbibliothek

des Simulationswerkzeugs, das selbst keine Modelle vorsieht. Sie spielen somit eine Sonderrolle dieses Ansatzes.

6.1.2.3. *Umgebungsmodelle.* Die eine Software ausführende Umgebung wird durch Umgebungsmodelle beschrieben. Diese können zum Beispiel durch UML-Implementierungsdiagramme (*Deployment-Diagramms*) dargestellt werden. Die Spezifikation der Simulationsumgebung durch Implementierungsdiagramme, wie sie hier beispielhaft gezeigt wird, erbt die Struktur der klassischen Umgebungsmodelle, benötigt aber zusätzlich spezifische Annotationen, die Ressourcenmodelle festlegen, sie mit Namen versehen und gegebenefalls initialisieren.

6.1.2.4. *Dynamische Performanz-Modelle.* Aus den dynamischen Modellen der klassischen Artefakte müssen Modelle gewonnen werden, welche den Kontrollfluß genau wiedergeben, aber von qualitativen Aspekten abstrahieren. Es sind Annotationen nötig, die auszuführende Aktivitäten quantifizieren. Dazu müssen Lastmodelle gewählt werden, die im Modellpaket des Simulationswerkzeugs vorhanden sind. Eine weitere Abhängigkeit existiert bei Umgebungsmodellen bezüglich des Namensraumes. Im Performanz-Modell spezifizierte Lasten müssen auf adäquate Ressourceninstanzen des Umgebungmodells lokalisiert werden.

6.1.2.5. *Szenarien.* Die meist zahlreichen Szenarien hängen nur vom dynamischen Performanz-Modell ab. Ein Szenario legt genau einen möglichen Bearbeitungsablauf eines dynamischen Performanz-Modells fest. Im Regelfall werden Szenarien von einem Performanzmodell direkt abgeleitet. In manchen Fällen können Szenariospezifikationen auch auf Performanz-Modelle angewendet werden, die ihrer Erstellung nicht zugrunde lagen. Dies ist dann möglich, wenn Modelle sich zum Beispiel nur in der Lokalisation der Lasten unterscheiden, Permutationen in rein sequentiellen Bereichen aufweisen oder sich nur in der Annotation von Lastmodellen oder deren Parametern unterscheiden.

6.1.2.6. *Performanz Analyse.* Das Ergebnis einer Simulation ist eine quantitative Bewertung eines Tripels aus dynamischem Modell, Umgebungsmodell und einem spezifischen Szenario. Es besteht meist aus einem numerischen Wert und einer Einheit. Ein solcher Wert ist mit der Spezifikation einer Anforderung in der Anforderungsanalyse oder dem Analysemodell assoziierbar. Hieraus kann auch eine sinnvolle Zusammenfassung mehrerer Werte sowie tabellarische oder graphische Aufbereitung abgeleitet werden. Eine solche Aufbereitung sollte sinnvolle Anmerkungen beinhalten, die es dem Leser ermöglichen, den Inhalt richtig zuzuordnen. Solcherart aufbereitete Artefakte werden als Performanz-Analyse-Dokumente bezeichnet und fließen wieder in den klassischen Software-Entwicklungsprozeß ein.

6.2. Prozeßablauf

6.2.1. Hauptprozeßstrang. Zunächst wird die Entstehung der Artefakte verfolgt, die parallel zu den klassischen Dokumenten, die direkt mit der Umsetzung funktionaler Anforderungen in Zusammenhang stehen. Gesondert wird später noch die Klasse an Artefakten betrachtet, die zum Kontext gerechnet werden können, da sie zur technischen Realisierung der Simulationsumgebung gehören. Tatsächlich müssen sie spezifischen Vorgaben entsprechen, die aus den Hauptartefakten abgeleitet werden. Sie sind dadurch mittelbar auch mit den funktionalen Anforderungen verbunden.

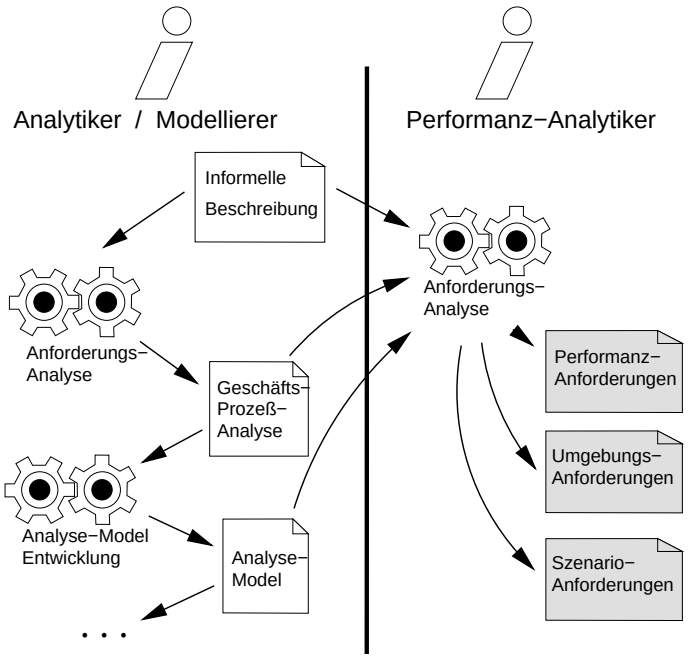


ABBILDUNG 6.2.1. Anforderungsanalyse

6.2.1.1. *Anforderungsanalyse.* Die bereits geforderten speziellen Performanz-, Umgebungs- und Szenarioanforderungsanlysen müssen in der Analysephase erstellt werden. Abbildung 6.2.1 weist eine eigene Rolle eines *Performanz-Analytikers* aus, die später noch eingeführt wird. Weiterhin zeigt sie schematisch die Einflußnahme der Erkenntnisse aus der Geschäftsprozeßanalyse und dem gesamten Analysemodell. Die unterschiedlichen Klassen von Anforderungen sollten in gesonderten Dokumenten formuliert werden. Später begründen sie Quelldokumente für verschiedene Rollenprofile.

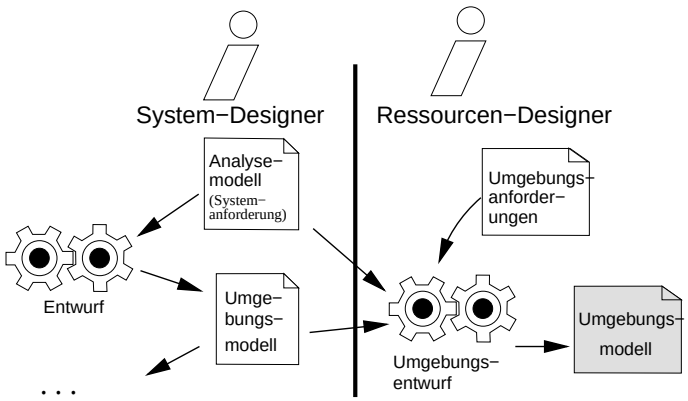


ABBILDUNG 6.2.2. Umgebungsanlyse

6.2.1.2. *Umgebungsmodell.* Zunächst wird wieder davon ausgegangen, daß alle notwendigen Lastmodelle und adäquaten Ressourcen bereits spezifiziert sind und verwendet werden können. Betrachtet man erneut die in Abbildung 6.1.2 dargestellten Abhängigkeiten, ist erkennbar, daß zuerst das Umgebungmodell entwickelt werden muß. Da diese Reihenfolge nicht immer üblich ist, sondern technische Entscheidungen über die Umgebung meist nach der Wahl der Software-Technologie getroffen werden, muß dem bereits während der Anforderungsanalyse durch den Anforderungsanalytiker Rechnung getragen werden. Unter Einbeziehung des Analysemodells, des technischen Umgebungsentwurfes und der in den Umgebungsanforderungen identifizierten performanzrelevanten Einschränkungen wird ein Umgebungsmodell entwickelt, wie in Abbildung 6.2.2 dargestellt. In diesem Dokument werden alle Ressourcentypen identifiziert. Deshalb sollten diese Ressourcentypen in den Ressourcenmodell-Archiven der Simulationsumgebung bereits bekannt und verfügbar sein. Auch Ressourcen, die noch unbekannt sind, können aber hier modelliert werden. Ihre Beschreibung muß später nachgefordert werden. Hier liegt ein Umgebungsmodell beispielhaft als *UML*-Implementierungsdiagramm (*Deployment Diagram*) vor.

In der Darstellung sind die zwei parallel existierenden Umgebungsmodelle zu erkennen. Eines das die technischen Spezifikationen der späteren Ressourcen bezüglich ihre funktionalen Anforderungen enthält und das Andere welches davon abstrahiert und sich auf Annotationen zum zeitlichen Verhalten beschränkt.

6.2.1.3. *Dynamische Performanz-Modelle.* Liegt ein Umgebungsmodell vor, kann ein dynamisches Performanz-Modell entwickelt werden. Hier wird ein Entwurf bezüglich Kontrollfluß und Lasttypen entwickelt, aber auch die Zuordnung der einzelnen Lasten zu Ressourcen spezifiziert. Dazu muß das Umgebungsmodell vorliegen. In das

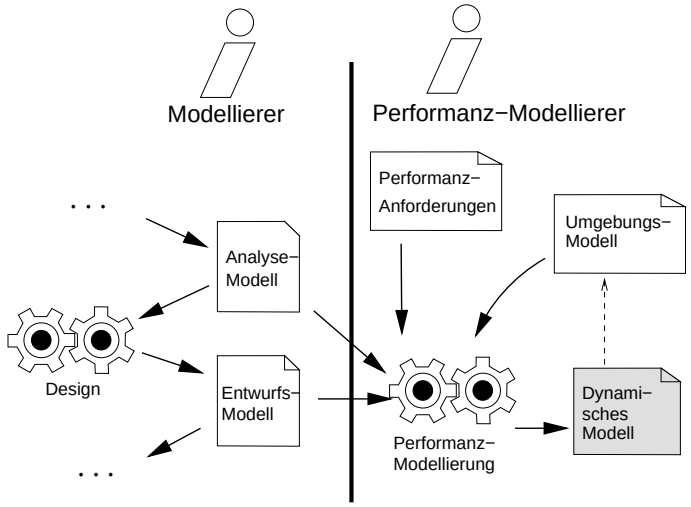


Abbildung 6.2.3. Performanz Modellierung

Performanz-Modell fließen die Erkenntnisse des Analysemodells und die Entscheidungen des Software-Entwurfs mit ein. Wie in Abbildung 6.2.3 dargestellt, muß der Performanz Modellierer auch Sorge tragen, daß das entwickelte Performanzmodell bezüglich aller identifizierter Anforderungen ausreichend aussagekräftig ist.

6.2.1.4. *Szenarien.* Aufbauend auf das Performanz-Modell können Szenarien für die Performanzanalyse entwickelt werden. In die Szenarien fließen die Erkenntnisse aus der Anforderungsanalyse und dem Entwurf mit ein. Eigentlich können Szenarien aber unabhängig von klassischen Artefakten, aufbauend auf dem Performanzmodell, die Performanz- und Szenarioanforderungen formulieren. Abbildung 6.2.4 zeigt diese Unabhängigkeit schematisch. Szenarien sind für das dynamische Performanz-Modell, für das sie entwickelt wurden spezifisch. Jedoch ist eine Wiederverwendung nicht ausgeschlossen, falls keine Änderungen am Kontrollfluß des dynamischen Modells vorgenommen worden sind.

6.2.1.5. *Performanz-Analyse.* Liegen Umgebungsmodell, dynamisches Performanzmodell und Szenarien vor, kann die Performanz-Simulation durchgeführt werden, und das zeitliche Verhalten des Softwaremodells bewertet werden. Dieser Schritt soll durchgeführt werden

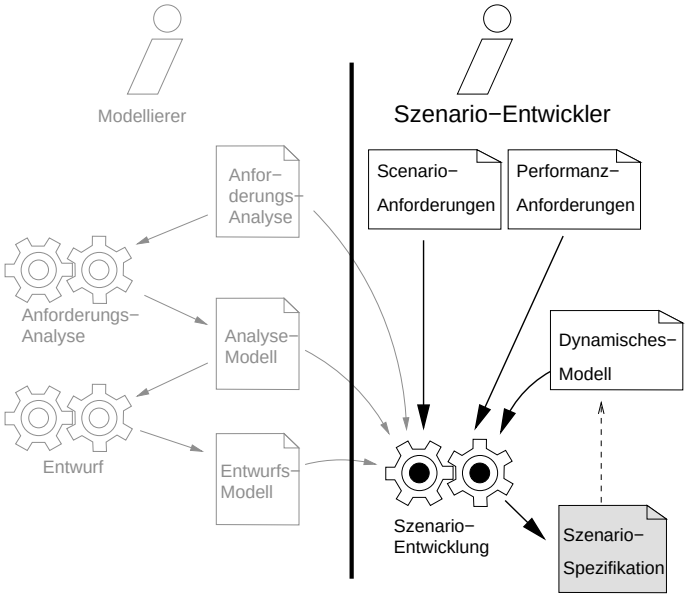


ABBILDUNG 6.2.4. Szenario-Entwicklung

bevor die Implementierung beginnt. In der Herleitung der Dokumente war zu erkennen, daß eine Implementierung (unter der Annahme, Last- und Ressourcenmodelle seien vorhanden) bis zu diesem Zeitpunkt nicht nötig war. Das weitere Vorgehen im Prozeß wird von den Ergebnissen

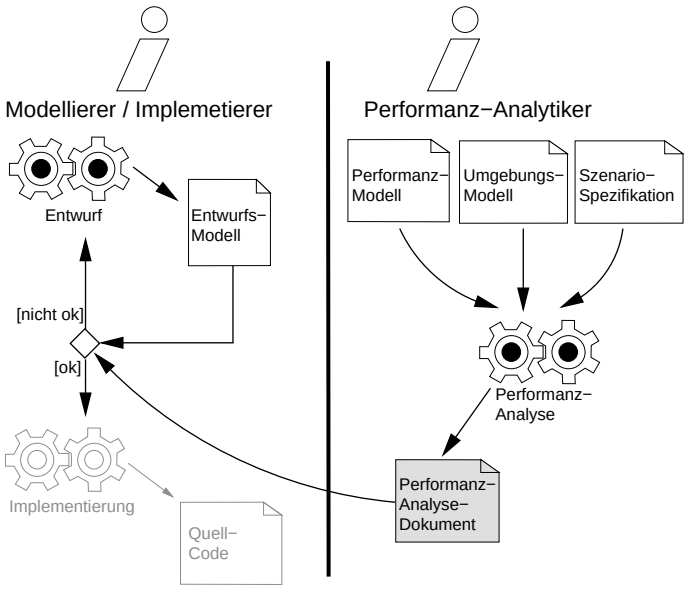


ABBILDUNG 6.2.5. Performanzanalyse

der Performanzanalyse abhängig gemacht. Abbildung 6.2.5 zeigt schematisch die Entscheidung mit der Implementierung zu beginnen oder nochmals Verbesserungen am Modell vorzunehmen. Auch die bereits angesprochene Möglichkeit des Projektabbruchs kann an dieser Stelle begründet sein.

6.2.2. Kontextprozeß. Es wurde bis zu dieser Stelle immer angenommen, daß adäquate Last- und Ressourcenmodelle in der Bibliothek der Simulationsumgebung ausreichend zur Verfügung stehen. Eine der Stärken der hier vorgestellten Technik liegt in der Ausdruckstärke und Anpassbarkeit der Last und Ressourcenmodelle. Diese sollten, um diesen Vorteil zu nutzen, auch spezifische, leistungsentscheidende Charakteristika ausdrücken. Deshalb folgt eine Auseinandersetzung mit der Erzeugung solcher Modelle in einem eigenen, sogenannten Kontextprozeß.

6.2.2.1. *Initiale Lastmodell-Entwicklung.* Am Anfang des ersten Software-Entwicklungsprozesses existieren keine Lastmodelle, die zur Simulation verwendet werden können. Dennoch wird, wie vorher beschrieben, ein dynamisches Performanzmodell erstellt. In diesem Modell werden alle benötigten Lastmodelle verwendet, als würden sie bereits existieren. Zusätzlich kann eine Dokumentation informell Aufschluß über die gewünschten Eigenschaften der Lastmodelle geben. Die Menge der verwendeten Lastmodelle muß nun von einem Simulationsingenieur realisiert werden. Er erstellt für jedes Modell eine Lastklasse in der Werkzeugbibliothek und gibt gleichzeitig bei einem Software-Entwickler die Implementierung eines Codefragments in Auftrag. Sollten die Informationen, die der Simulationsingenieur dem Modell-Dokument entnehmen kann, unzureichend sein, können eigene Interviews und Analysesitzungen anberaumt werden.

Eine Bearbeitung des entsprechenden kompilierten Codefragments soll genau ein Beispiel für die Bearbeitung einer Instanz der modellierten Last sein. Diese Fragment dienen später zur Erstellung von Laufzeittests mit deren Ergebnissen die Ressourcenmodelle kalibriert werden. Abbildung 6.2.6 zeigt diesen Vorgang schematisch. Die Entwicklung und Kalibrierung der Ressourcenmodelle, die im nächsten Abschnitt beschrieben wird, kann eine iterierte Abarbeitung dieses Teilprozesses anstoßen, was in dieser Abbildung nicht explizit dargestellt ist. Die möglichen Zusatzinformationen des Modellierers sind hier nicht extra aufgeführt.

6.2.2.2. *Initiale Ressourcenmodell-Entwicklung.* Für jeden im Umgebungsmodell ausgewiesene Ressourcentyp muß ein Simulationsmodell

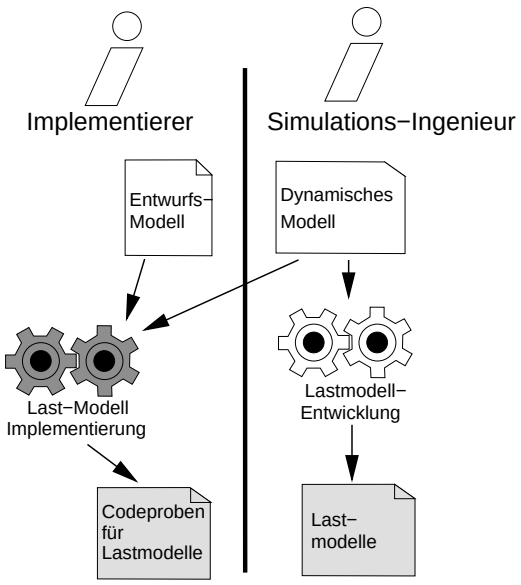


ABBILDUNG 6.2.6. Lastmodellentwicklung

kalibriert werden. Die Modelle sind prinzipiell alle gleichartig und werden nur durch die beiden Funktionen zur zeitlichen Bewertung eines Rechenschritts und einer Funktion zur Ermittlung des nächsten Folgezustandes realisiert. Der Definition dieser Funktionen liegen im hier ausgearbeiteten Beispiel Messungen echter Laufzeiten als Stützstellen einer Interpolationsfunktion zugrunde. Um diese Messungen durchführen zu können, muß eine repräsentative technische Umgebung installiert werden, die der im Umgebungsmodell spezifizierten entspricht. Aus dem dynamischen Performanzmodell muß ermittelt werden, in welchem Intervall welche Dichte an Stützstellen nötig ist, um zuverlässige Schätzungen zu erhalten. Eine vorausschauende Planung solcher Grenzen kann sinnvoll sein. In dieser Voraussicht sollten Bereiche abgedeckt werden, die Modellierer bei erwartbaren Erweiterungen oder für die Auslotung der Modelleigenschaften in Problembereichen nutzen werden. Gemäß den so festgelegten Bereichen werden für jedes Lastmodell eine Testlauf-Matrix entwickelt.

Ein Systemtechniker muß nun die in dieser Matrix geforderten Werte zusammentragen. Dies kann durch Erfahrungswerte geschehen, manuellen Berechnungen, die auf Leistungsangaben des Herstellers beruhen, oder willentliche Festlegungen von Performanz-Verhalten. In diesem Fall stellt eine solche Performanz-Funktion an sich eine Anforderungs-Spezifikation für die Hardwarebereitstellung dar.

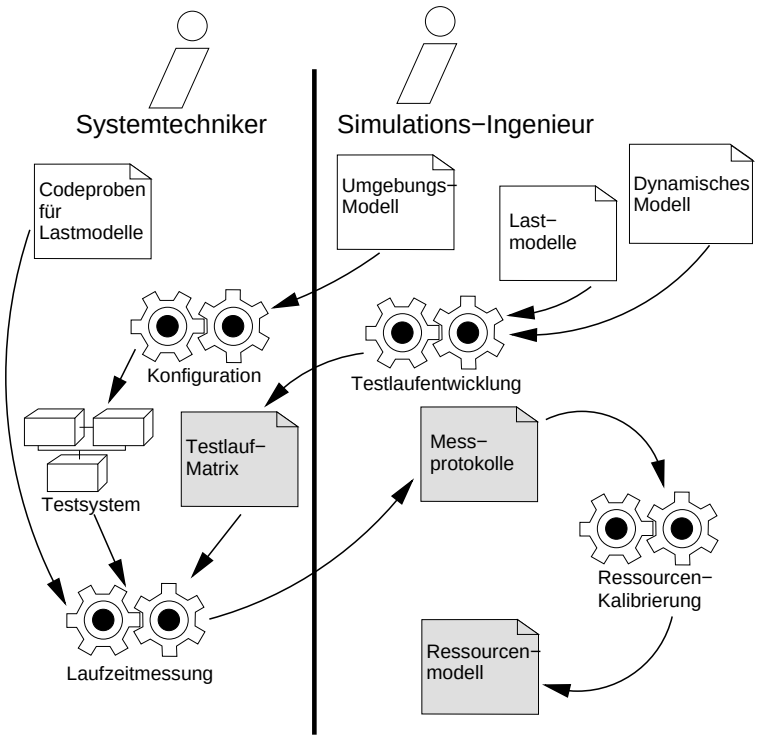


ABBILDUNG 6.2.7. Ressourcenmodell Entwicklung

Ein typischer Vorgang ist die Erstellung der Funktion durch Messung des zeitlichen Verhaltens existierender Hardware. Entsprechend der Testlauf-Matrix werden die dem Lastmodell zugeordneten Codeproben, in dem in der Matrix festgelegten Kontext ausgeführt. Die gemessenen Bearbeitungszeiten werden in einem Meßprotokoll dokumentiert. Dieses Meßprotokoll ist die Grundlage der Modellfunktionen, die der Simulator bei der Interpretation des dynamischen Performanzmodells zugrundelegt. In Abbildung 6.2.7 ist dieser komplexe Vorgang schematisch dargestellt.

6.2.2.3. *Iterierte Simulationsmodellentwicklung.* Abbildung 6.2.6 stellt eine Neuentwicklung von Codeproben zur Repräsentation von Lastmodellen dar. Die Kosten dieser Entwicklung lassen sich später nicht in der Software wirksam bezüglich qualitativer Eigenschaften einbringen, da sie nur dem Betrieb der Simulationstechnologie dienen. Befindet sich der Software-Entwicklungsprozeß in einer höheren Iteration, kann diese gesonderte Codeerstellung entfallen, falls repräsentative Codefragmente aus der letzten Iteration isoliert werden können. In diesem Fall muss durch den Simulations-Ingenieur nur sichergestellt werden, daß die Fragmente repräsentativ für die verwendeten Lastmodelle sind.

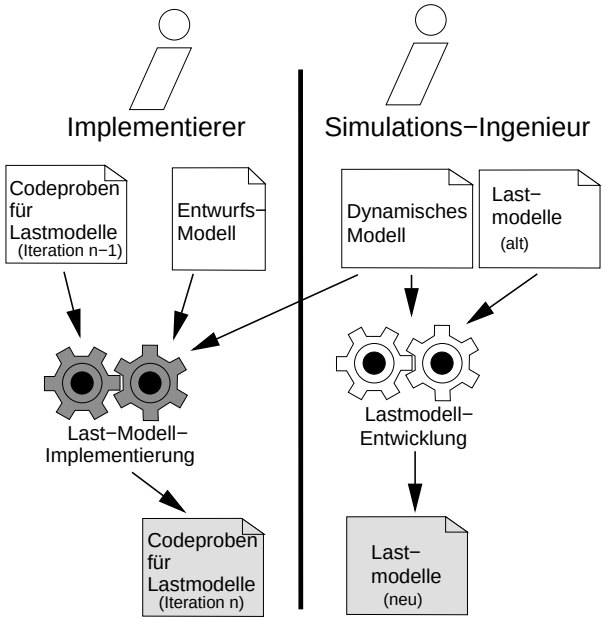


ABBILDUNG 6.2.8. Iteration der Lastmodell Entwicklung

Möglicherweise ist es sinnvoll, die Performanzbewertung insgesamt erst in der zweiten Iteration zu starten, um diese zusätzlichen Kosten zu vermeiden. Grundlage für eine solche Entscheidung muss eine Risikoanalyse sein. In einem größeren Software-Projekt, das beispielsweise nach dem Schema des *Unified Software Development Process* [**40**] durchgeführt wird, können aussagekräftige Codebeispiele möglicherweise sehr spät, etwa am Anfang der *Construction*-Phase zur Verfügung stehen. Zwischen diesem erhöhten Fehlschlagsrisiko und den nicht unmittelbar produktiven Aufwendungen für Performanzmodelle ist eine kaufmännisch vertretbare Abwägung durchzuführen. Die strukturellen Abhängigkeiten solcher Managemententscheidungen werden im nächsten Kapitel diskutiert. Abbildung 6.2.8 zeigt die Verwendung von existierendem Code für die Entwicklung von Lastmodell-Codeproben. Existierender Code aus einer vorhergegangenen Iteration ist ein wichtiger Anhaltspunkt für den Implementierer, der es erlaubt, spezifischere Modelle zu schaffen, die bezüglich der interessanten Aspekte genauere Aussagen erlauben. Die Verfügbarkeit von echten Codeproben aus einer vorhergehenden Iteration bedeutet dabei nicht notwendig eine Arbeitszeitersparnis.

Die Ressourcenmodelle müssen nach einer Veränderung der Lastmodelle, adaptiert und gegebenenfalls rekalibriert werden. Der Prozeß geht dabei unverändert nach dem in Abbildung 6.2.7 dargestellten Schema

vor. Die Ausgangsartefakte sind jeweils in der neuesten Version gemäß des Iterationsschritts zu verwenden. Organisatorisch handelt es sich zwar um eine fortschreitende Versionierung, aber die Kontextmodelle sind vollständig unabhängig. Lediglich eine Analyse der Veränderungen längs ihrer geschichtlichen Entwicklung wäre interessant, womit sich diese Arbeit aus Platzgründen nicht beschäftigen kann.

6.3. Rollen

Neben neuen Artefakten wurde vor allem auch von neuen Rollen gesprochen, die eine besondere Beachtung verdienen. Folgend werden verwandte Tätigkeitsbereiche zu Rollen zusammengefaßt und fachliche Anforderungsprofile skizziert.

6.3.1. Klassische Rollen. Mitarbeiter in klassischen Rollen übernehmen in der Entwicklung performanzkritischer Software gegenüber Mitarbeitern in neuen Rollen häufig eine interne *Kundenfunktion*. Oft überschneiden sich fachliche Anforderungsprofile, sodaß sie mit neuen Rollen verschmolzen werden können. Darauf wird im Folgenden gesondert eingegangen. Bei manchen Tätigkeiten, wie etwa der Entwicklung und Durchführung von Performanz-Tests, ist aber auch eine personelle Trennung begründet. Die Aspekte der Kundenfunktion überwiegen bei Modellierern und Analytikern. Bei Systemtechnikern und Implementierern nach dem klassischen Rollenverständnis dagegen überwiegt eher der Kooperationscharakter. Um diese Aspekte zu beleuchten, werden folgend die neuen Rollenbilder umrissen.

6.3.2. Neue Rollen im Hauptprozeßstrang.

6.3.2.1. *Performanz Analytiker.* In chronologischer Betrachtung erscheint zuerst der Performanz-Analytiker. Er muß die Beachtung performanzrelevanter Fragestellungen von Anfang an unterstützen und durchführen. Dazu sollte er auch bei Kundengesprächen und Interviews anwesend sein. Seine Aufgabe ist es, performanzrelevante Sachverhalte zu identifizieren und durch zusätzliche Fragen wichtige primäre Informationen gewinnen. Er beobachtet die Integration der identifizierten Sachverhalte in die Geschäftsprozeßanalyse und die Analysemodelle durch den Analytiker. Gegebenenfalls muß er in der Lage sein, antagonistische Forderungen des Kunden aufzuzeigen und zu visualisieren.

Durch eigene Reviews der in der Analyse entstehenden Artefakte muß er garantieren, daß Performanzanforderungen und alle relevanten Aspekte in den Artefakten bewahrt werden. Alle Analyse-Dokumente müssen durch den Performanz-Analytiker abgenommen werden. In Abbildung 6.3.1 werden die Tätigkeiten schematisch aufgezeigt.

