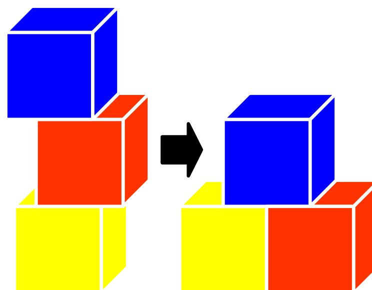
Verhaltensinvariante Transformation von Entwurfsmodellen Reaktiver Systeme

Eine Adaption der Refactoring-Technik auf gezeitete Modelle unter Verwendung eines formalen Verhaltensäquivalenzbegriffs

Alexander Karl Wißpeintner



Lehrstuhl für Software & Systems Engineering
Fakultät für Informatik
Technische Universität München

Verhaltensinvariante Transformation von Entwurfsmodellen Reaktiver Systeme

Eine Adaption der Refactoring-Technik auf gezeitete Modelle unter Verwendung eines formalen Verhaltensäquivalenzbegriffs

Alexander Karl Wißpeintner

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Bernd Radig
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Dr. h.c. Manfred Broy
	2.	Univ.-Prof. Bernd Brügge, Ph.D.

Die Dissertation wurde am 13. Februar 2006 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 10. Juli 2006 angenommen.

Für Monika und Peter.

Danksagung

Das Gelingen einer wissenschaftlichen Arbeit ist nicht nur von den individuellen Fähigkeiten seines Verfassers abhängig, sondern bedarf auch eines optimalen Nährbodens und der Unterstützung derer, die den Schaffensprozess auf die eine oder andere Weise begleiten. Herrn Professor Manfred Broy steht daher ganz besonderer Dank zu, nicht zuletzt, weil er es mir ermöglicht hat, diese Arbeit anzufertigen. Ich möchte mich bei ihm insbesondere für die Geduld, die er aufgebracht und die Zeit, die er investiert hat und für die vielen guten Kommentare zu meiner Arbeit bedanken. Ebenso möchte ich Herrn Professor Bernd Brügge danken, der sich nicht nur als Zweitgutachter der Arbeit zur Verfügung gestellt, sondern mir auch viele Anregungen mit auf den Weg gegeben hat.

Wissenschaftliches Arbeiten erfolgt nicht nur im Elfenbeinturm und daher bin ich folgenden Personen für regen Austausch und viele Diskussionen sehr dankbar: Dr. Peter Braun, Alexander Gruler, Tobias Hain, Franz Huber, Dr. Frank Marschall, Jan Philipps, Dr. Wolfgang Prenninger, Dr. Alexander Pretschner, Dr. Martin Rappl, Jan Romberg, Professor Bernhard Rumpe, Dr. Bernhard Schätz, Maurice Schoenmakers, Dr. Oscar Slotosch und Stefan Wagner.

Des Weiteren möchte ich mich bei Herrn Jürgen Jeitner, der meine Arbeit kontinuierlich begleitet hat, für die vielen fachlichen und nicht fachlichen Gespräche bedanken. Frau Silke Müller danke ich für die tolle Organisation an unserem Lehrstuhl.

Meinem Vater Karl Wißpeintner danke ich für die viele Unterstützung. Zu guter Letzt danke ich meiner Frau Monika dafür, dass sie einfach toll ist.

Zusammenfassung[1]

Refactoring ist eine Technik zur verhaltensinvarianten Restrukturierung von Programmcode mit dem Ziel der Verbesserung dessen Qualität. Diese Arbeit überträgt am Beispiel der Modellierungssprache AutoFocus die Refactoring-Technik auf die Ebene von Entwurfsmodellen Reaktiver Systeme unter Verwendung eines formalen Verhaltensäquivalenzbegriffs.

Für gezeitete Modelle wird ein Äquivalenzbegriff für Schnittstellenverhalten unter Zeitabstraktion festgelegt. Die Eignung von zeitsynchronen und zeitasynchronen Semantiken für das Refactoring wird betrachtet. Die Eigenschaft der Zeitrobustheit zeitsynchroner Modelle, die die Möglichkeiten der Durchführung von Refactorings begünstigt, wird identifiziert und auf Basis dieser Eigenschaft wird eine Semantik konstruiert, die weitgehendes Modell-Refactoring unterstützt.

Für AutoFocus Systemstrukturdiagramme und Zustandsübergangsdiagramme werden Refactorings definiert, deren Verhaltensinvarianz nachgewiesen wird. Diese Refactorings ermöglichen weitgehende Architekturveränderungen bestehender Softwaresysteme. Darüber hinaus wird die Modell-Refactoring-Technik in einen modellbasierten iterativen Entwicklungsprozess integriert.

[1] Dieses Dokument basiert auf Arbeiten, die zum Teil durch das U.S. Army Research Laboratory und das U.S. Army Research Office unter der Fördernummer DAAD19-03-1-0197 unterstützt wurden.

Summary[2]

Behavior Preserving Transformation of Design Models of Reactive Systems—An Adaption of the Refactoring Technique on Timed Models by using a Formal Notion of Behavior Equivalence.

Refactoring is a technique for restructuring program code to improve its quality. This work adapts refactoring to the layer of design models of reactive systems by using a formal notion of behavior equivalence. Therefore the modeling language AutoFocus is exemplary used.

We define a notion of equivalence of interface behavior under time abstraction. The suitability of time synchronous and time asynchronous semantics for applying refactoring is examined. We define the property of time robustness of time synchronous models, that eases the realization of refactoring. Based on this property we construct a semantics that extensively supports model refactoring.

A catalog of refactorings of AutoFocus System Structure Diagrams and State Transition Diagrams are defined including an evidence of behavior preservation. These refactorings enable vast architectural changes of existing software systems. Finally we integrate the model refactoring technique into a model based iterative development process.

[2]This material is based upon work supported in part by the U.S. Army Research Laboratory and the U.S. Army Research Office under grant number DAAD19-03-1-0197.

Inhaltsverzeichnis

Danksagung 7

Zusammenfassung 9

Summary 11

1. Einleitung 17
1.1. Reaktive Systeme und modellbasierte Softwareentwicklung 18
1.2. Refactoring von Programmcode 20
1.3. Nutzen von formalem Modell-Refactoring 21
1.4. Ableitung der Modell-Refactorings 24
1.5. Forschungsbeitrag 25
1.6. Aufbau der Arbeit 28

2. Grundlagen 31
2.1. Stand der Forschung im Themengebiet Refactoring 31
2.2. AutoFocus 34
2.3. Focus 36

3. Die AutoFocus Semantik, Schnittstellenverhalten und Abstraktionen 39
3.1. Übersetzung von Systemstrukturdiagrammen in Focus 40
3.2. Abbildung von AutoFocus Zustandsmaschinen (STDs) in Focus Relationen 42
3.3. Semantik von AutoFocus Spezifikationen 52
3.4. Das AutoFocus Schnittstellenverhalten in Focus Strömen 56
3.5. Äquivalenz von Schnittstellenverhalten 59
3.6. Komposition konkreten Schnittstellenverhaltens 62
3.7. AutoFocus Zeitabstraktionen 62
3.7.1. Sequenzdiagramme zur Darstellung von zeitabstraken Abläufen 63
3.7.2. Abstraktion von nicht verarbeiteten Eingaben 69
3.7.3. Zeitabstraktion von leeren Ausführungsschritten 70
3.7.4. Zeitabstraktion von aufeinander folgenden Eingaben beziehungsweise Ausgaben 72
3.7.5. Die Focus Zeitabstraktion angewendet auf AutoFocus Schnittstellenabläufe 76
3.7.6. Zeitkonkretisierung von Schnittstellenverhalten 78
3.7.7. Verhaltensäquivalenz unter Zeitabstraktion 79
3.7.8. Komposition abstrakten Schnittstellenverhaltens 79

4. Formale Definition von Refactoring 83
4.1. Beobachtbares Verhalten und Modelltransformation 83
4.2. Allgemeine Definition von Refactoring 84
4.3. Refactoring von AutoFocus Modellspezifikationen 87

5. Strukturelles Refactoring ohne Änderung des zeitlichen Verhaltens 91
5.1. Refactoring von AutoFocus Systemstrukturdiagrammen 92
5.1.1. Push Down Component Refactoring 93
5.1.2. Pull Up Component Refactoring 98
5.1.3. Wrap Components Refactoring 102
5.1.4. Move Component Refactoring 104
5.1.5. Remove Single Component Hierarchy Refactoring 105
5.1.6. Verhaltensinvarianz der SSD Struktur-Refactorings 106
5.2. Refactoring von hierarchischen AutoFocus Zustandsmaschinen 108
5.2.1. Push Down State Refactoring 110
5.2.2. Wrap States Refactoring 113
5.2.3. Verhaltensinvarianz der STD Struktur-Refactorings 115

6. Zeitsynchronität versus Zeitasynchronität 119
6.1. Zeitsynchrones AutoFocus und zeitliche lokale Verhaltensänderungen . 120
6.2. Modelleigenschaften zur Vereinfachung von Refactoring 123
6.2.1. Kompositionalität der Abstraktion 124
6.2.2. Zeitrobustheit 126
6.2.3. Refactoring von zeitrobusten AutoFocus Modellen 129
6.3. Anforderungen an die neue Semantik der AutoFocus Modellierungssprache 130
6.4. Semantikansätze und deren Auswirkungen auf das Refactoring 131
6.5. Wahl von geeigneten Zeitabstraktionen für das AutoFocus Refactoring . 133
6.6. Eine zeitasynchrone AutoFocus Semantik 135
6.6.1. Notationserweiterung 135
6.6.2. Zeitliche Entkopplung 137
6.6.3. Pufferkomponenten 139
6.6.4. Das Verhalten der Pufferkomponente 140
6.6.5. Verarbeitung von Nachrichten 141
6.6.6. Zwei zu eins Kommunikationsbeziehungen 142
6.6.7. Sequentielles Senden von Nachrichten 143
6.7. Erfüllung der Zeitrobustheitseigenschaft 144
6.7.1. Wirkung von Stuttering Steps und Verzögerung der Kommunikation 146
6.7.2. Lokale Wirkung zeitlicher Veränderungen 148
6.7.3. Auswirkungen lokaler zeitlicher Verhaltensänderungen auf komponiertes Verhalten 150
6.7.4. Praktische Anwendbarkeit der zeitasynchronen Semantik 152

7. Refactoring mit Veränderung des zeitlichen lokalen Verhaltens 155
7.1. Remove Intermediate State Refactoring . . . 156
7.2. Insert Intermediate State Refactoring . . . 159
7.3. Split Component Refactoring . . . 160
7.4. Remove Inactive States Refactoring . . . 174
7.5. Nachweis der Verhaltensinvarianzeigenschaft unter Zeitabstraktion . . . 177
7.5.1. Verhaltensinvarianz des Remove Intermediate State Refactorings 178
7.5.2. Verhaltensinvarianz des Insert Intermediate State Refactorings . 179
7.5.3. Verhaltensinvarianz des Split Component Refactorings . . . 180

8. Weitere Modelloptimierungen 189
8.1. Entfernen unbenutzter Ports, Kanäle und Variablen. . . . 189
8.2. Entfernen nicht erreichbarer Zustände und nicht ausführbarer Transitionen. . . . 190
8.3. Minimalisierung von Zustandsmaschinen. . . . 190
8.4. Berechnung eines Produktautomaten . . . 191

9. Entwicklungsprozess und Werkzeuge 193
9.1. Modell-Refactoring im Entwicklungsprozess . . . 193
9.1.1. Iterativer modellbasierter Entwicklungsprozess . . . 193
9.1.2. Prozess der Anwendung eines Modell-Refactorings . . . 196
9.1.3. Modell-Refactoring in einem inkrementellen Entwicklungsprozess 198
9.2. Metriken zur Bewertung der Qualität von Entwurfsmodellen . . . 199
9.3. Nachweis der Verhaltensinvarianzeigenschaft . . . 200
9.4. Werkzeugunterstützung . . . 201

10. Schlussfolgerungen 205
10.1. Bedeutung des Modell-Refactorings für die Entwicklung Reaktiver Systeme . . . 205
10.2. Bezug zu anderen Modellierungssprachen für Reaktive Systeme . . . 207
10.3. Zukünftige Arbeiten im Umfeld von Modell-Refactoring . . . 208
10.4. Vision der Softwareentwicklung in der Zukunft . . . 211

A. Bezeichner und Symbole 213

B. Definition von Refactorings auf Basis der AutoFocus Modell API 219
B.1. Push Down Component Refactoring . . . 219
B.2. Wrap Components Refactoring . . . 221
B.3. Move Components Refactoring . . . 222
B.4. Remove Single Component Hierarchy Refactoring . . . 223

C. Puffer in der zeitasynchronen AutoFocus Semantik 225
C.1. Datentyp- und Funktionsdefinitionen der Pufferkomponente . . . 225
C.2. Transitionen des Pufferkomponente . . . 225
C.3. Transitionen der Zustandsmaschine der Multiplexing Komponente . . . 227

D. Kombinierte Betrachtung von eins zu zwei Kommunikation **229**

Literaturverzeichnis **233**

1. Einleitung

1999 ging ein tragischer Unfall durch die deutsche Presse, bei dem ein Kind durch einen fehlgezündeten Airbag getötet wurde [Wüs99]. Obwohl der frontale Beifahrerairbag sachgemäß deaktiviert war, wurde er bei einem leichten Aufprall ausgelöst und dadurch das Kind in dem gegen die Fahrtrichtung montierten Kindersitz nach hinten katapultiert. Die Ermittlungen haben ergeben, dass ein Softwarefehler innerhalb des Airbag Steuergerätes für die Fehlfunktion verantwortlich war. Vermutlich wurde die Funktion des verwendeten Steuergerätes von der Steuerung zweier Frontairbags um die Steuerung von Seitenairbags erweitert. Auf Grund von sicherheitstechnischen Anforderungen erfolgt die Zündung der Front- und Seitenairbags nicht zeitgleich. Nachdem korrekter Weise zunächst beim Auslösen des Frontairbags auf der Fahrerseite auf die Zündung des Beifahrerairbags verzichtet wurde, wurde zusammen mit der Zündung der in dem betreffenden Auto gar nicht vorhandenen Seitenairbags nochmals das Steuerkommando zur Zündung der Frontairbags gegeben. Bei der zweiten Zündung wurde die Deaktivierung des Beifahrerairbags ignoriert.

Der Fall zeigt, dass die Erweiterung bestehender Software sehr risikoreich ist. Bereits kleine Funktionserweiterungen können unbeabsichtigte Wechselwirkungen hervorrufen, die sich in Softwarefehlern manifestieren. Bei sicherheitskritischen Systemen können Softwarefehler zu fatalen Folgen führen. In dieser Anwendungsdomäne sind die Softwareentwickler daher dazu verpflichtet, die aktuellen technischen und methodischen Möglichkeiten auszureizen, um Softwarefehler bei der Erweiterung zu vermeiden, zu erkennen und zu beseitigen.

Diese Arbeit adaptiert die aus der objektorientierten Programmierung bekannte Technik des Refactorings [FBB+99], die eine verhaltensinvariante Restrukturierung bestehender Software mit dem Ziel der Qualitätssteigerung ermöglicht, auf Entwurfsmodelle Reaktiver Systeme. Das Modell-Refactoring kann vor der Erweiterung angewendet werden, um die Qualität des Entwurfs in Hinblick auf Lesbarkeit, Verständlichkeit und Wartbarkeit zu verbessern und hierdurch das Risiko von Spezifikationsfehlern bei der anschließenden Erweiterung zu minimieren. Durch eine formale Fundierung der Modell-Refactoring-Technik kann die Verhaltensinvarianzeigenschaft der Refactorings garantiert und somit Fehler beim Refactoring ausgeschlossen werden.

In Abschnitt 1.1 wird zunächst das in dieser Arbeit betrachtete Anwendungsgebiet der Reaktiven Systeme definiert. In dieser Anwendungsdomäne wird verstärkt die Methodik der modellbasierten Entwicklung eingesetzt. Eine kurze Einführung in das Refactoring von Programmcode ist in Abschnitt 1.2 angegeben. In Abschnitt 1.3 wird schließlich der Einsatz der in dieser Arbeit vorgestellten Modell-Refactoring-Technik moti-

viert. Der Abschnitt 1.4 beschreibt die Art und Weise, wie die Refactorings dieser Arbeit entstanden sind. Der Forschungsbeitrag dieser Arbeit ist in Abschnitt 1.5 aufgeführt. Schließlich wird in dem Abschnitt 1.6 der Aufbau der Arbeit angegeben.

1.1. Reaktive Systeme und modellbasierte Softwareentwicklung

Ein Modell ist ein vereinfachtes Abbild der Wirklichkeit. In der Softwareentwicklung werden Modelle eingesetzt, um Aspekte der zu entwickelnden Software auf bestimmten Abstraktionsebenen zu beschreiben. Entwurfsmodelle dienen der Beschreibung der logischen Architektur der Software bestehend aus der Systemstruktur, die die Aufteilung des Systems in Teilsysteme beschreibt, sowie der Verhalten dieser Teilsysteme. Entwurfsmodelle sind typische Artefakte, die innerhalb der Entwurfsphase eines Softwareentwicklungsprozesses erstellt werden.

In dieser Arbeit wird das Refactoring von Entwurfsmodellen der Systemklasse Reaktiver Systeme betrachtet. Zunächst wird eine Definition der Klasse der Reaktiven Systeme unter einer Abgrenzung zu anderen Systemklassen festgelegt.

Definition 1.1 (Reaktives System)**.** Entsprechend Harel und Pnueli [HP85] sind Reaktive Systeme offene Systeme die mit ihrer Umgebung fortwährend kommunizieren. Bei Reaktiven Systemen wird der Zeitpunkt der Interaktion von der Umgebung bestimmt.

Reaktive Systeme besitzen meist einen internen Kontrollzustand und terminieren in der Regel nicht. Reaktive Systeme operieren häufig nicht auf komplexen Daten, sondern die Beziehungen zwischen Ein- und Ausgaben stehen bei diesen Systemen im Vordergrund. Bei Reaktiven Systemen handelt es sich häufig um verteilte Systeme, die sich durch Parallelverarbeitung auszeichnen. Der Aspekt der Echtzeit nimmt auf Grund der Eigenschaft, dass die Umgebung den Zeitpunkt der Interaktion bestimmt, häufig eine wichtige Rolle ein.

Definition 1.2 (Proaktives System)**.** Unter proaktiven Systemen verstehen wir Systeme, die mit ihrer Umgebung interagieren, wobei der Zeitpunkt der Interaktion durch das proaktive System, beispielsweise durch eine Eingabeaufforderung an den Benutzer, bestimmt wird.

Definition 1.3 (Interaktives System)**.** Interaktives System ist der Oberbegriff für Reaktive Systeme und proaktive Systeme.

Definition 1.4 (Transformatives System beziehungsweise Batch-System)**.** Im Gegensatz zu interaktiven Systemen verstehen wir unter einem transformativen System beziehungsweise einem Batch-System ein System, das eine gegebene Datenbasis in eine andere Datenbasis überführt und anschließend terminiert.

Definition 1.5 (Eingebettetes System)**.** Ein Eingebettetes System ist ein aus Hardware- und Softwareanteilen bestehendes Computersystem, das in ein anderes technisches System integriert ist.

Heute ist ein Großteil der technischen Systeme mit Eingebetteten Systemen ausgestattet. Populäre Beispiele für den Einsatz von Eingebetteten Systemen finden sich im Automobilbereich. Hier werden eine Vielzahl von Eingebetteten Systemen in Form von Steuergeräten verteilt eingesetzt, um Funktionen in den Bereichen Motorsteuerung, Fahrzeugsicherheit, Komfortelektronik und Infotainment zu realisieren. Bei einem Großteil der Eingebetteten Systeme handelt es sich um Reaktive Systeme. Das Eingebettete System reagiert fortlaufend auf Eingaben des umgebenden technischen Systems mit Ausgaben an das umgebende System.

In dieser Arbeit wird das Gebiet des Softwareentwurfs für Reaktive Systeme in der Anwendungsdomäne der Eingebetteten Systeme betrachtet. Wie vorangehend bei der Definition des Begriffs der Reaktiven Systeme erläutert, unterscheidet sich dieses Gebiet des Softwareentwurfs stark von dem Entwurf von Software in anderen Anwendungsbereichen. Aus diesem Grund wird in diesem Bereich nur in begrenztem Umfang das Paradigma der Objektorientierung in der Softwareentwickung, wie es in dem Buch *Object-Oriented Software Engineering: Using UML, Patterns and Java* [BD03] von Brügge et al. beschrieben ist, eingesetzt. Die Modellierungssprache *Unified Modeling Language* (UML) [FS03, RJB04, OMG04] ist in dem Bereich der Reaktiven Systeme wesentlich schwächer vertreten als in den anderen Bereichen der Softwareentwicklung.