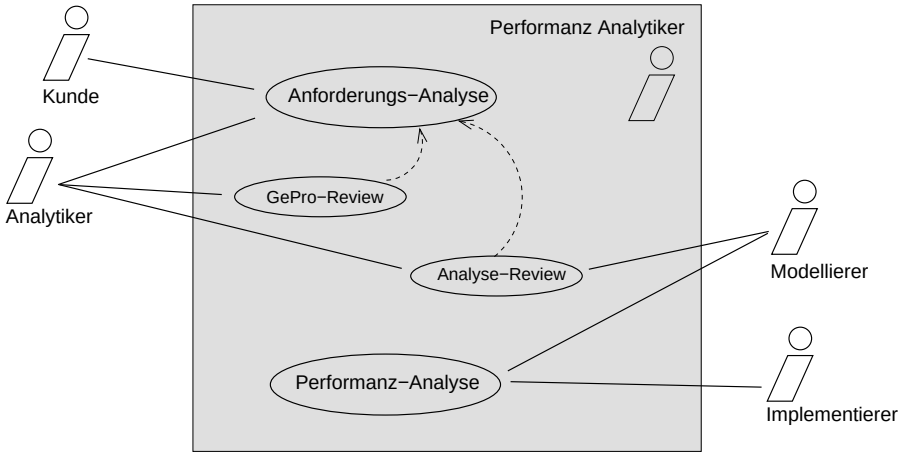


Abbildung 6.3.1. Der Performanz-Analytiker

Vom Performanz-Analytiker werden soziale Kompetenz, angenehmer Umgang mit Menschen und besonderes psychologisches und sprachliches Geschick im Gestalten von Interviews erwartet. Er benötigt essentielle Kenntnisse des Ziel-Geschäftsbereiches, der eingesetzten Modellierungstechniken und Formalismen. Außerdem muß er mit der hier beschriebenen Bewertungstechnik gut vertraut sein. Er benötigt, um vorausschauend analysieren zu können, profunde Kenntnisse in Hardware, Implementierungstechniken, Laufzeitkomplexität von Algorithmen und umfangreiche Erfahrung zum zeitlichen Verhalten von Systemen. Wie oft am Anfang eines Prozesses sind die Anforderungen an fachliche Qualifikation überdurchschnittlich hoch, da jetzt Fehler begründet werden können, welche die Kosten aller folgenden Fehler deutlich übersteigen.

6.3.2.2. *Performanz-Modellierer.* Ressourcen- und Performanz-Modellierer sollten eng zusammen arbeiten, haben viele Überschneidungen in ihren fachlichen Anforderungen, sodaß diese beiden Rollen verschmolzen werden können.

In Abbildung 6.3.2 sind beide Rollen zusammengelegt. Der Performanz-Modellierer extrahiert aus den dynamischen Modellen des Entwurfes den Kontrollfluß, eliminiert rein funktionale Annotationen und reichert das entstehende Modell um performanzrelevante Annotationen an. Das so entstehende dynamische Performanz-Modell muß dem Entwurf äquivalent sein und unmittelbar für die Simulation verwendet werden können. Deshalb sind hier bereits die Allokation der Lasten auf

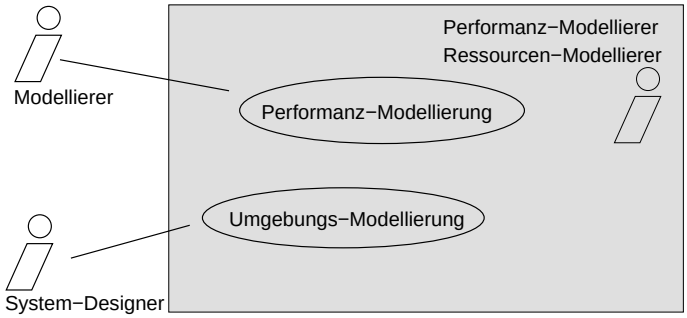


ABBILDUNG 6.3.2. Der Performanz-Modellierer

Ressourcen durchzuführen, und es ist das Umgebungsmodell als bekannt vorauszusetzen. Der Ressourcen-Modellierer baut seine Artefakte ebenso auf Analyse- und Entwurfs-Dokumente auf, entnimmt aber andere Aspekte und muß seine Arbeit bereits beendet haben, bevor der Performanz-Modellierer beginnt. Auch diese Abhängigkeit erlaubt es, beide Rollen zu verschmelzen, da es hier zu keinen Engpässen bzw. Überforderungen in der Arbeitsbelastung kommt.

Performanz- und Ressourcen-Modellierer müssen nicht bei Kundengesprächen anwesend sein. Ihre fachlichen Anforderungen sind aber dennoch hoch. Es müssen profunde Kenntnisse in Hardwaretechnik, zeitlichem Verhalten von Systemen vorhanden sein und das Wissen, wie Entwürfe implementiert werden würden.

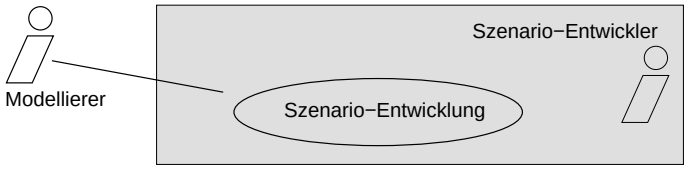


ABBILDUNG 6.3.3. Der Szenario-Entwickler

6.3.2.3. *Szenario-Entwickler.* Szenarien sind Testfälle der Performanz. Es ist von Vorteil, den Entwickler dieser Testfälle auch personell von den Entwicklern der Software-Modelle zu trennen. Ein Szenarien-Entwickler sollte sich unbeeinflußt von den Entwurfsideen ein eigenes Verständnis der dynamischen Strukturen machen. Darauf aufbauend, sollen die Szenarien Testfälle darstellen, die Standard- oder auch Ausnahmefälle und Extremsituationen simulieren. Der Entwickler bezieht in seine Testfall-Entwicklung die vorhandenen Artefakte ein, je nachdem, ob es sich um Black- oder Whitebox-Testvorgehen handelt.

In Abbildung 6.3.3 ist eine Beteiligung des Modellierers dargestellt, die sich aber ausdrücklich nur auf eine nachrangige Referenzfunktion

beschränkt. Der Modellierer sollte nur bei einer zusätzlichen Garnitur an Szenarien unmittelbar mit in die Entwicklung einbezogen werden.

Der Szenario-Entwickler sollte Modellierungskenntnisse besitzen und ein guter Implementierer sein, damit er typische Grenzfälle identifizieren kann. Theoretische Kenntnisse über das Testen und das Ableiten von Testfällen aus dem Kontrollflußgraphen des dynamischen Modells müssen beherrscht werden. Ebenso ist eine selbstständige Einschätzung des Anwendungsgebietes der entstehenden Software von Vorteil. Dadurch kann eine zweite Meinung zu den kritischen Grenzfällen weitere Erkenntnisse bringen. Trotzdem ist es sinnvoll, dem Szenario-Entwickler keine intensive Einbindung in den kreativen Entwicklungsprozeß zu gewähren.

ABBILDUNG 6.3.4. Der Ressourcen-Entwickler

6.3.2.4. *Ressourcen-Entwickler.* Ähnlich wie beim dynamischen Modell entspricht das Performanz-Umgebungsmodell bis auf die Anotationen dem Umgebungsmodell aus dem klassischen Prozeß. Die Anotationen, die über funktionale Eigenschaften Auskunft geben, müssen durch performanzrelevante Annotationen ersetzt werden. Der Systementwickler muß dem Performanz-Umgebungsentwickler dazu Auskünfte geben. Diese können in der ersten Iteration aus Erfahrungswerten bestehen oder später aus Performanz-Messungen entnommen werden. Abbildung 6.3.4 zeigt als einzigen Ansprechpartner den Systementwickler. Tatsächlich ist der Ressourcenentwickler aber ein entscheidendes Bindeglied zwischen Performanzmodellierer, Systementwickler und dem Simulationsingenieur

6.3.3. Neue Rollen im Kontextprozeßstrang. Die neuen Artefakte werden im Kontextprozeß erstellt. Eine Aufteilung in mehrere Rollen mit spezifischen Anforderungen wäre zwar möglich, es ist aber auch wegen des engen Zusammenhangs von Lasten und Ressourcen, eine Zusammenfassung zu einer Rolle des Simulationsingenieurs begründbar.

6.3.3.1. *Simulationsingenieur.* Seine Aufgabe besteht in der Entwicklung von Lastmodellen und den zugehörigen Beispielimplementierungen. Diese Codebeispiele dienen als Referenzmodelle. Ist eine Last

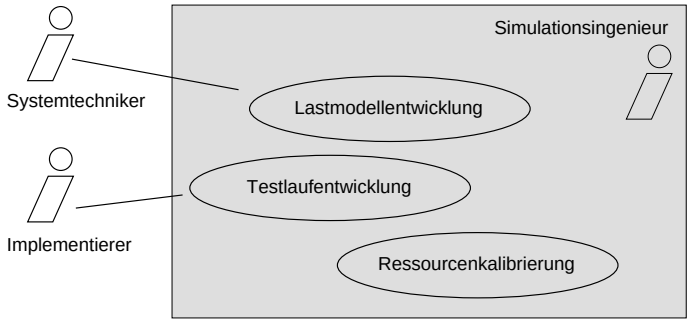


ABBILDUNG 6.3.5. Der Simulationsingenieur

beispielsweise einfach parametrisiert, so müssen den verschiedenen Parameterwerten jeweils Codebeispiele zugeordnet werden, die den Wert des Parameters auch quantitativ repräsentieren. Fehler, die hier in der Zuordnung der Referenzmodelle gemacht werden, setzen sich in der Bewertung fort. Aus der Anforderungsanalyse muß extrahiert werden, in welchen Wertebereichen der jeweiligen Parameter bei der Durchführung der Simulation Bewertungen durchgeführt werden. In diesen Bereichen müssen durch Laufzeitmessungen mit den Codebeispielen Stützstellen für eine Interpolation geschaffen werden. Auch die Wahl einer Interpolationsfunktion, die dem Modellverhalten gerecht wird, ist Bestandteil der Modellierung. Im Aufgabenbereich des Simulationsingenieurs ist es auch, die nötige Dichte der Stützstellen festzulegen. Abbildung 6.3.5 zeigt, neben der Zusammenarbeit mit einem Implementierer, den Systemtechniker als Kooperationspartner. Durch zahlreiche Serien von Laufzeitmessungen werden Meßwert-Matrizen mit Stützstellen aufgefüllt. Diese Tabellen dienen dann den Ressourcenmodellen als Grundlage für die zeitliche Bewertung der Lastbearbeitung in den verschiedensten Situationen.

Da die Artefakte des Kontextprozesses nur dem Betrieb des Simulationswerkzeugs dienen, sind alle Aufwendungen zu deren Erstellung mit Kosten verbunden, die nicht unmittelbar den funktionalen Eigenschaften der entstehenden Software zugute kommen. Falls Performanzeigenschaften wichtige Anforderungen an die Software selbst sind, können diese Aufwendungen mittelbar gegen einen geldwerten Vorteil aufgerechnet werden. Ist Performanz aber nur ein Risikofaktor, der das Scheitern des Softwareprozesses betrifft, sind die oft nicht unerheblichen Kosten für eine Performanzanalyse im Zusammenhang mit dem Gesamtrisiko des Projektes zu sehen. Aufwendungen für eine vollständige Performanzanalyse können nur durch ein erheblich größeres

Gesamtrisiko des Prozesses gerechtfertigt werden. Im nächsten Kapitel werden die wichtigsten Abhängigkeiten bei einer Risikobewertung diskutiert.

6.3.4. Zusammenfassung. In diesem Kapitel wurde skizziert, welche zusätzlichen Artefakte die in dieser Arbeit vorgestellte Bewertungstechnik erfordert und wie diese erstellt werden. Neue Rollen im Software-Entwicklungsprozeß wurden entwickelt und Aktivitäten in einen klassischen Prozeß, der Analyse, Entwurf, Implementierung und Test unterscheidet, integriert. Wichtige Abhängigkeiten zwischen den unterschiedlichen Artefakten wurden diskutiert. Wichtigstes Ergebnis ist die Möglichkeit, den Prozeßablauf nach der Erstellung einer Performanzanalyse zu unterbrechen und den Implementierungsentwurf in einem Mikrozyklus so lange zu iterieren, bis die Ergebnisse der Performanzanalyse eine größere Erfolgswahrscheinlichkeit versprechen. Auch die Möglichkeit den gesamten Prozeß zu stoppen, wenn sich andeutet, daß das Projekt nicht erfolgreich sein kann, besteht an dieser Stelle. Im ersten Fall besteht der Nutzen dieser Methodik in den eingesparten Aufwendungen für die Implementations- und Testphase, die nicht in die Microzyklen der folgenden Iterationen eingeschlossen werden mußten. Im letzteren Fall ist der Nutzen durch die Menge der Kosten bewertbar, die nicht zusätzlich sinnlos aufgewendet wurden, bis der Fehlschlag des Projektes in den Tests des implementierten Systems festgestellt werden kann.

Performanzmodellierung ist kostenintensiv. Artefakte, die zur Risikobewertung angefertigt werden, tragen zur Problembewältigung eines Projektes nicht bei. Somit ist die Performanzmodellierung an sich ein Kostenrisiko und muß durch ein Fehlschlagsrisiko gerechtfertigt werden. Prozeßkennzahlen, Fehlschlagsrisiko und Quantifizierung von Kosten für die Einführung einer Technik zu bewerten, ist problematisch. Da sie aus Erfahrungen vergangener Prozesse ermittelt werden und die Struktur kommender Prozesse beeinflussen soll, ist eine Einflußnahme dieser Parameter auf den Prozeß nur möglich, falls dieser im höchsten Reifegrad betrieben wird. (Vergleiche CMM - Level 5, in [**28**, Dymond]). Da es in ganz Europa nur wenige Firmen gibt, welche diesen Reifegrad erreichen, liegt eine wissenschaftliche Untersuchung, die auf echten Maßen basiert, in weiter Ferne. In Kapitel 8 werden dennoch grundlegende Zusammenhänge einer ausgewählte Risikosituation beispielhaft skizziert.

Entwicklung und Bewertung von Software-Modellen in der Praxis

Im ersten Teil dieser Arbeit wurden Techniken zur Spezifikation von dynamischen Performanz-Modellen, Szenarien und virtuellen Simulationsumgebungen vorgestellt. Im zweiten Teil wurde eine Methodik zur Anwendung und die relevanten Begriffe diskutiert. Die Entwicklung der Technik verwendet formale Methoden nicht unmittelbar, um Ergebnisse zu erzielen, sondern um die Präzision der Spezifikation zu erhöhen. Erwünschte technische Eigenschaften wurden auf formaler Ebene gezeigt. Die Spezifikation sollte eine technische Umsetzung in ein Werkzeug mit minimalem Aufwand (dh. einer minimalen Anzahl an Iterationen) und mit maximalem Erfolg (dh. umfangreicher Funktionalität und hervorragenden Stabilitätseigenschaften) ermöglichen. Schließlich sollte die Ausdruckskraft der entwickelten Modellierungstechnik in aktuelle Technologien integrierbar sein und mit dem Prozeß einer Software-Entwicklung nach dem Stand der Kunst harmonieren.

Als Beispiel für die Anwendbarkeit auf aktuelle Modellierungstechniken wird der Einsatz der Modellierungssprache UML mit der hier vorgestellten Technologie begleitet. Im Abschnitt 7.1 werden Vorschläge zur Integration von Dokumenten der UML Version 1.X vorgestellt. Zur Umsetzung dieses Vorschlages wird in Abschnitt 7.2 eine komponentenbasierte Software-Architektur entwickelt. Die zu erwartende Paßgenauigkeit der Komponenten wird durch die Kompatibilität der einzelnen Spezifikationen erzielt. Einzelne Abbildungen dokumentieren die Ausgaben des am Institut entwickelten Werkzeugs |SEMPER|. Dieses Werkzeug stellt eine stabile Software dar, welche die Anwendung aller vorgestellten Techniken und die Auswertung der Ergebnisse auf komfortablem graphischem Weg erlaubt. Die Applikation wurde in nur einer Iteration, basierend auf den Ergebnisse dieser Arbeit, entwickelt. Durch einen kleines Einsatzbeispiel in Abschnitt 7.3 wird die Funktionsfähigkeit des Werkzeugs und damit der Erfolg dieser Arbeit prototypisch belegt.

KAPITEL 7

Betrachtungen zur Anwendbarkeit

7.1. UML - Eine Anwendung

Schon in der Einleitung wurden Aktivitätsdiagramme der UML verwendet, um verschiedne Fragestellungen zu formulieren. Die weite Verbreitung der UML auch in der Industrie legt eine Anwendung der hier entwickelten Beschreibungsmöglichkeit für dynamische Modelle auf die UML nahe. Da sich Aktivitätsdiagramme zur Beschreibung von implementierungsnahen Modellen erfahrungsgemäß größerer Beliebtheit erfreuen als Zustandsautomaten, waren diese die Notation der Wahl. In den UML Versionen 1.X laden Aktivitätsdiagramme aufgrund der freizügigen Verwendbarkeit der syntaktischen Elemente dazu ein, Modelle zu entwickeln, die auf höherer Abstraktionsebene betrachtet mehrdeutige Interpretationen zulassen. Dies betrifft erfahrungsgemäß vor allen die Wohleinbettung nebenläufiger Regionen. Bei der Verwendung von Zustandsmaschinen wird der Modellierer durch *Concurrent States* besser in seinem Entwurf geleitet. So lassen sich an Aktivitätsdiagrammen die Vorteile der hier eingeführten dynamischen Modelle eindrucksvoll zeigen.

Gegenwärtig wird die Version 2.0 der UML in Literatur und in der Praxis eingeführt. In Abschnitt 2.1.3 wurden bereits Problemfälle angedeutet, die ihre Ursache in der unausgereiften Semantik der Aktivitätsdiagramme nach der Lesart von UML 2.0 haben. In Zukunft kann man dort eine deutliche Nachbesserung (etwa zum Tokenbegriff) erwarten. In weiten Bereichen der Industrie wurde auch aus diesen Gründen die Entscheidung getroffen, die UML 1.X Version weiterhin zu verwenden.

In dieser Anwendungsstudie steht daher die UML 1.X im Mittelpunkt. Sollte der Tokenbegriff in Zukunft besser spezifiziert werden, ist die Erweiterung dieses Ansatzes auf die UML 2.0 und dort neu eingeführter Begriffe, wie beispielsweise die Ausnahmebehandlung, ein interessanter Zukunftsaspekt.

7.1.1. Aktivitätsdiagramme. Das in Abschnitt 3.3 eingeführte Kartuschenbeispiel, hier noch einmal als Abbildung 7.1.1 dargestellt,

zeigt alle definierten syntaktischen Elemente und eignet sich dadurch als Demonstrationsobjekt.

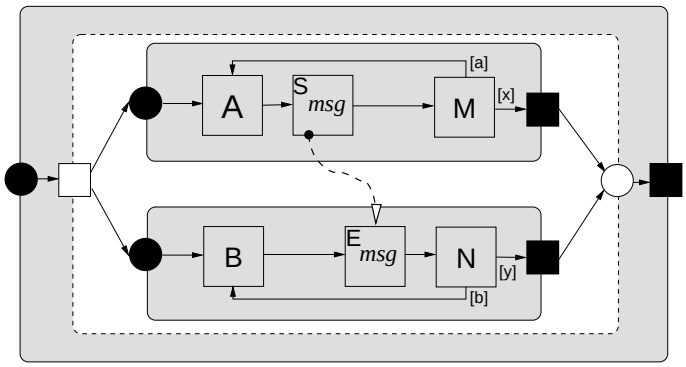


ABBILDUNG 7.1.1. Beispiel mit Kommunikation

Das UML-Aktivitätsdiagramm, welches die gleiche Semantik hat, ist intuitiv leicht zu bestimmen. Abbildung 7.1.2 zeigt einen Vorschlag als Diagramm.

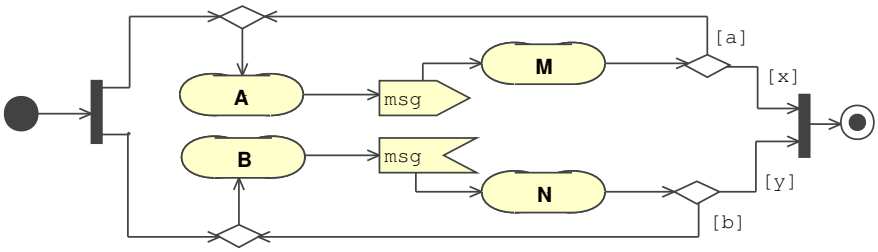


ABBILDUNG 7.1.2. UML-Ursprungsmodell

Die *Decision*- bzw. *Merge*-Knoten werden oft weggelassen. Die Pentagon für das Senden und Empfangen visualisieren nach der UML Spezifikation nur einen Ereignis, bzw. einen Auslöser einer Transition. Somit bilden nach der UML-Spezifikation das Symbol, die eingehende und die ausgehende Kante eine erweiterte Darstellung einer einzigen Kante. Die Trennung der beiden nebenläufigen Bereiche muß sich der Betrachter im UML Diagramm selbst dazudenken. In der hier gewählten Darstellung wurde dies durch ein günstiges Diagrammlayout erleichtert.

Folgend sind die einzelnen syntaktischen Elemente dem dynamischen Performanzmodell zugeordnet.

Die Übersetzung der in Tabelle 1 dargestellten Last- und Datenzustände bedürfen keiner weiteren Erläuterung. Es sei nur darauf hingewiesen, daß eine Übersetzung nur dann erfolgen muß, falls die Zustände

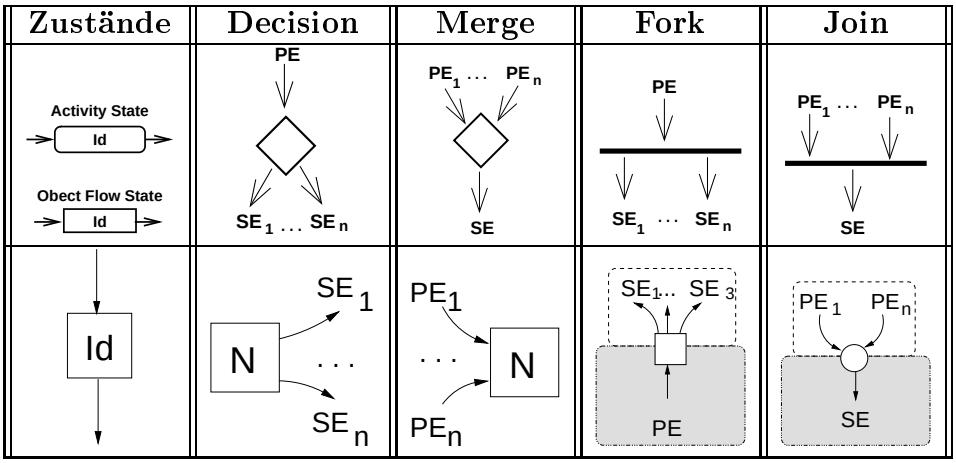


TABELLE 1. Übersetzungstabelle 1. Teil

im UML-Diagramm als Zeit-relevante Lasten ausgewiesen sind. Wie das geschieht, wird im Abschnitt 7.1.2, bei den Performanz-Annotationen erläutert. Die eingehende Kante eines *Decision*-Zustand und der Zustand selbst werden reduziert und die mit Wächtern bewehrten Transitionen an das vorhergehende Element angefügt, das in der Tabelle mit der Last `N` angedeutet ist. Der *Merge*-Zustand wird analog behandelt.

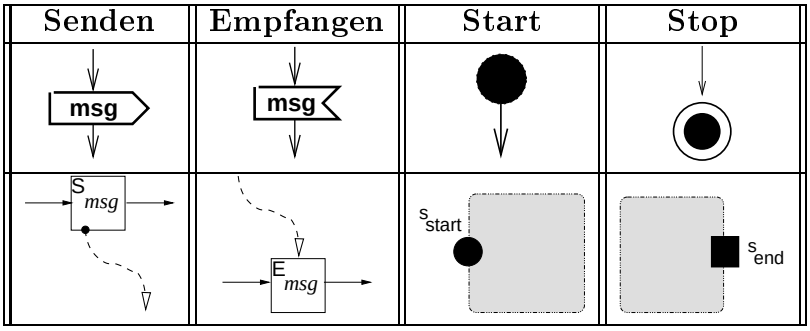


TABELLE 2. Übersetzungstabelle 2. Teil

Ein *Fork*-Knoten verläßt eine Sequenz-Kartusche und tritt in eine *eingebettete* Parallel-Kartusche ein. Dadurch wird das Betreten einer nebenläufigen Region formuliert. An dem korrespondierenden *Join*-Knoten verläßt man diese Parallel-Kartusche und tritt wieder in die übergeordnete Sequenz-Kartusche ein.

Sollte sich ein Kontrollfluß in Folge nicht mehr isoliert vereinen, kann als ein Sonderfall ein *Fork*-Knoten auch eine Sequenz-Kartusche endgültig schließen. Dazu wird auf das Kapitel 2 über dynamische Modelle verwiesen.

Das Senden und Empfangen einer Nachricht kann in der UML auch durch einen Datenfluß mit einem Objektzustand spezifiziert werden. Diesen als Last zu betrachten, ist in der derzeitigen Version noch nicht vorgesehen. In Tabelle 2 wird nur das Pentagon der UML dargestellt. Der durch die gestrichelte Linie im Performanz-Modell dargestellte Zusammenhang ergibt sich aus der Namensgleichheit der Nachricht. *Start*- und *Stop*-Zustand eines Aktivitätsdiagramms werden in die Eintritts- und Austrittspunkte der Wurzelkartusche übersetzt.

7.1.2. Performanz-Annotationen. Um Aktivitätsdiagramme aus allen Phasen und Iterationen eines Softwareprozesses unverändert übernehmen zu können, ist es sinnvoll, keine vordefinierten Elemente des Metamodells für diese Simulation zu verwenden. Es werden *Tagged Values* zur Spezifikation verwendet, das sind Werte, die flexibel in einem Diagramm mit einem Bezeichner spezifiziert werden können. Jedem Bezeichner wird ein Wert in Form einer Zeichenkette beigefügt. Für Aktivitätsdiagramme sind folgende Bezeichner mit spezifischen Bedeutungen belegt:

- `StartL`: Name der Performanz-Lastmodell-Instanz
- `TypL`: Typ der Lastinstanz
- `ParamL`: Parameterstring
- `LocR`: Ressource auf der die Last ausgeführt wird
- `Cond`: Name eines Wächters

Der Parameterstring selbst ist nach folgendem Schema aufgebaut: `ParamL : PName "=" PWert`. `PName` und `PWert` sind dabei Zeichenketten. Dabei muß `PWert` zusätzlich je nach Ressource gültig interpretierbar sein, dh. zum Beispiel als ganze Zahl ausgewertet werden können.

In Abbildung 7.1.3 ist ein Aktivitätsdiagramm dargestellt. Ein Aktivitätszustand und zwei Transitionen sind mit Annotationen ausgestattet, die während der Performanz-Simulation im Werkzeug interpretiert werden. Die Werte für den `Cond`-Wert aller von einem Zustand ausgehenden Transitionen müssen natürlich eindeutig sein. Die Last `Funktion_X`, so der Name im Modell, trägt für die Simulationsumgebung den Namen `meinJob` und ist vom Typ `exec_meth_01`. Ein Parameter namens `Groesse` ist für diesen Lasttyp vorgesehen und erwartet einen ganzzahligen Wert. Der hier spezifizierte Wert 10 sollte dabei im

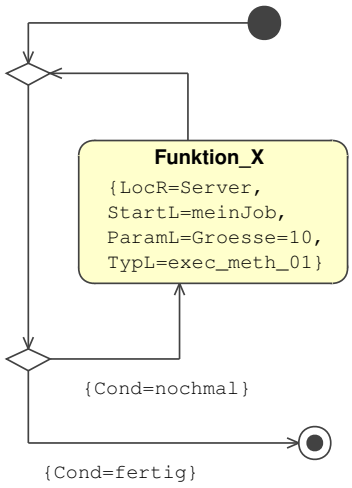


ABBILDUNG 7.1.3. Annotiertes Aktivitätsdiagramm

Gültigkeitsbereich dieses Lastmodells liegen. Diese Last soll während der Simulation von einer Ressource namens `Server` bearbeitet werden.

Die beiden `Cond`-Werte, `fertig` und `nochmal`, sind die Zeichenketten mit denen der Testfall-Ablauf, also das Szenario, spezifiziert werden kann.

7.1.3. Testfall-Spezifikation. Als Spezifikation für Einzelabläufe bietet sich in der UML das Sequenzdiagramm oder das Kollaborationsdiagramm an. Zur Simulation ist aber lediglich die Auflösung der Konflikte an den *Decision*-Knoten interessant. Beide UML-Diagrammtypen sind also deutlich zu aufwendig, weshalb hier auf die Verwendung einer UML-Quelle verzichtet wird. Die Speicherung eines Testfalls wird in einem XML-Format durchgeführt, das geeignet ist, `Cond`-Sequenzen nach einer in Kapitel 3 eingeführten Spurzerlegung zu speichern.

7.1.4. Implementationsdiagramme. Für die Beschreibung der Simulationsumgebung sind `Deployment-` oder Implementationsdiagramme hervorragend geeignet. Zur Beschreibung einer Ressource kann beispielsweise eine Knoteninstanz verwendet werden. Die Assoziationen bleiben uninterpretiert, ebenso, analog zu der Entscheidung für die Aktivitätsdiagramme, alle UML-spezifischen Attribute. Zur Interpretation während der Simulation werden wieder `Tagged Values` verwendet.
Für Implementationsdiagramme sind folgende Bezeichner mit spezifischen Bedeutungen belegt:

- `LocR`: Name einer Ressource
- `TypR`: Typ einer Ressource

Durch die Übereinstimmung der `LocR`-Werte in Aktivitäts- und Implementationsdiagramm wird die Last der entsprechenden Ressource zugeordnet.

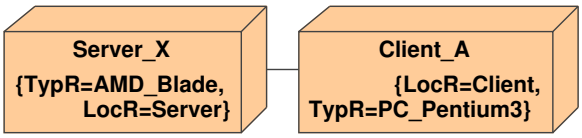


ABBILDUNG 7.1.4. Annotiertes Implementationsdiagramm

In Abbildung 7.1.4 ist beispielhaft ein Server und ein Client spezifiziert. Im Modell, das funktionale Aspekte beschreibt, hat der Server den Namen `Server_X`. Für die Simulationsumgebung ist seine Bezeichnung `Server`. Der für den Typ `AMD_Blade` muß natürlich in der Simulationsumgebung eine Ressource spezifiziert und kalibriert sein. Analog sind die Namen für den Client-Knoten vom Namen, den die Simulation verwendet, unabhängig.

7.2. Werkzeugimplementation

Eine Werkzeugstudie sollte alle Aspekte umfassen, die in dieser Arbeit angeschnitten wurden. Daraus ergeben sich einige Vorgaben für die Auswahl der Technologien und Architektur. Der Zugriff auf UML-Dokumente, die durch eine große Bandbreite an kommerziellen Werkzeugen bearbeitet werden können, legt die Verwendung von XML-Formaten nahe. Die meisten dieser Werkzeuge verwenden selbst den von der OMG vorgeschlagenen Standard XMI, siehe [**50**]. In der Einleitung wurde die Lösung über eine formale Semantik, die die Petrinetz-Theorie als semantische Domäne verwendet, mit der unmittelbaren Implementierbarkeit als vorteilhaft beworben. Eine unmittelbare Implementierung der Semantik als Petrinetz-Maschine liegt also nahe. Im Kapitel 6 wurden außerdem die Rollen und Artefakte beschrieben, welche die Integration dieser Technik in eine klassische Softwareentwicklungs-Methode erlauben. Ein Werkzeug sollte ein Rollenverständnis von Benutzern mit unterschiedlichen Aufgaben unterstützen. Funktionalität und Zugriff auf Dokumente und Artefakte in diesem Prozeß sollten durch das Werkzeug sinnvoll bereitgestellt werden.

7.2.1. Architektur. Am Institut für Informatik der Universität München wurde dazu |SEMPER|, eine Werkzeugstudie entwickelt. Das Werzeug soll dynamische Modelle aus UML-Diagrammen erzeugen können. Es soll die Entwicklung von Szenarien, Last und Ressourcenmodellen unterstützen und schließlich Modelle in einer Simulation bewerten. Die so gewonnen Ergebnisse sollten mit dem Werkzeug graphisch aufbereitet werden können. Die Vorgaben, die für die Entwicklung einer Plattform und deren Architektur abgeleitet werden können, werden folgend vorgestellt.

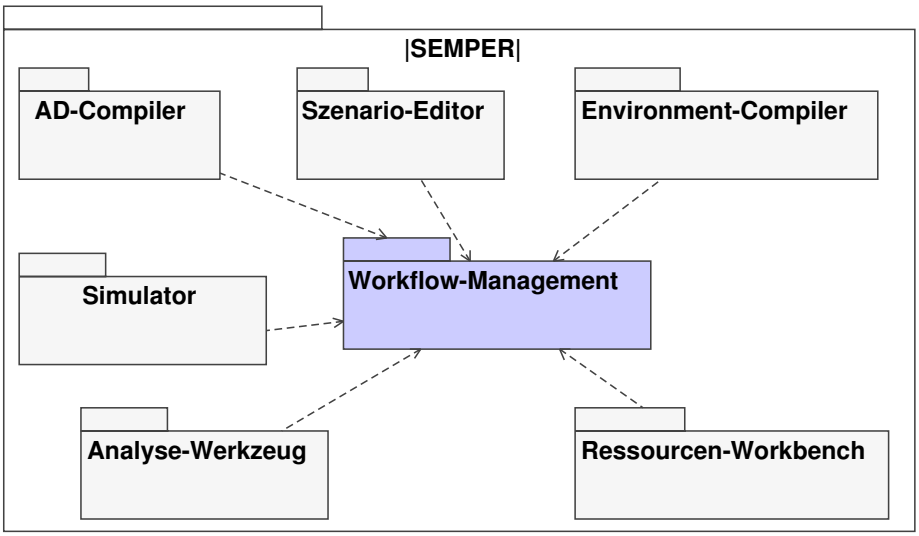


ABBILDUNG 7.2.1. Architektur mit Einzelkomponenten

Der statische Architekturanteil läßt sich praktisch aus den Kapiteln der einzelnen Bestandteile einer Simulation ableiten. Ein *Aktivitätsdiagramm-Compiler* ermöglicht die Erzeugung dynamischer Performanzmodelle. Ohne Anwendung in der UML können in einem *Szenario-Editor* Testfall-Abläufe entwickelt werden. Die Simulationsumgebung, bestehend aus den Instanzen der Ressourcen und Lastmodelle, wird durch einen *Environment-Compiler* aus den UML-Deploymentdiagrammen erstellt. Eine weitere Komponente, der *Simulator*, muß die Simulation eines einzelnen Ablaufes für ein Modell in einer spezifischen Umgebung durchführen, also die Bewertungssemantik implementieren und berechnen. Diese Ergebnisse müssen, wie in Kapitel 6 gefordert, in anschaulichen Analyse-Dokumenten (möglichst graphisch ansprechend) aufbereitet werden. Dafür ist ein *Analyse-Werkzeug* erforderlich, das durch eine eigene Komponente realisiert

wird. Die im Kapitel 4 auf abstrakter Ebene beschriebenen Funktionen, die eine Ressource modellieren, müssen nun konkret vorgegeben werden können. Dies wird durch eine *Ressourcen-Workbench* realisiert. Schließlich sollte das Zusammenspiel dieser Komponenten im Sinne eines geordneten Prozeßablaufs durch ein *Workflow-Management* realisiert werden. Abbildung 7.2.1 zeigt diese Komponenten im Überblick.

7.2.1.1. *Aktivitätsdiagramm-Compiler.* Zur Erstellung von dynamischen Modellen steht aus den Ergebnissen dieser Arbeit eine graphische Modellierungssprache zur Verfügung. Eine Komponente zur Handhabung dynamischer Modelle kann also einen graphischen Editor für dynamische Performanz-Modelle anbieten. Die Verwendbarkeit eines Werkzeugs steht und fällt mit der Integrierbarkeit in eine bestehende Werkzeuglandschaft. Somit ist es unumgänglich, die Artefakte der zahlreichen im Handel befindlichen UML-Modellierungswerkzeug bearbeiten zu können. Die Komponente für dynamische Performanz-Modelle benötigt eine Import-Schnittstelle für UML-Aktivitätsdiagramme. Diese müssen in das Performanz-Modell übersetzt werden können, falls sie den strukturellen Anforderungen entsprechen.

Da großer Aufwand zur Definition einer Petrinetz-Semantik betrieben wurde, liegt es nahe, die Übersetzung in die semantische Domäne früh durchzuführen und das erhaltene Netz in einem geeigneten Format zu speichern. Vor einigen Jahren hat sich durch die von der Deutschen Forschungsgemeinschaft geförderten Arbeiten der Humboldt Universität zu Berlin der XML Standard *PNML* (siehe [**20, 63**]) zur Verarbeitung von Petrinetzen durchgesetzt. Im Rahmen dieser Arbeiten wurde eine frei verfügbare Version eines Netzsimulators entwickelt [**52, 64**], die sich zur Verwendung anbietet. Die geschickte Wahl eines Petrinetzes als semantische Domäne erlaubt die unmittelbare Implementierung der Semantik. Der Nachweis einer korrekten Implementierung wird dadurch erheblich vereinfacht. Selbst die Verwendung bereits vorhandener Implementationen ist möglich, von denen das Werkzeug durch die weit fortgeschrittene Standardisierung unabhängig bleibt.

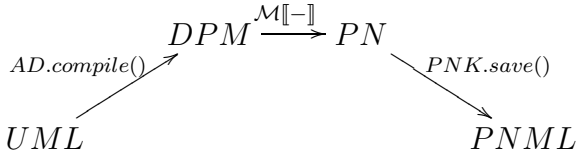


ABBILDUNG 7.2.2. Übersetzungsfunktionen.

Der Compiler selbst funktioniert nach dem Schema aus Abbildung 7.2.2. Der *Compiler* übersetzt die Elemente des Aktivitätsdiagramms (UML) in das dynamische Performanz-Modell (DPM), dem Schema der Übersetzungstabellen 1 und 2 folgend. Mit einer Implementierung der semantischen Funktion $\mathcal{M}[\![-]\!]$, die in Kapitel 2 eingeführt wurde, wird das Petrinetz erzeugt und durch den Petrinetzkernel der Humboldt Universität (PNK) in PNML gespeichert.

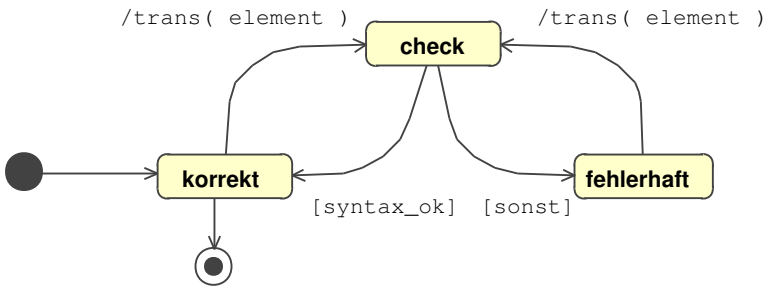


ABBILDUNG 7.2.3. Zustandssystem für den Compiler

Der Übersetzungsvorgang selbst ist ausführlich in den technischen Berichten zu der Werkzeugimplementierung dokumentiert [**48, 49**]. Nach grundlegendem Schema erfolgt die Übersetzung der Darstellung in Abbildung 7.2.3. Elementweise wird die Übersetzung durchgeführt. Eine Tabelle an Wohlgeformtheitsbedingungen formuliert die durch die Definitionen in Kapitel 2 gegebenen Einschränkungen. Die Auswertung aller Wohlgeformtheitsbedingungen über das gesamte bislang übersetzte Modell erlaubt, die korrekte Syntax zu prüfen. Sind keine weiteren Elemente mehr zu übersetzen, befindet sich der Compiler in einem der beiden Zustände *korrekt* oder *fehlerhaft*. Je nachdem wird das UML-Modell als korrekt angenommen oder verworfen.

7.2.1.2. *Szenarioeditor.* Sequenzdiagramme und Kollaborationsdiagramme werden im klassischen Software-Entwicklungsprozeß verwendet, um typische und kritische Einzelfälle dynamischer Abläufe in Hinsicht auf die funktionalen Eigenschaften der Software zu spezifizieren. Diese Einzelfälle sind meistens von den typischen und kritischen Einzelfällen in Hinsicht auf die *nicht*-funktionalen Eigenschaften grundlegend verschieden. Sequenzdiagramme aus dem klassischen Prozeß können also *nicht* durch einfache Annotationen unverändert in die Performanz-Analyse übernommen werden. Eigene Sequenzdiagramme für die Ablaufspezifikation zu entwickeln, scheint nicht sinnvoll, da die

Auflösung von Konflikten an Verzweigungspunkten nur ein verschwinden kleiner Bestandteil dieses UML-Diagrammtyps und dadurch ungeeignet ist.

Ein Szenarioeditor soll dem Performanz-Tester möglichst eine intuitiv verständliche und effiziente Möglichkeit geben, einzelne Ablaufe eines dynamischen Modells auszuwählen und zu speichern. Diese Form soll dann unmittelbar zur Steuerung eines Simulationslaufs (nach Kapitel 3) verwendet werden können. Da ein einmal gespeichertes Szenario möglicherweise auf ein bis zur Durchführung der Simulation modifiziertes Modell trifft, soll eine Funktion die Kompatibilität mit einem dynamischen Modell[1] feststellen können. Falls dynamische Modelle Kommunikations- und Synchronisationszustände verwenden, soll außerdem vor einem Simulationslauf entschieden werden können, ob eine Verklemmung vorliegt. Die entscheidenden Aufgaben sind also:

(1) Interaktion mit einer graphischen Modellrepräsentation
(2) Strukturierte Auswahlmöglichkeit
(3) Persistente Form von Szenarien
(4) Implementierung eines Simulationsinterfaces
(5) Implementierung eines Kompatibilitätstests
(6) Implementierung eines Tests auf Verklemmungsfreiheit

Zu (1) muß der Ablauf von nebenläufigen Vorgängen so visualisiert werden, daß die jeweilige Position einem spezifischen Element aus dem UML-Diagramm oder der Kartusche des dynamischen Performanz-Modells zugeordnet werden kann. Zu (2) müssen an Konflikten Auswahlmöglichkeiten[2] gegeben werden. Eine mögliche Iteration über den bereits beschrittenen Pfad (Schleifenerkennung) soll die Möglichkeit bieten, die Anzahl der Iterationen unmittelbar zu spezifizieren. Zu (3) bietet sich als Grundlage einer XML-Datenstruktur die bereits daraufhin gestaltete Spurzerlegung nach Definition 50 in Kapitel 3 an. Daraus ergibt sich eine Menge an Einzelsequenzen, die jeweils genau einer Sequenz-Kartusche zugeordnet werden können. Um die Größe der persistenten Form in Grenzen zu halten, sollten iterierte Sequenzen nur einfach gespeichert und mit ihrer Kardinalität annotiert werden. Zu (4) muß eine mit der Simulationskomponente abgestimmte Implementierung die in Kapitel 3 in Definition 52 spezifizierte Ablaufsteuerung übernehmen und im Verlauf der Simulation den Zustand des Szenarios gemäß der Definition 53 fortschreiben. Zu (5) muß eine Funktion

[1] Kompatibilität heißt formal die Entscheidung des Wortproblems der durch das dynamische Modell spezifizierten Spursprache

[2] Beispielsweise durch *Pulldown*- oder Kontextmenüs

verklemmungsfrei ermitteln können, ob durch das Lesen des Szenarios der Endzustand eines spezifischen Modells erreicht werden kann. Unabhängig davon kann (zu (6)) das Szenario erst als vollständig gespeichert werden, falls es bezüglich der Kommunikation eines Modells keine bereits vorab erkennbaren Verklemmungen beinhaltet.

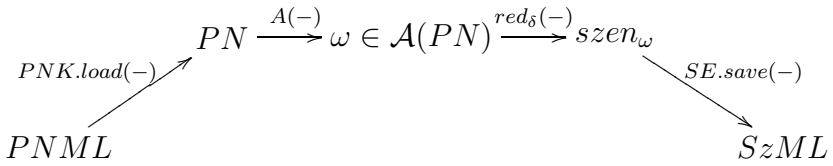


ABBILDUNG 7.2.4. Funktionsschema des Szenario-Editors.

Eingebettet in die formale Betrachtung ergibt sich das in Abbildung 7.2.4 dargestellte Schema. Der Szenario-Editor (SE) nutzt eine Funktion des Petrinetz-Kernels, um das Petrinetz aus dem `PNML` zu erzeugen (hier als `PNK.load()`-Methode dargestellt). Die Erzeugung der Spursprache entfällt nur fast, da sie durch das Petrinetz unmittelbar verfügbar ist. Ein graphischer Editor erlaubt dem Benutzer, eine Schaltfolge zu selektieren. Der Ablauf wird auf die Sequenz der relevanten Transitions-Anschriften nach der Reduktionsfunktion red_δ aus Definition 48 reduziert und in einem geeigneten XML-Format (hier `SzML` genannt) durch eine im Szenario-Editor zu implementierende Funktion (hier durch eine `SE.save()`-Methode dargestellt) gespeichert.

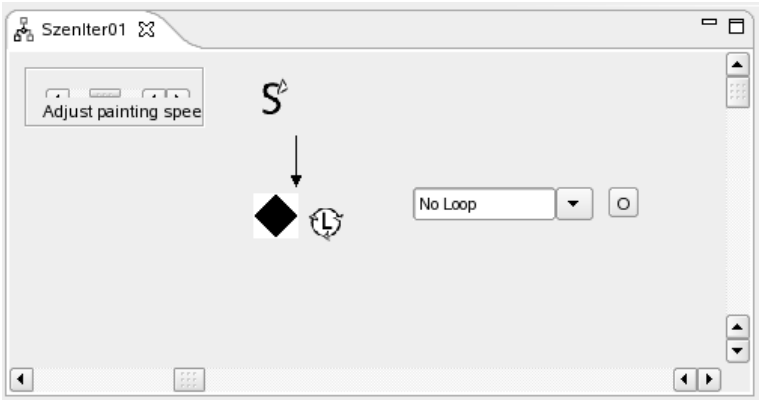


ABBILDUNG 7.2.5. Schleifenerkennung

Der Szenario-Editor der |SEMPER|-Implementierung realisiert diese geforderten Funktionen. Das Beispiel der einfachen Schleife über eine

Aktivität aus Abbildung 7.1.3 dient jetzt beispielhaft als Modell, für das ein Szenario entwickelt wird.

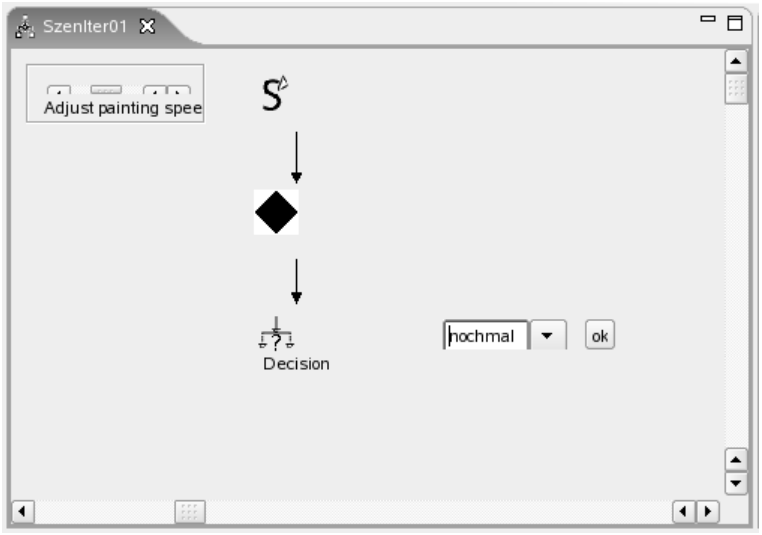


ABBILDUNG 7.2.6. Manuelle Auswahl

Abbildung 7.2.5 zeigt die Ansicht im Editor, welche die mögliche Schleife erkannt hat und die Iteration der gesamten Schleife oder die manuelle Auflösung der einzelnen Konflikte zur Auswahl stellt.

Wählt man die manuelle Auflösung, kann im nächsten Schritt zwischen den mit `nochmal` oder `fertig` annotierten Kanten ausgewählt werden. Abbildung 7.2.6 zeigt diese Interaktionsmöglichkeit. Wird die im Bild sichtbare Auswahl der mit `nochmal` annotierten Kante getroffen, erfolgt eine einmalige Ausführung der Aktivität `Funktion_X`. Um dem Bearbeiter des Szenarios die Orientierung zu erleichtern, wird die Aktivität mit dem Namen des UML-Modells gekennzeichnet. Abbildung 7.2.7 zeigt diese einmalige Ausführung und erkennt am *Decision*-Knoten erneut die Möglichkeit einer Schleife. Soll die Schleife n-Mal durchlaufen werden, genügt hier die Eingabe von n. Wird einmal die mit `fertig` annotierte Kante gewählt, ist der Endzustand erreicht und das Szenario kann im Repository gespeichert werden.

7.2.1.3. *Ressourcen-Workbench.* Ressourcen und deren spezifische Lasten wurden in Kapitel 4 durch eine Reihe an Funktionen höherer Ordnung spezifiziert, zu deren möglicher Realisierung ein großer Freiraum geben wurde. Die Aufgabe einer *Ressourcen-Workbench* ist

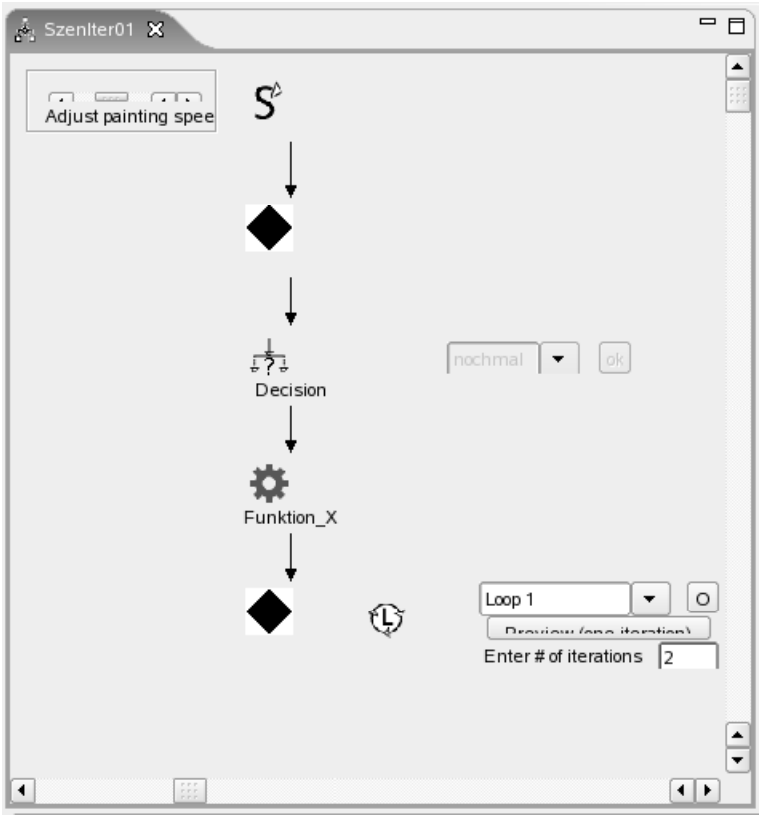


ABBILDUNG 7.2.7. Wiederholte Schleifenerkennung

es, vorrangig Möglichkeiten zur konkreten Entwicklung solcher Funktionen anzubieten. Alle Wege, die in dieser Werkzeugstudie beschritten werden, sind spezifisch und können nur als Einzelbeispiele gewertet werden. Für die eingehende Umsetzung wird auf das Projekt-Dokumentationsmaterial verwiesen.

In |SEMPER| werden zwei grundsätzliche Möglichkeiten angeboten. Die Bearbeitungszeitfunktion f_t aus Definition 61 wird durch eine Regressionsgerade oder als kubischer Spline realisiert. Im ersten Ansatz der Performanz-Analyse durch *White-Box*-Modellierung aus [**13**] waren beide Verfahren bereits sehr erfolgreich. Die grundlegende Idee ist für beide Möglichkeiten gleich.

Jede Last wird durch eine Zahl an unabhängigen Parametern gekennzeichnet, die ihre Größe beschreiben. Für jeden Parameter wird in einer Messung oder Schätzung die abschließende Bearbeitungszeit durch eine Ressource für eine Menge an Stützstellen bestimmt. Das Intervall zwischen der Stützstelle mit dem kleinsten und dem größten Wert ist der Gültigkeitsbereich dieser Ressource. Die Berechnung der

benötigten Bearbeitungszeit für eine Last mit einem beliebigem Wert innerhalb dieses Intervalls erfolgt nun durch eine Funktion f_t die durch Berechnung der Regressionsgeraden oder eines Splines über die Stützstellen gewonnen wird.

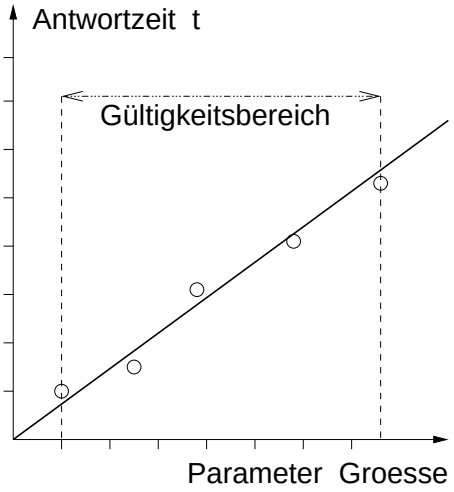


ABBILDUNG 7.2.8. Regressionsfunktion aus Stützstellen

Abbildung 7.2.8 zeigt schematisch die Bestimmung einer Regressionsfunktion durch gegebene Stützstellen. Die *Ressourcen-Workbench* hält für jede dieser Methoden generische Klassen bereit. Jede Klasse wird mit einen Satz an Meßdaten aus einer Basisdatenbank instantiiert. Jedes dieser generischen Ressourcen-Modelle bietet dann eine Schnittstelle mit den beiden Funktionen f_t und f_s an. Die berechneten Werte lassen sich individuell für jeden Ressourcentyp aus den im Konstruktor übergebenen Stützstellen ableiten. Jedes Lastmodell ist für ein Ressourcenmodell kompatibel, wenn es die passende Anzahl an Parametern aufweist und sich dessen Initialwerte im Gültigkeitsbereich des Ressourcenmodells befinden. Ein einfaches Typsystem ist technisch in |SEMPER| durch zusätzliche Bezeichner realisiert, die den Parameter-Werten beigefügt sind.

Abbildung 7.2.9 zeigt schematisch die von der *Ressourcen-Workbench* bereitgestellten Funktionen. Aus einer Datenbasis für Meßwerte, werden die Datensätze für einzelne Ressourcen- und Lastmodelle zusammengestellt. Aufbauend auf diesen Datensätzen arbeiten die generischen Implementierungen der Ressourcenmodelle und berechnen die jeweiligen Werte für f_t und f_s. Außerdem stellt die *Ressourcen-Workbench* eine Fabrik für Lastmodelle zur Verfügung.

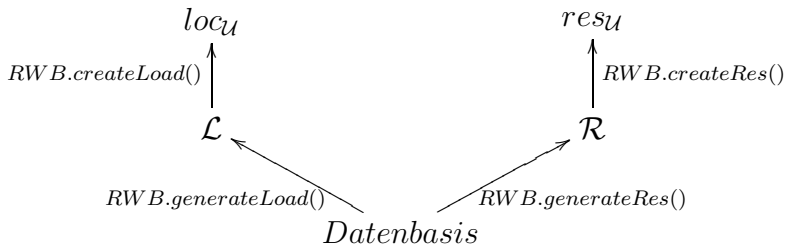


ABBILDUNG 7.2.9. Funktionsschema der Ressourcen-Workbench.