Die Idee der modellbasierten Softwareentwicklung, wie sie von der *Model Driven Architecture* (MDA) [OMG03a] und im AutoFocus Entwicklungsprozess [SPHP02, SBHW03] vorgeschlagen wird, ist die der verstärkten beziehungsweise ausschließlichen Verwendung von Modellen auf unterschiedlichen Abstraktionsebenen. Modelle sind hier das zentrale Beschreibungsmittel der Entwicklungsartefakte und der Übergang zwischen den verschiedenen Modellen kann durch gegebenenfalls automatisierte Modelltransformationen erfolgen. Modellierungssprachen, die in diesem Kontext eingesetzt werden, müssen eine in Form eines Metamodells klar definierte Notation beziehungsweise Syntax und eine präzise definierte Semantik besitzen. Durch die Existenz einer präzise definierten Modellierungssprache eröffnen sich eine Vielzahl von Anwendungsmöglichkeiten für Modelle. Modelle können in einer Simulation ausgeführt, Eigenschaften von Modellen können verifiziert, Modelle können getestet und Modelle können transformiert werden. Auf Grund der Möglichkeit der automatischen Generierung von Programmcode aus Modellen kann der Entwickler in der modellbasierten Softwareentwicklung ausschließlich auf Modellebene arbeiten, ohne Programmcode bearbeiten zu müssen. Die Modellbasierung ist bei der Entwicklung Reaktiver Systeme in der Praxis heute wesentlich stärker etabliert als in anderen Softwareentwicklungsbereichen. In vielen Fällen ist auf Grund einer zu herkömmlichen Softwaresystemen geringeren Komplexität von Reaktiven Systemen eine vollständige Verhaltensmodellierung auf Basis von Entwurfsdiagrammen möglich.

Bei Reaktiven Systemen handelt es sich häufig um sicherheitskritische Systeme im Sin-

ne der Funktionssicherheit. Bei diesen Systemen muss die korrekte Funktion garantiert werden, da im Falle einer Fehlfunktion schwerwiegende Folgen, wie beispielsweise der Tod von Menschen, verursacht werden können. Entwurfsmodelle Reaktiver Systeme werden hier eingesetzt, um Modelleigenschaften durch Verifikationstechniken nachzuweisen. Sind verifizierte Code Generatoren, Betriebssysteme und Micro-Controller vorhanden, dann können die nachgewiesenen Modelleigenschaften auch für die Implementierung garantiert werden.

Auf Grund der Besonderheiten Reaktiver Systeme werden heute im Entwurf spezielle für diesen Bereich entwickelte Modellierungssprachen eingesetzt. Beispiele für Modellierungssprachen und *Computer Aided Software Engineering* (CASE) Werkzeuge für Reaktive Systeme sind:

- *StateChart* [HN96] und *Activity* Diagramme in StateMate und Rhapsody in MicroC von I-Logix.[1]
- *Real-Time Object-Oriented Modeling* (ROOM) [SGW94] und *UML Real-Time* (UML-RT) in Rational Rose Technical Developer von IBM.[2]
- Mathlab / StateFlow von Mathworks.[3]
- Specification and Description Language (SDL) [ITU02] in Tau SDL Suite von Telelogic.[4]
- AutoFocus (siehe Abschnitt 2.2).[5]

Alle in diesem Bereich verwendeten Modellierungssprachen ermöglichen die Spezifikation von hierarchischen Systemstrukturen bestehend aus Komponenten unter strenger Datenkapselung und expliziter Modellierung der Datenabhängigkeiten durch Kommunikationskanäle. Das Verhalten von Komponenten wird in diesen Modellierungssprachen durch Zustandsmaschinen spezifiziert. In dieser Arbeit wird stellvertretend für alle in dem Bereich der Reaktiven Systeme eingesetzten Modellierungssprachen AutoFocus verwendet.

1.2. Refactoring von Programmcode

Refactoring ist eine Technik zur Restrukturierung von Programmcode mit dem Ziel der Qualitätsverbesserung des Codes in Hinblick auf Lesbarkeit, Verständlichkeit und

[1] Die *Activity* Diagramme stellen Datenflussdiagramme dar und sind grundlegend von den UML Aktivitätsdiagrammen verschieden. Web-Seite: `http://www.ilogix.com`.

[2] Web-Seite: `http://www.ibm.com/software/awdtools/developer/technical`.

[3] Web-Seite: `http://www.mathworks.com`.

[4] Web-Seite: `http://www.telelogic.se/corp/products/tau/sdl/index.cfm`.

[5] AutoFocus 1 Web-Seite: `autofocus.in.tum.de`. AutoFocus 2 Web-Seite: `http://www4.in.tum.de/~af2`.

Wartbarkeit. Bekannt wurde diese Technik durch das Buch *Refactoring - Improving the Design of Existing Code* [FBB+99] von Fowler et al., das einen Katalog von Refactorings für die Programmiersprache Java enthält. Fowler definiert den Begriff des Refactorings wie folgt:

Definition 1.6 (Refactoring (Substantiv) nach Fowler, [FBB+99, S. 53])**.** A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Definition 1.7 (Refactor (Verb) nach Fowler, [FBB+99, S. 54])**.** To restructure software by applying a series of refactorings without changing its observable behavior.

Die zentrale Eigenschaft von Refactorings ist die Erhaltung des von außen beobachtbaren bestehenden Verhaltens. Diese Eigenschaft ermöglicht es, große Veränderungen der Struktur eines vorhandenen Softwaresystems vorzunehmen und dabei sicher zu sein, die Funktionalität von diesem durch die Änderungen nicht zu zerstören. Refactorings sind in der Regel kleine Veränderungsschritte, die kombiniert sequentiell angewendet werden können. Dies hat den Vorteil, dass für jede kleine Veränderung des Codes deren Wirkung auf das Verhalten betrachtet werden kann. Wird festgestellt, dass sich auf Grund der Veränderung des Codes eine Verhaltensänderung ergibt, dann kann diese Änderung rückgängig gemacht werden. Die Refactoring-Technik wird eingesetzt, um die Erweiterung eines bestehenden Softwaresystems vorzubereiten. Es werden zunächst Änderungen des Programms durchgeführt, die das Verhalten nicht verändern und der Qualitätssteigerung des Codes dienen. Die Gleichheit des beobachtbaren Verhaltens vor und nach dem Refactoring wird durch strukturiertes Testen sichergestellt. In einem zweiten Schritt wird anschließend die eigentliche Erweiterung des Programmcodes vorgenommen. Durch die durch das Refactoring erzielte bessere Qualität des Codes wird das Risiko von Spezifikations- und Implementierungsfehlern bei der Durchführung der Erweiterung reduziert.

Der agile Entwicklungsprozess *Extreme Programming* [Bec99] von Beck et al. propagiert den Einsatz der Refactoring-Technik. Da in diesem flexiblen Prozess der Schwerpunkt auf die Durchführung der Programmierung gelegt wird und auf die Erstellung von Entwurfsmodellen weitgehend verzichtet wird, verwendet Beck die Refactoring-Technik zur Änderung der Softwarearchitektur auf Basis des Programmcodes vor der Realisierung neuer Funktionalitäten. Die Verwendung von ausgiebigen Test Suiten wird hier nicht nur im Rahmen des Refactorings vorgeschlagen, sondern ist in diesem Prozess der wichtigste Bestandteil der Qualitätssicherung.

1.3. Nutzen von formalem Modell-Refactoring

Die Refactoring-Technik wird sehr stark im Umfeld des *Extreme Programming* Entwicklungsprozesses eingesetzt, da dieser den weitgehenden Verzicht auf Entwurfsmodelle in der Softwareentwicklung vorschlägt. Es ist jedoch heute innerhalb eines Großteils

der Softwareentwicklungsprojekte üblich, Entwurfsmodelle unter Verwendung einer Modellierungssprache einzusetzen. Möchte man in einem solchen Umfeld weiterhin die Refactoring-Technik einsetzen, stößt man auf zwei Fragen:

- Wie können die durch das Programmcode Refactoring bewirkten Änderungen der Softwarestruktur in die Diagramme des Systementwurfs übernommen werden?
- Ist es möglich und sinnvoll, die Refactoring-Technik direkt auf die Diagramme des Softwareentwurfs anzuwenden?

Zur ersten Frage existieren eine Reihe von Arbeiten, die sich mit Konsistenzerhaltung zwischen Programmcode und Entwurfsdiagrammen beschäftigen (siehe Abschnitt 2.1).

Bis heute sind nur wenige Arbeiten vorhanden, die die Übertragung der Refactoring-Technik auf die Modellebene betrachten. Aus der Definition von Fowler [FBB+99] können wir eine informelle Definition von Modell-Refactoring ableiten:

Definition 1.8 (Modell-Refactoring (informell))**.** Ein Modell-Refactoring ist eine strukturelle Veränderung einer Modellspezifikation, die ihre Qualität in Hinblick auf Lesbarkeit, Verständlichkeit und Wartbarkeit erhöht und dabei das außen beobachtbare Verhalten der Spezifikation nicht verändert.

In dieser Arbeit wird das Thema des Modell-Refactorings betrachtet. Zunächst möchten wir die Anwendung von Refactorings auf Modellebene motivieren. Die Refactoring-Technik dient zur Restrukturierung von Programmcode. Durch das Refactoring lassen sich aber nicht nur kleinere Restrukturierungsveränderungen durchführen, sondern Refactoring kann auch eingesetzt werden, um ganze Softwarearchitekturen zu verändern. Solche weitreichenden Änderungen sind auf einer abstrakteren Ebene als der Programmcodeebene, den Entwurfsmodelle bieten, für den Entwickler wesentlich besser verständlich. Werden solche großen Änderungen auf der Ebene der Entwurfsmodelle durchgeführt, dann sinkt gegenüber dem Refactoring auf Programmcodeebene auf Grund der besseren Verständlichkeit das Risiko von Fehlern. Durch eine Werkzeugunterstützung ist es möglich, Modell-Refactorings sehr effizient auf Modellspezifikationen anzuwenden. Das werkzeuggestützte Modell-Refactoring bietet dem Entwickler die Möglichkeit, spielerisch große Architekturen umzustrukturieren, auf diese Weise verschiedene Architekturalternativen auszuprobieren und eine den gegebenen Anforderungen entsprechende optimale Architektur zu finden.

Die Durchführung des Refactorings auf der Ebene der Entwurfsmodelle hat zunächst die Konsequenz, dass die Entwurfsdiagramme immer aktuell den Entwicklungsstand wiedergeben. Diese Veränderungen müssen sich, vorausgesetzt der Entwickler arbeitet ebenfalls auf Basis des Programmcodes, natürlich auch in dem Programmcode widerspiegeln. Es werden also weiterhin Konsistenzerhaltungsmechanismen zwischen den Modellen und dem Programmcode benötigt. Wird hingegen der Programmcode der Implementierung automatisch aus den Entwurfsmodellen generiert, dann können die

Änderungen, die durch das Refactoring bewirkt werden, automatisch durch eine Neugenerierung nach jeder Anwendung eines Modell-Refactorings in den Programmcode übernommen werden. In diesem Fall werden keine Konsistenzerhaltungsmechanismen zwischen Modell und Code benötigt.

In dem Anwendungungsgebiet der Reaktiven Systeme werden heute verstärkt Modelle in der Softwareentwicklung eingesetzt. Die verwendeten Modellierungssprachen ermöglichen eine vollständige Verhaltensmodellierung und CASE Werkzeuge in dieser Anwendungsdomäne bieten bereits heute Code Generatoren an, die es ermöglichen, automatisch aus einem Modell den Programmcode der Implementierung zu erzeugen. Die Komplexität von heutigen Reaktiven Systemen beziehungsweise Eingebetteten Systemen ist zwar im Vergleich zu anderen Softwaresystemen geringer, in den letzten Jahren ist die Komplexität dieser Systeme jedoch sprunghaft angestiegen. Wie in allen Bereichen der Softwareentwicklung werden auch Reaktive Systeme sehr häufig erweitert. Heute werden also neue Qualitätssicherungsmaßnahmen benötigt, die den geänderten Anforderungen der modellbasierten Entwicklung im Bereich der Reaktiven Systeme Rechnung tragen und die Erweiterung von Software unterstützen. Das in dieser Arbeit vorgestellte Modell-Refactoring für Reaktive Systeme ist genau eine solche Qualitätssicherungsmaßnahme, die hilft, Fehler bei der Erweiterung bestehender Systeme zu vermeiden.

Die zentrale Eigenschaft des Refactoring ist die Verhaltensinvarianz der durchgeführten Änderungen. Kann für ein Refactoring diese Verhaltensinvarianz allgemein für alle in der Modellierungssprachen ausdrückbaren Spezifikationen nachgewiesen werden, dann hat dies große Vorteile. Für solche Refactorings ist, eine korrekte Anwendung vorausgesetzt, automatisch die Verhaltensinvarianz garantiert. Ein Nachweis der Verhaltensäquivalenz des Modells vor und nach dem Refactoring ist nicht erforderlich. Hierdurch wird die Effizienz des Refactorings enorm gesteigert. Heute wird üblicherweise beim Refactoring das Testen zum Nachweis der Verhaltensäquivalenz des Modells vor und nach dem Refactoring eingesetzt. Beim Testen wird fast immer nur ein Teil des Gesamtverhaltens des Systems betrachtet. Die Testtechnik kann folglich im Gegensatz zu einem formalen Nachweis der Verhaltensinvarianz nicht garantieren, dass keine Verhaltensänderung durch das Refactoring bewirkt wurde. Die vorliegende Arbeit liefert einen formalen Verhaltensbegriff auf dessen Basis der Nachweis der Verhaltensinvarianz der definierten Refactorings erfolgt.

Modellierungssprachen für Reaktive Systeme besitzen üblicherweise einen Zeitbegriff. Dieser Zeitbegriff wirkt sich auf das zeitliche und gegebenenfalls auch auf das nicht zeitliche Verhalten der Modellspezifikation aus und muss folglich bei dem Refactoring berücksichtigt werden.

Vor allem in dem Anwendungsgebiet der sicherheitskritischen Systeme im Sinne von Funktionssicherheit bietet sich der Einsatz von formal fundiertem Modell-Refactoring an. In dieser Anwendungsdomäne muss die Korrektheit der Funktion des Systems zum Teil auf Grund von staatlichen Reglementierungen nachgewiesen werden. Der Einsatz von Modellen ist in diesem Umfeld zwingend erforderlich. Modelle sicher-

heitskritischer Systeme lassen sich durch das formal fundierte Modell-Refactoring in ihrer Qualität in Hinblick auf Lesbarkeit, Verständlichkeit und Wartbarkeit steigern, wodurch Spezifikationsfehler vermieden beziehungsweise besser erkannt werden können. Durch die Erfüllung der Verhaltensinvarianzeigenschaft können die vor dem Refactoring formal nachgewiesenen Eigenschaften des Systems auch auf das veränderte System nach dem Refactoring übertragen werden.

1.4. Ableitung der Modell-Refactorings

Die in dieser Arbeit in den Kapiteln 5 und 7 aufgeführten AutoFocus Modell-Refactorings sind aus Erfahrungen, die in Modellierungsprojekten im universitären Umfeld gemacht wurden, abgeleitet. Insbesondere wurden in folgenden Projekten Erfahrungen gesammelt, die in diese Arbeit eingeflossen sind:

- Im Rahmen des Seminars „CASE-Werkzeuge für Eingebettete Systeme“[6], das im Wintersemester 2002/2003 stattfand, wurde die Eignung von CASE Werkzeugen für die Entwicklung Eingebetteter Systeme untersucht. Eines der betrachteten Werkzeuge war AutoFocus. Gegenstand der durchgeführten Fallstudie war die inkrementelle Entwicklung eines Türsteuergerätes von DaimlerChrysler, entsprechend einer modifizierten Anforderungsspezifikation von Houdek und Paech [HP02]. Das Türsteuergerät realisiert Komfortfunktionen im Auto, wie beispielsweise die Steuerung der Zentralverriegelung und die Steuerung der elektrischen Sitzverstellung. Durch eine in der Fallstudie durchgeführte Erweiterung, bei der die Anzahl der zu steuernden Sitzachsen und eine Priorisierung der Verstellachsen realisiert werden mussten, konnte die Eignung der CASE Werkzeuge in Hinblick auf die Veränderung von Modellen untersucht werden. Das Ergebnis war ernüchternd. Außer einer teilweise sehr rudimentären *Cut & Paste* Funktion bieten die innerhalb der Studie betrachteten Werkzeuge keinerlei Unterstützung für das Verändern von Modellen. Die Ergebnisse der Studie sind in [SRS+03, SHH+03] publiziert.

- Innerhalb eines Softwareentwicklungsprojektes wurde von Thaler eine Fallstudie zur Service-basierten Entwicklung Eingebetteter Systeme durchgeführt. Im Rahmen der Fallstudie wurde die Steuerung einer Sitzverstellung, ähnlich zu der vorangehend erwähnten Anforderungsspezifikation, aus rein funktionaler Sicht ohne Verwendung von Komponenten beschrieben. Die Ergebnisse der Fallstudie sind in [KSTW04] dokumentiert.

- In [SW99] wurde ein Steuersystem für einen Personenaufzug in AutoFocus entwickelt.

[6] Web-Seite `http://www4.in.tum.de/~schaetz/Werkzeugseminar/case-seminar02.html`.

- Im Rahmen des FairPay Projektes wurde in [WW01] eine Erweiterung der Modellierungssprache AutoFocus für sicherheitskritische Systeme im Sinne der Datensicherheit entwickelt und zur Modellierung eines elektronischen Geldkartensystems eingesetzt.
- In dem Softwaretechnikpraktikum „*Palm based Money Exchange PalME*“[7] im Sommersemester 2002 wurde eine elektronische Geldbörse für den *Palm PDA* unter Berücksichtigung der *Common Criteria* [Com05], einem Katalog von Anforderungen an die Entwicklung von in Bezug auf Datensicherheit kritischen Anwendungen, realisiert. Zur formalen Modellierung der sicherheitskritischen Teile der Applikation wurde AutoFocus eingesetzt. Die Ergebnisse des Praktikums sind in [VWW02] publiziert.
- Im Rahmen der Diplomarbeit [Ben03] wurde eine Qualitätsmetrik für AutoFocus Modellspezifikationen definiert.

Der in dieser Arbeit aufgeführte Katalog von Modell-Refactorings erhebt nicht den Anspruch, wie beispielsweise der Refactoring Katalog von Fowler, dass die vorgestellten Refactorings wirklich der Satz der in der Praxis am häufigsten benötigten verhaltensinvarianten Transformationen sind. Da die in dieser Arbeit definierten Refactorings jedoch sehr grundlegende Transformationen von Modellen ohne Spezialisierung auf bestimmte Spezialanwendungsdomänen darstellen und diese Transformationen bei der Modellierung von unterschiedlichen Systemen mit AutoFocus häufig „per Hand“ durchgeführt wurden beziehungsweise häufig der Wunsch vorhanden war, solche Transformationen durchzuführen, diese aber auf Grund des Aufwandes für die Modellrestrukturierung dann doch nicht durchgeführt werden konnten, gehen wir davon aus, dass diese Refactorings auch in der Praxis häufig eingesetzt werden können. Diese Einschätzung ist jedoch subjektiv und nicht durch Fallstudien belegt. Die Identifikation der Refactorings erfolgte insbesondere nicht auf Basis einer empirischen Studie, die die Bedeutung dieser Restrukturierungsmaßnahmen zeigen würde. Der in dieser Arbeit definierte Katalog von Refactorings stellt keinen Anspruch auf Vollständigkeit dar, in dem Sinn, dass alle wichtigen Modell-Refactorings der Modellierungssprache AutoFocus aufgeführt sind. Des Weiteren wird nicht der Anspruch erhoben, dass der Satz der aufgeführten Refactorings es ermöglicht, Modellspezifikationen durch kombinierte Anwendung der Refactorings in alle entsprechend des gewählten Äquivalenzbegriffs verhaltensäquivalenten Modellspezifikationen überführen zu können.

1.5. Forschungsbeitrag

Der Hauptbeitrag dieser Arbeit ist die Adaption der Refactoring-Technik auf Entwurfsmodelle Reaktiver Systeme unter einer formalen Betrachtung der Verhaltensäquivalenzeigenschaft. Zu dem Thema Modell-Refactoring existieren Arbeiten, an die diese

[7] Web-Seite `http://www4.in.tum.de/~palme/`.

Arbeit anknüpft. Die vorliegende Arbeit grenzt sich aber klar von den anderen Arbeiten zum Thema Modell-Refactoring ab. Wie Philipps et al. in [PR01, PR03] zeigen, existierten bereits vor dem Bekanntwerden der Refactoring-Technik Arbeiten zu den Themen Programm- und Modelltransformation unter Einhaltung bestimmter Eigenschaften. Die Arbeiten zur Verfeinerung von Datenflussdiagrammen von Philipps et al. [PR97, PR99] und *StateCharts* von Rumpe [Rum96, Rum98] lassen sich hierbei als Modell-Refactorings auffassen, falls für das Refactoring ein Verhaltensäquivalenzbegriff gewählt wird, der ein Erweitern von Verhalten beziehungsweise von Schnittstellen toleriert. Diese Arbeiten sind nicht unter der Zielsetzung der Ermöglichung von Refactoring entstanden. Vielmehr steht dort die Zielsetzung der Erweiterung von Modellen im Vordergrund.