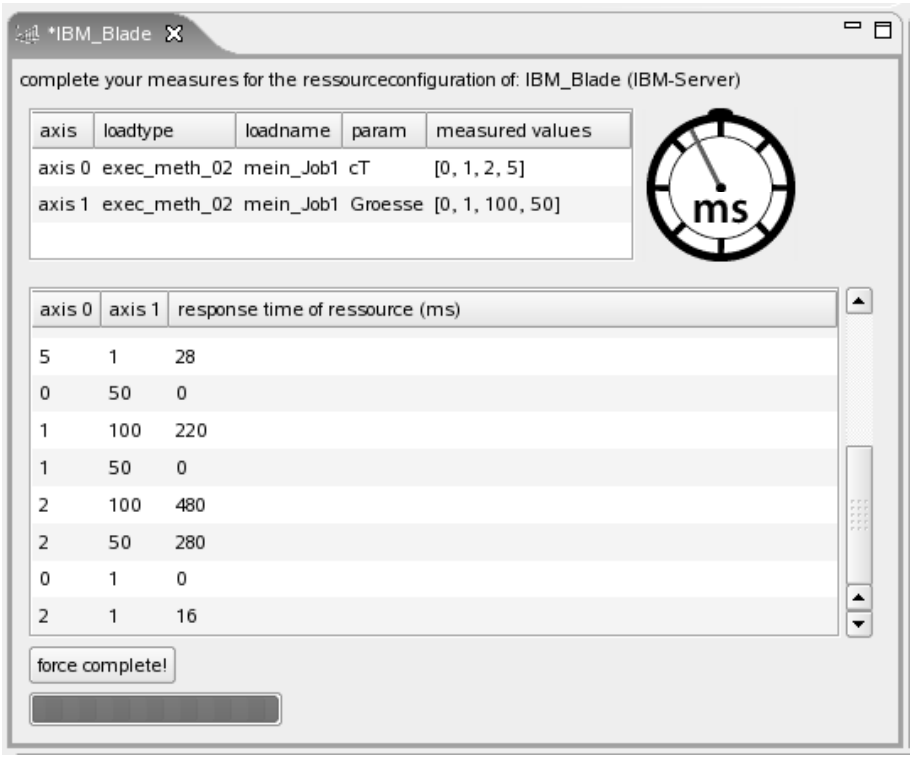


ABBILDUNG 7.2.10. Editor für Meßwerttabellen

Der größte Implementierungsaufwand liegt in der Realisierung eines leicht bedienbaren Editors zur Eingabe der Stützstellen und Pflege der Datenbasis. Da die Größe der Meßwerttabellen mit der Anzahl der modellierten Parameter exponentiell wächst, bringt die Pflege der Datenbasis ein erheblicher Aufwand. Die Visualisierung einer

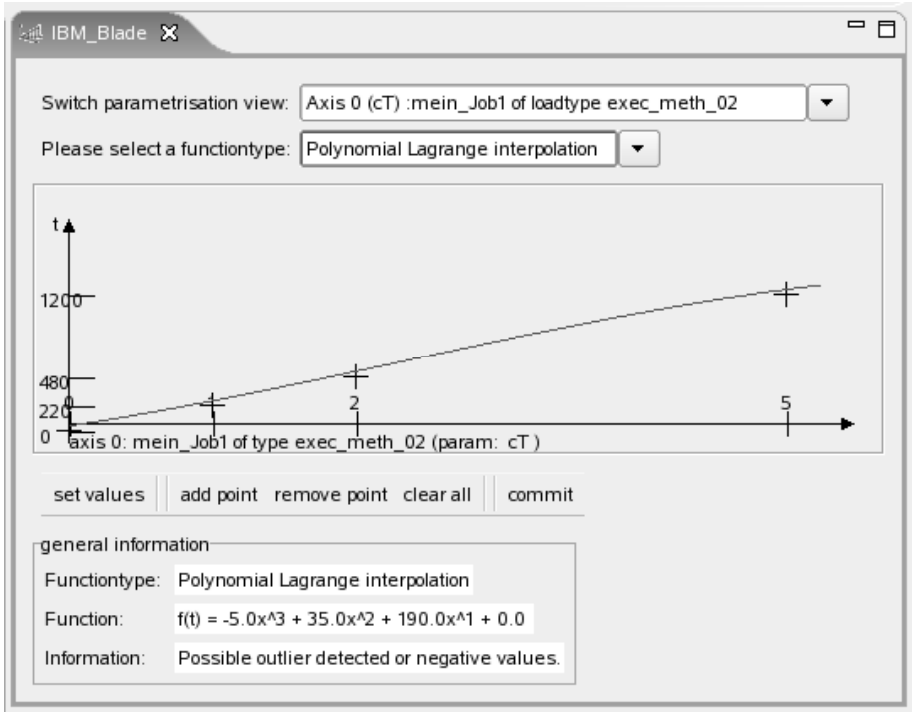


ABBILDUNG 7.2.11. Splinefunktion im Editor

mehr-dimensionalen Funktion ist weiterhin eine bleibende Herausforderung. Abbildung 7.2.10 zeigt den Editor der |**SEMPER**|-Ressourcen-Workbench, der die Eingabe der Performanz-Meßwerte oder Schätzwerte in tabellarischer Form erlaubt. Nach der Eingabe der Meßwerte kann für jede einzelne Achse des Systems, welches das zeitliche Verhalten der Ressource modelliert, eine eingene Interpolationsfunktion (Spline oder Regressionsgerade) gewählt werden. Ein eigener Editor erlaubt die getrennte Visualisierung der Funktion auf der ausgewählten Achse gegen die Bearbeitungszeit. Die Funktion kann hier nochmals graphische dargestellt und nachbearbeitet werden.

Abbildung 7.2.11 zeigt beispielhaft die Abbildung einer Erhöhung der Antwortzeit in *ms* in Abhängigkeit von der Anzahl der nebenläufigen Prozesse. Hier werden Messungen für einen, zwei und fünf nebenläufige Prozesse als Stützstellen zugrundegelegt.

7.2.1.4. *Environment-Compiler.* Eine Umgebung besteht aus einer Menge an Ressourcen - soweit die Definition in Kapitel 4. Die Umgebung ist somit nur ein Container für Ressourcen. Die Implementierung

muß nur die Lokalisierung einer Ressource anhand ihres Namens bewerkstelligen. Die Instanz eines Ressourcenmodells muß die in Definition 61 festgelegten Methoden für Schritt- und Zeitfunktion bereitstellen. Die Instantiierung dieses Modells übernimmt die *Ressourcenworkbench*. Der richtige Typ, der Name und die Initialparameter sind aber in einem UML-Implementationsdiagramm spezifiziert. Jede Ressource ist im Laufe einer Simulation mit der Bearbeitung verschiedener Lasten beschäftigt. Aus dem dynamischen Performanzmodell läßt sich die gesamte Menge der möglichen Lasten ermitteln. Die Instanzen der Lastmodelle müssen initialisiert sein, der Ressource zugeordnet werden und bezüglich des Ressourcentyps adäquat sein.

ABBILDUNG 7.2.12. Modell einer Simulationsumgebung

Abbildung 7.2.12 zeigt schematisch das Modell einer solchen virtuellen Simulationsumgebung. Zur Erzeugung dieser virtuellen Simulationsumgebung gehören verschiedene Funktionalitäten, die eng gekoppelt sind. Obwohl im ersten Teil keine formale Begrifflichkeit für das gesamte Simulationsmodell entwickelt wurde, ist eine Einführung einer solchen Datenklasse sinnvoll. Aus dieser Architekturentscheidung erwachsen an diese Datenstruktur und dem sie erzeugenden *Environment-Compiler* folgende Funktionsanforderungen:

(1) Ermittlung aller Ressourcenmodelle aus einem UML-Implementationsdiagramm (*Deployment-Diagramm*)
(2) Erstellung passender Ressourcen-Modellinstanzen
(3) Zusammenfassung zur Umgebung mit Konsistenzprüfung
(4) Laden des Performanzmodells
(5) Ermittlung aller Lastmodelle aus dem Petrinetz
(6) Erstellung passender Last-Modellinstanzen
(7) Zuordnung zu den richtigen Ressourcen-Modellinstanzen
(8) Umfassende Konsistenz und Typprüfung
(9) Persistenzform zur Wiederverwendung

Der *Environment-Compiler* erzeugt also die in Definition 60 beschriebene Umgebung. Dieser Vorgang wird nicht weiter spezifiziert.

Analog zu den vorangegangenen Abbildungen sind diese Vorgänge wie in Abbildung 7.2.13 schematisierbar. Dabei sind die Funktionen aufgeteilt in Methoden, die durch den *Environment-Compiler* (EC) zu

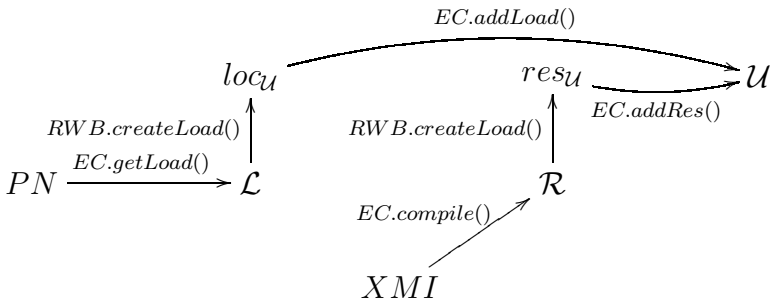


ABBILDUNG 7.2.13. Funktionsschema des Umgebungscompilers.

implementieren sind, und solche, die der *Environment-Compiler* als Funktionalität der *Ressourcen-Workbench* (RWB) nutzt.

7.2.1.5. *Simulation und Analyse.* Bei der Simulation wird aus einer Petrinetzsemantik eines dynamischen Modells PN dessen Erreichbarkeitsgraph $G(PN)$ erzeugt. Es wird ein Pfad im Erreichbarkeitsgraph bestimmt. An Knoten mit mehreren ausgehenden Kanten wird der jeweils nächste entweder durch ein Szenario $in \in \mathcal{I}_{Szenario}$ oder durch die Ermittlung der Last mit der minimalen Bearbeitungszeit durch die Funktion $res_{\mathcal{U}} \in \mathcal{U}$ bestimmt.

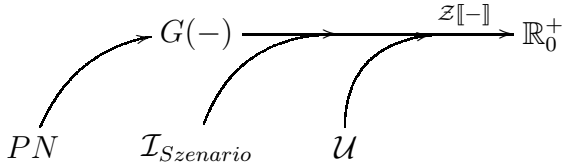


ABBILDUNG 7.2.14. Funktionsschema des Simulators.

Dieses in Kapitel 5 ausführlich beschriebene Verfahren ist in Abbildung 7.2.14 nochmals schematisch zusammengefaßt. Jeder Simulationslauf ermittelt durch das Inkrement über die f_t-Funktion einen Zeitwert für die Bearbeitung.

Die einzelnen simulierten Zeitwerte sollen in einem Analyse-Werkzeug zu Dokumenten zusammengefaßt und graphisch ansprechend gestaltet werden. Durch die nahe Koppelung beider Vorgänge, der Gestaltung der Graphiken und der Auswahl der im Sinnzusammenhang stehende Simulationen, sind beide Komponenten in |SEMPER| zusammengefaßt.

Abbildung 7.2.16 zeigt (ohne Diskussion des Inhalts) ein Beispiel eines Diagramms, welches mit dem |SEMPER|-*Analyse-Werzeug* erstellt worden ist.

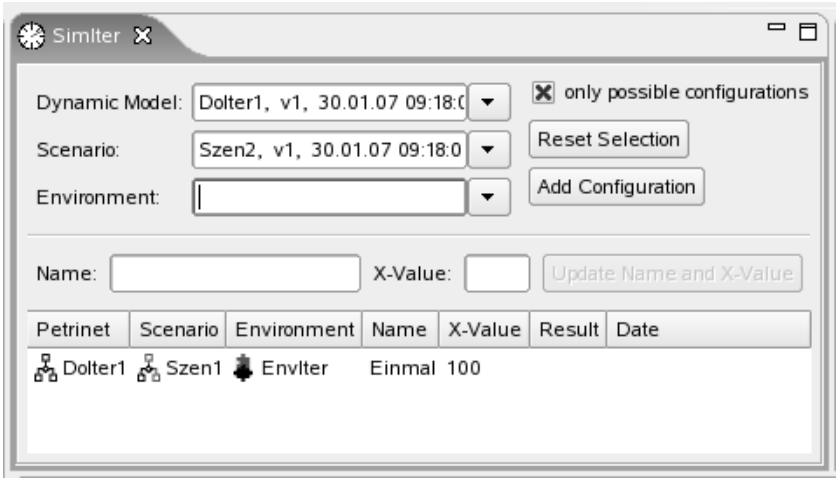


ABBILDUNG 7.2.15. Zusammenstellung einer Simulation

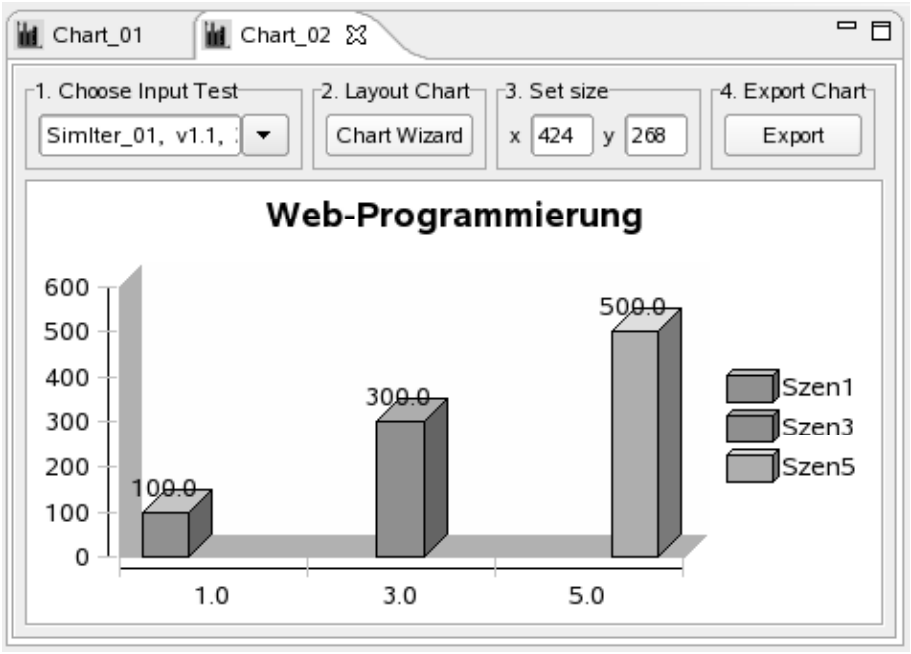


ABBILDUNG 7.2.16. Chart-Erstellung im Analyse-Werkzeug

Graphisch aufbereitete Dokumente sollen zu jedem in der Requirements-Analyse identifizierten Testfall angefertigt werden. Diese Dokumente können als Grundlagen für die Bewertung verschiedener Entwürfe oder der Erfolgs- und Fortschrittsbewertung der in den Iterationen des Projektes wachsenden Artefakte herangezogen werden. Auch

eine Risikobewertung eines Projektes, welches eine kritische Abhängigkeit von Performanz-Eigenschaften hat, kann sich auf diese Dokumente stützen. Im Kapitel 8 werden auf abstrakter Ebene einige Aspekte des Managements eines kritischen Prozesses diskutiert.

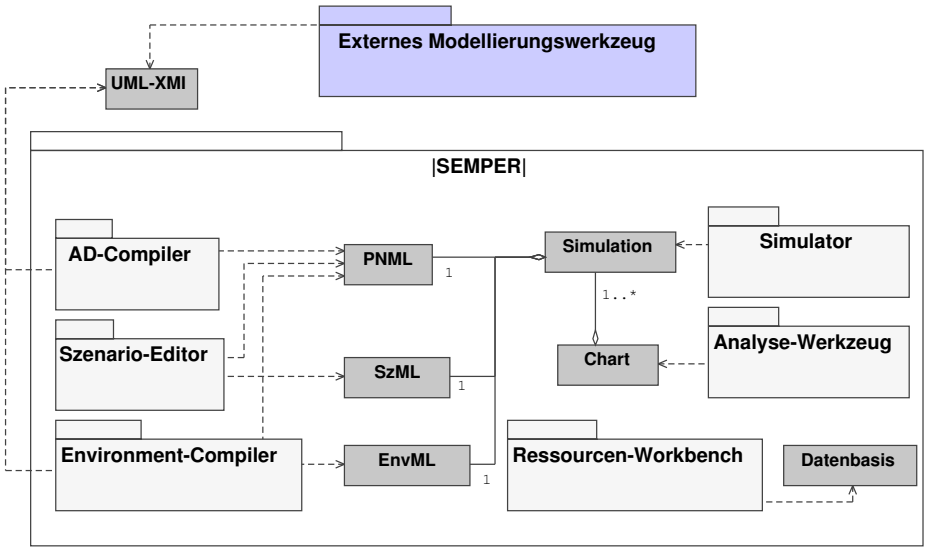


ABBILDUNG 7.2.17. Architektur mit Artefakttypen

7.2.2. Architektur aus Datensicht. Bevor die Anforderungen an eine Datenhaltung formuliert werden können, werden in Abbildung 7.2.17 die Komponenten in Abhängigkeit zu den von ihnen verwendeten Dokumenttypen dargestellt.

Für |SEMPER| kann ein einfaches Datenverwaltungskonzept beispielsweise einen Bearbeitungszustand und einen abgeschlossenen Zustand für alle Dokumente vorsehen. Jedes Dokument kann in den primären Verantwortungsbereich einer Komponente gestellt werden. Diese Komponente bearbeitet das Dokument und stellt es schließlich durch ein *commit* den anderen Komponenten in der Datenhaltung bereit. Danach bauen diese Komponenten weitere Dokumente auf den Zustand auf, weshalb es nicht mehr verändert werden darf. Eine Veränderung setzt eine neue Versionierung voraus und damit eine neue Instanz des Dokumentes. Folgende Zuständigkeiten wurden für |SEMPER| festgelegt:

- AD-Compiler → PNML (XMI-Import)
- Szenario-Editor → SzML
- Environment-Compiler → EnvML (XMI-Import)

- Simulator (& Analyse-Werkzeug) → Simulationsparameter, Charttabellen
- Ressourcen-Workbench → Meßwert-Datenbasis

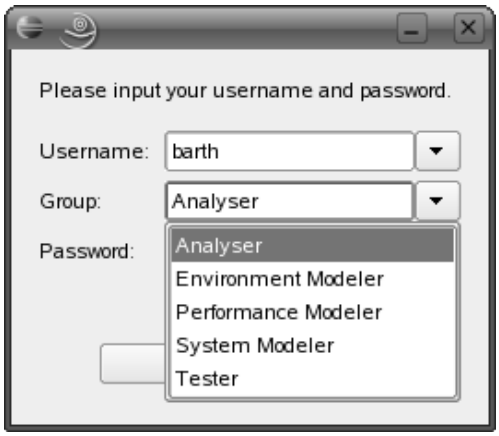


ABBILDUNG 7.2.18. Rollenbasierter Nutzerzugang

Eine weitere Komponente, das *Workflow-Management*, vervollständigt die Werkzeug-Architektur. Eine Aufgabe ist die Bereitstellung und Verwaltung der Dokumente nach dem hier skizzierten Schema. Einesteils sind die Zugriffsrechte vom Zustand der Dokumente selbst abhängig, andererseits aber auch von der Rolle des Nutzers, die noch nicht diskutiert wurde. Die im Kapitel 6 eingeführten Rollen müssen hier durch Steuerung der Rechte über Funktionalität und Dokumentenzugriff realisiert werden.

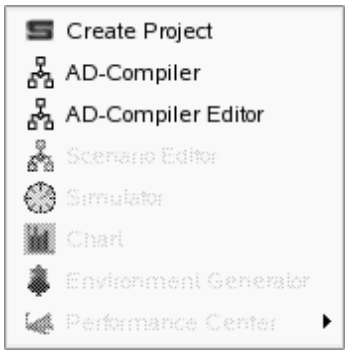


ABBILDUNG 7.2.19. Rollenbasierter Funktionszugang

Abbildung 7.2.18 zeigt den Login zu |SEMPER|, bei dem nicht nur Nutzername und Passwort angegeben werden müssen, sondern auch die gewünschte Rolle. Die hier realisierten Rollen tragen nicht nur den in Kapitel 6 geforderten Rollen Rechnung, sondern auch den in der Praxis auftretenden, häufigen Doppelbesetzungen. Der *Performanz-Modeler* hat die Befugnis, Projekte anzulegen, beliebige Dokumente zu importieren und dynamische Performanz-Modelle bereitzustellen. Die

Rolle ist nahezu deckungsgleich mit der Beschreibung des Performanz-Modellierers[3]. Abbildung 7.2.19 zeigt ein rollenbasiert gesteuertes Funktionsmenu. Nur die Funktionalität, die der Nutzerrolle im gegenwärtigen Prozeßzustand zukommt, ist aktiviert.

Als Feinheit wurde der Workflow nachdem hier erarbeitetem Prozeßmodell in |SEMPER| mittels eines Petrinetzes formuliert. Alle Software-Funktionalität ist jeweils an eine Transition gekoppelt. Die entsprechende Funktion ist nur dann aktiviert, wenn die zugeordnete Transition konzessioniert ist. Somit ist das Prozeßmodell leicht anpaßbar. In kleineren Firmen, in denen die Übernahme mehrerer Rollen durch einen Arbeiter unvermeidlich ist, läßt sich der Prozeß so umgestalten, daß zahlreiche Rollenwechsel im Arbeitsablauf vermieden werden können.

7.2.3. Implementierung. |SEMPER| wurde selbst in Java entwickelt. Die Verfügbarkeit einer großen Zahl leistungsfähiger Werkzeuge und die Integration in die Technologien, die durch |SEMPER| selbst erweitert werden sollen, legen diese Wahl nahe. Die Aufteilung in unabhängige Einzelkomponenten, deren Realisierung durch Editoren und deren Kommunikation über Dokumentenaustausch legte die Verwendung eines Komponenten-Frameworks nahe. |SEMPER| wurde in der Eclipse-Plattform realisiert, die eine weite Verbreitung und Beliebtheit in den Java- und UML-Technologien genießt. Die Implementierung der Petrinetz-Maschine wurde, nicht zuletzt um die Plattformunabhängigkeit des Konzeptes zu belegen, mit einem externen Werkzeug, dem Petrinetz-Kernel der Humboldt-Universität, Berlin **[64, 52]** umgesetzt.

Die Realisierung erfolgte im Jahr 2005 und 2006 durch Arbeiten des Autors und diverser Praktika und Diplomarbeiten, siehe **[29, 30, 48, 49, 56, 46, 55, 12]**. Die Abbildung der Bildschirmausgaben stammen von der |SEMPER| Version 0.9, Ende 2006.

7.3. Anwendungsbeispiel

An einem einfachen Beispiel soll nun der Ablauf einer Modellbewertung dargestellt werden. Dazu gehören die Erstellung einer Simulationsumgebung aus Meßwerten, die Annotation eines zu bewertenden UML-Aktivitäts- und Implementations-Diagramms, einer Festlegung einer ausdrucksstarken Menge an Testfällen und einer Auswertung durch |SEMPER|. Abschließend erfolgt eine Interpretation des Ergebnisses.

[3]Der Performanz-Analytiker arbeitet nach diesem Konzept hauptsächlich mit externen UML-Werkzeugen.

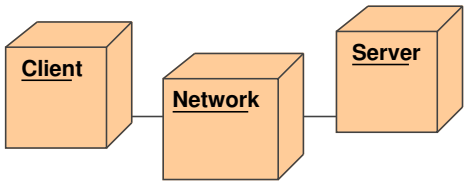


ABBILDUNG 7.3.1. Beispielumgebung

7.3.1. Ressource. Dem Beispiel liegt die einfache, aber häufige Fragestellung zugrunde, ob eine Verarbeitung effizienter auf dem Client ausgeführt wird oder es Vorteile bringt, eine Übertragung von Daten über ein Netzwerk in Kauf zu nehmen, um auf den schnelleren Server zugreifen zu können. Die im Implementationsdiagramm in Abbildung 7.3.1 dargestellte Umgebung liegt dieser Bewertung zugrunde.

ABBILDUNG 7.3.2. Erzeugung eines Rohdatensatzes

Beispielhaft wird hier die Erzeugung eines Simulationsmodells für den Server gezeigt. Als erstes wird im Werkzeug ein Rohdatensatz angelegt. Abbildung 7.3.2 zeigt den Eintrag im Repository als Bildschirmausschnitt.

In diesem Beispiel wurde für die Ressource ein Lasttyp vorgesehen, der einen Sortiervorgang (bzw. die Erstellung eines binären Index) modelliert. Er verfügt über einen Parameter `item`, der die Anzahl der zu sortierende Elemente festlegt. Zur Messung der Ressource wurde ein Codefragment implementiert, welches Testdaten nach dem Quicksortalgorithmus sortiert. Der Parameter `item` bezeichnet die Größe des Testdatensatzes. Es wurden zwei Messungen zur Kalibrierung der Funktion durchgeführt. Die Stützstellen waren `item=16384` und `item=65536`. Die Messung wurde einige Male wiederholt und aus den Meßwerten das arithmetische Mittel gebildet. Diese Meßdaten werden in den Editor für die Tabelle des Rohdatensatzes eingetragen.

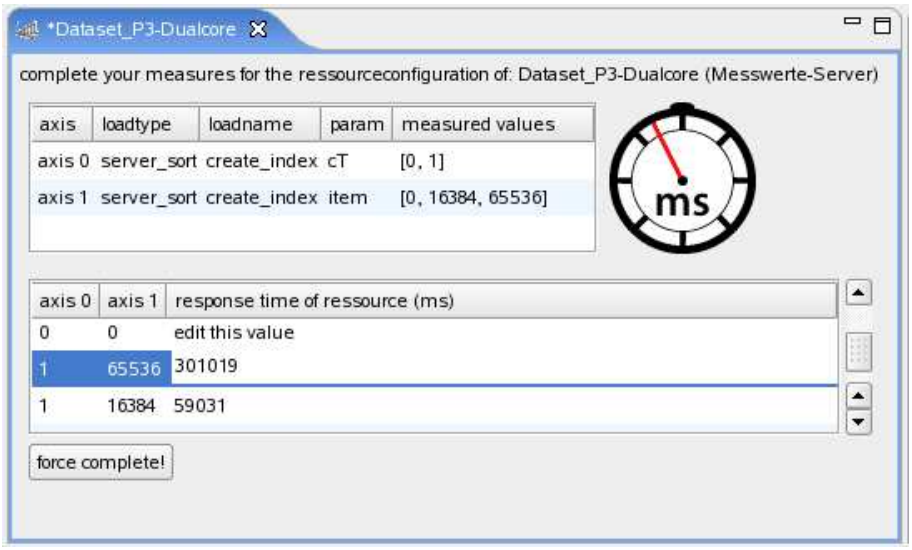


ABBILDUNG 7.3.3. Erstellen der Meßwerttabelle

Abbildung 7.3.3 zeigt einen Bildschirmausschnitt des Editors und die eingegebenen Meßwerte in *ms*. Nach vollständiger Eingabe aller Meßwerte kann die Tabelle gespeichert werden. Beim nächsten Öffnen des Rohdatensatzes erscheint ein Editor, in dem die Interpolationsfunktionen für die entsprechenden Sichten erzeugt werden können.

Da es sich bei der Performanzfunktion um eine n-stellige Funktion handelt, ist allein die Darstellung der eingegebenen Werte eine anspruchsvolle Aufgabe. In einer zweidimensionalen Graphik, kann stets nur ein freier Parameter gegen die Antwortzeit der Ressource aufgetragen werden. In dem hier dargestellten Beispiel haben wir zwar nur einen freien Parameter geschaffen, aber durch die mögliche nebenläufige Bearbeitung mehrerer Lasten implizit als weiteren Parameter die Anzahl der Lasten erzeugt. In der Tabelle wird dieser als `cT` (countThreads) abgekürzt. Das hier gezeigte Beispiel ist stark vereinfacht. Da wir im Modell nur eine einzige Last annehmen, die gleichzeitig bearbeitet wird, kann `cT` nur die Werte `0` und `1` annehmen. Längs der Achse, der Anzahl der zu sortierenden Elemente, muß aber eine Interpolationsfunktion ermittelt werden, die auch Antwortzeiten zwischen den Stützstellen berechenbar macht. In Abbildung 7.3.4 ist die Oberfläche des Editors dargestellt, mit der diese Interpolationsfunktionen gestaltet werden können. Im ersten Auswahlfeld kann die Achsensicht bestimmt werden. Hier wurde die Anzahl der Elemente gewählt. Im

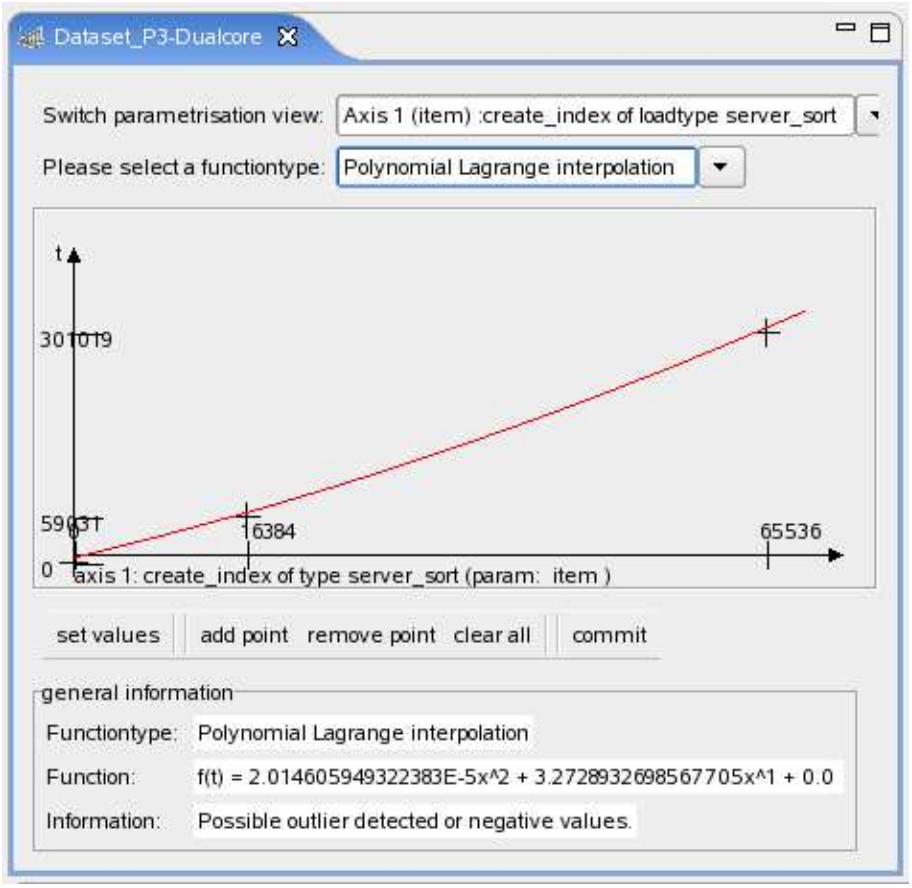


ABBILDUNG 7.3.4. Erzeugung der Interpolationsfunktion

zweiten Auswahlfeld können vorbereitete Funktionen ausgewählt werden. In diesem Fall wurde eine Lagrange Interpolation gewählt. Diese Entscheidung ist eine bewußte Wahl des Simulationsingenieurs, der weiß, daß ein Quicksort-Algorithmus zur Anzahl der verarbeiteten Elemente eine Laufzeitkomplexität von $n * log_2 n$ hat. Eine quadratische Interpolationsfunktion liefert hier gute Ergebnisse. Die Abbildung im Editor zeigt die berechnete Kurve und die zugrundeliegenden Stützstellen. Die Parameter der Funktion sind zusätzlich angegeben.