Das Refactoring von UML Klassendiagrammen, die die durch das Code Refactoring bewirkten Veränderungen wiederspiegeln und als ein Refactoring von Modellen verstanden werden können, wurde verstärkt untersucht [Ast02, BPPT03, GSMD03]. Die Arbeiten, die der in dieser Arbeit verfolgten Idee des Modell-Refactorings, das dem einer vollständigen Modellierung des Systems, bestehend aus Struktur- und Verhaltensmodellierung, am nächsten kommt, sind die Arbeiten zu dem Refactoring von UML *StateCharts* von Sunye et al. in [SPTJ01] und Boger et al. in [BSF02]. In diesen beiden Arbeiten erfolgt die Definition von einfachen Refactorings von Zustandsmaschinen, die verwandt zu den in dieser Arbeit in Abschnitt 5.2 vorgestellten rein strukturellen Refactorings von AutoFocus Zustandsübergangsdiagrammen sind. Im Gegensatz zu den in dieser Arbeit festgelegten Refactorings wurden jedoch in den beiden vorangehend genannten Arbeiten diese StateChart Refactorings in einem informellen Kontext gesehen, ohne hierbei die *StateCharts* zur vollständigen Verhaltensbeschreibung von Komponenten einzusetzen. Die genannten Arbeiten legen Refactorings ohne dem Vorhandensein einer formalen Semantik der *StateCharts*, ohne einen definierten Verhaltensäquivalenzbegriff und ohne einen Verhaltensäquivalenznachweis fest.

In den vorhandenen Arbeiten zur formalen Semantikbetrachtung von Code-Refactorings, wie beispielsweise den Arbeiten von Mens et al. [MDJ02, MEDJ05] (siehe Abschnitt 2.1), werden auf Grund der Komplexität der Semantiken heutiger in der Praxis eingesetzter objektorientierter Programmiersprachen vereinfachte Äquivalenzbegriffe angewendet, die nur bestimmte häufig statisch überprüfbare Aspekte der Verhaltensäquivalenz berücksichtigen. Hierbei werden die Wirkungen von Refactorings auf den Datenzustandsraum des Programms betrachtet.

Der Hauptbeitrag der semantisch fundierten Adaption der Refactoring-Technik auf Entwurfsmodelle Reaktiver Systeme wird durch einige Neuerungen in diesem Kontext erzielt, die ihrerseits Beiträge zum aktuellen Forschungsstand darstellen. Die vorliegende Arbeit definiert erstmals eine formale Verhaltensäquivalenzeigenschaft für das Modell-Refactoring. Von Modell-Refactorings wird die Gleichheit des Schnittstellenverhaltens der Modellspezifikation vor und nach Durchführung des Refactorings unter Anwendung einer Abstraktion gefordert. Der in dieser Arbeit für Modelle definierte Verhaltensäquivalenzbegriff unterscheidet sich grundlegend von den vorhandenen

Arbeiten zur Formalisierung von Code Refactorings. Die Einfachheit der Semantik der Modellierungssprache AutoFocus und die strikte Trennung der Datenzustandsräume einzelner Komponenten ermöglicht im Gegensatz zu den Arbeiten in Bezug auf Code-Refactoring eine formale Definition eines umfassenden Verhaltensäquivalenzbegriffs über das abstrahierte Schnittstellenverhalten von Systemen und Komponenten. Der Verhaltensäquivalenzbegriff schließt nicht, wie bei den Arbeiten zum formalen Code-Refactoring, die Betrachtung der internen Datenzustandsräume der Komponenten beziehungsweise Systeme mit ein und legt somit die Mengen der nach außen beobachtbar verhaltensäquivalenten Modelle fest. Über diesen Begriff wird die Menge von verhaltensinvarianten Modell-Transformationen, den Modell-Refactorings, ebenfalls präzise festgelegt.

Die Besonderheiten von Modellierungssprachen für Reaktive Systeme werden explizit in dieser Arbeit berücksichtigt. Modellierungssprachen für Reaktive Systeme unterscheiden sich von heute hauptsächlich bei der Entwicklung herkömmlicher Software eingesetzten objektorientierten Modellierungssprachen dahingehend, dass nicht die Datenmodellierung, sondern die Modellierung des Kontrollflusses des Systems im Vordergrund steht. Verteilung, Parallelausführung und Ausführungszeit sind in dieser Anwendungsdomäne wichtige Konzepte, Kommunikationsbeziehungen zwischen verteilten Komponenten werden explizit dargestellt und das Verhalten von Komponenten wird in dieser Anwendungsdomäne oft vollständig modelliert, so dass aus den Modellen direkt Programmcode automatisch generiert werden kann.

Die Arbeit betrachtet erstmals das Refactoring gezeiteter Modelle unter Verwendung von Zeitabstraktionen. Hierbei werden unterschiedliche Zeitabstraktionen auf den Ebenen der Komponenten- und Systemschnittstelle eingesetzt. Die Wirkungen von zeitsynchronen und zeitasynchronen Semantiken auf das Modell-Refactoring wird betrachtet. Die Klasse der zeitrobusten zeitsynchronen Modelle wird identifiziert, die das Refactoring begünstigen. Auf Basis dieser Eigenschaft erfolgt in dieser Arbeit erstmals die Definition einer Semantik einer Modellierungssprache unter der zentralen Anforderung der Unterstützung von Modell-Refactoring. Es wird ein komplexes Modell-Refactoring detailiert definiert, das das Zerteilen einer atomaren Komponente, einschließlich der deren Verhalten beschreibenden Zustandsmaschine, in zwei Komponenten, ermöglicht. Dieses Refactoring bewirkt im Gegensatz zu den bisher bekannten Modell-Refactorings gleichzeitige komplexe Veränderungen der Komponentenstruktur und der Verhaltensbeschreibung und ermöglicht auf diese Weise sehr weitreichende Architekturveränderungen. Es belegt die Eignung der in dieser Arbeit konstruierten Semantik der Modellierungssprache für das Refactoring. Bei den bisherigen Arbeiten zum Thema Modell-Refactoring wurde nicht die Verhaltensinvarianzeigenschaft nachgewiesen. Die Verhaltensinvarianz der in dieser Arbeit definierten Modell-Refactorings wird auf Basis der Semantik der Modellierungssprache gezeigt.

Neben dem Hauptbeitrag zusammen mit den hierfür geleisteten Teilbeiträgen liefert die Arbeit die nachfolgend aufgeführten Nebenbeiträge. Die Arbeit definiert für die Modellierungssprache AutoFocus einen Katalog von Refactorings von Systemstruktur-

und Zustandsübergangsdiagrammen. Der Refactoring Katalog enthält methodische Richtlinien für die Erstellung qualitativ hochwertiger AutoFocus Spezifikationen und ermöglicht es dem Entwickler, umfassende Änderungen von AutoFocus Modellspezifikationen zur Verbesserung ihrer Qualität durchzuführen. Die Modell-Refactoring-Technik wird in dieser Arbeit erstmals in einen modellbasierten Entwicklungsprozess eingeordnet. Der Einsatz von Entwurfsmetriken zur Steuerung des Modell-Refactorings und der Einsatz von Verifikationstechniken zum Nachweis der Verhaltensinvarianz der Refactoring-Transformation beziehungsweise der Verhaltensäquivalenz der Modellspezifikation vor und nach dem Refactoring wird erstmalig in dieser Arbeit diskutiert. Eine umfassende Werkzeugunterstützung für das Modell-Refactoring wird ebenfalls skizziert.

Die ersten Ideen zu dieser Arbeit wurden in [Wiß03] vorgestellt.

1.6. Aufbau der Arbeit

Das Kapitel 2 beschreibt die in dieser Arbeit verwendeten Grundlagen. Der Stand der Forschung im Bereich des Refactorings wird aufgezeigt. Des Weiteren werden die Modellierungssprache AutoFocus und die mathematisch fundierte Spezifikationssprache Focus, die in dieser Arbeit verwendet werden, vorgestellt.

Um eine formale Definition von Verhaltensäquivalenz und damit eine präzise Definition des Refactoring von AutoFocus Modellspezifikationen angeben zu können, wird ein Begriff von AutoFocus Schnittstellenverhalten benötigt. Zu diesem Zweck wird zunächst in Kapitel 3 die Semantik von AutoFocus durch eine Übersetzung in Focus formalisiert und auf Basis dieser Formalisierung der Begriff des Schnittstellenverhaltens zusammen mit einem Begriff der Äquivalenz von Schnittstellenverhalten und einem Kompositionsoperator von Schnittstellenverhalten festgelegt. AutoFocus Modelle sind zeitliche Modelle. Um bei der Veränderung von AutoFocus Spezifikationen gewisse Freiheitsgrade zu gewähren, wird das AutoFocus Schnittstellenverhalten unter einer Abstraktion betrachtet. Zu diesem Zweck werden AutoFocus Zeitabstraktionen unter Verwendung von Focus stromverarbeitenden Funktionen definiert. Zur Betrachtung von zeitabstrakten Systemabläufen wird auf Basis der gezeiteten AutoFocus Sequenzdiagramme eine zeitabstrakte Sequenzdiagrammart definiert. Für abstrakte Schnittstellenverhalten werden eine Konkretisierungsabbildung, ein abstrakter Kompositionsoperator sowie eine abstrakte Verhaltensäquivalenzeigenschaft festgelegt.

Unter Verwendung der in Kapitel 3 festgelegten Eigenschaften und Abbildungen erfolgt in Kapitel 4 die formale Definition von Refactoring. Eine allgemeine Definition von Refactoring, die sowohl für Programmcode Refactoring als auch für Modell-Refactoring gilt, fordert neben der Qualitätsverbesserung der Spezifikation eine Verhaltensäquivalenz der Systemschnittstellenverhalten unter Abstraktion vor und nach dem Refactoring. Ein Refactoring wird als allgemeingültig bezeichnet, falls eine syntaktische Vorbedingung des Refactorings existiert, die die Erfüllung der Verhaltensäquivalenzei-

genschaft impliziert. Die Definition von AutoFocus Modell-Refactoring sieht die Veränderung von Teilspezifikationen vor. Das Schnittstellenverhalten des Gesamtsystems wird hierbei unter Verwendung einer der in Kapitel 3 definierten Zeitabstraktionen beobachtet. Wird keine Abstraktion zur Beobachtung des Schnittstellenverhaltens eingesetzt, dann reicht beim Refactoring von AutoFocus Komponentenspezifikationen die Betrachtung des Schnittstellenverhaltens der veränderten Komponente für den Verhaltensäquivalenznachweis des Systems aus.

In Kapitel 5 werden schließlich allgemeingültige Modell-Refactorings für AutoFocus Systemstrukturdiagramme und AutoFocus Zustandsübergangsdiagramme definiert, die das zeitliche Schnittstellenverhalten nicht verändern. Die hier vorgestellten Refactorings ermöglichen die Restrukturierung der hierarchisch angeordneten Modellelemente. Die Erfüllung der Verhaltensäquivalenzeigenschaft der dort vorgestellten Refactorings wird mit den Eigenschaften der Semantik von AutoFocus begründet.

AutoFocus besitzt eine zeitsynchrone Semantik (siehe Abschnitt 2.2). Das Kapitel 6 befasst sich mit der Frage, welche Auswirkungen diese Semantik auf die Möglichkeiten der Definition allgemeingültiger Refactorings, die das zeitliche Schnittstellenverhalten von Komponenten verändern und dabei das zeitabstrahierte Systemverhalten unverändert lassen sollen, hat. Es wird festgestellt, dass in der zeitsynchronen Semantik von AutoFocus jede lokale zeitliche Veränderung eines Komponentenverhaltens, die unter einer Zeitabstraktion nicht sichtbar ist, ein zeitabstrahiertes Gesamtsystemverhalten verändern kann. Die AutoFocus Zeitabstraktionen sind in Bezug auf die zeitsynchrone AutoFocus Semantik nicht kompositional. Diese Eigenschaft verhindert für die Menge aller in AutoFocus ausdrückbaren Modellspezifikationen die Definition allgemeingültiger Refactorings, die zeitliches Komponentenverhalten verändern. Über die Zeitrobustheitseigenschaft wird eine Unterklasse von AutoFocus Modellspezifikationen identifiziert, die flexibel gegenüber zeitlichen Verschiebungen der Eingaben sind. Bei dem Refactoring von zeitrobusten AutoFocus Komponentenspezifikationen reicht der Nachweis der Verhaltensinvarianzeigenschaft des durch das Refactoring veränderten Komponentenschnittstellenverhaltens unter Anwendung einer Zeitabstraktion aus, um daraus auf die abstrakte Verhaltensinvarianz des Gesamtverhaltens schließen zu können. Für diese Unterklasse können die vorangehend angesprochenen allgemeingültigen Refactorings, die lokale zeitliche Verhaltensänderungen bewirken, festgelegt werden. Nach der Auswahl der anzuwendenden Zeitabstraktionen und einer Betrachtung verschiedener Semantikkonzepte in Bezug auf die Eignung für das Refactoring wird eine zeitasynchrone Semantik durch eine Abbildung in das zeitsynchrone AutoFocus definiert. Es wird gezeigt, dass für alle entsprechend der neuen zeitasynchronen Semantik interpretierten AutoFocus Modellspezifikationen die Zeitrobustheitseigenschaft erfüllt ist.

In Kapitel 7 werden AutoFocus Modellrefactorings definiert, die lokales zeitliches Schnittstellenverhalten verändern, ohne hierbei eine Wirkung auf das zeitabstrahierte Gesamtschnittstellenverhalten zu haben. Die in diesem Kapitel beschriebenen Refactorings sind für zeitasynchrone AutoFocus Modellspezifikationen als auch in ab-

gewandelter Form für zeitrobuste zeitsynchrone AutoFocus Modellspezifikationen allgemeingültig, das heißt sie garantieren, eine korrekte Anwendung vorausgesetzt, die Erfüllung der Verhaltensinvarianzeigenschaft. Für herkömmliche zeitsynchrone AutoFocus Modellspezifikationen sind die hier definierten Refactorings, wie in Kapitel 6 festgestellt, nicht allgemeingültig. Neben einfachen Refactorings von Zustandsübergangsdiagrammen, die beispielsweise das Entfernen von Zwischenzuständen realisieren, wird ein sehr komplexes Refactoring detailliert definiert, das die Zerteilung einer atomaren Komponente, das heißt einer Komponente, die ihrerseits nicht aus Unterkomponenten besteht, einschließlich der dieser Komponente zugeordneten Zustandsmaschine ermöglicht. Ein auch für das zeitasynchrone AutoFocus nicht allgemeingültige Refactoring ermöglicht das Entfernen des Aktivierungsmechanismus, der durch das Refactoring zum Zerteilen atomarer Komponenten eingefügt wird.

Das Kapitel 8 führt schließlich verschiedene Modelloptimierungsmechanismen, die als Modell-Refactoring verstanden werden können, an. Neben dem Entfernen unbenutzter Teile von Schnittstellen (Ports) und Kommunikationspfaden (Kanälen) in AutoFocus Systemstrukturdiagrammen, dem Entfernen von nicht erreichbaren Zuständen in Zustandsmaschinen und der Berechnung von minimalen Zustandsmaschinen wird auch die Berechnung von Produktautomaten, die zur Verschmelzung von zwei atomaren Komponenten zu einer atomaren Komponente eingesetzt werden kann, angesprochen.

Die Modell-Refactoring-Technik wird in Kapitel 9 in einen iterativen modellbasierten Entwicklungsprozess eingeordnet. Die Verwendung von Metriken zur Bewertung der Qualität von Entwurfsmodellen zur Identifikation von Teilspezifikationen schlechter Qualität und der Auswahl geeigneter Refactorings zur Verbesserung der Qualität wird diskutiert. Darüber hinaus sind in Kapitel 9 verschiedene Techniken zur Durchführung des Nachweises der Verhaltensinvarianzeigenschaft bei dem Modell-Refactoring aufgeführt. Eine umfassende Werkzeugunterstützung für das Modell-Refactoring wird skizziert.

In dem abschließenden Kapitel 10 werden die Schlussfolgerungen dieser Arbeit gezogen. Die Übertragbarkeit der in dieser Arbeit erzielten Ergebnisse auf andere Modellierungssprachen wird diskutiert und es werden weiterführende zukünftige Arbeiten im Umfeld des Modell-Refactorings angesprochen. Abschließend wird eine Vision der Softwareentwicklung in der Zukunft aufgezeigt.

2. Grundlagen

In dieser Arbeit wird die Refactoring-Technik auf Modelle in der Softwareentwicklung unter Verwendung eines formalen Verhaltensäquivalenzbegriffs übertragen. In dem Grundlagenkapitel wird in Abschnitt 2.1 zunächst der aktuelle Stand der Forschung auf dem Gebiet des Refactorings aufgezeigt. Der Abschnitt 2.2 stellt die Modellierungssprache und den CASE Werkzeug Prototyp AutoFocus vor, an dessen Beispiel in dieser Arbeit das Modell-Refactoring demonstriert wird. Zur Formalisierung von Verhalten und zur Definition von Zeitabstraktionen verwendet diese Arbeit die mathematisch fundierte Spezifikationssprache Focus, die in Abschnitt 2.3 vorgestellt wird.

2.1. Stand der Forschung im Themengebiet Refactoring

Der Begriff des Refactorings wurde erstmals von Opdyke in seiner PhD Arbeit [Opd92] verwendet. Dort sind verhaltensinvariante Transformationen von Smalltalk Programmcode definiert. Bekannt wurde die Technik durch Fowler, der einen Katalog vordefinierter Refactorings für die Programmierspache Java erstellt hat [FBB+99].

Die Idee der Durchführung von verhaltensinvarianten Transformationen ist jedoch nicht ganz so neu. In [PR01] und [PR03] werden die Ursprünge der Refactoring-Technik betrachtet. Philipps und Rumpe stellen die Refactoring-Technik in Bezug zu dem Konzept der Verfeinerung, mit dessen Pionieren Dijstra [Dij71], Wirth [Wir71] und Bauer [BW82]. In dem Münchner *Computer-Aided Intuition-Guided Programming* (CIP) Projekt [BBB+85] wurde ein transformationsbasierter Ansatz zur Programmentwicklung verfolgt. Die dort verwendeten Transformationen erfüllen zum Teil die Eigenschaft der Verhaltensinvarianz, wie sie auch bei dem Refactoring gefordert wird. Philipps et al. haben in [PR97] und [PR99] Transformationen zur Verfeinerung von Datenflussdiagrammen angegeben, die als Refactorings verstanden werden können. Verfeinerungsregeln für Zustandsmaschinen beziehungsweise *StateCharts* [Har87] wurden von Rumpe in [Rum96], [Rum98] und [Sch98] definiert. Stefănescu definiert in [Ste00] eine allgemeine Netzwerk Algebra. Die Algebra legt Axiome fest, die die invariante Umformung von Netzwerkstrukturen ermöglichen. Diese allgemeine Algebra lässt sich auf spezielle Strukturen, wie beispielsweise Endliche Automaten, Petri Netze und Datenflussdiagramme anwenden.

Einen weit umfassenden Überblick über den aktuellen Stand der Forschung im Bereich des Refactorings liefert der Beitrag [MT04] von Mens et al.. Betrachten wir zunächst verschiedene formale Techniken, die für das Refactoring eingesetzt werden. Opdyke

verwendet Vorbedingungen, die erfüllt sein müssen, um ein Refactoring anwenden zu dürfen [Opd92]. Roberts spezifiziert erstmals Vorbedingungen für Refactorings formal unter Verwendung von Prädikatenlogik und schlägt die Verwendung von Nachbedingungen und Invarianten für Refactorings vor [Rob99]. Ursprünglich wurden Refactorings als Programmtransformationen definiert. Es hat sich jedoch gezeigt, dass sich für die Durchführung und die formale Überprüfung von Refactorings Graphtransformationssysteme sehr gut eignen [Hec95]. Der Einsatz von Softwaremetriken zur Bewertung der Qualität des Programmcodes vor und nach dem Refactoring ist von Simon et al. in [SSL01] beschrieben.

Die Verhaltensinvarianz von Programmtransformationen wurde auf unterschiedliche Weise unter verschiedenen Verhaltensäquivalenzbegriffen untersucht. Auf Grund der Komplexität der Semantiken der in der Praxis eingesetzten Programmiersprachen werden vereinfachte Äquivalenzbegriffe angewendet, die nur bestimmte, häufig statisch überprüfbare, Aspekte der Verhaltensäquivalenz berücksichtigen. Ein Beispiel hierfür sind die Arbeiten von Mens et al. [MDJ02] [MEDJ05], die auf Basis von Graphtransformationen eine Formalisierung von Code Refactorings angeben. Dort werden folgende statisch überprüfbare Eigenschaften in Bezug auf Verhaltensinvarianz eingesetzt:

Access Preservation
: Jede Methode greift nach dem Refactoring mindestens auf die selben Variablen zu, wie vor dem Refactoring. Bei dieser Eigenschaft werden die Variablenzugriffe, die von Untermethoden durchgeführt werden, mit berücksichtigt.

Update Preservation
: Jede Methode aktualisiert unter Berücksichtigung von Untermethoden nach dem Refactoring mindestens die selben Variablen, wie vor dem Refactoring.

Call Preservation
: Jede Methode führt unter Berücksichtigung von Untermethoden nach dem Refactoring mindestens die selben Methodenaufrufe durch, wie vor dem Refactoring.

Bernstein hat die Eigenschaft der *Object Equivalence* [Ber91] als die Eigenschaft, die gleiche Menge von Objekten vor und nach einer Programmtransformation instanziieren zu können, definiert. Als *language-preserving* bezeichnet Bernstein in [Ber97] auf Basis der Theorie der Formalen Sprachen die Eigenschaft, dass das Objektgeflecht eines Programms vor und das Objektgeflecht nach der Programmtransformation eine semantische Äquivalenz aufweist. Bernstein verwendet hierzu für Programmcode, der in einer vereinfachten objektorientierten Pseudosprache vorliegt, eine Repräsentation, die eine Grammatik darstellt. Definieren zwei Grammatiken die gleiche Sprache, dann sind die repräsentierten Objektgeflechte semantisch äquivalent.