Sind alle Interpolationsfunktionen festgelegt und abgespeichert, können diese zur Initialisierung einer generischen Ressourcen- und Lastmodell-Kasse herangezogen werden. Zur Identifikation der richtigen Funktion wird ein Bezeichner festgelegt, der dann im dynamischen Modell unmittelbar als Typbezeichnung Verwendung findet. Gleiches

[3]Natürlich könnte eine *log*-Funktion auch unmittelbar implementiert werden.

erfolgt für die Last. Im Beispiel ist der Bezeichner des Typs der Last `server_sort` und der Ressource `System_Netfinity5000_Dualcore`.

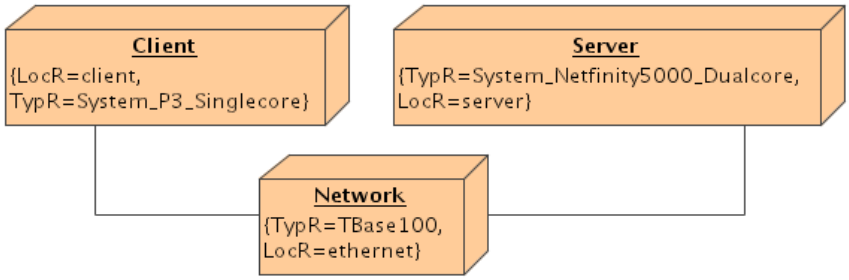


ABBILDUNG 7.3.5. Spezifikation der virtuellen Umgebung

7.3.2. Umgebungsspezifikation. Das bereit eingeführte Implementationsdiagramm wird mit Eigenschaftswerten versehen, die Auskunft für die Simulation geben. In Abbildung 7.3.5 ist unter anderem der Typ `System_Netfinity5000_Dualcore` verwendet. Weiterhin werden unter dem Eigenschaftswert `LocR` Namen vergeben. Über diese Namen kann später im dynamischen Modell spezifiziert werden welche Ressource eine Last bearbeitet.

7.3.3. Modellspezifikation. Das zu untersuchende Modell wird als UML-Aktivitätsdiagramm spezifiziert. In Abbildung 7.3.6 ist der Graph dargestellt. Die zusätzlichen Eigenschaftswerte sind hier mit angegeben. `StartL` bezeichnet dabei einen frei wählbaren, aber eindeutigen Namen für die Last. Für das Sortieren auf dem Server ist der Wert hier `server_sort_activity`. Der Lasttyp wird als `TypL` mit `server_sort` spezifiziert. Dadurch wird das Lastmodell verwendet, dessen Erzeugung im vorletzten Abschnitt erläutert wurde. Der Name der Ressource wird durch `LocR` mit `server` angegeben. Eine Bewertung dieses Modells kann nur erfolgen, wenn das Implementationsdiagramm unter dem Namen `server` eine Ressource spezifiziert, deren Typ mit dem Typ `server_sort` der Last kompatibel ist. Zusätzlich ist ein notwendiger Parameter als `ParamL` mit dem String `"item=10000ßpezifiziert`. Dabei ist vorausgesetzt, daß der Bezeichner des Lastparameters `item` existiert. In diesem Fall ist `item` der Bezeichner, mit dem die Anzahl der zu sortierenden Elemente spezifiziert wurde. Damit das Modell bewertet werden kann, muß auch der Parametername bekannt sein. Außerdem muß der Wert, der dafür angegeben wird, innerhalb des Intervalls der zur Generierung des Modells angegebenen Stützstellen liegen.

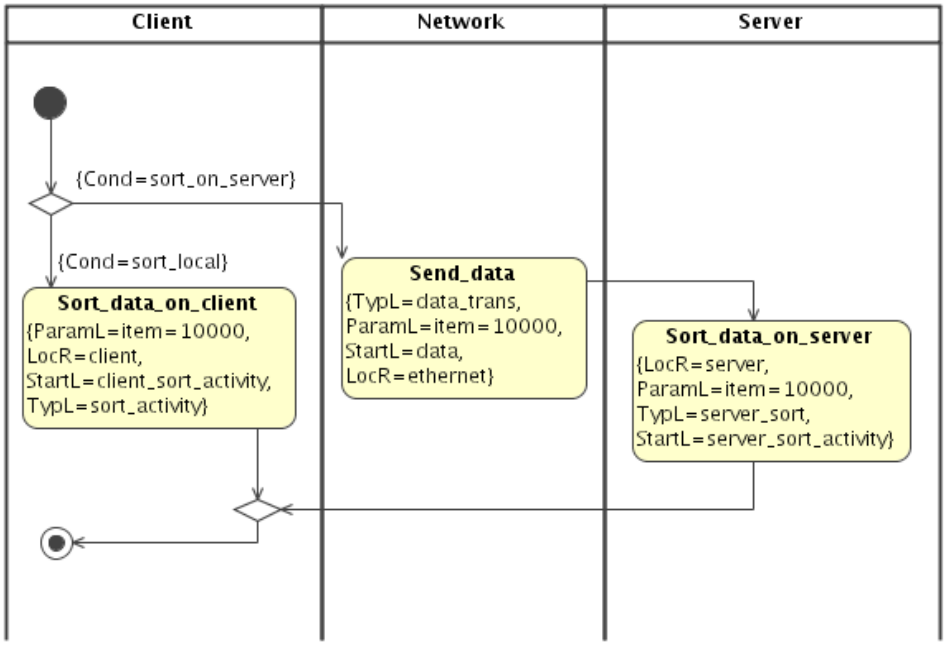


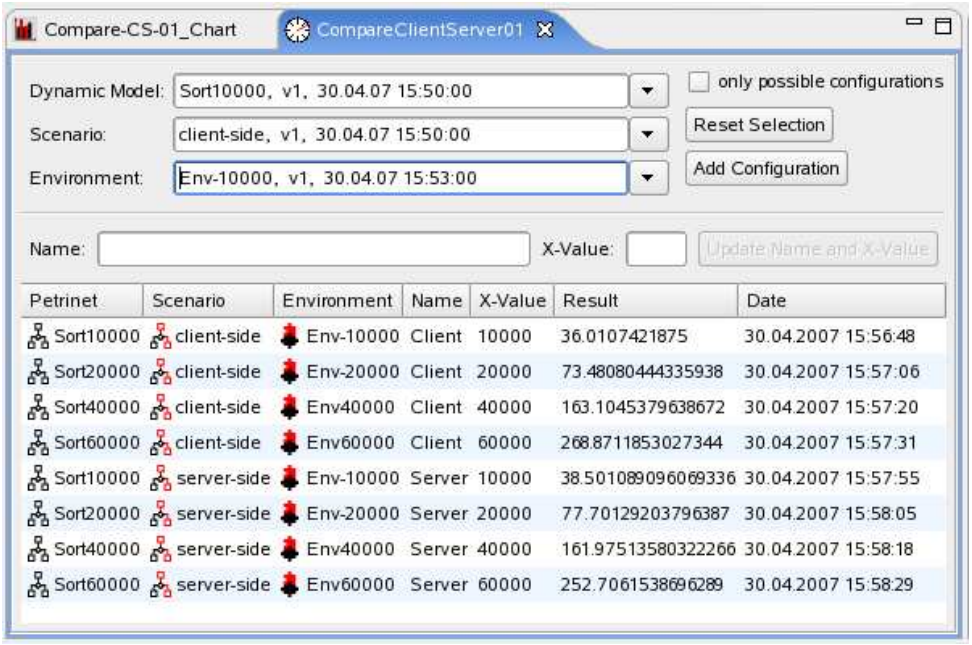
ABBILDUNG 7.3.6. Spezifikation des dynamischen Modells

Alle beschriebenen Vorgänge müssen für alle Parameter, Lasten und Ressourcen dieses Diagramms durchgeführt worden sein.

7.3.4. Testfallspezifikation. Das Modell stellt inhaltlich zwei alternative Abläufe gegenüber. Wird die Alternative `sort_local` gewählt, wird eine Sortieraktivität auf dem Client durchgeführt. Nach Abschluß der Bearbeitung ist der Ablauf beendet. Die andere Alternative wird durch die Wahl der mit `sort_on_server` bezeichneten Transition gewählt. Danach werden die Daten zunächst über das Netzwerk zum Server transportiert und dann dort sortiert. Die beiden Abläufe können also gegenübergestellt werden. Zwei Szenarien, *client-side* und *server-side*, sind hier denkbar.

Weiterhin sollen verschiedene Last-Quantitäten verglichen werden. Hier ergeben sich quantitative Unterschiede nicht aus unterschiedlich häufigen Iterationen im Graphen, sondern aus den Lastparametern. Für verschiedene Datensatzgrößen müssen die Werte für `ParamL` und `item` verändert werden. In diesem Beispiel wurden Diagramme für 10000, 20000, 40000 und 60000 Elemente spezifiziert (ohne Darstellung).

Jedes Diagramm wird später mit jeweils den Szenarien *client-side* und *server-side* simuliert.



Petrinet	Scenario	Environment	Name	X-Value	Result	Date
Sort10000	client-side	Env-10000	Client	10000	36.0107421875	30.04.2007 15:56:48
Sort20000	client-side	Env-20000	Client	20000	73.48080444335938	30.04.2007 15:57:06
Sort40000	client-side	Env40000	Client	40000	163.1045379638672	30.04.2007 15:57:20
Sort60000	client-side	Env60000	Client	60000	268.8711853027344	30.04.2007 15:57:31
Sort10000	server-side	Env-10000	Server	10000	38.501089096069336	30.04.2007 15:57:55
Sort20000	server-side	Env-20000	Server	20000	77.70129203796387	30.04.2007 15:58:05
Sort40000	server-side	Env40000	Server	40000	161.97513580322266	30.04.2007 15:58:18
Sort60000	server-side	Env60000	Server	60000	252.7061538696289	30.04.2007 15:58:29

ABBILDUNG 7.3.7. Zusammenstellung der Simulationen

7.3.5. Simulation. Es ergeben sich für die Durchführung des Vergleiches acht Tripel aus dynamischem Modell, Szenario und passendem Environment. Diese werden im Simulator konfiguriert und können einzeln oder zusammen durch Simulation nach der Laufzeitsemantik bewertet werden. Abbildung 7.3.7 zeigt die Konfiguration im Simulator.

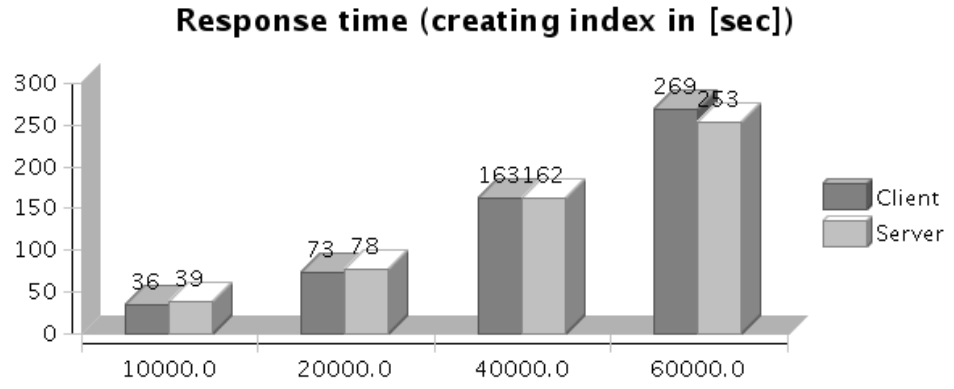


ABBILDUNG 7.3.8. Simulationsauswertung

7.3.6. Auswertung. Durch ein dem Simulator angeschlossenes Auswertungswerkzeug können die simulierten Meßwerte graphisch ansprechend aufbereitet werden. In Abbildung 7.3.8 ist eine Darstellung als zwei dimensionales Balkendiagramm gewählt. Die Bewertung in dem hier gezeigten Beispiel ergibt, daß ein Szenario, welches das Sortieren serverseitig durchführt, bei Feldgrößen über 40000 Elemente bezüglich der Performanz vorteilhaft ist. Diese Erkenntnis kann verwendet werden, um den Wächter an dem Verzweigungszustand (*Decision State*) zu formulieren oder um diese Entscheidung grundsätzlich zu treffen und in einer entsprechende Architektur festzugießen. Beispielsweise kann die Entscheidung sinnvoll sein eine Übertragungsmöglichkeit zum Server nicht zu implementieren, falls aus der Geschäftsprozeß-Analyse feststehen sollte, daß Datensätze über 40000 Elemente selten auftreten. Außerdem kann der aus der Graphik erkennbare marginale Laufzeitunterschied diese Entscheidung, das der Laufzeitgewinn keine weiteren Implementierungskosten rechtfertigt.

Interpretationen setzen aber voraus, daß alle impliziten Annahmen beachtet worden sind. Beispielsweise kann eine Messung der Performanz-Eigenschaften am unbelasteten Netzwerk unrealistische Bewertungen hervorbringen. Die Interpretation einer solchen Auswertung sowie die Gestaltung aller nötigen Artefakte ist ein Vorgang, der zusätzliche Kunst und Erfahrung benötigt. Das Werkzeug |SEMPER| gewährleistet nur die eindeutige Berechnung einer Konfiguration.

7.4. Auswertungsklassen

Eine nicht vollständige Klassifikation von Auswertungsklassen schließt dieses Kapitel ab. Grundsätzlich eignet sich das Werkzeug zu einer Darstellung der Antwortzeit bei einer Reihe von Abläufen. Diese Abläufe sollten in eine Ordnung gebracht werden, die auf der Abszisse in der graphischen Darstellung maßgenau oder zumindest die Ordnung erhaltend, aufgetragen werden können.

Diese Ordnung kann auf einzelnen oder zusammengesetzten Kriterien beruhen. Zur Klassifikation können diese aber auf ihren Ursprung in einem Element aus dem Simulationstripel zurückgeführt werden. In Tabelle 3 sind die Quellen zusammengefasst, nach denen die Organisation der Einzelsimulationen zu einem geordneten Satz erfolgen kann.

Simulationssätze, die bereits mit dieser eindimensionalen Ordnung auskommen, können typischerweise in einer zweidimensionalen Graphik durch ein Balkendiagramm oder einen Polygonzug dargestellt werden. Hier wird eine, meist manuell zu bestimmende Ordnung in der Menge der unterschiedlichen Simulationen als Reigenfolge für den Auftrag an

Lokalisation	**Ausdrucksmöglichkeit**	**Interpretation**
Szenario	- Kardinalität von Schleifen - Ranking nach Geschäftsprozeßanalyse - Abgeleitetes ablauf-bezogenes Größenmaß.	- Indikation kritischer Anforderungen - Bestimmung von Wächtern und Konstanten - Absolute Bewertung einer Architektur
Dynamisches Modell	- Quantität von Parametern - Serie an alternativen Architektur- Entwürfen	- Belastungsgrenzen von Entwürfen - Auswahl der richtigen Architektur - Erfüllung der Anforderungsanalyse
Umgebung	- Serie alternativer Ressourcen-Typen - Serie alternativer Hardwarearchitekturen	- Bestimmung notwendiger Hardware - Optimierung der Systemarchitektur

TABELLE 3. Ordnungsklassen der Simulationssätze

der Abszisse verwendet. Auf der Ordinate werden dann die durch den Simulator bestimmten Antwortzeiten angetragen. So können in einem Polygon, oder durch die abgebildeten Balken, Minima, Maxima oder in der Analyse identifizierte kritische Grenzwerte identifiziert werden.

In Tabelle 4 sind Möglichkeiten zur Zusammensetzung von Bewertungen zu Simulationsgruppen zu einem Typmuster zusammengestellt.

Satz→ ↓Gruppe	**Szenario**	**Dynamisches Modell**	**Umgebung**
Szenario	Typ 1	Typ 2	Typ 3
Dynamisches Modell	Typ 2	Typ 4	Typ 5
Umgebung	Typ 3	Typ 5	Typ 6

TABELLE 4. Klassifikation der Simulationsgruppen

Eine Simulationsgruppe ist dabei eine geordnete Menge an Simulationssätzen. Die Klassifikation wird dabei eingeteilt durch das Artefakt,

nachdem sich die Ordnung der Gruppe bzw. des Satzes richtet. Simulationsgruppen eignen sich besonders zu folgenden Auswertungen (nach Typ geordnet):

(1) *Szenario/Szenario*: Diese Gruppierung bietet sich dann an, wenn Abläufe nach mehreren unabhängigen Kriterien geordnet werden können. Beide sind in der Anforderungsanalyse identifiziert worden, und das Modell muß in einem zweidimensionalen Feld an kritischen Konstellationen beurteilt werden.
(2) *Dynamisches Modell/Szenario*: Zwei Architekturen oder implementationsabhängige Lastkonstellationen sollen bezüglich unterschiedlicher Abläufe miteinander verglichen werden[4].
(3) *Umgebung/Szenario*: Fragestellungen nach der richtigen Architektur der Hardwareumgebung, der notwendigen Leistungsparameter der Ressourcen oder der Optimierung von Ressourceneinsatz und Verteilung der Applikationen kann durch diesen Auswertungstyp beurteilt werden. Voraussetzung: Die Abhängigkeit besteht zu der Art des Ablaufes der Bearbeitung.
(4) *Dynamisches Modell/Dynamisches Modell*: Sollen die Architektur unabhängiger Komponenten in einer größeren Software oder die zeitlichen Effekte zweier voneinander unabhängiger Lastgrößen miteinander verglichen und quantitativ abgeschätzt werden, eignet sich die Gruppierung mehrerer dynamischer Modellmengen, die in ihrer jeweils eigenen Ordnung gegeneinander aufgetragen werden.
(5) *Umgebung/Dynamisches Modell*: Der Vergleich verschiedener Architekturen oder implementationsabhängiger Lastkonstellationen mit einer Verteilung, Hardware-Architektur oder Hardware-Umgebung werden durch diesen Gruppentyp am besten dargestellt. Wobei es gleichgültig ist, ob die Architekturen der Software oder der Hardware variabel sind.
(6) *Umgebung/Umgebung*: Das dynamische Modell ist bekannt, unveränderbar und bereits implementiert. Die Abläufe sind gegeben und beschreibbar. Nur eine Verteilung auf einer existierenden Umgebung, die Modifikation der Umgebung oder einzelner Umgebungskomponenten kann durchgeführt werden.

Auswertungen dieser zusammengesetzten Typen müssen meist in dreidimensionalen Diagrammen mittels Polygonzug oder Balkendiagrammen dargestellt werden. Sollte die Größe der Gruppe nur zwei

[4]Dieser Typ lag dem vorangegangenen Beispiel in Abschnitt 7.3 zugrunde

oder drei Alternativen aufweisen, kann einer zweidimensionale Graphik ausreichen, wenn die Alternativen durch Farbgebung anschaulich getrennt werden können.

Die anfangs besprochenen alleinstehenden Simulationssätze können als einelementige Gruppen aufgefaßt werden. Ihr Gruppenordnungstyp gibt dann dennoch Aufschluß darüber, welche Interpretationen dadurch intendiert sind.

Das Beispiel in Abschnitt 7.3 und die Auswertung in Abbildung 7.3.8 zeigen ein Beispiel für den Typ 2 und die zweidimensionale Darstellung[5], die hier im Druck nur durch Grauwerte unterschieden werden kann. Die Gruppe hat hier zwei Elemente, der Satz mit den serverseitigen und den Satz mit den clientseitigen Simulationen. In dieser Fragestellung ist die Größe der Datensätze interessant, ab der die Sortierung der jeweils anderen Hardware-Komponente effizienter durchgeführt werden kann. Als Polygonzug wäre dies der Schnittpunkt der Züge. Durch die Balkendiagramme ist nur das Intervall indem dieser Wechsel angesiedelt ist bestimmbar. Der Schnittpunkt des Polygonzuges würde nur einen exakten Wert suggerieren, der durch die Simulation und die zugrundeliegenden Schätzwerte nicht belegt werden kann.

Eine später durch geführte Implementierung dieses Beispiels bestätigte die Lage des tatsächlichen Punktes gleicher Performanz im vorhergesagten Intervall.

7.5. Zusammenfassung

Die wesentlichen Erwartungen an den in dieser Arbeit gezeigten Ansatz waren:

(1) Praktische Anwendbarkeit und Integrierbarkeit in verbreitete Modellierungstechniken für funktionale Eigenschaften.
(2) Ausreichend große Ausdruckskraft, Flexibilität und Skalierbarkeit.
(3) Sicherung wichtiger technischer Eigenschaften durch geeignete Spezifikation.
(4) Zweckmäßigkeit der Spezifikation für eine umfangreiche Werkzeugentwicklung.

Hier haben die problemlose Übertragbarkeit auf die Aktivitätsdiagramme und der Zugewinn an begrifflicher Klarheit für nebenläufige und sequentielle Regionen den Beleg für den Erfolg der ersten Anforderung erbracht. Die Ausdruckskraft übersteigt die der meisten Kalküle, die auf reinen Netzgraphen bestehen. Weitere wichtige Leistungen

[5]Die dreidimensionale Darstellung ist hier kein inhaltstragender Bestandteil

sind die Behandlung zustands-behafteter Systeme, die freie Wahlmöglichkeit der Grenze zwischen Ressourcen und dynamischem Modell, sowie der Granularität der betrachteten Einheiten. Dadurch wurden eine überdurchschnittliche Ausdruckskraft, Flexibilität und Skalierbarkeit erreicht.

Die Implementierung des Werkzeugs |SEMPER| zeigt eine große Stabilität und gute Reife der Entwicklung, bei Betrachten der hohen Komplexität der Implementierung kleinem Arbeitsaufwandes. Die Integration der einzelnen Komponenten erfolgte im Anschluß an die Entwicklung in einer einzigen Iteration innerhalb von ca. zwei Wochen. Dieser geringe Zeitaufwand für ein Projekt dieser Größe ist der "Paßgenauigkeit"der einzelnen Komponenten zu verdanken. Während der Implementierung wurde durch zahlreiche Tests die Einhaltung der Spezifikationen sichergestellt. Somit konnte die exakte Spezifikation im ersten Teil dieser Arbeit unmittelbar in einen Qualitäts- und Zeitvorteil umgesetzt werden. Die Realisierung eines Projektes dieser Größenordnung wäre ohne die in dieser Arbeit demonstrierte strukturierte Vorgehensweise unmöglich gewesen.

An dem |SEMPER|-Anwendungsbeispiel konnte man vor allem auch die vielleicht wichtigste Eigenschaft dieser Technik beobachten. Keines der nötigen Artefakte war in seinem Ausdruck durch die verwendete Technik qualitativ eingeschränkt. Finden beispielsweise Markov-Prozesse Verwendung, so bietet sich zur Darstellung der Prozeßverhalten nur die Exponentialfunktion an, die sie die Markov-Eigenschaft erfüllt. Die Beschränkung der Gestaltung der Ressourcenfunktionen auf Lagrange-Interpolation und Regressionsrechnung, beispielsweise, ist nur der begrenzten Funktionalität der prototypischen Implementierung geschuldet. Die Ressourcen-Workbench ist beliebig erweiterbar.

Die gleichzeitig entwickelte Methodik hat die Technik in ihrer Gestaltung entscheidend beeinflußt: Sie war grundlegend für die Möglichkeit, ein Werkzeug zu schaffen, welches stabil funktioniert, intuitiv bedienbar ist und die, von der Natur der Problematik bestimmten Arbeitsabläufe in solcher Art unterstützt, daß für eine marktreife Implementierung eine hohe Akzeptanz im Arbeitsprozeß zu erwarten ist.

Risikomanagement im Prozeßablauf

Zwischen dem Ablauf einer Software in einer Umgebung und dem Prozeß, in dem Software selbst gefertigt wird, gibt es Ähnlichkeiten. Ethische und rechtliche Gründe mögen es erschweren, einen Arbeiter als Ressource zu betrachten und seine täglichen Aufgaben als Last. Dennoch ist, nicht zuletzt aufgrund enormer Kosten in der Software-Herstellung, eine quantitative Modellierung des Software-Prozeßes selbst wünschenswert. Die hier vorgestellte Technik bietet sich auch für diese Anwendung an. Dies würde eine vollständige Entwicklung eines Metamodells für die Begriffe im Software-Entwicklungsprozeß, für funktionale und nicht-funktionale Aspekte, wirtschaftliche Gesichtspunkte, Risiko-Klassifikation, Leistungstypen von Mitarbeitern, Klassifikation von Mitarbeiter-Qualifikationen und vieles mehr erfordern. Dies im Rahmen dieser Arbeit zu leisten, ist unmöglich. Es darf sogar bezweifelt werden, ob es in einer eigenständigen Arbeit bewältigt werden kann. Gründe finden sich in den Kosten und der Langwierigkeit der Prozeßabläufe. Theorien in der Praxis zu testen ist nahezu unfinanzierbar. Erfahrungswerte aus Industrieprojekten werden meist als vertraulich klassifiziert und entziehen sich so dem Zugriff.

Im letzten Kapitel sollen dennoch Begriffe an einem abstrakten Beispiel angesprochen werden, das durch die Diskussion gewebt ist. Erfahrungs- und Meßwerte aus zwei kleinen, nicht repräsentativen, Versuchsimplementierungen, die noch unter dem ersten Ansatz der *White-Box*-Modellierung entstanden sind [**13**], sollen Annahmen über Größenordnungen stützen. Wenn sich auch echte Prozeße der exakten Vermessung entziehen, ist der Versuch, Erkenntnisse allein durch Plausibilität zu gewinnen, nicht von vorn herein verwerflich. Die Bestätigung muß dann im realen Projekt erfolgen. Mögen die im folgenden diskutierten Begriffe die Aufmerksamkeit eines Managers mit Projektverantwortung schärfen und Ideen zu seiner Arbeit beisteuern.

KAPITEL 8

Risikozentrierter Prozeßentwurf

8.1. Einführung

Anforderungen an zeitliches Verhalten können in mannigfacher Weise gestellt werden. Ebenso vielfältig sind die Abhängigkeiten zwischen den Anforderungen an andere Komponenten einer entstehenden Software und genauso vielfältig die Möglichkeiten, wie im Software-Entwicklungsprozeß darauf reagiert werden kann. Es wird anhand der Auswahl typischer Muster ein Überblick gegeben.

Im folgenden Kapitel wird ein Prozeß betrachtet, der zum Ziel hat, neue Software zu entwickeln, die in einzelnen Funktionen performancekritisch ist, in anderen nicht. Die Performanzanforderung bei diesem Typ sind so geartet, daß beispielsweise die Einhaltung von Antwortzeiten gewisser Funktionen eine unverzichtbare Notwendigkeit ist. Eine nicht ausreichende Performanz ausgwählter Funktionen würde zum Scheitern des Projektes führen. Durch diese Vorgabe lassen sich alle Teile des Systems entweder denjenigen zuordnen, die von diesen Anforderungen betroffen sind, oder denjenigen, an die keine unmittelbaren Performanzanforderungen gestellt werden. Im letzten Kapitel wurde dargestellt, wie ein erheblicher Teil der Kosten einer Implementierungs- und Testphase vermieden werden kann, indem der Prozeß nach einer erfolglosen Performanzanalyse nach der Entwurfsphase abgebrochen oder verbessert wird. Da in dieser Darstellung aber noch das gesamte System modelliert werden mußte, waren Einsparungen nur im Bereich der Implementierung möglich. Durch die Zweiteilung der Untereinheiten bietet sich eine weitere Verfeinerung des Prozesses an, die es erlaubt, im Fehlschlagsfall sogar die Modellierungskosten der unkritischen Komponenten gespart zu haben. Die Idee ist, performancerelevante Teile vorrangig und unkritische Teile nachrangig zu modellieren. Zur Performanzanalyse ist ein vollständiges dynamische Modell notwendig. Es müssen nach dieser Vorgehensweise fragmentarische Modelle entwickelt werden, um die Abhängigkeiten mit den unkritischen Teilen aufzulösen. Dieser (neue) Artefakttyp stellt aber wiederum einen Kostenfaktor dar, der betrachtet werden muß. Bei geringem Fehlschlagsrisiko

kann es sinnvoll sein, einen Fehlschlag in Kauf zu nehmen, statt größere Investitionen zur Vermeidung zu tätigen, die keinen unmittelbaren wirtschaftlichen Wert an sich generieren. Die Quantifizierung der dazu nötigen Kennzahlen bleibt stets problematisch. Dennoch soll folgend ein differenzierter Prozeß diskutiert werden.