Heute wird die Refactoring-Technik hauptsächlich im Umfeld der objektorientierten Sprachen Java [FBB+99] und C++ [SGMZ98] eingesetzt. Es existieren aber auch Refactoring Unterstützungen für imperative Sprachen, wie beispielsweise C [GJ03] und

funktionale Sprachen, wie beispielsweise Haskell [LRT03]. Tichelaar et al. beschreiben in [TDDN00] einen Ansatz für sprachunabhängiges Refactoring.

Im Bereich der Entwurfsdiagramme werden heute in erster Line die durch das Code Refactoring bewirkten Veränderungen von UML Klassendiagrammen betrachtet. In [Ast02] sind Refactorings von großen UML Klassendiagrammen aufgeführt. Objektorientierte Entwurfsmuster [GHJV95, BMR+96] werden heute verstärkt im Softwareentwurf eingesetzt. Douglass definiert in seinem Buch *Doing Hard Time* [Dou99] Entwurfsmuster für eingebettete Echtzeitsysteme. Der Einsatz von Code Refactorings zur Integration von Entwurfsmustern wurde von Batory et al. in [BT95] und von Schulz et al. in [SGMZ98] untersucht. France et al. zeigen in [FGSK03] einen Ansatz zum Refactoring von UML Klassendiagrammen durch die Anwendung von Entwurfsmustern. Ansätze zur Konsistenzerhaltung zwischen Programmcode und UML Klassendiagrammen beim Refactoring sind von Bottoni et al. in [BPPT03] und von Gorp et al. in [GSMD03] dargestellt.

Es existieren Ansätze zur Übertragung der Refactoring-Technik auf die anderen Diagrammarten der UML. Sunye et al. legen in [SPTJ01] Refactorings von UML Klassendiagrammen und erstmals Refactorings von UML *StateCharts* Diagrammen fest. Die Refactorings werden dort durch syntaktische Vor- und Nachbedingungen in der *Object Constraint Language* (OCL) [OMG03b] definiert. Einfache Refactorings von UML *StateCharts* und UML Aktivitätsdiagrammen sind von Boger et al. in [BSF02] zusammen mit einer prototypischen Werkzeugunterstützung skizziert. In [Pre05, Abschnitte 6.1 und 6.2.2] wird ein Verfahren zur verhaltensinvarianten Umwandlung von Variablen- und Kontrollzuständen in AutoFocus Zustandsmaschinen angegeben. Das Refactoring von UML *Use Case* Diagrammen ist in [RB03] beschrieben.

Für die formale Definition von Modell-Refactorings lassen sich Transformationsdefinitionssprachen, wie die *Bidirectional Object oriented Transformation Language* (BOTL) [BM02, Mar05] oder die *Operation Description Language* (ODL) [Sch01], eine Erweiterung der *Object Constraint Language* (OCL) [OMG03b], einsetzen. Engels et al. zeigen in [EHKG02] den Einsatz von konsistenzerhaltenden Modelltransformationen.

Auf Seiten der Werkzeugunterstützung wird heute in vielen Refactoring Werkzeugen und IDEs eine automatisierte Ausführung von durch den Benutzer ausgewählten Refactorings angeboten (siehe Abschnitt 9.4). Es existieren Arbeiten zum vollständig automatisierten Refactoring auf Programmcodeebene einschließlich der automatischen Identifikation von Programmteilen und der automatischen Auswahl geeigneter Refactorings [Moo96, Cas94]. Porres demonstriert in [Por03] die Verwendung eines regelbasierten Transformationssystems für das Refactoring von UML Modellen. Zhang et al. demonstrieren in [ZLG05] den Einsatz des Metamodellierungswerkzeugs *Generic Modeling Environment* (GME) [LBM+01] und des Modelltransformationswerkzeugs *Constraint-Specification Aspect Weaver* (C-SAW)[1] für das Modell-Refactoring. Der Einsatz des Modelltransformationsframeworks Aqua, das in dem CASE Werkzeug AutoFocus integriert ist, ist von Schätz et al. in [SBHW05] beschrieben.

[1] Web-Seite `http://www.gray-area.org/Research/C-SAW`.

2.2. AutoFocus

AutoFocus [BHS99] ist eine Modellierungssprache und ein CASE Werkzeug Prototyp für den Entwurf Reaktiver Systeme.[2] Das Modell-Refactoring wird in dieser Arbeit exemplarisch am Beispiel von AutoFocus demonstriert. Ein kurzer Überblick über die Modellierungssprache und das CASE Werkzeug AutoFocus ist in [HMS+98] an Hand eines Beispiels aufgeführt. Eine Einführung in die Modellierungssprache AutoFocus ist in dem Buch *Modellbildung in der Informatik* [BS04] von Broy und Steinbrüggen sowie in [LPS+02] zu finden.

Die Sprache AutoFocus besitzt zur Spezifikation der Komponentenstruktur eines Systems hierarchische Systemstrukturdiagramme (*System Structure Diagrams* (SSDs)). In diesen Diagrammen werden die einzelnen Komponenten in Form von Rechtecken, deren syntaktische Schnittstellen durch Eingabe-Ports in Form von weißen und Ausgabe-Ports in Form von schwarzen Kreisen sowie die Kommunikationsbeziehungen zwischen den Schnittstellen der einzelnen Komponenten durch Kanäle in Form von Pfeilen dargestellt (Abbildung 2.1a auf der nächsten Seite).

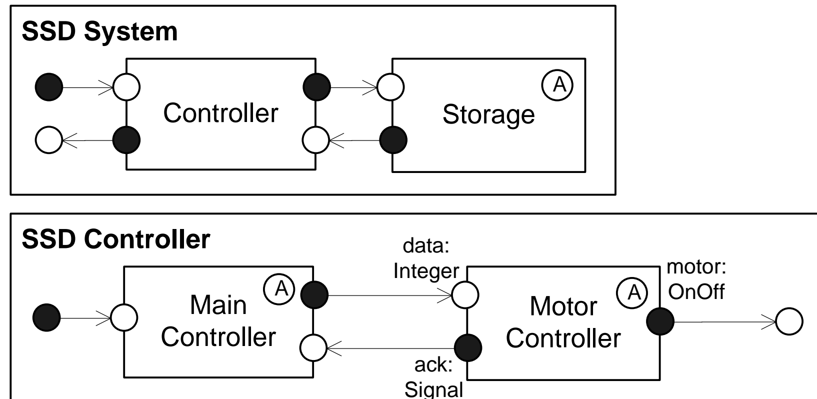
Das Verhalten von atomaren, das heißt nicht weiter in Unterkomponenten gegliederten, Komponenten wird in AutoFocus durch spezielle Zustandsmaschinen beschrieben. Die Zustandsmaschinen werden Zustandsübergangsdiagramme (*State Transition Diagrams* (STDs)) genannt (Abbildung 2.1b auf der nächsten Seite). STDs können hierarchisch aufgeteilt werden. Eine Transition in einem STD besitzt eine Vorbedingung an die lokalen Variablen der Komponente und eine Bedingung an die Eingaben der Komponente. Mit dem Feuern einer Transition wird eine Ausgabe getätigt und es erfolgt eine Variablenzuweisung.

Exemplarische Abläufe der Kommunikation zwischen den einzelnen Komponenten des Systems werden in AutoFocus durch spezielle gezeitete Sequenzdiagramme (*Enhanced Event Traces* (EETs)) dargestellt (Abbildung 2.1c auf der nächsten Seite). Durch horizontale gestrichelte Linien werden in diesen Diagrammen die AutoFocus Zeitschritte repräsentiert. Datentypdefinitionen und Funktionsdefinitionen erfolgen in AutoFocus in der funktionalen Sprache Quest/F.

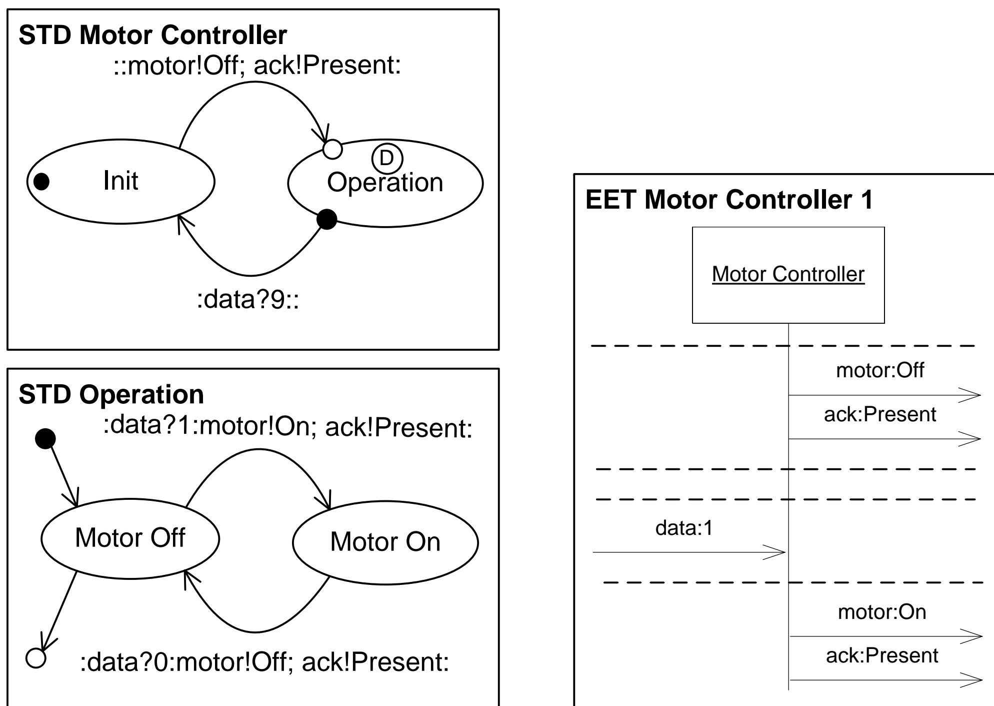
Die sich gerade in der Entwicklung befindende Version 2 von AutoFocus bietet, neben den hier aufgeführten und in dieser Arbeit verwendeten Beschreibungstechniken, *Mode Switching Charts* (MSCs) und *Data Flow Diagrams* (DFDs) zur Spezifikation an.

Die Modellierungssprache AutoFocus besitzt ein auf UML Klassendiagrammen basierendes Metamodell, das die in den verschiedenen Diagrammarten der Modellierungssprache vorkommenden Konzepte und deren Zusammenhänge definiert. Dieses Metamodell ist auf der Meta-Ebene *M2* der *Meta-Object Facility* (MOF) [OMG02] angeordnet. Entsprechend der in [SBHW03] aufgeführten Klassifikation handelt es sich bei dem AutoFocus Metamodell um ein konzeptuelles Modell. Die Abbildung 2.2 auf Seite 37 zeigt

[2] AutoFocus 1 Web-Seite `http://autofocus.in.tum.de`. AutoFocus 2 Web-Seite: `http://www4.in.tum.de/~af2/`.



(a) AutoFocus Systemstrukturdiagramme.



(b) AutoFocus Zustandsübergangsdiagramme.

(c) AutoFocus Sequenzdiagramm.

Abbildung 2.1.: Die AutoFocus Modellierungssprache.

in stark vereinfachter Form den Ausschnitt des AutoFocus Metamodells, der die Systemstrukturdiagramme beschreibt.

Das Metamodell wird in der Werkzeugimplementierung zur automatischen Generierung der Modelldatenstruktur genutzt und ist in der AutoFocus Modell API Dokumentation [Aut06b] und der AutoFocus 2 Code Dokumentation [Aut06a] detailliert beschrieben. Die Refactoring-Operationen werden in dieser Arbeit auf Basis dieses Metamodells definiert.

Die Semantik von AutoFocus ist in [HSE97] auf Basis des μ-Kalküls und in Kapitel 3 in dieser Arbeit auf Basis der Spezifikationssprache Focus definiert. Die einzelnen atomaren Komponenten einer AutoFocus Spezifikation werden parallel ausgeführt. Die Semantik von AutoFocus wird nach der Klassifikation in [KS03] als nachrichtenasynchron und zeitsynchron bezeichnet. Nachrichtenasynchron heißt in diesem Zusammenhang, dass die eine Nachricht sendende Komponente nicht bis zur Verarbeitung dieser Nachricht in der empfangenden Komponente blockiert wird. Unter Zeitsynchronität verstehen wir die Eigenschaft, dass die Ausführung der einzelnen parallel arbeitenden Komponenten unter zeitlicher Synchronisierung geschieht. In AutoFocus wird pro Zeitschritt genau eine Transition in jeder der die Verhalten der atomaren Komponenten beschreibenden Zustandsmaschinen gefeuert. Die Ausgaben werden in AutoFocus nach der standardmäßigen Kommunikationssemantik immer um einen Zeitschritt verzögert, das heißt Eingaben zu dem Zeitpunkt t können erst die Ausgaben zum Zeitpunkt $t+1$ beeinflussen. AutoFocus bietet darüber hinaus die spezielle *Immediate* Kommunikation, die durch rauteförmige Ports in den Systemstrukturdiagrammen dargestellt ist, und keine Verzögerung der Kommunikation bewirkt.

Das CASE Werkzeug AutoFocus bietet graphische Editoren, automatische Konsistenzprüfung, Simulation und Unterstützung für Requirements Engineering [SFGP05a, SFGP05b]. Mit dem Aqua Framework [Sch01, SBHW05] bietet AutoFocus Unterstützung für Modelltransformationen. Mit der AutoFocus Erweiterung Quest [BLS00], die von der Firma Validas AG[3] vertrieben wird, werden die Codegenerierung für unterschiedliche Zielplattformen, die Testfallgenerierung [PS01] und die Anbindungen an Verifikationswerkzeuge [Slo00] an AutoFocus unterstützt.

2.3. Focus

Focus [BS01, BDD$^+$92] von Broy und Stølen et al. ist eine mathematisch fundierte Sprache und Methodik zur Spezifikation von Softwaresystemen. Als Basiskonzept verwendet Focus Sequenzen von Ein- und Ausgaben, die Ströme genannt werden. Focus bietet verschiedene Arten von Beschreibungen zur Spezifikation von Systemen an. Im relationalen Stil wird die Ein-/Ausgaberelation eines Systems durch eine prädikatenlogische Formel festgelegt. Im Gleichungsstil werden rekursive Funktionen zur Spezifikation von Ausgabeströmen in Abhängigkeit der Eingabeströme verwendet. Diese Funktio-

[3]Web-Seite `http://www.validas.de`.

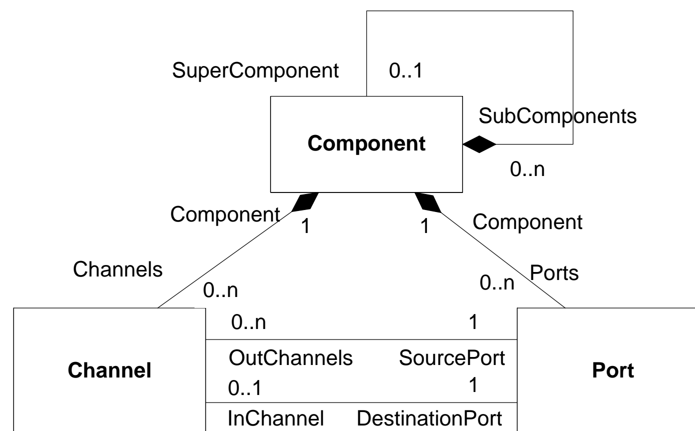


Abbildung 2.2.: Ausschnitt der Systemstrukturdiagramme innerhalb des AutoFocus Metamodells.

nen werden stromverarbeitende Funktionen genannt. Darüber hinaus bietet Focus die Möglichkeit, Annahme/Garantie Spezifikationen und graphische Spezifikationen festzulegen. Focus bietet drei verschiedene Semantiken an: Ungezeitetes Focus, gezeitetes Focus und zeitsynchrones Focus. Durch die Integration der Semantiken von ungezeitetem und zeitsynchronem Focus in das gezeitete Focus ist der Übergang zwischen den verschiedenen Zeitmodellen lückenlos möglich.

Durch die Fokusierung auf die Betrachtung von Ein- und Ausgabeströmen eignet sich Focus sehr gut für die Darstellung von Schnittstellenverhalten. In [Bro95] wurde von Broy das Verhalten Reaktiver Systeme durch Mengen von Strömen, dort *Traces* genannt, dargestellt und charakterisiert. Zentrale Konzepte der Focus Methodik zur Spezifikation von Systemen sind die Verhaltensverfeinerung, die es ermöglicht, kontrolliert zusätzliche Funktionalität in eine Spezifikation zu integrieren und die Schnittstellenverfeinerung, die die Erweiterung von Komponentenschnittstellen betrachtet. In den Arbeiten von Broy zur Zeitverfeinerung [Bro97, Bro01] werden die Übergänge von diskreter zu kontinuierlicher Zeit und von diskreter zu dichter Zeit dargestellt und der Übergang von einer zeitasynchronen Semantik in eine zeitsynchrone Semantik wird an Hand einer Verzögerungsfunktion untersucht. Zeitabstraktionen in Focus werden von Broy in den Arbeiten [Bro93, Kapitel 6], [Bro03] und [Bro04] betrachtet.

In dieser Arbeit wird gezeitetes Focus verwendet, um einen formalen Begriff des Schnittstellenverhaltens von AutoFocus Spezifikationen festzulegen. Zu diesem Zweck wird eine Übersetzung von AutoFocus in Focus im relationalen Stil festgelegt (Kapitel 3). Die Focus Begriffe der Komposition von Verhalten und der Verhaltensäquivalenz können auf diese Weise auf AutoFocus Schnittstellenverhalten angewendet werden. Unter Verwendung von Focus stromverarbeitenden Funktionen werden Zeitabstraktionen von AutoFocus Verhalten definiert (Abschnitt 3.7).

Auf eine detailliertere Beschreibung von Focus wird an dieser Stelle verzichtet, da mit dem Buch *Specification and Development of Interactive Systems—Focus on Streams, Interfaces, and Refinement* [BS01] von Manfred Broy und Ketil Stølen eine sehr umfassende und detaillierte Referenz von Focus existiert. In diesem Buch findet sich im Anhang B ein Überblick über die in Focus verwendete Notation, die in dieser Arbeit nicht gesondert beschrieben wird. In dieser Arbeit sind im Anhang A die abweichend von der Standard Focus Notation verwendeten Symbole, Funktionen und Operatoren aufgelistet und erläutert.

3. Die AutoFocus Semantik, Schnittstellenverhalten und Abstraktionen

In dieser Arbeit wird die Verhaltensinvarianzeigenschaft von Refactoring auf einer formalen Ebene betrachtet. Um verschiedene Verhalten miteinander vergleichen zu können, werden ein Begriff von Schnittstellenverhalten sowie Abstraktionen dieses Schnittstellenverhaltens benötigt. Die Modellierungssprache AutoFocus besitzt eine formale Semantik, die in [HSE97] auf Basis des μ-Kalküls definiert wurde. Die formale Spezifikationssprache Focus [BDD$^+$92, BS01] besitzt im Gegensatz zu der vorhandenen Formalisierung von AutoFocus in dem μ-Kalkül einen Begriff von Schnittstellenverhalten in Form von Ein- und Ausgabeströmen als grundlegendes Konzept. Des Weiteren bietet Focus einen Zeitbegriff sowie eine Zeitabstraktion. Eine Semantikdefinition von AutoFocus in der Sprache Focus ist bezüglich des Einsatzes in dieser Arbeit folglich besser geeignet als die bestehende Formalisierung. In den folgenden Abschnitten wird die Semantik von AutoFocus durch eine Abbildung in die Sprache Focus definiert. Zur Vereinfachung der Semantikdefinition wird hierbei das AutoFocus Konzept des Endzustands, das in der Praxis nur sehr selten in der Anwendungsdomäne der Reaktiven Systeme beziehungsweise Eingebetteten Systeme genutzt wird, nicht berücksichtigt und es werden jeweils unendlich lange Modellabläufe angenommen.

In Abschnitt 3.1 erfolgt zunächst die Übersetzung von AutoFocus Systemstrukturdiagrammen (SSDs) in Rümpfe von Focus Spezifikationen. Die Übersetzung der Zustandsmaschinen (STDs) erfolgt in Abschnitt 3.2. In Abschnitt 3.3 wird schließlich die Semantik einer AutoFocus Modellspezifikation durch die Denotation dieser in Focus übersetzten Spezifikation festgelegt. In Abschnitt 3.4 wird das Schnittstellenverhalten einer AutoFocus Modellspezifikation durch Mengen von Tupel von Focus-Strömen dargestellt. Schließlich wird in Abschnitt 3.5 ein Äquivalenzbegriff von Schnittstellenverhalten definiert und in Abschnitt 3.6 erfolgt die Definition eines Kompositionsoperators für konkretes Schnittstellenverhalten. In Abschnitt 3.7 werden schließlich verschiedene Abstraktionen, in erster Line Zeitabstraktionen, für konkretes AutoFocus Schnittstellenverhalten unter Verwendung von Focus stromverarbeitenden Funktionen definiert. Darüber hinaus erfolgt in diesem Abschnitt auch die Definition einer Konkretisierungsabbildung von abstrakten Schnittstellenverhalten in Mengen möglicher konkreter Schnittstellenverhalten, die Definition von zeitabstrakten Sequenzdiagrammen, die Definition der Eigenschaft der Verhaltensäquivalenz von abstrakten Schnittstellenverhalten und die Definition des abstrakten Kompositionsoperators.