Klassifikation von Projekten. Bezüglich der zu entwickelnden Software läßt sich die in Tabelle 1 dargestellte Einteilung treffen.

Software	Aufgabe
Neue Software	Vorgehensmodell nach dem hier gezeigten Schema.
Erweiterung existierender Software	Entwicklung neuer Komponenten. Es bestehen Analogien zu der hier beschriebenen Methodik. Zusätzlich müssen existierende Komponenten analysiert und remodelliert werden. Die Interoperabilität der neuen Komponenten ist meist eine zentrale Eigenschaft. Dadurch wird ein gründliches Verständnis der Funktion aller bereits existierender Komponenten zu einem Erfolgsfaktor.
Software-Optimierung	Analyse, Reversmodellierung und Dokumentation der existierenden Software. Neuentwurf und Revision.

TABELLE 1. Basistypen verschiedener Softwareprojekte

Eine weitere Fragestellung kann auch die Verbesserung von Hardware-Ressourcen sein. Die Software wäre nur ein konstanter Faktor. Die hier beschriebene Technik läßt sich auch für diese Problemstellung einsetzen. Ein dynamisches Modell dient hier als Referenz. Das Umgebungsmodell steht auf dem Prüfstand. Die Messungen während der Ressourcenkalibrierung erzeugen die Artefakte, deren Variabilität Ziel der Performanzanalyse ist.

Die verschiedenen Ausgangssituationen sind in Tabelle 2 aufgeführt. Methodiken zur prozeßgesteuerten Entwicklung von Hardware sind ein eigenes Thema, weshalb es bei dieser schematischen Aufzählung bleibt.

8.2. Prozeß mit Fehlschlagsrisiko

Beispielhaft wird nun ein Softwareprozeß diskutiert, in dem neue Software erstellt werden soll. Gegeben seien mehrere Anwendungsfälle,

Hardware	Aufgabe
Neue Hardware	Erstellung eines Performanz-Softwaremodells nach dem im vorigen Kapitel beschriebenen Verfahren.
Hardware-Entwicklung	In diesem interessanten Fall wird ein weiterer Schritt in der Hardware-Modellierung notwendig. Ein vollständiges Performanzmodell der Software wird vorausgesetzt. Erste Ressourcenmodelle bauen auf Schätzungen auf, die während der Hardware-Entwicklung iterativ nachgeführt wird.
Vorhandene Hardware	Die Vorgaben für die Ressourcenmodelle liegen fest. Die Modellierung erfordert nur einen Schritt (Erstellung des Umgebungsmodells). Iterationen erfolgen nur für die Ressourcenkalibrierung.

TABELLE 2. Basistypen bei der Hardwareentwicklung

die als Anforderungen spezifiziert und von denen einzelne zeitkritisch sind. Dabei stelle ein Nichterreichen einer vorgegebenen Antwortzeit für die spezifizierte Funktionalität das gesamte Projekt in Frage stellt.

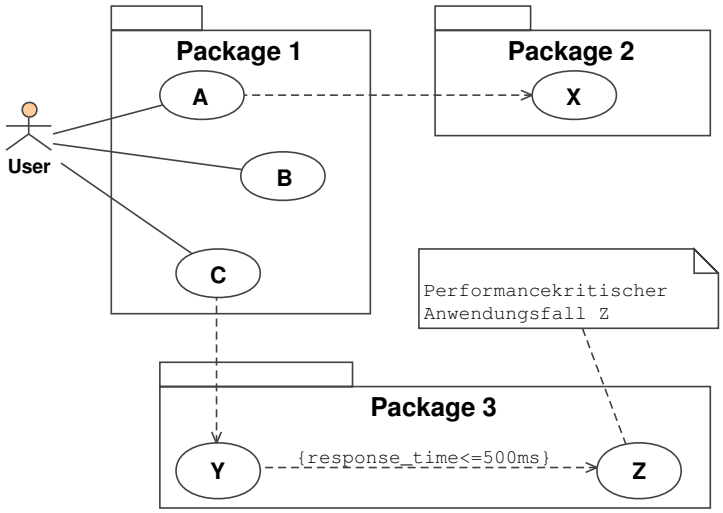


ABBILDUNG 8.2.1. Anwendungsfälle im Beispiel

8.2.1. Beispiel zeitkritischer Anwendungsfälle. Gegeben seien verschiedene Anwendungsfälle auf Subsysteme oder Pakete in der

Analyse verteilt worden. In den Anwendungsfällen sind jeweils zahlreiche funktionale Anforderungen formuliert. Abbildung 8.2.1 zeigt ein schematisches Beispiel. Zunächst ist erkennbar, daß der Anwendungsfall A auf den Anwendungsfall X, sowie der Anwendungsfall C auf Y aufbaut. Diese Abhängigkeiten funktionaler Eigenschaften werden von den in der Literatur bekannten Prozeßmodellen hinreichend behandelt. Dies würde nahelegen, die Pakete 2 und 3 nebenläufig, aber mit Vorrang vor Paket 1 zu implementieren. Werden die Abhängigkeiten aus funktionalen Eigenschaften ignoriert, ergäbe sich die Möglichkeit, alle drei Pakete nebenläufig zu implementieren. Abbildung 8.2.2 zeigt die nebenläufige Modellentwicklung als UML-Aktivitätsdiagramm.

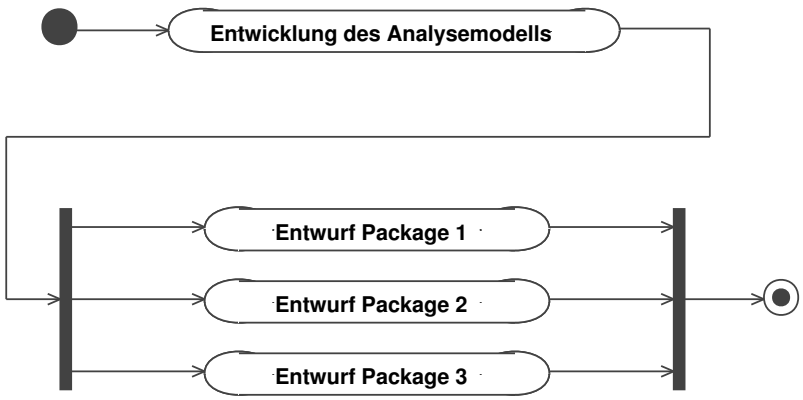


ABBILDUNG 8.2.2. Nebenläufiger Entwurf mehrerer Pakete

Ein fiktiver Anwendungsfall Z ist dabei mit einer Vorgabe für eine Mindest-Antwortzeit (hier 500 ms) versehen und stellt eine Performanzanforderung dar. Gegeben sei weiterhin, daß das gesamte System nicht zufriedenstellend funktioniert, falls diese Performanzanforderung verletzt wird. Eigentlich könnten die Pakete nach erfolgter Analyse unabhängig voneinander implementiert werden. So ergibt sich eine abstrakte Abhängigkeit aus dieser nicht funktionalen Anforderung. Die Implementierung der Pakete 1 und 2 ist unsinnig, falls die nicht funktionalen Anforderungen in Paket 3 nicht erreicht werden.

Die Modellierung von Untereinheiten eines Systems vorrangig zu behandeln, könnte hier bedeuten, daß alle Funktionalität, die mit Anwendungsfall Z zusammenhängt, modelliert wird, bevor Fragestellungen aus anderen Systemteilen behandelt werden. Es sind Abhängigkeiten der Anwendungsfälle untereinander dargestellt. Etwa die Abhängigkeit der Anwendungsfälle Y und Z. Daraus folgt, daß der Anwendungsfall Z

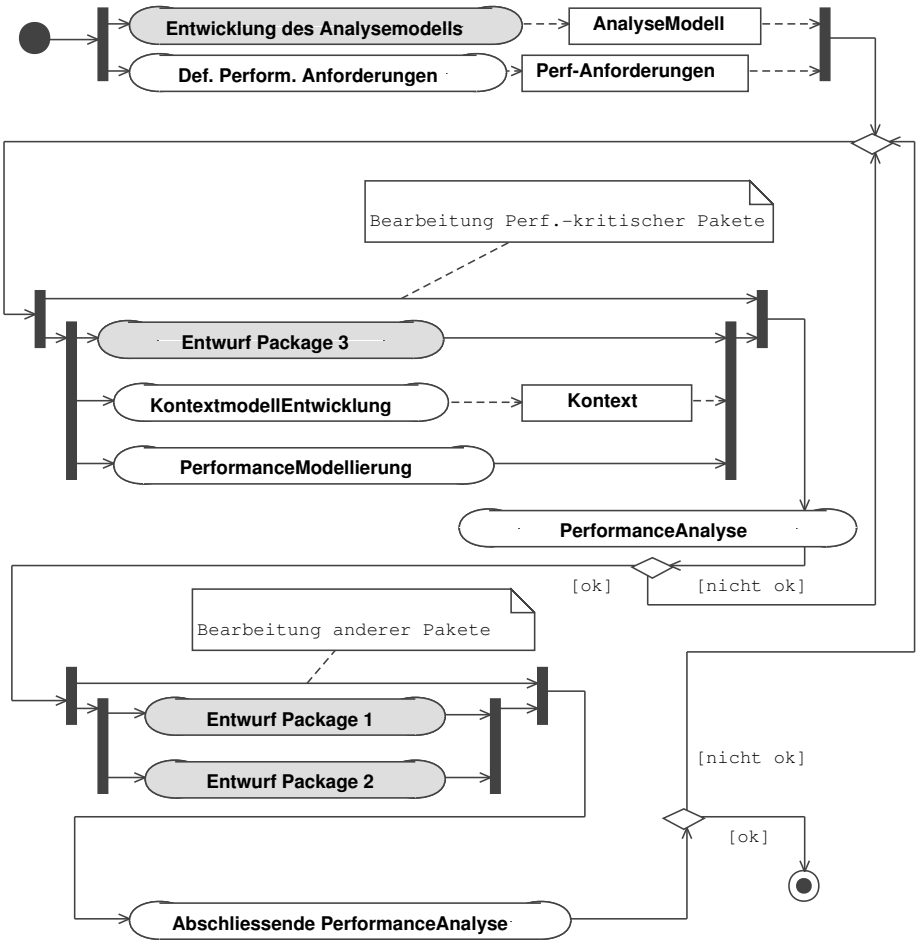


ABBILDUNG 8.2.3. Zusammenfassung der Entwicklung kritischer und unkritischer Subsysteme

möglicherweise auf Vorbedingungen aufbaut, die durch Nachbedingungen des Anwendungsfalls Y sicher gestellt werden. Dieser Zusammenhang kann möglicherweise eine zusätzliche Last der Ressourcen darstellen, die von der Bearbeitung dieser Anwendungsfälle betroffen sind. Weitere Abhängigkeiten, wie etwa die Bearbeitung durch eine geteilte Ressource, können zu ähnlichen Effekten führen.

Solche Abhängigkeiten können aufgelöst werden, indem die Funktionalität, die den Anwendungsfall Y realisiert, ebenso vorrangig behandelt wird. Oder indem, ein (weniger aufwendiges) Kontextmodell erstellt wird, das eine quantitative Abschätzung der Mehrbelastung erlaubt, aber die vollständige Modellierung umgeht. Abbildung 8.2.3 zeigt den modifizierten Prozeß, der die Abhängigkeiten auflöst, indem er das

ganze *Package 3* vorrangig behandelt. Mit der Modellierung der beiden anderen Pakete wird erst begonnen, falls eine Performanz-Analyse erfolgreich verlaufen ist.

Dennoch soll ein Projekt meist in kurzer Zeit abgeschlossen werden, sodaß Nebenläufigkeit erwünscht ist. Selbst wenn ein Projekt nicht so zeitkritisch ist, daß eine Bearbeitung nach dem Schema aus Abbildung 8.2.2 zwangsweise gegeben ist, muß ein Verfahren nach dem Schema aus Abbildung 8.2.3 wirtschaftlich begründbar sein. Diesen Konflikt zu bewerten, ist eine schwierige Aufgabe. Die dabei relevanten Begriffe können hier nur thematisiert werden.

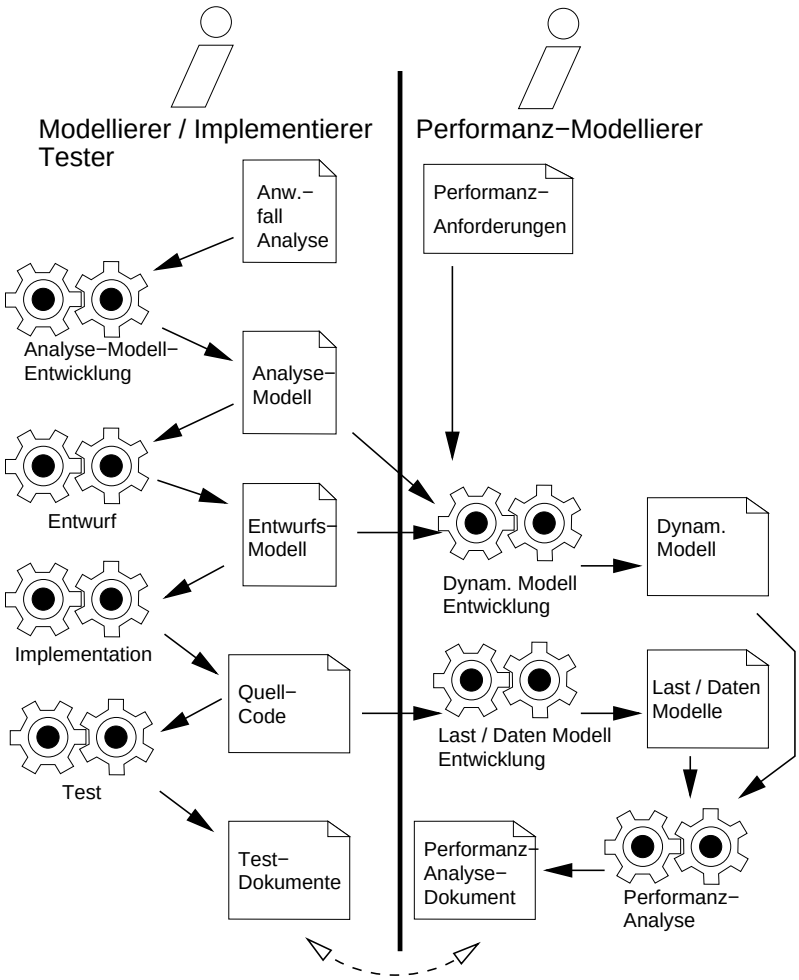


ABBILDUNG 8.2.4. Software Entwicklung

8.2.2. Softwaremodellierung. Die im vorhergehenden Abschnitt bereits eingeführten Artefakte werden hier nochmals im Überblick in Abbildung 8.2.4 gezeigt. Die Entwicklung eines dynamischen Modells, die Last- und Datenmodelle und die Erstellung einer Performanz-Analyse, vor allem ihre Herstellungskosten werden nachfolgend diskutiert.

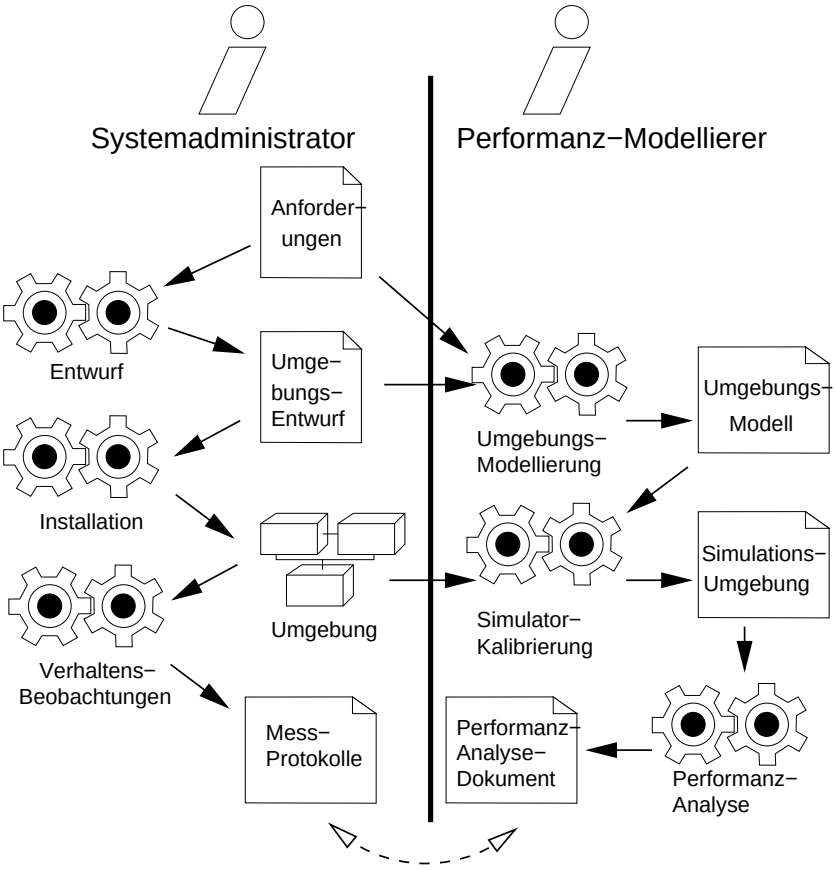


ABBILDUNG 8.2.5. Hardware Administration

8.2.3. Umgebungsmodellierung. Die quantitative Abschätzung des zeitlichen Verhaltens verlangt vor allem die aufwendige Modellierung der Laufzeitumgebung. In Abbildung 8.2.5 wird das Verfahren schematisch dargestellt, welches beim Aufbau neuer Hardware anwendbar ist. Nach der Spezifikation der Anforderungen an Leistungsfähigkeit, Kapazität und Architektur der Umgebung kann die benötigte Hardware eingekauft und installiert werden. Auf Messungen basierende Performanz-Eigenschaften können nicht verfügbar sein, bevor die

lauffähige Umgebung verfügbar ist. Nur durch Schätztechniken können diese Informationen vorweg genommen werden.

Aus dem Umgebungsmodell kann eine Simulationsumgebung erzeugt werden, deren Kalibrierung nach den gemessenen Verhaltensbeobachtungen den entscheidenden, erheblichen Zeitaufwand fordert. Da eine solche Messung dynamische Modelle, Last- und Datenmodelle und authentische Quellcode-Beispiele erfordert, kann sie typischerweise erst sehr spät durchgeführt werden.

8.3. Zusätzliche Aspekte des Prozeßentwurfs

In der klassischen Literatur [**40**] wird dem Aspekt der Umsetzung funktionaler Anforderung weiter Raum eingeräumt. Die Erfassung und Umsetzung von Performanz-Eigenschaften, die Thema dieser Arbeit sind, ist ein Beispiel der Betrachtung nicht-funktionaler Eigenschaften. In diesem Abschnitt wird ein weiterer nicht-funktionaler Aspekt behandelt: die Kosten der Umsetzung und deren strukturelle Abhängigkeiten zur Architektur. Anders als Performanz-Eigenschaften sind diese nur mittelbar mit den Artefakten der Anforderungsanalyse verbunden.

Aus Platzgründen wird hier nur ein spezifisches Beispiel dargestellt. Ein Software-Entwicklungsprozeß kann über mehrere Iteration gesteuert werden, um weiche Kriterien wie harmonisches Gesamtverhalten, gute Balance in der Nutzung der Ressourcen oder Minimierung der Antwortzeiten zu erfüllen. Das aktuelle Beispiel konstruiert ein fiktives hartes Kriterium, welches das Projekt scheitern läßt, falls das Ziel nicht erreicht wird. Der Aspekt der Entwicklungskosten rückt so plastischer in den Vordergrund.

8.3.1. Frühzeitige Abschätzung. In jedem Fall ist eine frühzeitige Bewertung von Modellen wünschenswert. Wie bereits in Kapitel 6 eingeführt, unterliegen die Simulations-Artefakte ebenso wie die Modellartefakte einer inkrementellen Verbesserung. Da die Steuerung des Entwicklungsprozesses bereits in der ersten Iteration greifen muß, ist man hier auf Schätzungen angewiesen, die später durch gemessene Artefakte ersetzt werden.

Abbildung 8.3.1 zeigt schematisch den Ablauf dieser ersten Iteration. Im Wiederholungfall werden die Schätz-Artefakte durch die Artefakte der jeweils vorhergehenden Iteration ersetzt bzw. Schätzungen durch Rekalibrierungsaktivitäten. Von den mit Stern markierten Aktivitäten und Artefakten betrachten wir diejenigen Instanzen, die nur der Steuerung des Prozeßflußes selbst dienen.

In der Kongruenzanalyse soll einerseits die Nähe zum erstrebten Ziel quantifiziert und damit der Eintritt in die nächste Phase angestoßen

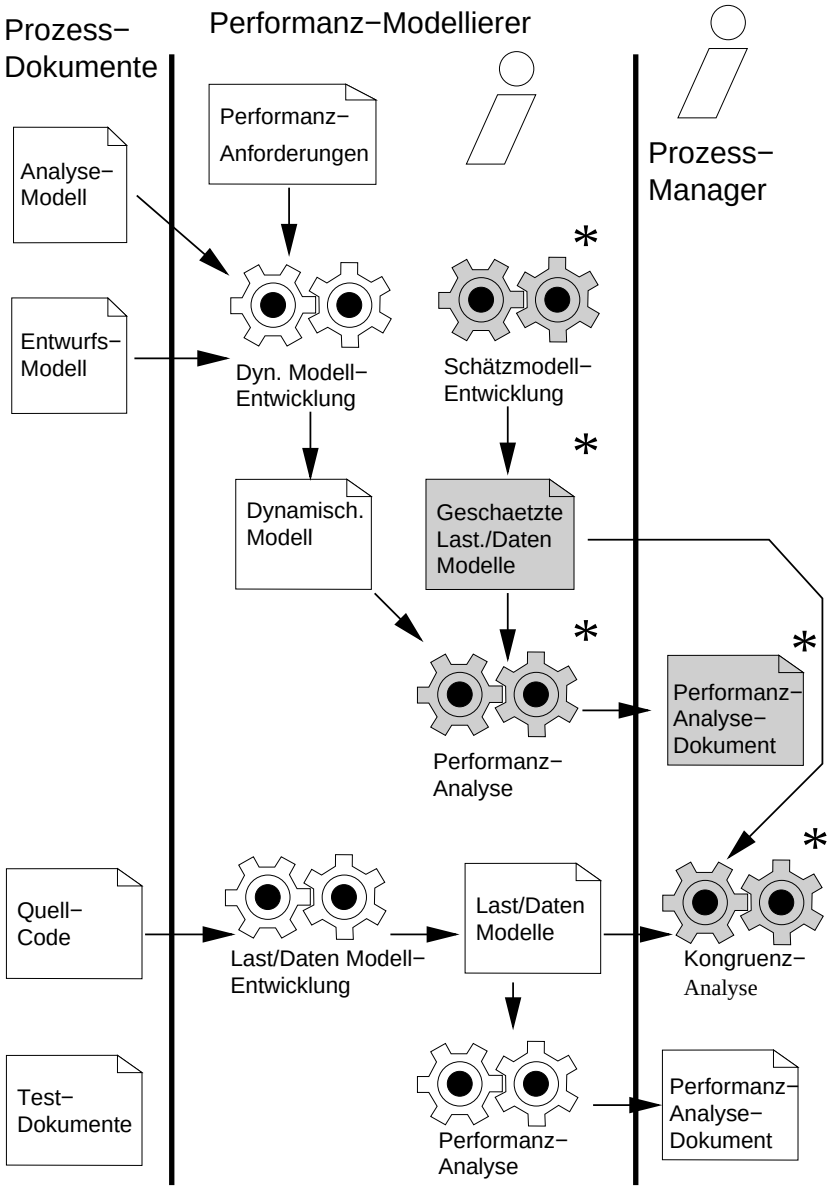


ABBILDUNG 8.3.1. Prozeßsteuerungs-Artefakte innerhalb einer Iteration

werden. Andererseits sollen mit den Artefaktinstanzen, die qualitative Projektinformationen erhalten, die Arbeitsaufgaben einer folgenden Iteration identifiziert werden.

Die Instanzen der grau dargestellten Artefakte und Aktivitäten müssen zur Justierung des Prozesses zusätzlich erzeugt werden. Dies verursacht erhebliche Kosten. Die Berechnung der Kosten als eigener

Aspekt und deren Einfluß auf die Prozeßsteuerung ist ebenso Gegenstand der Kongruenzanalyse, kann aber im Rahmen dieser Arbeit nicht erörtert werden. Die Klassifikation aller Parameter und die Formalisierung eines Prozeßmodells würde eine eigene Theorie mit erheblichem Ausmaß erfordern. Deshalb beschränkt sich die Einführung dieses Kapitels mit einer informellen Thematisierung.

8.3.2. Nebenläufigkeit bei der Paketentwicklung. Wie bereits in Abbildung 8.2.2 dargestellt, kann die Entwicklung unabhängiger Pakete nebenläufig durchgeführt werden.

Parallelisierung abhängiger Pakete. Das abstrakte Beispiel aus Abbildung 8.2.1 zeigt sequentielle Abhängigkeiten, die auch eine sequentielle Entwicklung abhängiger, dh. aufeinander aufbauender Pakete und Komponenten nach sich ziehen. Unter Zeitdruck kann eine nebenläufige Entwicklung ermöglicht werden, indem Abhängigkeiten durch Kontextkomponenten aufgelöst werden. Beispielweise durch Test an Frameworks oder Simulationsumgebungen, die bei Übersetzung und Test von Teilkomponenten die Aufgaben fehlender Bestandteile übernehmen. Diese Kontextkomponenten werden später verworfen. Ihre kostspielige Entwicklung muß also immer dem erreichten Zeitgewinn wirtschaftlich gegenübergestellt werden.

Im hier diskutierten speziellen Beispiel eines kritischen Projektes ergibt sich ein entgegengesetztes Bild. Das mögliche Scheitern eines ganzen Projektes bei Nichterreichen eines Performanzwertes ist mitunter ein Argument für eine erzwungene Sequentialisierung der Paketentwicklung. Kann ein mit Vorrang entwickeltes Performanz-kritisches Paket nötige Anforderungen nicht erfüllen, wurden zur Entwicklung der als nachrangig eingestuften Pakete mit Einschränkungen noch keine finanziellen Mittel verwendet. Im behandelten Beispiel würde ein Scheitern der Implementierungen für Anwendungsfall Z dazu führen, daß die Anwendungsfälle A, B, C und X nicht entwickelt werden. Die Einschränkung ergibt sich aus den Kosten der auch hier bereitszustellenden Kontextkomponenten. Somit ist die Frage nach der Zweckmäßigkeit einer vorrangigen Behandlung kritischer Pakete nicht kategorisch zu beantworten.

Priorisierung von Prozeßsträngen. Die getrennte Behandlung von vorrangigen und nachrangigen Paketen wurde in Abbildung 8.2.3 schematisiert. Wesentlich ist die Darstellung zweier in Reihe geschalteter Schleifen. Kritische Pakete werden in der ersten Schleife vorrangig behandelt. Der Prozeß tritt in die zweite Schleife erst ein, wenn die Performanz-Analyse das Erreichen der kritischen Anforderungen für ausreichend sicher hält. Das Bild ist trotz einer abstrakten Darstellung

nur als allgemeiner Lösungsansatz für das hier diskutierte abstrakte Problem zu betrachten.

Da mit der im ersten Teil beschriebenen Technologie eine Bewertung bereits im Modellstadium erfolgen kann, gilt dieses Schema nicht nur für eine Iteration, sondern kann als Mikrozyklus jeweils auf Entwurf und Implementierung getrennt angewendet werden. Dadurch wird dem Wunsch einer möglichst frühzeitigen Bewertung in hohem Maße Rechnung getragen und eine Verringerung des wirtschaftlichen Risikos erreicht.

Planung nebenläufiger Prozeßstränge. Unter den Bedingungen des hier behandelten Beispiels ergeben sich für den Entwicklungsprozeß nur zwei Szenarien, ein erfolgreiches Projekt oder sein Fehlschlag.

Ein erfolgreiches Projekt rechtfertig alle Investitionen, die unmittelbar in Zusammenhang mit der Funktionalität und den Eigenschaften stehen, die den Erfolg dieses Projektes begründen. Der finanzielle Aufwand für Kontextkomponenten war aus Sicht des erfolgreichen Projektes aber umsonst, da sie nicht Bestandteil der Lösung sind und nur zur Risiko-Minimierung gedient haben, also auch zum Erreichen der Lösung nicht notwendig gewesen sind.

Im Falle eines Fehlschlages aber erlauben eben diese Komponenten die frühzeitige Entscheidung zum Projektabbruch. Die Differenz der Kosten für die Kontextkomponenten und den vermiedenen Kosten bis zur Fertigstellung des als Fehlschlag identifizierten Projektes können als finanzieller Erfolg (in Form von Kostenvermeidung) des Risikomanagments betrachtet werden. Aus der Vermutung, daß dieser Differenzbetrag mitunter negativ werden kann ergeben sich die in diesem Abschnitt folgenden Betrachtungen.