3.1. Übersetzung von Systemstrukturdiagrammen in Focus

Im Folgenden wird durch eine Übersetzung der Modellierungssprache AutoFocus in die Spezifikationssprache Focus eine formale Semantikdefinition von AutoFocus Modellen erstellt, die einen formalen Begriff von Schnittstellenverhalten aufweist und als Basis für die Definition von AutoFocus Zeitabstraktionen zur Festlegung eines Beobachtungsbegriffs für das Modell-Refactoring dient.

Definition 3.1 (Übersetzung AutoFocus nach Focus)**.** Sei $\mathcal{L}_A$ die Menge aller in der Sprache AutoFocus ausdrückbaren Modellspezifikationen und $\mathcal{L}_F$ die Menge aller in der Sprache Focus ausdrückbaren Spezifikationen.

Die Übersetzung von AutoFocus nach Focus ist die Abbildung $C_{Focus} \in \mathcal{L}_A \rightarrow \mathcal{L}_F$. Die Übersetzungsabbildung ist durch die in diesem und dem nächsten Abschnitt angegebenen Übersetzungsregeln definiert.

Hierbei wird auch auf eine Übersetzung der in AutoFocus einsetzbaren funktionalen Sprache Quest/F verzichtet, da sich diese Arbeit auf die Betrachtung von Refactoring der Modelldiagramme beschränkt und nicht auf das Refactoring von in dem Modell enthaltenen in Programmiersprachen formulierten Spezifikationsanteilen eingeht.

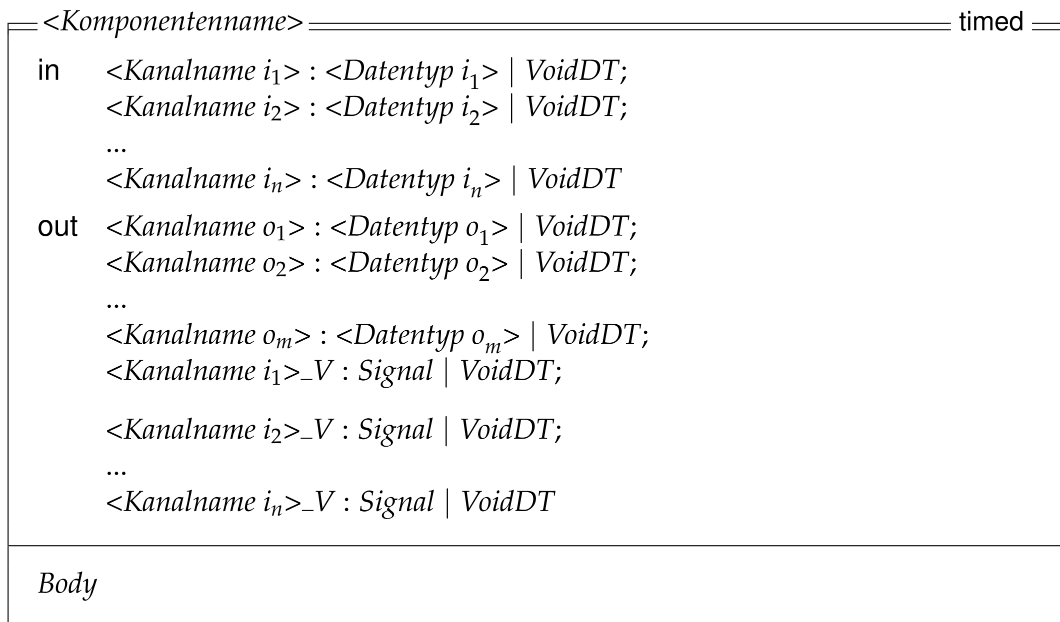
Zunächst werden die AutoFocus Systemstrukturdiagramme (SSDs) in Focus-Spezifikationsrümpfe übersetzt. Die Semantik von AutoFocus, wie sie in [HSE97] definiert wurde, weist der hierarchischen Strukturierung von Systemen in verschachtelten Systemstrukturdiagrammen keine Bedeutung zu. Eine hierarchisch strukturierte Spezifikation A, bestehend aus mehreren Systemstrukturdiagrammen (SSDs), verhält sich dementsprechend identisch zu einer Spezifikation B, bestehend aus einem SSD, welche alle atomaren Komponenten von A enthält und deren Ports äquivalent zur Spezifikation A über Kanäle unter Weglassung von Zwischen-Ports, die zur Verbindung zwischen den einzelnen hierarchischen SSDs verwendet werden, verbunden sind.

Aufgrund dieser Äquivalenz erfolgt die Übersetzung von AutoFocus Strukturdiagrammen in Focus nur unter Berücksichtigung der atomaren Komponenten. Komponenten, die wiederum aus Teilkomponenten bestehen, dienen in AutoFocus nur der Strukturierung des Systems und werden nicht in Focus übersetzt.

Sei A die Menge aller atomaren Komponenten, die in den verschiedenen hierarchischen SSDs einer AutoFocus Spezifikation S_A enthalten sind. Jede dieser AutoFocus Komponenten $a \in A$ wird zunächst in eine eigene Focus Spezifikation übersetzt. Die Übersetzung erfolgt in das allgemeine gezeitete Focus [BS01, S. 81, Definition 5.2.1].

Zur Vereinfachung der Übersetzung nehmen wir an, dass alle atomaren Komponenten $a \in A$ einer AutoFocus Spezifikation S_A einen zueinander eindeutigen Namen besitzen. Des Weiteren gehen wir zur Vereinfachung der Übersetzung davon aus, dass alle in einer Spezifikation enthaltenen AutoFocus Port Bezeichner über alle Komponenten hinweg eindeutige Namen besitzen. Die in der Focus Spezifikation verwendeten Ausdrücke, die in spitze Klammern gesetzt sind, stellen Platzhalter für Bezeichner dar. Jede

atomare Komponente $a \in A$ der AutoFocus Spezifikation wird in eine Focus Komponentenspezifikation mit gleichem Namen (<*Komponentenname*>) übersetzt, die den in der Spezifikation 3.1 gezeigten Spezifikationsrahmen besitzt.



Spezifikation 3.1: Spezifikationsrahmen einer von AutoFocus nach Focus übersetzten Modellspezifikation.

Für jeden Eingabe-Port der AutoFocus Komponente wird der *In* Ausdruck um einen Focus Eingabekanal (<*Kanalname* i_x>) mit entsprechendem Datentyp erweitert. Die Kommunikation zwischen Teilspezifikationen geschieht in Focus über gleich benannte Ein- und Ausgabekanäle in den verschiedenen Komponentenspezifikationen. Aus diesem Grund verwenden wir als Namen eines Eingabekanals nicht den Eingabe-Port-Namen der betrachteten AutoFocus Komponente, sondern den mit diesem Port über einen Kanal verbundenen Ausgabe-Port-Namen der sendenden Komponente. Alle Ausgabe-Ports der betrachteten AutoFocus Komponente werden in dem *Out* Ausdruck als Focus Ausgabekanal mit gleichem Namen und entsprechendem Datentyp aufgeführt. Die in den Focus *In* und *Out* Ausdrücken aufgelisteten Kanalnamen werden entsprechend ihrer alphabetischen Ordnung sortiert, um eine Zuordnung der einzelnen Focus Ein- und Ausgabeströme in der Ein-/Ausgaberelation zu gewährleisten und später den Vergleich der syntaktischen Schnittstelle und des Schnittstellenverhaltens von AutoFocus Spezifikationen eindeutig zu ermöglichen.

In AutoFocus können Ports zu bestimmten Zeitpunkten keinen Wert besitzen. Das Nichtvorhandensein eines Wertes wird in der Focus Übersetzung durch einen spezi-

ellen *void* Aufzählungswert angezeigt. Die AutoFocus Datentypen der Ein- / und Ausgabekanäle werden in der Übersetzung nach Focus jeweils um diesen speziellen *void* Wert erweitert. Hierzu wird ein neuer Datentyp *VoidDT* in Focus definiert.

```
type VoidDT = {void}
```

Nach der AutoFocus Semantik stehen Eingaben genau einen Systemtakt lang zur Verfügung und werden im darauf folgenden Takt automatisch gelöscht beziehungsweise überschrieben. Erfolgt zu dem Zeitpunkt der Eingabe keine Verarbeitung dieser durch die der Komponente zugeordneten Zustandsmaschine, das heißt weist der Eingabeterm der zu diesem Zeitpunkt gefeuerten Transition keine Abhängigkeit zur betrachteten Eingabe auf, dann hat die Eingabe keine Wirkung auf die betrachtete Systemausführung. Die Eingabe wird durch die Transition nicht ausgewertet und geht in diesem Sinn verloren. Um die Verarbeitung von Eingaben nach außen sichtbar zu machen, wird in der Focus Spezifikation für jeden Eingabekanal ein spezieller Ausgabekanal mit dem Namen *<Kanalname* i_x*>_V* von dem Datentyp *Signal* hinzugefügt, über den die Verarbeitung einer Eingabe angezeigt wird. Der Datentyp *Signal* besitzt in AutoFocus nur einen Aufzählungswert *Present*.

```
type Signal = {Present}
```

Das Verhalten der Komponente wird in dem Rumpf (*Body*) der Spezifikation beschrieben. Dieser ergibt sich aus der Übersetzung von Zustandsmaschinen in Focus Relationen (siehe Abschnitt 3.2).

Die Menge aller in Focus übersetzten, in einer AutoFocus Spezifikation enthaltenen atomaren Komponentenspezifikationen, wird mit *F* bezeichnet. Das Verhalten der gesamten AutoFocus Spezifikation ist festgelegt durch das Verhalten der durch die Komposition mit Rückkopplung $\otimes$ [BS01, S.86f] zusammengesetzten in *F* enthaltenen Komponentenspezifikationen. Das Verhalten ist in Focus durch die Denotation $[\![\, S \,]\!]$ angegeben:

$$[\![\, S \,]\!] \stackrel{\text{def}}{=} [\![\, \otimes_{f \in F} f \,]\!] \tag{3.1}$$

3.2. Abbildung von AutoFocus Zustandsmaschinen (STDs) in Focus Relationen

Das Verhalten atomarer AutoFocus Komponenten ist durch Zustandsübergangsdiagramme (STDs) beschrieben. Diese Zustandsübergangsdiagramme werden in Focus Spezifikationen, die ein zur AutoFocus Komponente äquivalentes Schnittstellenverhalten zeigen, dargestellt. Hierbei wird in Focus der relationale Spezifikationstil eingesetzt.

Hierarchische Zustandsübergangsdiagramme. Analog zu hierarchischen Systemstrukturdiagrammen (SSDs) bieten auch Zustandsübergangsdiagramme die Möglichkeit der hierarchischen Strukturierung. Hierbei existieren, ähnlich zu den Zwischen-Ports in den SSDs, Konnektoren zur Realisierung von Transitionen zwischen verschiedenen hierarchischen Zustandsübergangsdiagrammen. Zwei Transitionen, die über einen Konnektor verbunden sind, werden in der Semantik von AutoFocus als eine Transition betrachtet, deren Variablenbedingungen, Eingabebedingungen, Ausgaben und Variablenzuweisungen jeweils durch eine logischen Konjunktion verknüpft sind. Die Hierarchie hat in diesen Diagrammen nur eine sehr eingeschränkte Bedeutung für die Semantik.

Hierarchische AutoFocus Zustandsübergangsdiagramme können in verhaltensäquivalente nicht hierarchische Zustandsübergangsdiagramme übersetzt werden. Hierzu werden alle hierarchischen Zustände, das heißt Zustände, denen eine Unterzustandsmaschine zugeordnet ist, durch diese Unterzustandsmaschine ersetzt. Eine Transition, die ursprünglich über einen Konnektor mit einer Transition der Unterzustandsmaschine verbunden war, wird durch eine neue direkte Transition ersetzt, die durch eine logische Konjunktion der Vorbedingungen, Eingabebedingungen, Ausgabebedingungen und Zuweisungen aus der in den Konnektor eingehenden Transition und der von dem Konnektor ausgehenden Transition gebildet werden. Besitzt ein Konnektor mehrere ein- und ausgehende Transitionen, dann wird für jede Kombination von ein- und ausgehenden Transitionen eine äquivalente direkte Transition durch die logische Konjunktion gebildet.

Zeitsynchronität. Das nicht hierarchische AutoFocus Zustandsübergangsdiagramm wird in Focus Terme abgebildet. Nach der AutoFocus Semantik ist ein Port zu jedem Zeitpunkt t entweder genau mit einem Wert belegt oder nicht. In der Abbildung nach Focus wird ein spezieller *void* Wert zur Darstellung eines nicht belegten Ports verwendet. AutoFocus Ein- / Ausgabesequenzen entsprechen unter Focus folglich genau zeitsynchronen Focus Strömen, die zwischen zwei Ticks jeweils genau eine Nachricht enthalten (spezielle zeitsynchrone Variante von Focus) [BS01, Definition 5.2.3, S.83]. In AutoFocus ist für die Eingaben sichergestellt, dass diese zeitsynchron erfolgen und Ausgaben müssen dementsprechend ebenfalls zeitsynchron sein. Aus diesem Grund wird in dem *Body* der Focus Spezifikation für jeden Eingabekanal der Spezifikation (Menge der Eingabekanäle I) die Eigenschaft der Zeitsynchronität angenommen (Funktion ts). Von allen Ausgabeströmen der Spezifikation (Menge der Ausgabekanäle O) einschließlich der Verarbeitungsströme (Menge der speziellen Verarbeitungskanäle V) wird gefordert, dass diese zeitsynchron sein müssen.[1]

$$\bigwedge_{i \in I} \mathsf{ts}(i) \Rightarrow \bigwedge_{o \in O} \mathsf{ts}(o) \wedge \bigwedge_{v \in V} \mathsf{ts}(v) \wedge \qquad (3.2)$$

[1] Zur Erläuterung der Notation $\bigwedge_{i \in I} e(i)$ siehe Anhang A.

Kontrollzustand. Die Focus Spezifikation benötigt zur Repräsentation der Historie von durch die Zustandsmaschine erreichten Konstrollzuständen einen Strom *stateHistory*. Zunächst wird der Datentyp *<Component Name>_States* für diesen Strom definiert.

Die einzelnen Namen der Zustände müssen innerhalb des betrachteten Zustandsübergangsdiagramms eindeutig gewählt sein. Der Datentyp *<Component Name>_States* ist ein Aufzählungstyp, der alle Namen der Zustände der Zustandsmaschine auflistet.

```
type <Component Name>_States = { <State Name 1> ,
     <State Name 2> , ... , <State Name n> }
```

Wir fügen nun in den Focus *Body* die existenziell gebundene Variable *stateHistory* ein, die einen unendlich langen zeitsynchronen Strom von Zuständen repräsentiert.

$$\exists\, stateHistory \in \text{<Component Name>_States}^{\underline{\infty}} : \mathsf{ts}(stateHistory) \wedge \tag{3.3}$$

Lokale Variablen. Alle in der AutoFocus Komponente vorhandenen lokalen Variablen werden in der Focus Spezifikation als gebundene zeitsynchrone Ströme von Belegungen der Variablen dargestellt.

Für jede lokale Variable der AutoFocus Spezifikation wird folgender Ausdruck an den Focus *Body* angehängt:

$$\exists \text{<Variablenname>} \in \text{<Datentyp der Variable>}^{\underline{\infty}} : \mathsf{ts}(v) \wedge \tag{3.4}$$

Die Datentypen der Variablen werden im Gegensatz zu den Datentypen, die zur Kommunikation verwendet werden, nicht um ein *void* Element erweitert.

Initialisierung. Wir übersetzen nun das in der AutoFocus Zustandsmaschine beschriebene Verhalten in einen Focus Term. Zum Zeitpunkt $t = 1$ befindet sich die AutoFocus Zustandsmaschine im Startzustand, alle Ausgabekanäle, die keine *Immediate* Ports repräsentieren[2], sind leer, und alle lokalen Variablen sind zu diesem Zeitpunkt mit ihrem Initialwert belegt. Sei $P \subseteq O$ die Menge der Focus Ausgabekanäle, die keine *Immediate* Ports repräsentieren, und A die Menge der lokalen Focus Kanäle zur Darstellung der lokalen AutoFocus Variablen. In dem *Body* der Focus Spezifikation wird folgender Term angehängt:

$$\begin{gathered}
\overline{stateHistory}.1 = \text{<Name Startzustand>} \wedge \\
\textstyle\bigwedge_{p \in P} \overline{p}.1 = void \wedge \\
\textstyle\bigwedge_{a \in A} \overline{a}.1 = \text{<Initalwert der Variable a>} \wedge
\end{gathered} \tag{3.5}$$

[2]Hiervon sind folglich auch die speziellen Verarbeitungskanäle ausgenommen.

Bei der Selektion des Zeitpunktes innerhalb eines Stroms wird der Focus Zeitabstraktionsoperator $\bar{}$ [BS01, S. 65] verwendet. Da es sich bei allen verwendeten Strömen um zeitsynchrone Ströme handelt, bei denen zwischen zwei Taktsymbolen immer genau eine Nachricht existiert, enthält nach dem Filtern der Taktsymbole die Stelle t innerhalb des Stroms die Nachricht zum Zeitpunkt t.

Darstellung der Zeitpunkte. Zur Darstellung der einzelnen Zeitpunkte wird eine gebundene Variable t verwendet.[3] Der *Body* der Focus Spezifikation wird um folgenden Teilausdruck erweitert:

$$(\forall\, t \in \mathbb{N}_+ : \tag{3.6}$$

Übersetzung der Transitionen. Nach der AutoFocus Semantik wird in einer Zustandsmaschine, falls zu einem gegebenen Zeitpunkt im aktuellen Zustand keine Transition schaltbereit ist, eine implizite *Idle* Transition gefeuert. Hieraus folgt für die Focus Spezifikation, dass zu jedem Zeitpunkt in der Modellausführung eine Transition ausgehend vom aktuellen Kontrollzustand zum Zeitpunkt t existiert, so dass die Variablen- und Eingabebedingungen dieser Transition erfüllt sind. Zum Zeitpunkt $t+1$ müssen dann die Ausgaben auf herkömmliche AutoFocus Ports (nicht *Immediate* Ports) und die

[3] In der hier betrachten Semantikdarstellung werden unendliche Abläufe der Spezifikation betrachtet. Da in der Semantikdefinition das Konzept von AutoFocus Endzuständen in den Zustandsmaschinen, das in der Praxis sehr selten bei der Beschreibung reaktiver System eingesetzt wird, nicht berücksichtigt wird, sind alle gültigen Abläufe von AutoFocus Modellen nicht zeitlich begrenzt und somit ist die Betrachtung unendlicher Ströme in der nach Focus übersetzten Spezifikation korrekt und erforderlich. Durch eine Modifikation der in diesen Abschnitten beschriebenen Übersetzung von AutoFocus nach Focus ist es jedoch möglich, auch AutoFocus Verhalten unter zeitlicher Begrenzung zu betrachten, das heißt es werden in den Abläufen nur Nachrichten bis zu einem gegebenen maximalen Zeitpunkt betrachtet. Um eine Semantikdefinition von zeitlich begrenztem Verhalten zu realisieren, müssen in der gegebenen Übersetzung folgende Dinge geändert werden:

- Der Datentyp aller in der Focus Spezifikation auftretenden Ströme muss für die Berücksichtigung endlicher Ströme erweitert werden. Hierzu werden die in der Spezifikation vorkommenden Mengen von Strömen von $x^{\underline{\infty}}$ auf $x^{\underline{\omega}}$ geändert.
- Die Spezifikation darf nur Verhalten festlegen, wenn die Länge der Eingabeströme unter Verwendung des Focus Zeitabstraktionsoperators gleich dem als Konstante gegebenen maximal betrachteten Zeitpunkt ist. Die Erfüllung dieser Eigenschaft impliziert also den Rest der Spezifikation.
- Von der Länge aller Ausgabeströme einschließlich der speziellen Verarbeitungsströme sowie der in der Spezifikation intern verwendeten Ströme zur Repräsentierung der Kontroll- und Variablenzustandshistorien wird unter Anwendung des Focus Zeitabstraktionsoperators gefordert, dass diese gleich dem als Konstante gegebenen maximal betrachteten Zeitpunkt sind.
- Die die einzelnen Zeitpunkte repräsentierende Variable t wird *all* quantifiziert über die Menge der natürlichen Zahlen im Intervall von eingeschlossen eins bis zu dem gegebenen maximal betrachteten Zeitpunkt. Der maximal betrachtete Zeitpunkt ist in diesem Intervall ausgeschlossen, da Eingaben zum Zeitpunkt t entsprechend der herkömmlichen AutoFocus Kommunikationssemantik Ausgaben zum Zeitpunkt $t+1$ bewirken.