8.3.3. Fallbeispiele. Zunächst soll ein Gefühl für typische Größenordnungen und Proportionen von Arbeitszeitaufwendungen entwickelt werden. Dazu werden zwei kleine Fallbeispiele mit unterschiedlichen Zielsetzungen betrachtet.

Beispiel einer Treiberimplementierung. In einem kleinen exemplarischen Versuchsprojekt wurde eine Treibersoftware für ein Steuergerät implementiert und von der Anforderungsanalyse bis zum Test eine Arbeitszeiterfassung durchgeführt. Abbildung 8.3.2 zeigt quantitativ die Aufwendungen in Stunden aufgegliedert in die drei Iterationen [1] und nach ihrer Zugehörigkeit zu den Bereichen Modellierung/Requirementsanalyse, Implementierung/Test und gesondert die

[1] Die hier dargestellten drei Iterationen hätten nach der Nomenklatur des `Rational Unified Process` auch als 3 Phasen mit je einer Iteration dargestellt werden können.

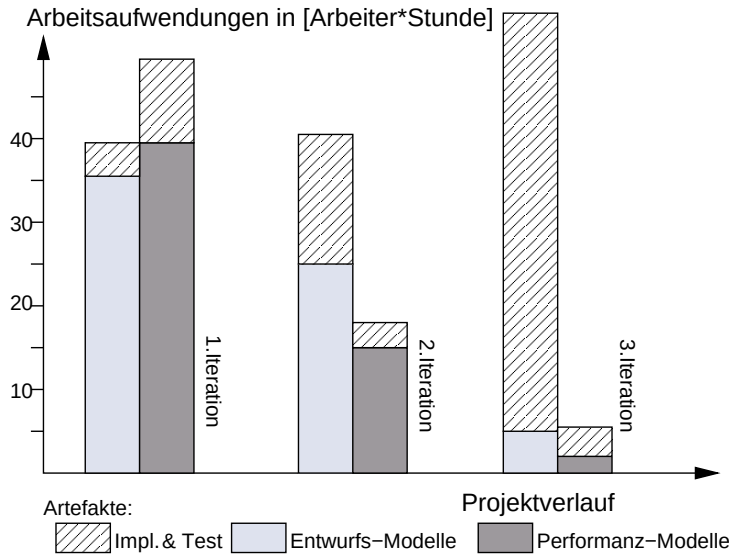


ABBILDUNG 8.3.2. Arbeitsaufwendungen im Prozeßverlauf

Entwicklung von Performanzmodellen. Implementierung und Test repräsentieren im Balken der Performanzmodelle quantitative Zeitaufwendungen für Testfallspezifikation, Simulation und Ergebnisauswertung.

Die Dynamik des Steuervorgangs erforderte eine strenge Einhaltung von Antwortzeiten, da sonst der Regelvorgang in Echtzeit undurchführbar ist. Hierzu wurden die kritischen Funktionen vorrangig implementiert und getestet. Da diese Software eine vollständige Neuentwicklung war, mußten zunächst mit dem Entwurfsmodell und der Analyse einige Quellcode-Fragmente entwickelt werden (Aufwand 4 Stunden), um grundlegende Hardwarefunktionen zu verstehen. Den größten Arbeitszeitanteil aber verschlang in der ersten Iteration die Entwicklung des dynamischen Performanz-Modells. Dazu mußten einige simple Codebeispiele implementiert werden, um Test- und Meßläufe auf dem Steuergerät auszuführen. Teile der Quellcodefragmente fanden hierbei Verwendung. Dadurch konnte das initiale Umgebungsmodell kalibriert werden. Der ersten Abschätzung des Laufzeitverhaltens lag die Annahme zugrunde, die angestrebte Implementierung würde aus Codebestandteilen aufgebaut sein, die in ihrem zeitlichen Verhalten den zur Kalibrierung verwendeten gleichen. Die so durchgeführte Simulation ergab ein positives Ergebnis und lies den Erfolg einer Implementierung nach diesem Modell erwarten. In der zweiten Iteration wurde das Modell verfeinert. Erst jetzt wurde der eigentliche Teil der Implementierung angefertigt. Lediglich ein Interpolationsalgorithmus zur Berechnung von

Einstellungen, die zwischen definierten Zuständen des Steuergerätes lagen, wurde auf die letzte Iteration verschoben. Umgebungs- und Lastmodelle wurden mit Fragmenten des tatsächlich verwendeten Codes rekalibriert.

Die Implementierung des Interpolationsalgorithmus stellte die größte Zeitaufwendung in der Entwicklung dieses Treibers dar, obwohl in der Literatur bereits gut beschriebene mathematische Verfahren (kubische Splines) als Modellgrundlagen verfügbar waren. Mit einer gut kalibrierten Testumgebung konnte jeder Teilfortschritt auf Einhaltung der Laufzeitobergrenzen getestet werden. Jeder Schritt erwies sich in diesem Beispiel als unkritisch.

Die Performanzmodellierung in einem solch überschaubaren Projekt ist wirtschaftlich nicht begründet. Das Beispiel belegt aber eine typische Verteilung von Arbeitsaufwand auf die einzelnen Artefakttypen. Augenfällig ist hier der große Aufwand für die Performanz-Modellierung. Wäre die Simulation nach Abschluß der ersten Iteration so deutlich negativ verlaufen, daß man von einer weiteren Entwicklung Abstand genommen hätte, wäre dieser hohe Aufwand dennoch lohnend gewesen. Der in den beiden folgenden Iterationen betriebene Aufwand für die Implementierung der funktionalen Eigenschaften belief sich auf 118 Stunden. Durch die Entscheidung, das Projekt abzubrechen, stehen diese eingesparten 118 Stunden den aufgewendeten 50 Stunden für die Performanz-Analyse gegenüber. Qualitative Verbesserungen wurden in diesem Beispiel nicht bewertet.

Bewertung unterschiedlicher Entwürfe. In einem zweiten Fallbeispiel war die Frage zu klären, ob eine gewisse Funktionalität (Datenmanagement) zweckmäßig lokal durchgeführt oder auf den Server transferiert werden sollte. In Abbildung 8.3.3 werden die Arbeitszeitaufwendungen dieses zweiten Fallbeispiels analog zu dem ersten aufgeschlüsselt. In der ersten Version wurde die Client-Version der Architektur modelliert, in der zweiten die Server-Variante. In der zweiten Iteration ist der geringere Aufwand für die Performanz-Modellierung. Beide Architekturen verwenden das gleiche Umgebungsmodell und ähnliche Last- und Datenmodelle. Ohne die Verwendung von simulationsbasierter Performanz-Modellierung hätten für einen Vergleich beide Modell-Varianten implementiert und getestet werden müssen.

In diesem Beispiel ergibt sich zwar kein typischer Vorteil für die Ausdruckskraft der in dieser Arbeit beschriebenen, dynamischen Modelle. Einige klassische Performanz-Modellierungstechniken wie Markovketten hätten ebenso verwendet werden können. Dennoch läßt sich aus der Anwendung von Aktivitätsdiagrammen zur Beschreibung von dynamischen Performanz-Modellen ein Vorteil ziehen. Diese können

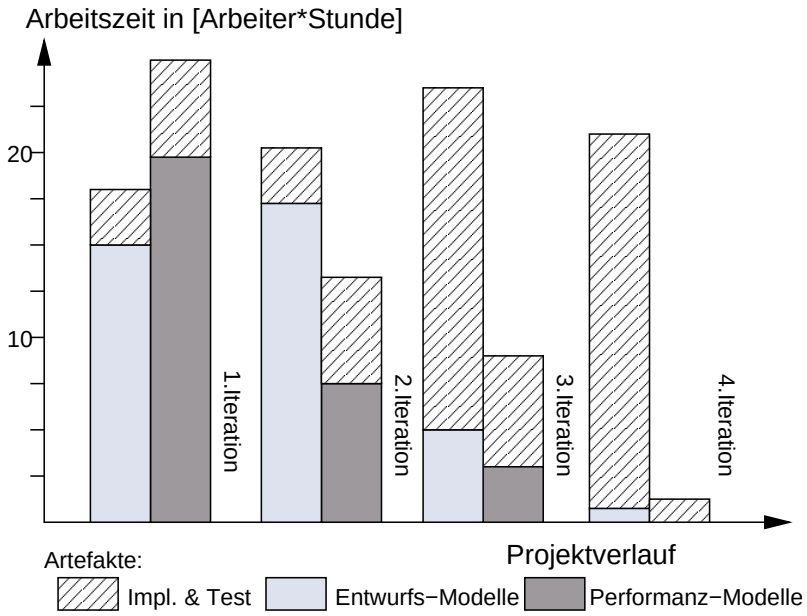


ABBILDUNG 8.3.3. Modellierung und Bewertung unterschiedlicher Entwurfsmodelle

schon als Implementierungsmodell betrachtet und fast vollständig zur Spezifikation einer Umsetzung in Quellcode genutzt werden. Eine zusätzliche Beschreibung des zeitlichen Verhaltens auf einer eigenen Abstraktionsebene (Entwicklung einer stochastischen Funktion) entfällt.

8.3.4. Gesteuerter Prozeß. Bislang wurden in diesem Kapitel (ohne den Anspruch auf Vollständigkeit) Artefakte des klassischen Prozesses, Artefakte der Performanz-Modellierung und zusätzliche Artefakte, die mit der vorrangigen oder nebenläufigen Bearbeitung von Aufgaben in Zusammenhang stehen, thematisiert. Folgt man der Strategie, kritische Pakete mit Vorrang zu behandeln, ist die Entscheidung zur nachrangigen Behandlung von unkritischen Paketen dann eindeutig gegeben, falls diese nicht von kritischen Paketen abhängig sind. Existiert eine solche Abhängigkeit ist die Zuordnung zu klären.

Diese Initialisierung des Software-Entwicklungsprozeßes ist einmalig für alle Pakete durchzuführen. Der Vorgang wird in Abbildung 8.3.4 schematisch dargestellt. Die Feststellung der Zuordnung wird im Folgenden diskutiert.

Wie im letzten Abschnitt angedeutet war, spielen dabei die Aufwendungen an Kosten zur Erstellung einzelner Artefakte eine große Rolle. Ist der einzige Zweck der Performanz-Modellierung die Verringerung des Fehlschlagsrisikos eines kritischen Projektes und übersteigen

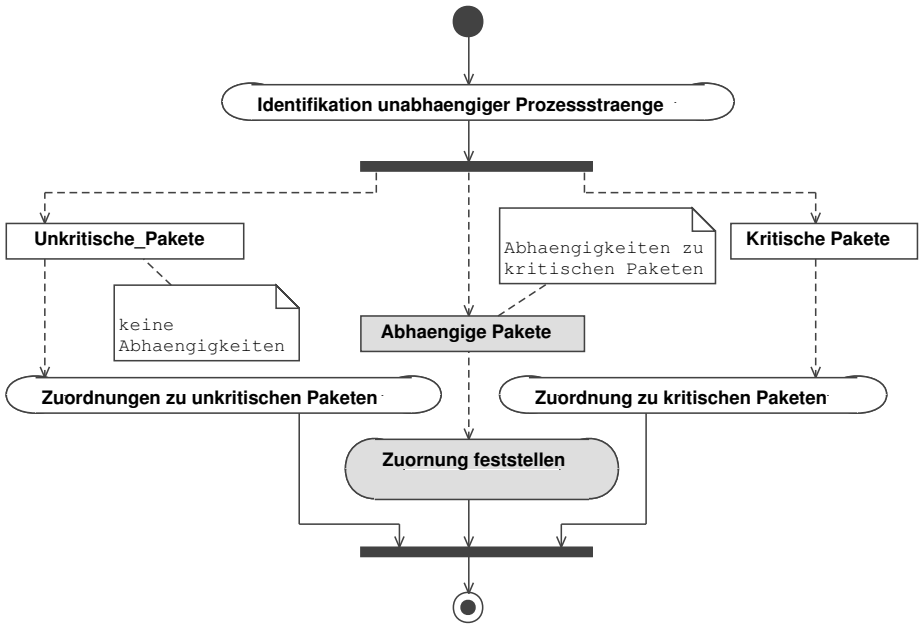


ABBILDUNG 8.3.4. Zuordnungsalgorithmus zu Workflow-Bereichen

die Kosten der Performanz-Modellierung die Kosten der Implementation, ist es offensichtlich nicht zweckmäßig, ein Risikomanagement durchzuführen. Im umgekehrten Fall gilt die Umkehrung der Entscheidung aber auch nicht zwingend. Zur vorausschauenden Abschätzung von Entwicklungskosten einzelner Teilfunktionen sind mehrere Techniken (z.B. Function-Point-Analysis) in der Literatur beschrieben worden. Diese Techniken sollen hier nicht erläutert werden. Wir gehen davon aus, daß ausreichend präzise Schätzungen und Erfahrungswerte zur Verfügung stehen. Hätte im ersten Beispiel in Abbildung 8.3.2 eine Aufwandsschätzung für die Performanzmodellierung 50 Stunden ermittelt und für die Modellierung einen kleineren Betrag, hätte man wohl auf die Entwicklung eines dynamischen Modells verzichtet.

Doch selbst wenn die Kosten des Risikomanagements durch Performanz-Modellierung wesentlich kleiner sind als die Aufwendungen zur Implementierung der Funktionalität, kann es sinnvoll sein, einen Fehlschlag ohne Risikomanagement zu riskieren. Dazu folgendes Gedankenexperiment: Betrachtet wird die Klasse an Paketen mit gleichem Fehlschlagsrisiko f_{risk}. Es sei f_{risk} abschätzbar oder aus Erfahrungswerten ermittelbar. Seien weiterhin die Kosten c_{Mod} ermittelbar, die aufgewendet werden Artefakte zu erzeugen, die ausschließlich

Bestandteil der Problemlösung sind. Seien schließlich die Kosten der Kontext-Modellierung c_{Cont} ermittelbar, die aufgewendet werden Artefakte zu erzeugen, die ausschließlich dem Risikomanagement dienen und nicht Bestandteil der Lösung sind.

- f_{risk} : Fehlschlagsrisiko, mit: $0 \leq f_{risk} \leq 1$,
  dabei ist 0 das sichere und 1 das unmögliche Ereignis,
- c_{Cont} : Absolute Kosten der zusätzlichen Artefakte,
- c_{Mod} : Kosten des Lösungs-Entwurfes.

Dann wäre durch eine Funktion *prio* eine Entscheidung für die Mehrkosten der Kontextmodellierung berechenbar nach:[2]

$$\begin{aligned} & prio \;\; \rightarrow \;\; \mathbb{B} \\ mit: \; prio = & \begin{cases} \texttt{false}, & \text{falls } ((1 - f_{risk}) * c_{Cont}) \; < \; (f_{risk} * c_{Mod}) \\ \texttt{true}, & \text{sonst.} \end{cases} \end{aligned}$$

In der Praxis werden Entscheidungen oft nach der einfacheren Intuition getroffen. Das priorisierte Modellieren sei dann zweckmäßig, falls die absoluten Kosten für die Kontextmodellierung geringer sind als die Modellierung des ganzen Paketes, also $c_{Cont} < c_{Mod}$. Die Differenz $c_{Mod} - c_{Cont}$ wäre positiv und damit etwas zu gewinnen. An einem abstrakten Rechenbeispiel sei gezeigt, daß diese Intuition falsch ist.

BEISPIEL 13. *Projekt mit aufwendiger Kontextmodellierung*

Sei ein Paket mit einer Fehlschlagswahrscheinlichkeit von 10% behaftet, dh. $f_{risk} = 0,1$. *Der Aufwand für die Modellierung des Paketes sei* $c_{Mod} = 10h$. *Der Aufwand für die Kontextmodellierung sei* $c_{Cont} = 5h$.

Nach der einfachen Intuition würden man sich für die Entwicklung eines Kontextmodells entscheiden, da die Kosten geringer sind als die Kosten für das vollständige funktionale Modell. Bezogen auf die richtige Grundgesamtheit wäre diese Entscheidung aber falsch und würde einen erheblichen Verlust bedeuten, was man leicht an Tabelle 3 ablesen kann: Erst durch die Einbeziehung des Risikofaktors f_{risk} *in die Entscheidungsfindung erhält man die richtig gewichtete Bewertung der Leistungsaufwendungen. Dem Wert von prio folgend würde man die Modellierung priorisieren und über einen längeren Zeitraum die geringeren Verluste erleiden, indem die gelegentlich vergeblich investierten Stunden in die Modellentwicklung toleriert werden.*

[2]Um sprachliche Verwirrungen zu vermeiden sei darauf hingewiesen: Priorisierung meint hier die frühzeitige Modellentwicklung durchzuführen und auf eine Kontextmodellierung zu **verzichten**.

10 Projekte ($f_{risk} = 0,1$):	*Nachrangige Behandlung mit Kontextmodellierung*	*Priorisierte vollständige Entwicklung*
9 erfolgreiche Projekte	$9 * c_{Cont} = 45h$ *(vergebl. Kontextmodellierung)*	*keine vergeblichen Aufwendungen*
1 erfolgloses Projekt	*keine vergeblichen Aufwendungen*	$1 * c_{Mod} = 10h$ *(vergebl. Modellentwicklung)*
Summe	*45 h*	*10 h*

TABELLE 3. Quantitativer Vergleich vergeblich aufgebrachter Arbeitsaufwendungen über mehrere Projekte

Nutzt man diese Erkenntnis bei der anfänglichen Entscheidung für die Zuordnung unkritischer, aber abhängiger Pakete ein, ergibt sich das Schema nach Abbildung 8.3.5. Der Risikofaktor klassifiziert das Fehlschlagsrisiko des gesamten Projektes. Kontextmodellierungskosten und Kosten der Modellentwicklung werden jeweils für die Pakete spezifisch geschätzt. Diese Zuordnung realisiert die Zuordnungsaktivität aus Abbildung 8.3.4.

Alle in diesem Abschnitt eingeführten Begriffe sind hoch problematisch. Jede aufgewendete Arbeitsstunde beinhaltet gewöhnlich das Schaffen von Artefakten, die allen hier klassifizierten Dokumenttypen in verschiedenem Maß zugeordnet werden können. Eine eindeutige Klassifikation ist praktisch nicht durchführbar. Ebenso ergeben sich aus den Abhängigkeiten der Dokumente untereinander Redundanzen.

Die konkrete Management-Entscheidung wird weiterhin auf Erfahrung gestützt sein. Mit solchen abstrakten und sehr punktuellen Überlegungen kann es aber gelingen, diese Intuition mit passenderen Mustern zu versorgen und ihre Verläßlichkeit deutlich zu verbessern.

8.4. Zusammenfassung

Über die Verwendung der Methodik. Oftmals werden formale Modellierungstechniken gerade dann zu komplex, wenn die Problembeschreibung interessant wird. Der Aufwand der hier vorgestellten Simulationstechnik ist zwar oft nicht weniger komplex. Sie bleibt aber in ihrer Abstraktion näher an den funktionalen Aspekten. Dadurch bringt ein Aufwand für die Modellierung nichtfunktionaler Eigenschaften mit der hier vorgestellten Methodik auch einen Gewinn für den klassischen Entwurf. Der hohe Aufwand läßt sich dadurch wirtschaftlich besser rechtfertigen. Nicht zuletzt die Anwendung der Methodik auf die UML hat daran für Projekte, die mit der UML modelliert werden entscheidenden Anteil. Durch den fließenden Übergang von den geschätzten

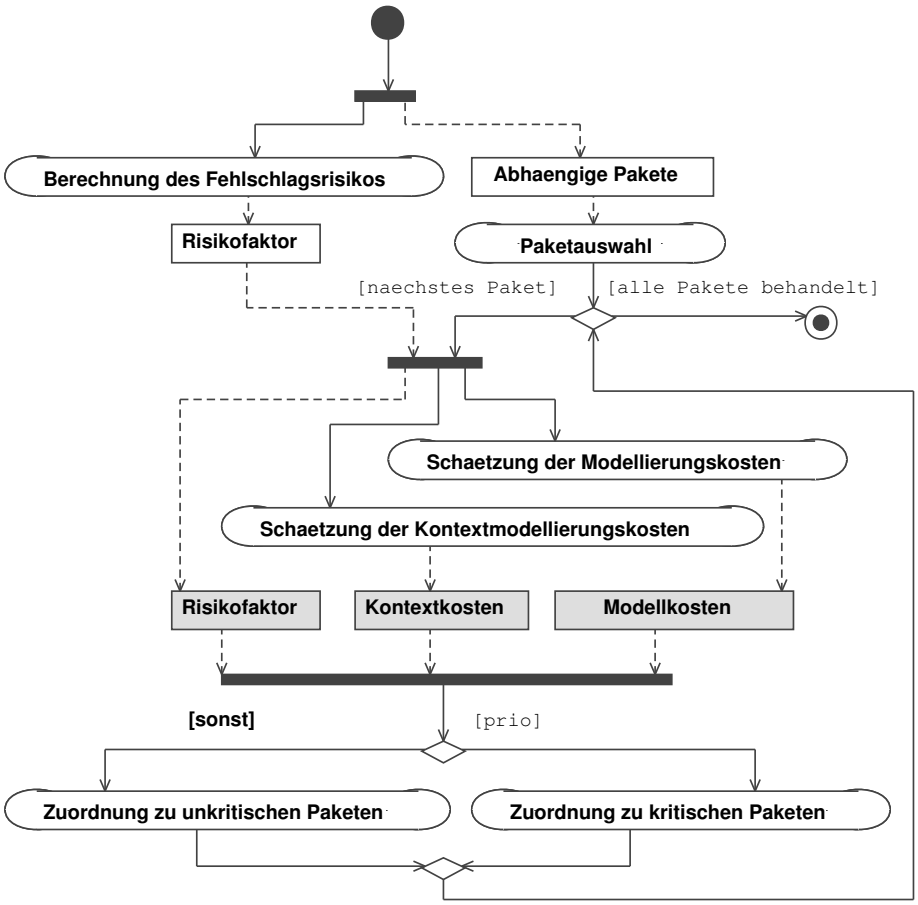


ABBILDUNG 8.3.5. Zuordnung unkritischer aber abhängiger Pakete

zu den gemessenen/kalibrierten Funktionen und der freien Wahl der gewünschten Abstraktionsebene ist diese Methodik flexibel einsetzbar.

Reproduzierbarkeit und Genauigkeit dieser Technik und der Methodik lassen sich durch exaktere Modelle verbessern und den Notwendigkeiten anpassen. Die Ausdruckskraft der dynamischen Modelle wurde gegenüber einfachen Netzmodellen deutlich verbessert. Die Möglichkeit, die nebenläufige Bearbeitung von Lasten durch Ressourcen als eigenen zustandsbehafteten Aspekt auszudrücken, ist eine große Verbesserung. Bei Markovprozessen ist man meist gezwungen, alle Vorgänge zustandslos zu formulieren. Sich erst in der Bewertung eines Ablaufes ergebende Nebenläufigkeiten könne nicht ausgedrückt werden, da die stochastischen Funktionen darauf nicht spezifisch reagieren.

Jedoch ist auch diese Technik nicht frei von Einschränkungen. Der Ansatz erfordert eine feste Spezifikation des Lastlokalisation und ist daher zur Modellierung von Load-Balancing-Systemen nicht geeignet.

Über die Integration in den klassischen Prozeß. In Kapitel 6 wurden neue Rollen und Artefakte eingeführt, die mit einer Behandlung der Performanz als nicht funktionale Eigenschaft einer zu entwickelden Software neu auftreten. Die Rollen mit ihren Aufgaben und Ansprechpartnern wurden skizziert und eine Integration in einen klassischen iterativen Softwareentwicklungsprozeß, der Analyse, Entwurf, Implementierung und Test unterscheidet, vorgeschlagen.

Der Performanz-Modellierer wurde als eigene Rolle empfohlen. Es ist sogar vorteilhaft, diese Rolle mit einer eigenen Person zu besetzen, die diese Rolle extern oder projektübergreifend innerbetrieblich wahrnimmt. Der Vorteil liegt darin, daß eine Person, die nicht zusätzlich mit der Modellierung funktionaler Eigenschaften betraut ist, Kritik und Änderungsvorschläge unbefangener entwickeln und beisteuern kann.

Die Steuerung des Prozesses durch quantitative Bewertungen und Vorhersagen bleibt auch mit einer Bewertungstechnik und Methodik, die stabile und repoduzierbare Werte liefert schwierig. Die Abschätzung der Wichtigkeit nicht funktionaler Eigenschaften und die Möglichkeit wichtige Abhängigkeiten zu übersehen, kann keine Methodik leisten.

Über das Prozeßmanagement. Im Kapitel 8 schließlich wurden verschiedene Risiken diskutiert und an einigen Beispielen Zusammenhänge aufgezeigt. Es ist deutlich geworden, welch großes Feld abgedeckt werden müßte, um alleine die verschiedenen Begriffe bezüglich eines wirtschaftlichen, performancetechnischen oder funktionalen Risikos zu klassifizieren. Eine auf Bewertung von Artefakten beruhende Formalisierung von Management-Entscheidungen ist eine Aufgabe, die eine eigene, erheblich größere wissenschaftliche Auseinandersetzung mit diesem Thema fordert. Zumal eine Validierung dort aufgestellter Theorie am praktischen Beispiel wegen der in der Softwareentwicklung extrem hohen Projektkosten rein zu experimentellen Zwecken kaum zu leisten sein wird.

Dennoch wurde versucht, durch abstrakte gedankliche Experimente aufzuzeigen, wie man im Projekt auch die Entscheidungen hinterfragen kann, bei denen man sich mangels Theorie auf Intuition verlassen muß. Wie beispielsweise in dem Fachgebiet der Medizin, gibt es auch in der Informatik Sachverhalte, bei denen sich nicht die Möglichkeit bietet, sie im Versuch zu erforschen.

Doch ist es nicht das Ende eines Weges zu neuer Erkenntnis.

Schlußwort

In dieser Arbeit wurde das zeitliche Verhalten von Software thematisiert. Es wurde eine Möglichkeit geschaffen, dynamische Modelle zu spezifizieren, die eine Klasse möglicher Abläufe beschreiben. Die Struktur dieser Klasse wurde näher untersucht und eine Möglichkeit zur Spezifikation einzelner Teilmengen dieser Klassen, sogenannte Szenarien zu spezifizieren. Dies wurde so gestaltet, das sich einzelne konkrete Abläufe aus einem Szenario nur in der Reihenfolge der Termination unabhängig konkurrierender Lasten unterscheiden und die konkreten Abläufe bezüglich der Interaktion mit der Außenwelt äquivalent sind. Diese Reihenfolge der Termination konnte erst bestimmt werden, indem Funktionen die Dauer jeder einzelnen Lastbearbeitung quantifizierbar machten. Die Menge dieser Funktionen, die als einzelne Ressourcen verstanden werden, wurde als Laufzeitumgebung aufgefaßt und eine Simulation als Bewertung eines Szenarios verstanden. Es wurde eine Bewertungsfunktion definiert, die für jedes Tripel aus dynamischem Modell, Szenario und Umgebung eindeutig und wiederholbar eine Ablaufzeit berechnet.

Im ersten Teil wurde auf eine hohe Abstraktionsebene wert gelegt und versucht jede Interpretation dessen, was eine Last oder eine Ressource ist zu vermeiden. Ebenso wurden keine Festlegungen über die Art der Quantifizierung von Lasten oder die Beschaffenheit und Realisierung einer Ressourcenfunktion gemacht, um eine größtmögliche Vielfalt an Anwendungsmöglichkeiten zu erreichen. Die ganze Technik wurde nicht aus einem einzigen Kalkül heraus entwickelt, sondern mittels einzelner, gut verstandener, adäquater Formalismen konstruiert. Ziel war es wichtige technische Eigenschaften zu erhalten, die eine möglichst unmittelbare Umsetzung in eine Werkzeugimplementierung erlauben. Der Erfolg bei der Implementierung der prototypischen Werkzeugstudie |SEMPER| belegen diesen Erfolg. Somit ist diese Arbeit auch ein erfolgreiches Praxisbeispiel für Softwareentwicklung unter Einbeziehung formaler Methoden.

Im zweiten Teil wurde die praktische Anwendung und die Integration dieser Technik in den Softwareentwicklungsprozeß diskutiert. Durch die Abstraktion der eingeführten Artefakttypen ergibt sich deren konkrete Bedeutung und Auswirkung erst an dieser Stelle. Dadurch wurde offenbar, daß diese Technik andere Einsatzgebiete, wie etwa die quantitative Untersuchung des Softwareentwicklungsprozeßes selbst, nicht ausschließt.

Weiterhin wurde eine Anwendung für die UML als weitverbreitete industrielle Spezifikationstechnik vorgeschlagen. Die hier vorgestellte

Modellierungstechnik ist auf einer Metaebene ausgeführt und damit nicht an den Einsatz in einem aktuellen Standard gebunden.

Als Hinweis für interessante Arbeiten in der Zukunft wurde das Themengebiet der maßbasierten Steuerung des Entwicklungsprozeßes performanz-kritischer Software thematisiert, wenn dies auch ein spannendes aber zu großes Themengebiet darstellt, um in dieser Arbeit erschöpfend behandelt zu werden.