Zuweisungen der lokalen Variablen entsprechend der Transition getätigt werden. Werden Ausgaben auf AutoFocus *Immediate* Ports getätigt, dann müssen diese Ausgaben nicht zum Zeitpunkt $t + 1$, sondern bereits zum Zeitpunkt t ausgeführt werden. Die nichtdeterministische Auswahl von Transitionen ist in der deskriptiven Focus Spezifikation enthalten, da in diesem Fall zu einem bestimmten Zeitpunkt t die Bedingungen an Eingaben und Variablenbelegungen mehrerer Transitionen und somit verschiedene alternative Abläufe erfüllt sind.

Nachfolgend wird das Symbol $\dashrightarrow$ verwendet, um Ersetzungsregeln für die Übersetzung zu definieren. Der Term links beziehungsweise oberhalb des Symbols $\dashrightarrow$ wird durch die Übersetzung in den Term rechts beziehungsweise unterhalb dieses Symbols abgebildet.

Sei R die Menge der in einem AutoFocus STD enthaltenen Transitionen. In der Focus Spezifikation wird zur Darstellung der Transitionen der Zustandsmaschine folgender Ausdruck an den *Body* angehängt:

$$\begin{aligned}
\bigvee_{r\in R} \quad & (\overline{stateHistory}.t = \textit{<Name des Quellzustands der Transition r>} \wedge \\
& \textit{<Variablenbedingung der Transition r>} \wedge \\
& \textit{<Eingabebedingung der Transition r>} \wedge \\
& \textit{<Ausgabeterm der Transition r>} \wedge \\
& \textit{<Verarbeitung der Eingaben durch Transition r>} \wedge \\
& \textit{<Variablenzuweisungsterm der Transition r>} \wedge \\
& \overline{stateHistory}.t + 1 = \textit{<Name des Zielzustands der Transition r>}) \qquad (3.7)
\end{aligned}$$

Wie eingangs in diesem Kapitel bereits angeführt, wird in der Definition der Übersetzung der Sprache AutoFocus in die Sprache Focus darauf verzichtet, auch eine Übersetzung von der von AutoFocus verwendeten funktionalen Sprache Quest/F in die Sprache Focus anzugeben, da sich die hier vorgestellte Arbeit auf die graphische Modellierungssprache beschränkt und Quest/F Ausdrücke nur am Rand betrachtet werden. Sind in den nachfolgend aufgeführten Transitionstermen Quest/F Ausdrücke enthalten, dann müssen diese in verhaltensäquivalente Focus Terme übersetzt werden.

Die Variablenbedingungen der Transitionen werden von AutoFocus übernommen, wobei jeder auftretende AutoFocus Variablenname durch folgenden Ausdruck ersetzt wird:

$$< \textit{AutoFocusVariablenname} > \dashrightarrow \overline{\textit{<AutoFocus Variablenname>}}.t \qquad (3.8)$$

Der Ausdruck der Eingabebedingung erfordert in der Übersetzung nach Focus wei-

tergehende Änderungen. In Abschnitt 3.1 wurden statt der Bezeichner der AutoFocus Eingabe-Ports für Focus Eingabekanäle die Bezeichner des mit dem Eingabe-Port verbundenen Ausgabe-Port-Namen der sendenden Komponente gewählt. Diese Änderung der Bezeichner ist nötig, um die Komposition einzelner Komponentenspezifikationen in Focus über gleichbenannte Focus Kanalnamen zu ermöglichen. Entsprechend werden in den Eingabebedingungen alle AutoFocus Port Bezeichner durch die äquivalenten Focus Eingabekanalbezeichner ersetzt, der Wert zum Zeitpunkt t innerhalb des Stroms referenziert und das AutoFocus Pattern Matching Symbol ? durch ein = ersetzt:

$$\begin{gathered} \textit{<AutoFocus Eingabe-Port-Bezeichner>}? \dashrightarrow \\ \overline{\textit{<Äquivalenter Focus Eingabekanalbezeichner>}}.t = \end{gathered} \tag{3.9}$$

Als Focus Ausgabekanalnamen werden einfach die entsprechenden AutoFocus Port-Namen der betrachteten Komponente verwendet. Handelt es sich bei dem betrachteten AutoFocus Port um einen normalen Port mit verzögerter Ausgabe, dann muss in Focus für die Zuweisung der Ausgabe der Zeitpunkt $t+1$ referenziert werden. Zur Darstellung von *Immediate* Ports wird statt dessen der Zeitpunkt t in der Formel referenziert. In den Ausdrücken der Transitionsausgabe werden folgende Ersetzungen bei Referenzierung herkömmlicher AutoFocus Ports vorgenommen:

$$\begin{gathered} \textit{<AutoFocus Ausgabe-Port-Bezeichner>}! \dashrightarrow \\ \overline{\textit{<Äquivalenter Focus Ausgabekanalbezeichner>}}.t+1 = \end{gathered} \tag{3.10}$$

Sind *Immediate* Ports im Ausgabeterm referenziert, dann werden diese Referenzierungen wie folgt übersetzt:

$$\begin{gathered} \textit{<AutoFocus Ausgabe-Port-Bezeichner>}! \dashrightarrow \\ \overline{\textit{<Äquivalenter Focus Ausgabekanalbezeichner>}}.t = \end{gathered} \tag{3.11}$$

Nach der AutoFocus Semantik ist zum Zeitpunkt t ein Port mit keinem Wert belegt, falls zu diesem Zeitpunkt keine Ausgabe auf den Port erfolgt. Dieses Nichtsenden von Nachrichten wird in der Repräsentation in Focus explizit dargestellt. In Abschnitt 3.1 sind die Datentypen der Focus Kanäle um ein *void* Element erweitert worden. Für jeden in einem Ausgabeausdruck nicht enthaltenen Ausgabekanal wird der Ausdruck um das Senden der *void* Nachricht erweitert. Stellt der betrachtete Ausgabekanal einen herkömmlichen AutoFocus Port dar, dann wird folgendes eingefügt:

$$\wedge\ \overline{\textit{<Focus Ausgabekanalbezeichner>}}.t+1 = \textit{void} \tag{3.12}$$

Repräsentiert der Kanal hingegen einen *Immediate* Port, dann wird der Ausgabeausdruck um folgende Bedingung erweitert:

$$\wedge\ \overline{\textit{<Focus Ausgabekanalbezeichner>}}.t = \textit{void} \tag{3.13}$$

Die Focus Komponentenspezifikation besitzt spezielle Ausgabekanäle, die zur Signalisierung der Verarbeitung von Eingaben dienen. Diese speziellen Verarbeitungskanäle verwenden die *Immediate* Kommunikationssemantik. Für jeden in dem Eingabeterm der Transition enthaltenen AutoFocus Port-Bezeichner wird der Focus Body um folgenden Ausdruck erweitert:[4]

$$\wedge\ \overline{\textit{<Äquivalenter Focus Eingabekanalbezeichner>_V}}.t = \textit{Present} \tag{3.14}$$

Für alle in dem Eingabeterm der Transition nicht enthaltenen AutoFocus Port-Bezeichner der Komponente wird folgender Ausdruck in Focus verwendet:

$$\wedge\ \overline{\textit{<Äquivalenter Focus Eingabekanalbezeichner>_V}}.t = \textit{void} \tag{3.15}$$

In den Variablenzuweisungsausdrücken wird für jede vorkommende Wertzuweisung in Focus der Zeitpunkt t+1 referenziert:

$$\textit{<Variablenname>} = \textit{<Wert>} \dashrightarrow \overline{\textit{<Variablenname>}}.t + 1 = \textit{<Wert>} \tag{3.16}$$

In dem Variablenzuweisungsausdruck einer AutoFocus Transition werden die einzelnen Zuweisungen durch das Symbol ; getrennt. In Focus wird diese parallele Zuweisung durch ein logisches Und ausgedrückt:

$$;\ \dashrightarrow \wedge \tag{3.17}$$

[4] Sind in einem Eingabeterm die geforderten Belegungen von zwei referenzierten Eingabekanälen durch eine Disjunktion verknüpft, dann wird nach der hier verwendeten Definition des Verarbeitungsbegriffs festgelegt, dass von beiden in dem Term vorkommenden Eingabekanälen jeweils eine Eingabe verarbeitet wird, obwohl tatsächlich nur eine der verarbeiteten Eingaben die Erfüllung der Eingabebedingung und somit das Feuern der Transition bewirkt. Der Begriff der Verarbeitung von Eingaben lässt sich auch restriktiver festlegen, so dass nur Eingaben, die tatsächlich eine Wirkung auf das Feuern der Transition haben, als verarbeitet angesehen werden. Auf Grund der Komplexität der Entscheidung, ob eine Eingabe verarbeitet wurde oder nicht, die eine dynamische Analyse zur Laufzeit erfordert, wird die restriktive Variante des Begriffs der Verarbeitung in dieser Arbeit nicht verwendet.

Für alle Variablen der AutoFocus Komponente, die nicht in der Variablenzuweisung enthalten sind, muss in Focus explizit spezifiziert werden, dass die Werte dieser Variablen erhalten bleiben. Hierzu wird für alle diese Variablen Folgendes an den Variablenzuweisungsterm der Transition angefügt:

$$\wedge\ \overline{<Variablenname>}.t+1 = \overline{<Variablenname>}.t \tag{3.18}$$

***Idle* Transitionen.** Nach der AutoFocus Semantik wird eine implizite *Idle* Transition gefeuert, falls zu einem bestimmten Zeitpunkt keine andere Transition schaltbereit ist. Bei der Ausführung einer *Idle* Transition wird keine Ausgabe getätigt und die Werte der lokalen Variablen sowie der aktuelle Kontrollzustand bleiben unverändert. Diese *Idle* Transitionen müssen zur Übersetzung in Focus explizit formuliert werden.

Sei Z die Menge aller in einem AutoFocus STD vorhandenen Zustände. Der *Body* der Focus Spezifikation wird um folgenden Ausdruck erweitert:

$$\begin{aligned}
&\vee \\
\bigvee_{z\in Z} (\quad &\overline{stateHistory}.t = <\textit{Name des Zustands z}> \wedge \\
&<\textit{Variablen- und Eingabebedingung der Idle Transition}> \wedge \\
&<\textit{Ausgabe der Idle Transition}> \wedge \\
&<\textit{Variablenzuweisung der Idle Transition}> \wedge \\
&\overline{stateHistory}.t+1 = <\textit{Name des Zustands z}>))
\end{aligned} \tag{3.19}$$

Die Variablen- und Eingabebedingung der *Idle* Transition des Zustands z wird aus der Konjunktion der einzelnen negierten durch ein logisches Und verknüpften Variablen- und Eingabebedingungen aller Transitionen mit Quellzustand z gebildet. Sei R die Menge aller Transitionen mit Quellzustand z. Dann ergibt sich die Variablen- und Eingabebedingung der *Idle* Transition des Zustands z in Focus zu folgendem Ausdruck:

$$\begin{aligned}
\bigwedge_{r\in R} \quad &\neg(<\textit{Variablenbedingung der Transition r}> \wedge \\
&<\textit{Eingabebedingung der Transition r}>)
\end{aligned} \tag{3.20}$$

Die Umwandlung der Bezeichner und Operatoren sowie das Einfügen des Zeitbezugs (Variable t) erfolgt an dieser Stelle analog zur Übersetzung der herkömmlichen AutoFocus Transitionen.

In dem Ausgabeterm der *Idle* Transition wird auf alle Focus Ausgabekanäle einschließlich der speziellen Verarbeitungskanäle der Komponentenspezifikation der Wert *void* geschrieben. Sei O_1 die Menge aller Namen der Ausgabekanäle der Focus Spezifikation, die herkömmliche AutoFocus Ausgabe-Ports repräsentieren und O_2 die Menge aller Namen der Ausgabekanäle der Focus Spezifikation, die AutoFocus *Immediate* Ausgabe-Ports repräsentieren. Dann ergibt sich der Ausgabeterm der *Idle* Transition in der Focus Spezifikation zu:

$$\left(\bigwedge_{a \in O_1} \overline{a}.t+1 = void\right) \wedge \left(\bigwedge_{b \in O_2} \overline{b}.t = void\right) \tag{3.21}$$

Die Variablenzuweisung der *Idle* Transition erhält die Werte aller lokalen Variablen V (ausgenommen der Variable *stateHistory*) der Focus Komponentenspezifikation.

$$\bigwedge_{v \in V} \overline{v}.t+1 = \overline{v}.t \tag{3.22}$$

Die Übersetzung von AutoFocus Modellspezifikationen in Focus Spezifikationen ist hiermit abgeschlossen.

Beispiel einer Übersetzung. Betrachten wir als konkretes Beispiel das in Abbildung 3.1 aufgeführte AutoFocus Modell der *PingPong* Komponente. Die Komponente nimmt abwechselnd von den Ports *i*1 und *i*2 Nachrichten entgegen und quittiert diese mit einer Nachricht auf dem Port *o*. Diese AutoFocus Spezifikation wird in die Focus Spezifikation 3.2 auf der nächsten Seite übersetzt. Zur Kennzeichnung von Kommentaren verwenden wir innerhalb der Focus Spezifikation das Symbol %.

Entsprechend der AutoFocus Semantik wird zu jedem Zeitpunkt in einer Zustandsmaschine genau eine Transition gefeuert. Ist keine der spezifizierten Transitionen der Zustandsmaschine schaltbereit, dann wird eine implizite *Idle* Transition ausgeführt. Diese

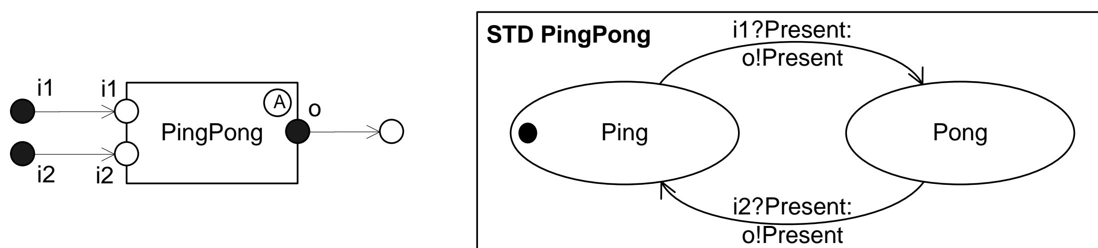


Abbildung 3.1.: Beispiel einer AutoFocus Modellspezifikation.

```
type PingPong_States = {Ping, Pong}
type Signal = {Present}
type VoidDT = {void}
```

PingPong — timed

in $i1 : Signal \mid VoidDT;\ i2 : Signal \mid VoidDT$

out $o : Signal \mid VoidDT;$
$i1_V : Signal \mid VoidDT;\ i2_V : Signal \mid VoidDT$

% *Zeitsynchrone Stroeme*
$\mathsf{ts}(i1) \wedge \mathsf{ts}(i2) \Rightarrow \mathsf{ts}(o) \wedge \mathsf{ts}(i1_V) \wedge \mathsf{ts}(i2_V) \wedge$
% *Kontrollzustand*
$\exists\, stateHistory \in PingPong_States^{\underline{\infty}} : \mathsf{ts}(stateHistory) \wedge$
% *Initialisierung*
$\overline{stateHistory}.1 = Ping \wedge \overline{o}.1 = void \wedge$
% *Zeitvariable*
$(\forall\, t \in \mathbb{N}_+ :$
% *Erste Transition*
$(\overline{stateHistory}.t = Ping \wedge \overline{i1}.t = Present \wedge \overline{o}.t+1 = Present \wedge$
$\overline{i1_V}.t = Present \wedge \overline{i2_V}.t = void \wedge \overline{stateHistory}.t+1 = Pong) \vee$
% *Zweite Transition*
$(\overline{stateHistory}.t = Pong \wedge \overline{i2}.t = Present \wedge \overline{o}.t+1 = Present \wedge$
$\overline{i1_V}.t = void \wedge \overline{i2_V}.t = Present \wedge \overline{stateHistory}.t+1 = Ping) \vee$
% *Idle Transition von Zustand Ping*
$(\overline{stateHistory}.t = Ping \wedge \neg(\overline{i1}.t = Present) \wedge \overline{o}.t+1 = void \wedge$
$\overline{i1_V}.t = void \wedge \overline{i2_V}.t = void \wedge \overline{stateHistory}.t+1 = Ping) \vee$
% *Idle Transition von Zustand Pong*
$(\overline{stateHistory}.t = Pong \wedge \neg(\overline{i2}.t = Present) \wedge \overline{o}.t+1 = void \wedge$
$\overline{i1_V}.t = void \wedge \overline{i2_V}.t = void \wedge \overline{stateHistory}.t+1 = Pong))$

Spezifikation 3.2: In Focus übersetztes AutoFocus Beispiel.

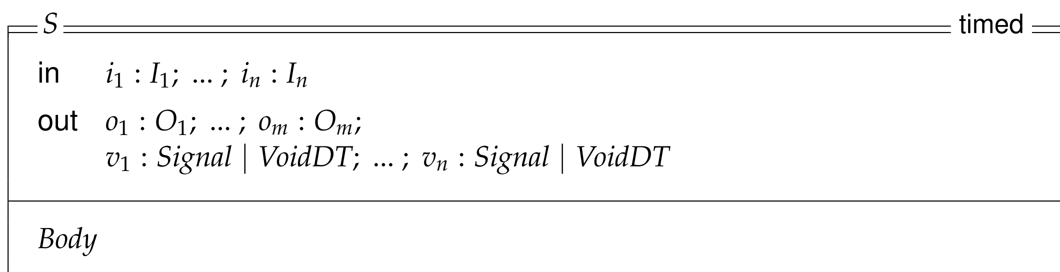
Idle Transition ist in der Übersetzung in Focus explizit dargestellt. Somit ist in der Focus Spezifikation zu jedem Zeitpunkt mindestens eine Transition schaltbereit und der Fall, dass zu einem Zeitpunkt keine Transition ausgeführt wird, kann nicht auftreten.

3.3. Semantik von AutoFocus Spezifikationen

Durch die Übersetzung der Sprache AutoFocus in die Sprache Focus lässt sich die Semantik einer AutoFocus Spezifikation direkt durch die Focus Semantik ausdrücken.

Betrachten wir zunächst eine atomare AutoFocus Komponentenspezifikation S_A, die mit der Übersetzungsabbildung C_{Focus} entsprechend den Abschnitten 3.1 und 3.2 in eine Focus Komponentenspezifikation S übersetzt wurde. Die Semantik der atomaren AutoFocus Komponentenspezifikation S_A lässt sich direkt aus der Focus Semantikdefinition ableiten.

In der Spezifikation 3.3 ist ein allgemein formulierter Focus Spezifikationsrahmen, der sich aus der Übersetzung einer atomaren AutoFocus Komponentenspezifikation ergibt, gegeben. Zur Vereinfachung der Definition der Semantik von AutoFocus Spezifikationen, von Funktionen auf AutoFocus Verhalten und Eigenschaften des AutoFocus Verhaltens nehmen wir im Folgenden an, dass jede von AutoFocus nach Focus übersetze Spezifikation die in diesem Spezifikationsrahmen verwendete Struktur und die dort verwendeten Bezeichner aufweist. Diese Annahme schränkt die Allgemeingültigkeit nur dahingehend ein, dass die Ein- und Ausgabekanäle und deren Datentypen feste Bezeichner besitzen und dadurch deren Referenzierung vereinfacht wird.



Spezifikation 3.3: Spezifikationsrahmen einer von AutoFocus nach Focus übersetzten Spezifikation unter Annahme fester Bezeichner für Kanäle und Datentypen.

Die Eingabekanäle sind in dieser Spezifikation mit i_j bezeichnet und sind von den Datentypen I_j. Die Ausgabekanäle haben die Namen o_j mit den Datentypen O_j und die speziellen Verarbeitungskanäle sind mit v_j bezeichnet.

Definition 3.2 (Syntaktische Schnittstelle einer atomaren AutoFocus Komponentenspezifikation)**.** Die syntaktische Schnittstelle einer atomaren AutoFocus Komponentenspe-

zifikation ist die syntaktische Schnittstelle dieser in Focus übersetzten Spezifikation. Die syntaktische Schnittstelle einer Focus Spezifikation ist in [BS01][S. 80] definiert. Sie umfasst eine Liste von Eingabekanalbezeichnern zusammen mit von diesen Kanälen verwendeten Datentypen sowie eine Liste von Ausgabekanalbezeichnern zusammen mit deren Datentypen. Die Abbildung Syn_{AF} bildet jede AutoFocus Spezifikation aus der Menge der möglichen AutoFocus Spezifikationen $\mathcal{L}_A$ auf ihre sich durch die Übersetzung nach Focus ergebende syntaktische Schnittstelle aus der Menge der in Focus möglichen sytaktischen Schittstellen $\mathcal{L}_{Syn}$ unter Verwendung der Focus Kanalbezeichner und deren Focus Datentypen ab. Die Abbildung ist für atomare AutoFocus Spezifikationen wie folgt definiert:[5]

$$\begin{aligned} Syn_{AF} \in \quad & \mathcal{L}_A \to \mathcal{L}_{Syn} \\ Syn_{AF}(S_A) \overset{\text{def}}{=} \quad & x \text{ so that } C_{Focus}(S_A) \in x \end{aligned} \tag{3.23}$$

Die syntaktische Schnittstelle des in dem vorangehend in Spezifikation 3.3 auf der vorherigen Seite aufgeführten Spezifikationsrahmens ergibt sich zu folgendem Ausdruck:

$$\begin{aligned} &(i_1 : I_1;\ \ldots;\ i_n : I_n \rhd o_1 : O_1;\ \ldots;\ o_m : O_m; \\ &v_1 : (Signal \cup VoidDT)\,;\ \ldots;\ v_n : (Signal \cup VoidDT)) \end{aligned} \tag{3.24}$$

Definition 3.3 (Schnittstellensignatur einer syntaktischen Schnittstelle)**.** Die Schnittstellensignatur einer syntaktischen Schnittstelle ist das Kreuzprodukt der in der syntaktischen Schnittstelle enthaltenen Datentypen. Sei $\underline{M}$ die Menge aller AutoFocus Schnittstellensignaturen. Die Abbildung $Sig \in \mathcal{L}_{Syn} \to \underline{M}$ liefert zu einer gegebenen syntaktischen Schnittstelle deren Schnittstellensignatur.