Schlagworte und Begriffe

$E_{Spontan}$-Menge, 132
$E_{Szenario}$-Menge, 132
E_{Term}-Menge, 132
T-Invarianten, 111
Δ-Funktion, 107
$\cong$-Operator, 95
δ_{min}-Funktion, 123
η_{in}-Funktion, 109
$\rho_{\mathcal{PK}}$-Funktion, 62
$\rho_{\mathcal{SK}}$-Funktion, 63
$\circ$ - Operator, 77
$\shortmid$ - Operator, 77
$\mathcal{Z}[\![-]\!]$-Funktion, 139
$\mathcal{L} - Menge$, 120
$\bigcup$-Operator, 76

$\cup$-Operator, 76
ν_i-Funktion, 135
ν_{Set}-Funktion, 135
$\overline{i}$-Folge, 137
$\overrightarrow{t}$-Funktion, 141
$\triangleright$-Operator, 117
$\mathcal{T}[\![-]\!]$-Funktion, 78
$\mathcal{K}[\![-]\!]$-Funktion, 79
$\mathcal{LE}[\![-]\!]$-Funktion, 78
$\mathcal{M}[\![-]\!]$-Funktion, 80
$\mathcal{PK}[\![-]\!](,-)$Funktion, 78, 79
$\mathcal{SK}[\![-]\!](,-)$Funktion, 78, 79
τ-Funktion, 133
$e(-)$-Prädikat, 114
elem-Element, 77
elem()-Funktion, 77, 101
f_s-Funktion, 122
f_t-Funktion, 122
g - Funktion, 136
$k(-)$-Funktion, 137
l_{act}-Funktion, 130
l_{term}-Funktion, 130
$loc_{\mathcal{L}}$-Funktion, 121
$loc_{\mathcal{U}}$-Funktion, 121
q_{init}-Funktion, 124
$r(-)$-Funktion, 137
red_δ-Funktion, 106
red_i-Funktion, 107
$res_{\mathcal{U}}$-Funktion, 121
$s(-)$-Prädikat, 114
sem-Funktion, 139
$sem0_i$-Funktion, 137
$sem1_i$-Funktion, 138
semS-Funktion, 139
$semS_i$-Funktion, 138
st_{Res}-Funktion, 136
st_{in}-Funktion, 110
st_{seq}-Funktion, 141
state-Funktion, 136
t^+-Funktion, 75
t^--Funktion, 75
type-Element, 77
v-Funktion, 111

Abgrenzung, 49
Abhängige Anwendungsfälle, 216
Abläufe, 105–112
Ablaufsteuerung, 109
Aktive Lasten, 130
Aktivierte Kanten, 55
Aktivierte Knoten, 55
Aktivierte Transition (Konzession), 75
Aktivitätsdiagramm, 154, 156, 177, 180
Aktivitätsdiagramm-Compiler, 184
Analysemodell, 153
Anforderungsanalyse, 153, 160

Anwendungsfälle, 156, 215
Anwendungsfall, zeitkritischer, 215
Artefakte, 153
Ausdrucksäquivalenz, 95
Austrittspunkte, graphische Darstellung, 58–61
Auswertungsklassen, 205

Bearbeitugszeit, 122
Brückenfreiheit, 51

Datenmodell, Werkzeug-, 196
Deterministischer EAA, 97
Dynamisches Performanz-Modell, 159, 161

Eindeutige Übergangsfunktion, 46
Eindeutigkeit der Pfadwahl, 139
Eindeutigkeit des Kontrollflusses, 53
Einzelschrittfunktion, 122
Empfangen, 114
Endliche asynchrone Automaten, 95–98
Endliche asynchrone Automaten, Definition, 95
Endlichkeit des Erreichbarkeitsgraphen, 86
Endpartition, 55
Endpunkte, graphische Darstellung, 58–61
Entwurf, 153
Environment-Compiler, 192
EnvML, 196
Ermittlung der Parallelkartuschen, 62
Ermittlung der Sequenzkartuschen, 63
Erreichbarkeit des Endzustandes, 56
Erreichbarkeitsgraph, 86–89

Fehlschlagsrisiko, 214, 226, 228
Flußanteil, 51
Folgeknoten, ausgewählter, 135
Folgeknoten, mögliche (Erreichbarkeitsgraph), 135
Folgepartition, 56
Folgezustand, Gesamt-, 136
Folgezustand, Last-, 136
Folgezustand, Ressourcen-, 136

Gesamt-Schrittfunktion, 139
Gewöhnliches Netz, 75
Grade von Kanten, 49

Implementationsdiagramm, 154, 156, 181
Implementierung, 153, 157
Indizierte Gesamt-Schrittfunktion, 138
Initiale Lastgröße, 124
Interpolationsfunktion, 192
Interviewprotokolle, 156
Iterierte Lesefunktion, 141
Iterierte Lesefunktion, Lemma, 141

Kantengruppen, 132
Kantensemantik, formal, 79
Klassendiagramme, 156
Kleinster Zeitschritt, 133
Knotenermittlung, 111
Kollaborationsdiagramm, 181
Kommunikation, 112–117
Kommunikation, graphische Darstellung, 112
Kommunikation, Petrinetz-Repräsentation, 113
Kommunikationsdeterminierte Verzweigungen, 88
Konfiguration, 157
Kontextprozeß, 164–168
Kontrollflußsemantik, informell, 66–73
Konzession, 75
Kopf- und Fußknoten, formal, 49
Kopf- und Restindexfolge, 137

Lastelement (graphische Darstellung im Kartuschenmodell), 43–45
Lastlokalisation, 121
Lastmenge, gesamte, 120
Lastmodell-Entwicklung, 164
Lastschrittfunktion, 136
Lastsemantik, formal, 78
Laufzeitsemantik, 139
Lesen eines Szenarios, 110

Meßdatensatz, 196
Mikrozyklen, 153
Minimaler Zeitschritt, 123
Modellbaum, graphisch, 61–62

Modellsemantik, formal, 78–80
Modellstruktur, formal, 61
Modellstruktur, graphisch, 61–62

Netzsumme, 76
Netzvereinigung, 76
Neutral-Schrittfunktion, 137
Neutraler Schritt, Lemma, 137, 142

Parallelkartusche, Darstellung im Kartuschenmodell, 47–48
Parallelkartusche, formal, 54
Parallelkartuschensemantik, formal, 78, 79
Peformanzszenario, Definition, 109
Performanz Analyse, 159, 162
Performanz-Analyse, 185
Performanz-Analytiker, 160, 168
Performanz-Annotation, 180
Performanz-Modellierer, 161, 169
Performanzbewertung, frühzeitige-, 220
Performanzdeterminierte Verzweigungen, 86
Performanzszenario, 106–109
Petrinetz, 75
Petrinetze, Einführung, 74–78
Petrinetze, Notation, 74–78
Petrinetzsemantik, informell, 66–73
Pfad (im PK-Graphen), 50
Phasen, 153
PK-Graph, formal, 48
PNML, 184, 196
Projektklassifikation, 214
Prozeß, 153
Prozeßablauf, 160–168
Prozeßkostenschätzung, 226–229
Prozeßoptimierung, 222–223, 226–229

Regressionsfunktion, 190
Ressourcen, 121
Ressourcen-Designer, 160, 171, 192

Sackgassenfreiheit, 50
Schaltsequenz, 92
Schleifen, 111, 187–188
Schleifen (graphische Darstellung im Kartuschenmodell), 43
Schnitt, 55
Schrittfunktion, Ressourcen-, 136
Semantische Bereiche, 74–78
Semantische Umkehrfunktion, 77, 101
Senden, 114
Sequenzdiagramm, 181
Sequenzkartusche (graphische Darstellung im Kartuschenmodell), 44
Sequenzkartusche, formal, 45–47
Sequenzkartuschensemantik, formal, 78, 79
Sicherer EAA, 97
Simulationsingenieur, 164, 165, 171, 188
Simulationsteilmenge, 136
Simulierbares Szenario, Definition, 117
SK-Automat, 45
Software-Entwicklungsprozeß, 153
Spaltung des Kontrollflusses, 52
Spline, 192
Sprünge (graphische Darstellung im Kartuschenmodell), 43
Spur, 95
Spurreduktion, 106
Spurreduktion auf Teilautomaten, 107
Spursprache (eines Modells), 101
Spursprache eines EAA, 97
Spursprachen, 94
Spurzerlegung, 107
Stützstellen, 190
Startpartition, 55
Startpunkte, graphische Darstellung, 58–61
Statische Struktur eines dynamischen Modells, formal, 61
Syntaktische Bereiche, 73–74
System-Designer, 160
Szenario, 159, 162, 186
Szenario, vollständiges, 109
Szenario-Entwickler, 162, 170
Szenarioeditor, 185
SzML, 196

Teilautomat, 102
Terminierende Lasten, 130

Test, 153
Testfall, 181
Testprotokoll, 157
Transitionsfolge-Funktion, 141
Transitionssequenz, 94

Uebergangsfunktionssemantik,
    formal, 78
Umgebung, 121
Umgebungsmodell, 159, 161
Umgebungsmodell, schematisch, 120
Unabhängigkeitsrelation, 94
Unmittelbare Folge (Szenario), 117

Vereinigung des Kontrollflusses, 53
Verklemmung, 115–117
Verklemmungsfreiheit, 186
Vollständiger Lauf, 46
Vollständiger Pfad, 50
Vollzusammenhang, 50

Werkzeug-Architektur, 183
Wiedereintrittspunkte, graphische
    Darstellung, 58–61
Wohlgeformte Indexfolge, 137
Wortproblem, Entscheidbares, 98
Wortsprache eines EAA, 96

zb-Funktion, 145
Zeiltliche Bestimmtheit, 122
Zeit-Schrittfunktion, 138
Zeitkorrelation, 122
Zeitschrittfunktion, 122
Zustandsdiagramm, 156
Zyklenfreiheit, 50

Abbildungsverzeichnis

1.2.1 Bedienstation 23
1.2.2 Geschlossenes Netz 24
1.2.3 Zustandssystem der Warteschlange 24
1.2.4 Bedienstation 28
1.3.1 Entwurfsabhängige Performanzeigenschaften 30
1.3.2 Modellsimulation 31
1.3.3 Schematischer Simulationsablauf 34

2.1.1 Kontrollfluß ohne Nebenläufigkeit 37
2.1.2 Nebenläufigkeit 38
2.1.3 Asynchrone Kommunikation und Event 38
2.1.4 Datenfluß 39
2.1.5 Teilweise Verschmelzung nebenläufiger Kontrollflüsse 40
2.1.6 Problematische Schleifen 40
2.1.7 Aktivitätsgraph-Beispiel aus der UML 2.0 Spezifikation 41
2.1.8 Alternative Join-Knoten 41
2.2.1 Sequentielle Ausführung 43
2.2.2 Bedingte und wiederholte Ausführung 44
2.2.3 Last 44
2.2.4 Schema einer Parallelkartusche 47
2.2.5 Kartusche mit nebenläufiger Region 48
2.2.6 Beispiel mit Synchronisation 53
2.2.7 Beispiel zweier nebenläufiger Kartuschen 54
2.2.8 Kein wohlgeformter PK-Graph 54
2.2.9 Einzelpartition einer Folge 58
2.2.10 Startpunkte 58
2.2.11 Wiedereintrittspunkte 59
2.2.12 Endpunkte 60

2.2.13 Vorübergehende Austrittspunkte 60
2.2.14 Modell-Baum 61
2.2.15 Zwei Arten von Teilbäumen 62
2.2.16 Beispiel-Kartusche mit paralleler Region 63
2.2.17 Beispiel einer Sequenzkartusche 65
2.3.1 Beispiel einer Sequenzkartusche 67
2.3.2 Beispielübersetzung einer Sequenzkartusche (1.Schritt) 68
2.3.3 Stellen-Substitutio 68
2.3.4 Lastzustandssemantik (verfeinert) 69
2.3.5 Lastzustandssemantik (vereinfacht) 69
2.3.6 Transitions-Substitution 70
2.3.7 Petri-Netz des Grundmodells 70
2.3.8 Kartusche mit paralleler Region 71
2.3.9 Übersetzung der S_0-Kartusche (1.Schritt) 72
2.3.10 Übersetzung der Parallelkartusche (2.Schritt) 72
2.3.11 Übersetzung der eingeb. Sequenzkartuschen (3. Schritt) 72
2.3.12 Reduktion unbeschr. Stellen und spontaner Transitionen 73
2.3.13 Unterscheidung laufender und bearbeiteter Lasten 73
2.3.14 Vollständige Übersetzung der Sequenzkartusche 85
2.4.1 Reduktion unbeschr. Stellen und spontaner Transitionen 86
2.4.2 Erreichbarkeitsgraph des Beispielmodells 87
2.4.3 Unterscheidung laufender und bearbeiteter Lasten 87
2.4.4 Erreichbarkeitsgraph des Beispielmodells 88
2.4.5 Beispielübersetzung der Sequenzkartusche (1.Schritt) 89
2.4.6 Erreichbarkeitsgraph des Beispielmodells 89

3.1.1 Beispiel 1 (Sequenzkartusche) 92
3.1.2 Erreichbarkeitsgraph des Beispielmodells 93
3.1.3 Beispiel 2 (Nebenläufigkeit) 93
3.1.4 Erreichbarkeitsgraph bei Nebenläufigkeit 94
3.2.1 Kartusche mit paralleler Region 99
3.2.2 Petrinetz des Modells 99
3.2.3 Asynchroner Automat des Beispiels 100
3.2.4 Komplexeres Beispiel 102
3.2.5 Erreichbarkeitsgraph zum komplexen Beispiel 103

3.2.6 Erreichbarkeitsgraph Fs. 104
3.2.7 Reduktions-Beispiel 108
3.3.1 Parallelkartuschen mit Kommunikation 112
3.3.2 Kommunikation in Petrinetz 113
3.3.3 Parallelkartuschen mit Kommunikation 115
3.3.4 Verklemmung, Beispiel 1 115
3.3.5 Verklemmung, Beispiel 2 116
3.3.6 Synchronisation in der Spurtheorie 116

4.1.1 Umgebung und Lasten 120
4.2.1 UML Beispiel 123
4.2.2 Erreichbarkeitsgraph des Beispiels 124

5.2.1 Lastmengenstruktur-Beispiel 131
5.2.2 Erreichbarkeitsgraph Kantengruppen 131
5.2.3 Erreichbarkeitsgraph in der Simulation 132
5.2.4 Simulation einfacher Nebenläufigkeit 134
5.2.5 Erreichbarkeitsgraph des Beispiels 134

6.0.1 Artefakte nach Prozeßphasen 153
6.0.2 Entwurfsabhängige Performanz-Eigenschaften 154
6.0.3 Inspektion mit Simulationstechniken 154
6.0.4 Modellsimulation (schematisch) 155
6.1.1 Artefakte des Simulators 157
6.1.2 Artefakteabhängigkeiten 158
6.2.1 Anforderungsanalyse 160
6.2.2 Umgebungsanlyse 161
6.2.3 Performanz Modellierung 162
6.2.4 Szenario-Entwicklung 163
6.2.5 Performanzanalyse 163
6.2.6 Lastmodellentwicklung 165
6.2.7 Ressourcenmodell Entwicklung 166
6.2.8 Iteration der Lastmodell Entwicklung 167
6.3.1 Der Performanz-Analytiker 169
6.3.2 Der Performanz-Modellierer 170
6.3.3 Der Szenario-Entwickler 170

6.3.4 Der Ressourcen-Entwickler 171
6.3.5 Der Simulationsingenieur 172

7.1.1 Beispiel mit Kommunikation 178
7.1.2 UML-Ursprungsmodell 178
7.1.3 Annotiertes Aktivitätsdiagramm 181
7.1.4 Annotiertes Implementationsdiagramm 182
7.2.1 Architektur mit Einzelkomponenten 183
7.2.2 Übersetzungsfunktionen, schematisch 184
7.2.3 Zustandssystem für den Compiler 185
7.2.4 Funktionsschema des Szenario-Editors 187
7.2.5 Schleifenerkennung 187
7.2.6 Manuelle Auswahl 188
7.2.7 Wiederholte Schleifenerkennung 189
7.2.8 Regressionsfunktion aus Stützstellen 190
7.2.9 Funktionsschema der Ressourcen-Workbench 191
7.2.10 Editor für Meßwerttabellen 191
7.2.11 Splinefunktion im Editor 192
7.2.12 Modell einer Simulationsumgebung 193
7.2.13 Funktionsschema des Umgebungscompilers 194
7.2.14 Funktionsschema des Simulators 194
7.2.15 Zusammenstellung einer Simulation 195
7.2.16 Chart-Erstellung im Analyse-Werkzeug 195
7.2.17 Architektur mit Artefakttypen 196
7.2.18 Rollenbasierter Nutzerzugang 197
7.2.19 Rollenbasierter Funktionszugang 197
7.3.1 Beispielumgebung 199
7.3.2 Erzeugung eines Rohdatensatzes 199
7.3.3 Erstellen der Meßwerttabelle 200
7.3.4 Erzeugung der Interpolationsfunktion 201
7.3.5 Spezifikation der virtuellen Umgebung 202
7.3.6 Spezifikation des dynamischen Modells 203
7.3.7 Zusammenstellung der Simulationen 204
7.3.8 Simulationsauswertung 204

8.2.1 Anwendungsfälle im Beispie 215
8.2.2 Nebenläufiger Entwurf mehrerer Pakete 216
8.2.3 Kritische vs. unkritische Subsysteme 217
8.2.4 Software Entwicklung 218
8.2.5 Hardware Administration 219
8.3.1 Prozeßsteuerungs-Artefakte innerhalb einer Iteration 221
8.3.2 Arbeitsaufwendungen im Prozeßverlauf 224
8.3.3 Modellierung und Bewertung untersch. Entwürfe 226
8.3.4 Zuordnungsalgorithmus zu Workflow-Bereichen 227
8.3.5 Zuordnung unkritischer aber abhängiger Pakete 230

Tabellenverzeichnis

1	Übersetzungstabelle der Sequenzkartuschen	67
2	Übersetzungstabelle der Parallelkartuschen	70
1	Meßwerttabelle einer Performanzmeßung	127
1	Übersetzungstabelle 1.Teil	179
2	Übersetzungstabelle 2. Teil	179
3	Ordnungsklassen der Simulationssätze	206
4	Klassifikation der Simulationsgruppen	206
1	Basistypen verschiedener Softwareprojekte	214
2	Basistypen bei der Hardwareentwicklung	215

3 Quantitativer Vergleich vergeblich aufgebrachter Arbeitsaufwendungen über mehrere

Literaturverzeichnis

[1] I. Aalbersberg, H. Hoogeboom: Decision Problems for Regular Trace Languages. LNCS 267, S. 250-259, Th. Ottmann (Hrg.), Springer, 1987

[2] I. Aalbersberg, G. Rozenberg: A Theory of Traces. TCS 60, S.1-82, 1988

[3] M. Aigner: Diskrete Mathematik. vieweg, 2006

[4] S. Balsamo, M. Simeoni: Software Performance Modeling: State of the Art and Perspectives. Research Report CS-2003-1, Dip. di Informatica - Univ. di Venezia, 2003

[5] S. Balsamo, M. Simeoni: Performance Evaluation at the Software Architecture Level. LNCS 2804, S. 207–258, M. Bernardo and P. Inverardi (Hrg.), Springer, 2003

[6] S. Balsamo, M. Simeoni: Deriving Performance Models from Software Architecture Specifications. Praga Proceedings of the ESM 2001, http://www.dsi.unive.it/ balsamo/saladin/bal-sim.2.01.pdf (download Nov. 2007)

[7] S. Balsamo, M. Simeoni: On Transforming UML Models into Performance Models. Proc. WTUML, Workshop on Transformations in UML, ETAPS 2001, http://ase.arc.nasa.gov/wtuml01/ (download Jan. 2003)

[8] S. Balsamo: A Simulation-based Approach to Software Performance Modeling. Proc. of ESEC 2003, S. 363 - 366, ACM Press, 2003

[9] S. Balsamo, M. Grosso, M. Marzolla: Towards Simulation Based Performance Modelling of UML specifications. Technical Report CS-2003-2, Universita di Venezia, 2003

[10] S. Balsamo, M. Simeoni: On Transforming UML Models into Performance Models. Technical report R-SAL-51 WTUML, Universita di Venezia, 2001

[11] S. Balsamo, M. Marzolla: Performance Evaluation of UML Software Architectures With Multiclass Queueing Network Models. WOSP-Proc., S. 37-42, ACM, 2005

[12] H. Bareth: Entwicklung eines Simulationskerns für SEMPER. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[13] M. Barth: Performance Assessment of Software Models In a Configurable Environment Simulator. Proc. of SERP'03, H. Arabnia (Hrg.), CSREA Press, 2003

[14] M. Barth: Integration of Simulation Based Performance Assessment In a Software Development Process. Proc. of SNPD 03, S.98-105, W. Dosch, R. Lee (Hrg.), ACIS, 2003

[15] M. Barth: Development of Performance-Critical Software Using UML Activity Models and Simulation Techniques. IJCIS, Volume 5, No. 3, S.200-211, ACIS, 2004

[16] M. Barth: A Formal Model for Performance Assessment in A Simulative Environment. Proc. of the SEEFM05, S. 196-208, G. Eleftherakis (Hrg.), SEERC, 2005

[17] B. Baumgartner: Petri-Netze. Spektrum - Akademischer Verlag, 2.Auflage, 1996

[18] S. Becker, H. Koziolek, R. Reussner: Modelbased Performance Prediction with the Palladio Component Model. WOSP'07, S.54-65, ACM, 2007

[19] E. Behrends: Introduction to Markov Chains. Vieweg, 2000

[20] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, M. Weber: The Petri Net Markup Language: Concepts, Technology, and Tools. März, 2003, http://www2.informatik.huberlin.de/top/pnml/, (download April 2007)

[21] P. Brémaud: Markov Chains. Springer, 1999

[22] K. Chandy, J. Misra, L. Haas: Distributed Deadlock Detection. Transactions on Computer Systems Vol.1, No2, S.144-156, ACM, 1983

[23] A. Clark and S. Gilmore: Evaluating Quality of Service For Service Level Agreements. FMICS 2006, http://www.dcs.ed.ac.uk/pepa/, (download März 2007)

[24] E. Coffman, M. Elphick, A. Shoshani: System Deadlocks. Computing Surveys, Vol.3, No.2, S. 67-78, ACM, 1971)

[25] V. Cortellessa, R. Mirandola: PRIMA-UML: A Performance Validation Incremental Methodology on Early UML Diagrams. Science of Computer

Programming, Vol. 44, Nr. 1, S.101-129, Elsevier, 2002

[26] V. Diekert: Compinatorics on Traces. LNCS 454, Springer, 1990

[27] V. Diekert: A Partial Trace Semantics for Petri Nets. TCS 134, S.87-105, Elsevier, 1994

[28] K. Dymond: CMM Handbuch - Das Capability Maturity Model für Software. Springer, 2002

[29] C. Empl: Spezifikation von Testfällen eines dynamischen Modells der Performanz-Analyse. Diplomarbeit, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, März 2007

[30] C. Empl: Entwicklung eines Szenario-Editors für Petrinetz-Modelle. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[31] W. Fokkink: Introduction to Process Algebra. EATCS, Springer, 2000

[32] S. Gaubert: Performance Evaluation of Timed Automata, INRIA RR-1922, 1993

[33] S. Gilmore, J. Hillston, L. Kloul, M. Ribaudo: Software Performance Modelling Using PEPA Nets. SIGSOFT, Vol.29, Nr.1, S.13-23, ACM, 2004

[34] S. Gilmore, J. Hillston: The PEPAWorkbench: A Tool to Support a Process Algebra-based Approach to Performance Modelling. LNCS 794, S. 353-368, G.Haring, G. Kotsis (Hrg.), Springer, 1994

[35] J. Hillston: Process Algebras for Quantitative Analysis. Proceedings of the 20th Annual Symposium on Logic in Computer Science (LICS05) 0-7695-2266-1/05 IEEE

[36] J. Hillston: Tuning Systems: From Composition to Performance. The Computer Journal, Vol.48 No.4, S.385-400, Oxford Univ. Press, 2005

[37] J. Hillston: A Compositional Approach to Performance Modelling. Cambridge University Press, 1996

[38] J. Hopcroft, j. Ullman: Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie. Addison Wesley, 2003

[39] H. Hoogeboom, G. Rozenberg: Dependence Graphs. Book of Traces S. 43-67, (Hrg.) V. Diekert, G. Rozenberg, World Scientific, Singapore, 1995

[40] I. Jacobson, G. Booch, J. Rumbaugh: The Unified Software Development Process. Addison Wesley, 1998

[41] H. Koziolek, J. Happe, S. Becker, R.Reussner: Evaluating Performance of Software Architecture Models with the Palladio Component Model. Technischer Bericht, http://sdqweb.ipd.uni-karlsruhe.de/wiki/Palladio_Component_Model, (download Nov. 2007)

[42] J. P. Lopez-Grao, J. Mersenguer, J. Campos: From UML Activity Diagrams To Stochastic Timed Petri Nets. SIGSOFT, Vol.29 Nr.1, S.25-36, ACM, 2004

[43] A. Mazurkiewicz: Introduction to Trace Theorie. World Scientific Publishing, 1995

[44] A. Mazurkiewicz: Trace Theorie, Advances in Petri Nets. LNCS 255, S.279-321, Springer, 1987

[45] A. Mazurkiewicz: Concurrent Program Schemes and Their Interpretations. Technischer Bericht, DAIMI-PB-78, Universität Aarhus 1977

[46] D. Mendez-Fernandez: Entwicklung eines Ressourcen-Modell-Generators. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[47] J. Mersenquer, J. Campos: Software Performance Modeling Using UML and Petri Nets. LNCS 2965, S.265-289, M. Calzarossa, E. Gelenbe (Hrg.), Springer, 2004

[48] M. Nerlich: Entwicklung eines Petrinetz-Compilers für UML Aktivitätsdiagramme. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[49] M. Nerlich: Entwicklung eines Editors für dynamische Performanz-Modelle. Diplomarbeit, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[50] Object Management Group: XML Metadata Interchange (XMI) version 1.1. formal/00-11-02, http:www.omg.org

[51] C. Petri: Kommunikation mit Automaten. Institut für Instrumentelle Mathematik, Bonn, IIM Nr.2, Dissertation, 1962

[52] Petri Net Kernel. http://www.informatik.hu-berlin.de/top/pnk, download Okt. 2006

[53] H. Reineke: Struktur und Verhalten von verteilten endlichen Automaten. Dissertation im Fachbereich Informatik, Carl.von-Ossietzky-Universität Oldenburg, (1995)

[54] F. Sauerwein: Mathematische Grundlagen der Warteschlangentheorie / Markov-Ketten. GRIN Verlag, 2007

[55] K. Schreiber: Entwicklung eines Workflow-Management-Tools zur Benutzerfhrung und Dokumenten-Verwaltung eines modular aufgebauten Werkzeuges. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[56] F. Sirtl: Entwicklung eines Environment-Compilers für UML-Deployment-Diagramme. Fortgeschrittenenpraktikum, Lehrstuhl für Programmierung und Softwaretechnik, Ludwig-Maximilians-Universität, München, Juni 2006

[57] P. H. Starke: Analyse von Petri-Netz-Modellen. Teubner, 1990

[58] E. Sultanow: Graphentheoretische Beschreibung von Petrinetzen. http://www.glenesoft.com/fhdw/Systemanalyse/Petrinetze.pdf, Universität Potsdam, 2005.

[59] P. Tittmann: Graphentheorie. fv Hanser, 2003

[60] UML Profile for Schedulability, Performance and Time. Object Management Group, Sept. 2003

[61] Unified Modelling Language-Version 2.0. Object Management Group, Final Adopted Specification, ptc/03-08-02, 2003

[62] Unified Modelling Language-Version 1.4. Object Management Group, IX 2001 (Unified Modelling Language, v1.4)

[63] M. Weber, E. Kindler: The Petri Net Markup Language. April, 2002, http://www2.informatik.hu-berlin.de/top/pnml/, (download April 2007)

[64] M. Weber, E. Kindler: The Petri Net Kernel: Documentation of the Application Interface. Humboldt Universität zu Berlin, Revision 1.1, 1998

[65] W. Zielonka: Notes on Finite Asynchronous Automata. Theoretical Informatics and Applicatios, Vol.21, Nr.2, S.99-135, EDP Sciences, 1987

[66] W. Zielonka: Safe Executions of Recognizable Trace Languages by Asynchronous Automata. LNCS 363, S.278-289, A. Meyer, M. Taitslin (Hrg.), Springer, 1989

Lebenslauf	**Michael Barth**
geb. 1966	in München
1972 bis 1976	Besuch der Grundschule in München
1976 bis 1985	Besuch der mathematisch-naturwissenschaftlichen Gymnasien in Augsburg und München
1985	Abitur
ab 1993	Studium der Informatik an der Ludwig-Maximilians-Universität, München
2000	Informatik Diplom
2008	Promotion in Informatik