Die Schnittstellensignatur des in Spezifikation 3.3 auf der vorherigen Seite aufgeführten Spezifikationsrahmens ergibt sich zu folgendem Tupel von Datentypen:

$$M = I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right) \tag{3.25}$$

Definition 3.4 (Bezeichner einer syntaktischen Schnittstelle)**.** Die Abbildung $Bez \in \mathcal{L}_{Syn} \to String \times ... \times String$ liefert zu einer gegebenen syntaktischen Schnittstelle das Tupel der darin verwendeten Variablenbezeichner.

Definition 3.5 (Semantik einer atomaren AutoFocus Komponentenspezifikation)**.** Sei $C_{Focus} : \mathcal{L}_A \to \mathcal{L}_F$ die Übersetzungsabbildung von der Sprache AutoFocus in die Sprache Focus. Die Semantik einer AutoFocus Spezifikation S_A ist definiert über die Menge

[5] Mit der Notation x so that $C_{Focus}(S_A) \in x$ ist ausgedrückt, dass x so belegt wird, dass die Übersetzung der AutoFocus Spezifikation S_A in Focus die syntaktische Schnittstelle x besitzt.

aller gültigen Belegungen der prädikatenlogischen Denotation ihrer übersetzten Focus Spezifikation $[\![\ C_{Focus}(S_A)\ ]\!]$.

Für eine atomare AutoFocus Komponentenspezifikation S_A, die einen Spezifikationsrahmen entsprechend der Spezifikation 3.3 auf Seite 52 besitzt, ergibt sich aus der Definition der Denotation von gezeitetem Focus für einzelne Spezifikationen [BS01, Definition 5.2.1, S. 81] folgende Denotation $[\![\ S_A\ ]\!]_{AF}$ als prädikatenlogische Formel:

$$\begin{aligned} [\![\ S_A\ ]\!]_{AF} \stackrel{\text{def}}{=} \quad & [\![\ C_{Focus}(S_A)\ ]\!] = \\ & i_1 \in I_1^{\underline{\infty}} \wedge \ldots \wedge i_n \in I_n^{\underline{\infty}} \wedge \\ & o_1 \in O_1^{\underline{\infty}} \wedge \ldots \wedge o_m \in O_m^{\underline{\infty}} \wedge \\ & v_1 \in (Signal \cup VoidDT)^{\underline{\infty}} \wedge \ldots \wedge v_n \in (Signal \cup VoidDT)^{\underline{\infty}} \wedge \\ & Body \end{aligned} \tag{3.26}$$

Die Semantik der atomaren AutoFocus Komponentenspezifikation S_A ist durch die Menge aller gültigen Belegungen dieser Formel, die alle durch die Spezifikation möglichen Abläufe von Ein- und Ausgabesequenzen repräsentieren, gegeben.[6]

$$\begin{aligned} & \mathcal{R}_S \subseteq I_1^{\underline{\infty}} \times \ldots \times I_n^{\underline{\infty}} \times O_1^{\underline{\infty}} \times \ldots \times O_m^{\underline{\infty}} \times \left(\times_{j=1}^{n} (Signal \cup VoidDT)^{\underline{\infty}} \right) \\ & (i_1, \ldots, i_n, o_1, \ldots, o_m, v_1, \ldots, v_n) \in \mathcal{R}_S \Leftrightarrow [\![\ S_A\ ]\!]_{AF} \end{aligned} \tag{3.27}$$

Die Semantik einer nicht atomaren AutoFocus Komponentenspezifikation beziehungsweise einer Systemspezifikation lässt sich aus der Semantik der einzelnen atomaren Komponentenspezifikationen und der Semantik des Focus Kompositionsoperators ableiten.

Zunächst wird von einer komponierten Spezifikation ermittelt, welche Ports beziehungsweise Kanäle der internen und welche der externen Kommunikation dienen.

Definition 3.6 (Syntaktische Schnittstelle einer komponierten AutoFocus Spezifikation)**.** Sei $\{S_1, \ldots, S_z\}$ die Menge der atomaren AutoFocus Komponentenspezifikationen, die in der komponierten Spezifikation S_A enthalten sind. Seien ferner $\mathcal{I}_1, \ldots, \mathcal{I}_z$ die Mengen der Eingabekanalvariablennamen der einzelnen in Focus übersetzen Spezifikationen. Mit $\mathcal{O}_1, \ldots, \mathcal{O}_z$ seien die Mengen der Ausgabekanalvariablennamen ohne die

[6] Die Notation $\times_{j=1}^{n}$ ist im Anhang A erläutert.
In [BS01, S. 81, Absatz nach der Definition 5.2.1] wird für die Abläufe, die in der Eingabe- / Ausgaberelation enthalten sind, gefordert, dass diese die logische Formel im Body der Focus Spezifikation erfüllen müssen. In dieser Arbeit wird hingegen gefordert, dass die Denotation der Spezifikation erfüllt sein muss. Beide Forderungen sind äquivalent, da die Denotation einer Spezifikation neben der Formel im Body der Spezifikation lediglich redundant fordert, dass die Ein- / Ausgabeströme der Abläufe Datentyp konform sind.

speziellen Verarbeitungsvariablen der einzelnen übersetzten Spezifikationen bezeichnet und die Mengen $\mathcal{V}_1, \ldots, \mathcal{V}_z$ enthalten die speziellen Verarbeitungskanalvariablennamen der einzelnen Spezifikationen.

Die Menge der Eingabekanalvariablen der komponierten Spezifikation besteht aus allen Eingabekanalvariablen der Teilspezifikationen, die nicht mit Ausgabekanalvariablen anderer Teilspezifikationen verknüpft sind:

$$\mathcal{I} = \bigcup_{j=1}^{z} \mathcal{I}_j \setminus \bigcup_{k=1}^{z} \mathcal{O}_k \tag{3.28}$$

Analog hierzu ist die Menge der Ausgabekanalvariablen der komponierten Spezifikation festgelegt durch die Ausgabekanalvariablen der Teilspezifikationen, die nicht mit Eingabekanalvariablen anderer Teilspezifikationen verknüpft sind:

$$\mathcal{O} = \bigcup_{j=1}^{z} \mathcal{O}_j \setminus \bigcup_{k=1}^{z} \mathcal{I}_k \tag{3.29}$$

Für jede Eingabekanalvariable der komponierten Spezifikation existiert auch nach Durchführung der Komposition eine spezielle Ausgabevariable zum Anzeigen der Verarbeitung der Eingaben. Die vorhandenen Verarbeitungskanäle, die sich nach der Durchführung der Komposition auf interne Kommunikation beziehen, werden hingegen nach außen versteckt. Die nach außen sichtbaren Verarbeitungsvariablen lassen sich aus der Menge der nach außen sichtbaren Eingabekanalvariablen $\mathcal{I}$ durch Ergänzung des Namens um den Präfix $_V$ ableiten:[7]

$$\mathcal{V} = \bigcup_{i \in \mathcal{I}} (i + ''_V'') \tag{3.30}$$

Die syntaktische Schnittstelle der komponierten AutoFocus Spezifikation wird aus den in den Mengen $\mathcal{I}$, $\mathcal{O}$ und $\mathcal{V}$ enthaltenen Bezeichnern und deren Datentypen abgeleitet. Wie bei der Übersetzung von atomaren Komponenten werden die in jeder dieser Mengen enthaltenen Bezeichner isoliert voneinander alphabetisch sortiert, um die korrekte Zuordnung beim Vergleich verschiedener syntaktischer Schnittstellen und verschiedener Schnittstellenverhalten zu ermöglichen.

Definition 3.7 (Komposition syntaktischer Schnittstellen)**.** Mit $\otimes_{Syn} \in \mathcal{L}_{Syn} \times \mathcal{L}_{Syn} \rightarrow \mathcal{L}_{Syn}$ ist die Komposition zweier syntaktischer Schnittstellen zu einer syntaktischen Schnittstelle entsprechend des vorangehend festgelegten Begriffs der syntaktischen Schnittstelle komponierter AutoFocus Spezifikationen bezeichnet.

[7] Mit $i + ''_V''$ ist die *String* Konkatenation des Variablennamens i mit dem Teil-String $_V$ bezeichnet.

Wir gehen im Folgenden davon aus, dass die syntaktische Schnittstelle einer betrachteten komponierten Spezifikation S_A die folgende Form aufweist:

$$\begin{aligned} Syn_{AF}(S_A) = \quad & (i_1 : I_1;\ \ldots;\ i_n : I_n \rhd o_1 : O_1;\ \ldots;\ o_m : O_m; \\ & v_1 : (Signal \cup VoidDT)\,;\ \ldots;\ v_n : (Signal \cup VoidDT)) \end{aligned} \tag{3.31}$$

Definition 3.8 (Interne Kommunikation einer komponierten Spezifikation)**.** Die Menge der Kanalvariablennamen, die der internen Kommunikation zwischen komponierten Teilspezifikationen dienen, wird mit $\mathcal{H}$ bezeichnet. Sie beinhaltet alle Kanalvariablennamen der Teilspezifikationen, die nach außen nicht sichtbar sind. $\mathcal{H}$ ergibt sich aus den Variablenbezeichnern der Teilspezifikationen, die weder in $\mathcal{I}$, in $\mathcal{O}$ noch in $\mathcal{V}$ enthalten sind.

$$\mathcal{H} = \left(\bigcup_{k=1}^{z} (\mathcal{I}_k \cup \mathcal{O}_k \cup \mathcal{V}_k) \right) \setminus (\mathcal{I} \cup \mathcal{O} \cup \mathcal{V}) \tag{3.32}$$

Definition 3.9 (Semantik einer komponierten AutoFocus Spezifikation)**.** Die Semantik einer komponierten AutoFocus Spezifikation S_A ist über die Semantik der darin enthaltenen atomarer AutoFocus Kompontenspezifikation $\{S_1, \ldots, S_z\}$ und die Semantik von komponierten Focus Spezifikationen [BS01, Definition 5.3.1, S. 85] gegeben.

Seien $h_1, \ldots, h_q$ die verschiedenen in $\mathcal{H}$ enthaltenen Kanalvariablen, deren Datentypen mit $H_1, \ldots, H_q$ bezeichnet sind.

Für die Denotation der komponierten AutoFocus Spezifikation $[\![\ S_A\ ]\!]$ ergibt sich folgende prädikatenlogische Formel, wobei die Variablen, die interne Kommunikationskanäle repräsentieren, durch Existenzquantoren gebunden werden und somit nach außen nicht sichtbar sind: [8]

$$[\![\ S_A\ ]\!]_{AF} \overset{\text{def}}{=} \exists h_1 \in H_1^{\infty}, \ldots, h_q \in H_q^{\infty} : \wedge_{j=1}^{z} [\![\ S_j\ ]\!]_{AF} \tag{3.33}$$

Durch die Menge aller gültigen Belegungen der freien Variablen dieser Formel ist die Semantik der komponierten AutoFocus Gesamtspezifikation S_A festgelegt.

3.4. Das AutoFocus Schnittstellenverhalten in Focus Strömen

Die Semantikdefinition von AutoFocus Modellspezifikationen auf Basis von Focus wird nun verwendet, um einen formalen Begriff des Schnittstellenverhaltens von Au-

[8] Zur Erläuterung der Notation $\wedge_{j=1}^{z}$ siehe Anhang A.

toFocus Modellen zu definieren, der als Beobachtungsbegriff von Modell-Refactorings dient. Das Verhalten eines AutoFocus Modells umfasst sowohl das nach außen sichtbare Ein- / Ausgabeverhalten, interne Kommunikation, als auch den Wechsel von internen Variablen- und Kontrollzuständen des Systems. Das Schnittstellenverhalten einer AutoFocus Spezifikation ist die Menge aller auf der Ebene der Ein- und Ausgabe-Ports der äußeren Schnittstelle der Spezifikation sichtbaren möglichen Abläufe von Ein- und Ausgaben. Von interner Kommunikation, internen Variablen und Kontrollzuständen wird hierbei abstrahiert.

Definition 3.10 (Datentyp zeitsynchroner Ströme)**.** Zur Definition des AutoFocus Schnittstellenverhaltens werden zeitsynchrone Focus Ströme verwendet [BS01, S. 66, Definition von ts]. Wir definieren die Menge aller entsprechend eines gegebenen Datentyps D möglichen zeitsynchronen unendlichen Ströme $D^{\underline{\infty},\mathsf{ts}}$ wie folgt: [9]

$$D^{\underline{\infty},\mathsf{ts}} \stackrel{\mathsf{def}}{=} \{x \in D^{\underline{\infty}} \text{ so that } \mathsf{ts}(x)\} \tag{3.34}$$

Definition 3.11 (Schnittstellenverhalten einer AutoFocus Spezifikation)**.** Das Schnittstellenverhalten $SV_{AF}(S_A)$ einer AutoFocus Spezifikation S_A ist direkt durch die Semantik von atomaren und komponierten AutoFocus Spezifikationen (Definition 3.5 auf Seite 53 und Definition 3.9 auf der vorherigen Seite) in Form der dort verwendeten Ein-/Ausgabe Relation $\mathcal{R}_S$ gegeben. Die Relation ist definiert als die Menge aller Belegungen der freien Variablen, die die Denotation der in Focus übersetzen AutoFocus Spezifikation $[\![\; S_A \;]\!]_{AF}$ erfüllen. Die Relation enthält alle nach außen sichtbaren Ein-/Ausgabesequenzen in Form von Tupel von zeitsynchronen Focus Strömen, die entsprechend der Spezifikation möglich sind. Sei M die Schnittstellensignatur der AutoFocus Spezifikation, $M^{\underline{\infty},\mathsf{ts}}$ die Menge aller entsprechend der festen Signatur M möglichen AutoFocus Abläufe, deren in den Tupel enthaltenen Ströme alle zeitsynchron sind, und $\mathcal{M}$ die Menge aller entsprechend der gewählten Signatur möglichen Ein-/Ausgabeverhalten. Das Schnittstellenverhalten SV ist wie folgt definiert:

$$M = \quad I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right) \tag{3.35}$$

$$M^{\underline{\infty},\mathsf{ts}} = \quad I_1^{\underline{\infty},\mathsf{ts}} \times \ldots \times I_n^{\underline{\infty},\mathsf{ts}} \times O_1^{\underline{\infty},\mathsf{ts}} \times \ldots \times O_m^{\underline{\infty},\mathsf{ts}} \times \left(\times_{j=1}^{n} (Signal \cup VoidDT)^{\underline{\infty},\mathsf{ts}} \right) \tag{3.36}$$

$$\mathcal{M} = \quad \mathcal{P}(M^{\underline{\infty},\mathsf{ts}}) \tag{3.37}$$

$$SV_{AF} \in \mathcal{L}_A \rightarrow \mathcal{M}$$

[9] Mit $\{x \in X \text{ so that } e(x)\}$ ist die Menge aller Elemente $x \in X$, die die Eigenschaft $e(x)$ erfüllen, bezeichnet.

$$SV_{AF}(S_A) \stackrel{\text{def}}{=} \mathcal{R}_S \in \mathcal{M} \text{ so that}$$
$$(i_1, \ldots, i_n, o_1, \ldots, o_m, v_1, \ldots, v_n) \in \mathcal{R}_S \Leftrightarrow [\![\ S_A\ ]\!]_{AF} \quad (3.38)$$

Die Abbildung 3.2 auf der nächsten Seite zeigt einen Ausschnitt von zwei möglichen Abläufen des *PingPong* AutoFocus Modells (Abbildung 3.1 auf Seite 50) unter Berücksichtigung der speziellen Verarbeitungsnachrichten als Sequenzdiagramme.

Diese beiden Abläufe sind als zwei Tupel von Focus Strömen in der Spezifikation 3.4 dargestellt, wobei dort mit $\mathcal{R}_T$ ein Teil der Ein-/Ausgaberelation des Gesamtverhaltens der *PingPong* Modellspezifikation unter Beschränkung der Länge der betrachteten Abläufe bezeichnet ist.

$$Syn_{AF}(PingPong) = (i1 : Signal \cup VoidDT;\ i2 : Signal \cup VoidDT \triangleright$$
$$o : Signal \cup VoidDT;\ i1_V : Signal \cup VoidDT;\ i2_V : Signal \cup VoidDT)$$
$$\mathcal{R}_T = \{($$
$$\langle Present, \checkmark, void, \checkmark, void, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, void, \checkmark, Present, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, Present, \checkmark, void, \checkmark, Present, \checkmark \rangle,$$
$$\langle Present, \checkmark, void, \checkmark, void, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, void, \checkmark, Present, \checkmark, void, \checkmark \rangle$$
$$), ($$
$$\langle Present, \checkmark, void, \checkmark, Present, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, void, \checkmark, void, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, Present, \checkmark, void, \checkmark, void, \checkmark \rangle,$$
$$\langle Present, \checkmark, void, \checkmark, void, \checkmark, void, \checkmark \rangle,$$
$$\langle void, \checkmark, void, \checkmark, void, \checkmark, void, \checkmark \rangle$$
$$)\}$$

Spezifikation 3.4: Darstellung von zwei Beispielabläufen des *PingPong* AutoFocus Modells in zwei Tupel von zeitsynchronen Focus Strömen.

Die Menge aller möglichen Abläufe, das heißt Tupel von unendlich langen Strömen, die die übersetzte Focus Spezifikation der *PingPong* Komponente erfüllen, beschreiben das Schnittstellenverhalten der AutoFocus *PingPong* Komponente.

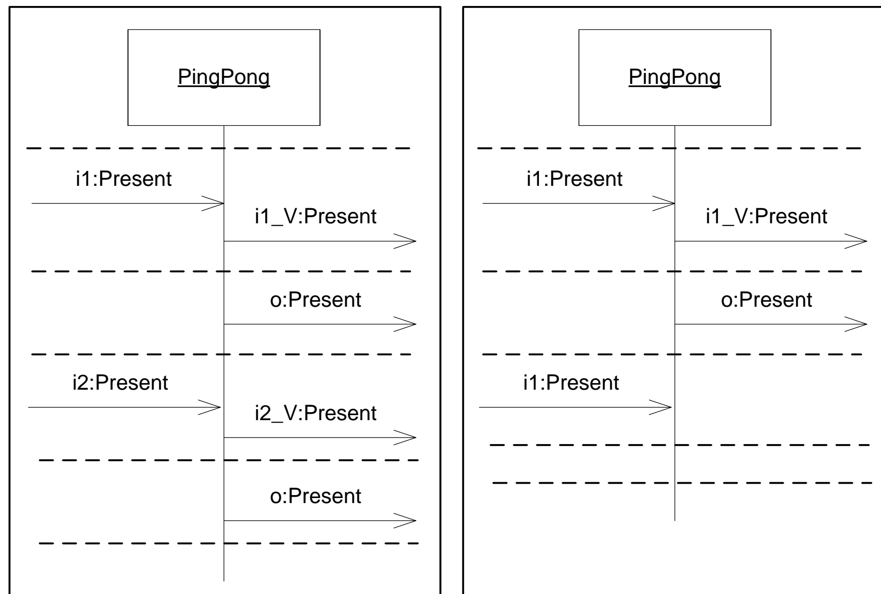


Abbildung 3.2.: AutoFocus Sequenzdiagramme zur *PingPong* Komponente.

Definition 3.12 (Verallgemeinerung von Abbildungen auf allgemeine Schnittstellensignaturen und allgemeines Verhalten)**.** Sei A eine Abbildung, die konkretes Schnittstellenverhalten aus der Menge $\mathcal{M}$ beziehungsweise abstraktes Schnittstellenverhalten aus der Menge $\mathcal{N}$ entsprechend der vorgegebenen Schnittstellensignatur M beziehungsweise N in ihrem Urbild oder beziehungsweise und ihrem Bild betrachtet. Wir legen fest, dass zu solch einer Abbildung die Abbildung $\underline{A}$ die Verallgemeinerung auf Verhalten von Spezifikationen, die alle in AutoFocus möglichen Schnittstellensignaturen umfasst, darstellt. Die Bild- und Urbildmengen der Abbildung werden von $\mathcal{M}$ auf $\underline{\mathcal{M}} \supset \mathcal{M}$ beziehungsweise von $\mathcal{N}$ auf $\underline{\mathcal{N}} \supset \mathcal{N}$ erweitert.

3.5. Äquivalenz von Schnittstellenverhalten

Für die vorangehend definierten Begriffe der syntaktischen Schnittstelle und des Schnittstellenverhaltens einer AutoFocus Spezifikation wird nun ein Verhaltensäquivalenzbegriff festgelegt. Zu diesem Zweck wird zunächst die stromverarbeitende Funktion *loescheVS* in Focus definiert, die es ermöglicht, die speziellen Verarbeitungsströme, die bei einem direkten Vergleich von Verhalten nicht berücksichtigt werden sollen, zu eleminieren. Die Abbildung bildet Abläufe, die spezielle Verarbeitungsströme enthalten, auf Abläufe ohne diese Ströme ab.

$$M = \quad I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right)$$

$$
\begin{aligned}
N = \quad & I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \\
N^{\underline{\omega},\mathsf{ts}} = \quad & I_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times I_n^{\underline{\omega},\mathsf{ts}} \times O_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times O_m^{\underline{\omega},\mathsf{ts}}
\end{aligned} \tag{3.39}
$$

$$
\begin{aligned}
& loescheVS \in M^{\underline{\omega},\mathsf{ts}} \rightarrow N^{\underline{\omega},\mathsf{ts}} \\
& \text{where } loescheVS \text{ so that} \\
& \forall\, i_1 \in I_1^{\underline{\omega},\mathsf{ts}}, \ldots, i_n \in I_n^{\underline{\omega},\mathsf{ts}}, o_1 \in O_1^{\underline{\omega},\mathsf{ts}}, \ldots o_m \in O_m^{\underline{\omega},\mathsf{ts}}, \\
& v_1 \in (Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}}, \ldots, v_n \in (Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}} : \\
& loescheVS(i_1, \ldots, i_n, o_1, \ldots, o_m, v_1, \ldots, v_n) = (i_1, \ldots, i_n, o_1, \ldots, o_m)
\end{aligned} \tag{3.40}
$$

Definition 3.13 (Erweiterung von Abbildungen von Schnittstellenabläufen auf Schnittstellenverhalten)**.** Wir legen allgemein fest, dass für jede Abbildung A, die Schnittstellenabläufe ineinander überführt, die Abbildung $\widetilde{A}$ die Abbildung bezeichnet, die die ursprüngliche Abbildung auf alle in einem Schnittstellenverhalten enthaltenen Abläufe anwendet und als Ergebnis wiederum ein Schnittstellenverhalten liefert. Als Urbildmenge $\mathcal{X}$ und Bildmenge $\mathcal{Y}$ können hierbei die Menge der konkreten Schnittstellenverhalten $\mathcal{M}$ und die Menge der abstrakten Schnittstellenverhalten $\mathcal{N}$ eingesetzt werden.[10]

$$
\begin{aligned}
& \widetilde{A} \in \mathcal{X} \rightarrow \mathcal{Y} \\
& \text{where } \widetilde{A} \text{ so that } \forall\, \mathcal{R}_S \in \mathcal{X} : \\
& \widetilde{A}(\mathcal{R}_S) = \bigcup_{r \in \mathcal{R}_S} \{A(r)\}
\end{aligned} \tag{3.41}
$$

Definition 3.14 (Äquivalenz von AutoFocus Schnittstellenverhalten)**.** Zwei Schnittstellenverhalten zweier AutoFocus Modellspezifikationen S_1 und S_2 sind äquivalent, wenn die Ein- / Ausgaberelationen beider in Focus übersetzten Spezifikationen unter Weglassung der speziellen Verarbeitungsströme identisch sind und beide AutoFocus Modellspezifikationen die gleiche syntaktische Schnittstelle aufweisen. Durch die Verwendung der Definition von Schnittstellenverhalten (Definition 3.11 auf Seite 57) und der Definition der syntaktischen Schnittstelle (Definition 3.2 auf Seite 52 und Definition 3.6 auf Seite 54) ergibt sich die Äquivalenzeigenschaft wie folgt:

$$
verhaltens\ddot{a}quivalentAF(S_1, S_2) \overset{\mathsf{def}}{=}
$$

[10] Mit $\bigcup_{r \in \mathcal{R}_S} \{A(r)\}$ wird die Vereinigung aller aus der Abbildung der in $\mathcal{R}_S$ enthaltenen Abläufen resultierenden Abläufe bezeichnet.

$$(Syn_{AF}(S_1) = Syn_{AF}(S_2)) \wedge$$
$$(\widetilde{\underline{loescheVS}}(\underline{SV}_{AF}(S_1)) = \widetilde{\underline{loescheVS}}(\underline{SV}_{AF}(S_2))) \quad (3.42)$$

Durch die Forderung der Gleichheit der syntaktischen Schnittstellen wird hier für die Verhaltensäquivalenz auch die Gleichheit der Focus Kanalbezeichner gefordert. Auf Grund der Struktur der Übersetzung von AutoFocus in Focus, die jeweils als Focus Eingabekanalbezeichner den Bezeichner des über AutoFocus Kanäle mit dem AutoFocus Eingabe-Port verbundenen AutoFocus Ausgabe-Ports verwendet, wird somit die Gleichheit der Ausgabe-Port Bezeichner von atomaren AutoFocus Komponenten gefordert.[11] Hierdurch lässt sich unter der Einschränkung, dass AutoFocus Port-Bezeichner zur Festlegung der Kommunikationsbeziehungen dienen und somit nicht verändert werden dürfen, durch die Forderung der Äquivalenz der syntaktischen Schnittstellen, sicherstellen, dass zwei betrachtete Teilspezifikationen unter Verwendung von AutoFocus Kanälen äquivalent mit einer Restspezifikation verschaltet sind.[12]

Unter Verwendung des Focus Verhaltensverfeinerungsbegriffs [BS01, Kapitel 15, S.253ff] ist das Schnittstellenverhalten äquivalent, falls unter Verstecken der speziellen Verarbeitungsströme[13] $C_{Focus}(S_2)$ eine Verhaltensverfeinerung von $C_{Focus}(S_1)$ und $C_{Focus}(S_1)$ eine Verhaltensverfeinerung von $C_{Focus}(S_2)$ darstellen:[14]

$$\exists\, v_1, \ldots, v_n, v'_1, \ldots, v'_{n'} \in (Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}} :$$
$$C_{Focus}(S_1) \rightsquigarrow C_{Focus}(S_2) \wedge C_{Focus}(S_2) \rightsquigarrow C_{Focus}(S_1) \Leftrightarrow$$
$$((\llbracket S_2 \rrbracket_{AF} \Rightarrow \llbracket S_1 \rrbracket_{AF}) \wedge (\llbracket S_1 \rrbracket_{AF} \Rightarrow \llbracket S_2 \rrbracket_{AF})) \quad (3.43)$$

[11] In Bezug auf Ports der Modellumgebung müssen die Eingabe-Port Bezeichner gleich sein.

[12] Bei der isolierten Betrachtung von AutoFocus Komponentenspezifikationen kann auch ein Verhaltensäquivalenzbegriff sinnvoll sein, der nicht die Gleichheit der Port-Bezeichner fordert. Grundsätzlich besitzt eine AutoFocus Komponente eine Menge von Eingabe- und eine Menge von Ausgabe-Ports, wobei die einzelnen Ports keine Ordnung zueinander besitzen und es in der Verantwortung des Entwicklers liegt, diese Ports korrekt mit der umgebenden Spezifikation durch Kanäle zu verknüpfen. Ein Verhaltensäquivalenzbegriff, der aussagt, dass eine korrekte Verschaltung der Ports existiert, so dass beide Spezifikationen sich gleich verhalten, kann daher sinnvoll sein. In dieser Arbeit betrachten wir jedoch Komponentenverhalten nicht isoliert, sondern immer in ein Restverhalten eingebettet. Deshalb benötigen wir den Bezug zum Restsystem und fordern daher die Äquivalenz der syntaktischen Schnittstellen von Teilsystemen.

[13] Seien mit $v_1, \ldots, v_n$ die Variablen bezeichnet, die die Verarbeitungsströme der Spezifikation S_1 repräsentieren. Die Variablen $v'_1, \ldots, v'_{n'}$ repräsentieren die Verarbeitungsströme der Spezifikation S_2.

[14] In den Denotationen der Spezifikationen $\llbracket S_1 \rrbracket_{AF}$ und $\llbracket S_2 \rrbracket_{AF}$ sind die Verarbeitungskanäle zunächst als freie Variablen vorhanden. Es wird dort gefordert, dass Belegungen dieser freien Variablen vom korrekten Datentyp sein müssen. Durch das Verstecken der Verarbeitungskanäle mit Hilfe des Existenz Quantors wird redundant gefordert, dass die Belegung der Variablen entsprechend den gewählten Datentypen erfolgen muss. Aus Gründen der einfacheren Schreibweise wird diese Redundanz beibehalten.

3.6. Komposition konkreten Schnittstellenverhaltens

In den vorangehenden Abschnitten wurde ein Begriff von Schnittstellenverhalten von AutoFocus Spezifikationen festgelegt. Durch diesen Begriff lässt sich sowohl das Schnittstellenverhalten einer Gesamtspezifikation als auch das Schnittstellenverhalten einzelner Komponenten betrachten.

Um von den Schnittstellenverhalten der einzelnen Komponenten auf ein Gesamtverhalten schließen zu können, wird ein Kompositionsoperator benötigt. Sowohl in AutoFocus als auch in Focus existieren auf der Ebene der Spezifikationssprache Kompositionsoperatoren. Im Folgenden wird eine Definition von Komposition konkreten Schnittstellenverhaltens auf Basis des Focus Kompositionsoperators angegeben.

Definition 3.15 (Komposition von konkretem AutoFocus Schnittstellenverhalten)**.** Seien zwei konkrete AutoFocus Schnittstellenverhalten (Ein-/Ausgaberelationen) $\mathcal{R}_{S_1}$ und $\mathcal{R}_{S_2}$ aus der Menge der möglichen AutoFocus Schnittstellenverhalten $\underline{\mathcal{M}}$ einschließlich deren syntaktische Schnittstellen Syn_1 und Syn_2 gegeben. Das gesuchte Gesamtschnittstellenverhalten ist mit $\mathcal{R}_S$ bezeichnet. Die Komposition von konkretem Schnittstellenverhalten $\otimes_K$ ist festgelegt durch die Komposition mit Rückkopplung [BS01, S.86f] zweier Focus Spezifikationen S_1 und S_2, die die Schnittstellenverhalten $\mathcal{R}_{S_1}$ und $\mathcal{R}_{S_2}$ aufweisen und dabei den syntaktischen Schnittstellen dieser Schnittstellenverhalten Syn_1 und Syn_2 entsprechen:

$$
\begin{aligned}
&\otimes_K \in (\underline{\mathcal{M}} \times \mathcal{L}_{Syn}) \times (\underline{\mathcal{M}} \times \mathcal{L}_{Syn}) \to \underline{\mathcal{M}} \\
&(\mathcal{R}_{S_1}, Syn_1) \otimes_K (\mathcal{R}_{S_2}, Syn_2) \stackrel{\text{def}}{=} \mathcal{R}_S \text{ so that } \exists\, S_1 \in \mathcal{L}_F, S_2 \in \mathcal{L}_F : \\
&(Bez(Syn_1) \in \mathcal{R}_{S_1} \Leftrightarrow [\![\, S_1 \,]\!]) \wedge \\
&(S_1 \in Syn_1) \wedge \\
&(Bez(Syn_2) \in \mathcal{R}_{S_2} \Leftrightarrow [\![\, S_2 \,]\!]) \wedge \\
&(S_2 \in Syn_2) \wedge \\
&(Bez(Syn_1 \otimes_{Syn} Syn_2) \in \mathcal{R}_S \Leftrightarrow [\![\, S_1 \otimes S_2 \,]\!])
\end{aligned}
\tag{3.44}
$$

3.7. AutoFocus Zeitabstraktionen

Die informelle Definition von Refactoring fordert die Verhaltensäquivalenz des beobachtbaren Systemverhaltens vor und nach Anwendung der Refactoring-Operation. In den folgenden Abschnitten werden verschiedene Zeitabstraktionen auf AutoFocus Schnittstellenverhalten definiert. Mit diesen Zeitabstraktionen kann der Begriff des beobachtbaren Systemverhaltens formal definiert werden. Die aufgeführten Abstraktionen bilden konkretes AutoFocus Schnittstellenverhalten, das in Form von Focus Ein-

/ Ausgaberelationen dargestellt ist, auf ein abstraktes Schnittstellenverhalten in Form von abstrakten Focus Ein-/Ausgaberelationen ab.

In Abschnitt 3.7.1 wird zunächst eine Notation für zeitabstrakte AutoFocus Sequenzdiagramme definiert, mit deren Hilfe abstrakte Abläufe kompakt darstellbar sind. In den darauf folgenden Abschnitten werden verschiedene Abstraktionen von AutoFocus Schnittstellenverhalten formal durch stromverarbeitende Funktionen in der Spezifikationssprache Focus definiert. Der Abschnitt 3.7.2 definiert eine Abstraktion von Eingaben, die von der Spezifikation nicht verarbeitet, sondern ignoriert werden. Anschließend wird im Abschnitt 3.7.3 eine Zeitabstraktion von AutoFocus Zeitschritten, in denen weder Ein- noch Ausgaben auftreten, festgelegt. Die in Abschnitt 3.7.4 definierte Zeitabstraktion abstrahiert von Reihenfolgen zwischen Eingaben und Reihenfolgen zwischen Ausgaben auf unterschiedlichen Ports, wobei die Reihenfolgen zwischen Ein- und Ausgaben erhalten bleiben. In Abschnitt 3.7.5 wird schließlich der Focus Zeitabstraktionsoperator auf AutoFocus Schnittstellenverhalten angewendet. Es wird eine Zeitabstraktion definiert, die von jeglichen zeitlichen Beziehungen zwischen Nachrichten auf verschiedenen Ports abstrahiert. Lediglich die Reihenfolge von Nachrichten, die auf einem Port auftreten, bleibt erhalten. Die gezeigten Zeitabstraktionen lassen sich nicht nur für AutoFocus Schnittstellenverhalten einsetzen, sondern können auch auf zeitsynchrone Focus Ein-/Ausgaberelationen angewendet werden, falls diese die speziellen Ströme zur Kennzeichnung der Verarbeitung von Eingaben besitzen und *void* Nachrichten zur Kennzeichnung von leeren Nachrichten verwenden. Im Abschnitt 3.7.6 wird zu den Abstraktionsabbildungen eine Zeitkonkretisierungsabbildung definiert und in Abschnitt 3.7.7 wird ein Verhaltensäquivalenzbegriff abstrakten Verhaltens festgelegt. Schließlich wird im Abschnitt 3.7.8 ein Kompositionsoperator für abstraktes AutoFocus Schnittstellenverhalten definiert.

3.7.1. Sequenzdiagramme zur Darstellung von zeitabstraken Abläufen

Zur besseren Darstellbarkeit von zeitabstrahierten Abläufen von Modellspezifikationen führen wir in diesem Abschnitt eine spezielle Notation von AutoFocus Sequenzdiagrammen ein, in der AutoFocus Zeittakte nicht explizit dargestellt werden müssen. Diese zeitabstrakten Sequenzdiagramme werden zur Unterscheidung von den herkömmlichen gezeiteten AutoFocus Sequenzdiagrammen mit dem Stereotyp *«time abstract»* gekennzeichnet. Sind in dieser Diagrammart zwei Ereignisse beziehungsweise Nachrichten untereinander angeordnet, dann bedeutet dies in der zeitsynchronen Semantik von AutoFocus, dass zwischen dem Auftreten dieser beiden Ereignisse beziehungsweise Nachrichten ein bis beliebig viele Zeitschritte vergehen. In Abbildung 3.3 auf der nächsten Seite ist diese Interpretation in Form einer Übersetzungsregel in die herkömmlichen zeitsynchronen AutoFocus Sequenzdiagramme dargestellt.

Eine Ausnahme von dieser Regel stellt die Sequenz einer Ausgabe gefolgt von einer Eingabe dar (siehe Abbildung 3.4 auf der nächsten Seite). Wird eine Ausgabe gefolgt von einer Eingabe in einem zeitabstrakten Sequenzdiagramm dargestellt, dann be-

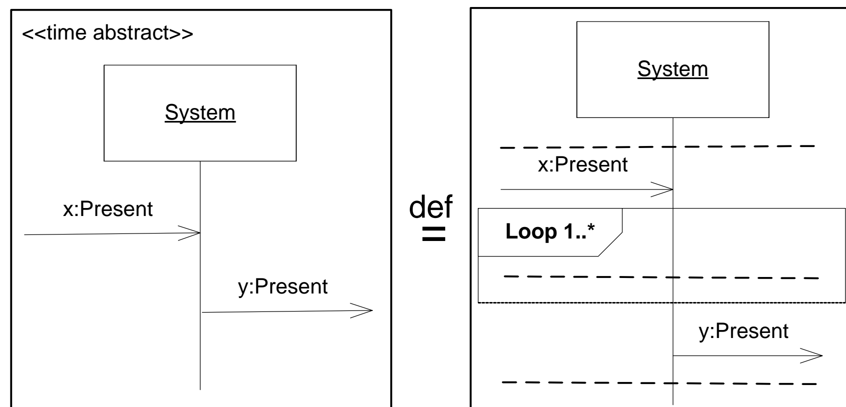


Abbildung 3.3.: Interpretation sequenzieller Nachrichten in zeitabstrakten Sequenzdiagrammen.

deutet dies, dass zwischen der Aus- und Eingabe beliebig viele Zeitschritte vergehen können, es ist damit aber insbesondere auch ausgedrückt, dass die Aus- und Eingabe gleichzeitig erfolgen kann. Dieser Sonderfall ist damit begründet, dass in dem zeitsynchronen AutoFocus Eingaben nicht zeitgleiche Ausgaben beeinflussen können.[15] Die Eingaben zum Zeitpunkt t beeinflussen erst zum Zeitpunkt $t + 1$ die Ausgaben.

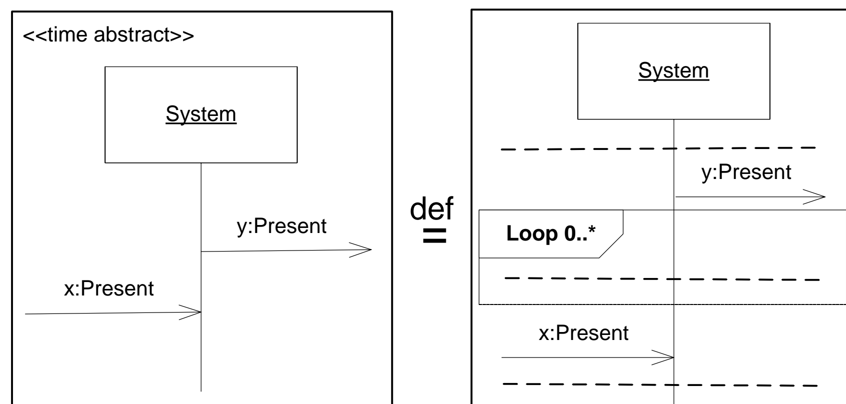


Abbildung 3.4.: Ausgaben gefolgt von Eingaben.

Es ist weiterhin möglich die Gleichzeitigkeit von Nachrichten in den hier definierten abstrakten Sequenzdiagrammen auszudrücken. Hierzu wird ein spezielles Symbol, das in der UML 2.0 zur Markierung von *Coregions* eingesetzt wird, mit dem Schlüsselwort *«simultaneous»* verwendet (siehe Abbildung 3.5 auf der nächsten Seite).

[15] Eine Ausnahme hiervon stellt das *Immediate* Kommunikationsparadigma von AutoFocus dar. Diese spezielle Kommunikationsart wird von den zeitabstrakten Sequenzdiagrammen nicht berücksichtigt.

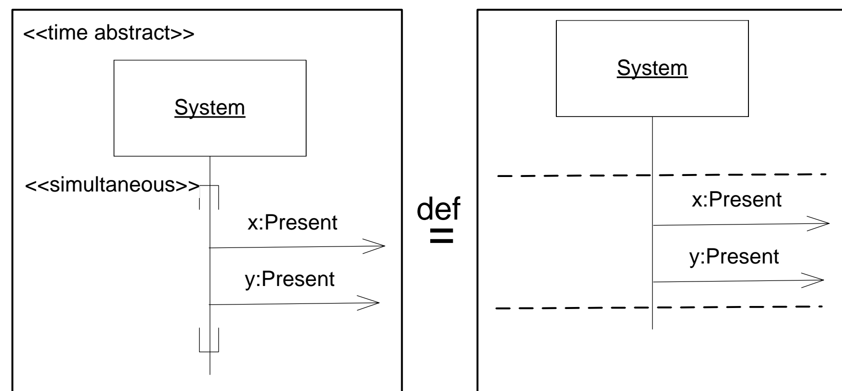


Abbildung 3.5.: Interpretation von simultanen Nachrichten in zeitabstrakten Sequenzdiagrammen.

Zur vereinfachten Ausdrückbarkeit mehrerer Abläufe in einem Sequenzdiagramm bieten die zeitabstrakten AutoFocus Sequenzdiagramme das von der UML bekannte Konzept der *Coregions*. Die Interpretation von *Coregions* ist in Abbildung 3.6 auf der nächsten Seite durch eine Abbildung in herkömmliche AutoFocus Sequenzdiagramme dargestellt. Alle in einer *Coregion* enthaltenen Nachrichten können in beliebiger zeitlicher Reihenfolge zueinander auftreten. Das gleichzeitige Auftreten von Nachrichten ist hierbei mit eingeschlossen.

Zur Beschreibung von sequentiellen Beziehungen von Nachrichten innerhalb einer *Coregion* wird eine spezielle Region verwendet, die mit dem Stereotyp *«sequential»* gekennzeichnet ist. Die Nachrichten innerhalb der *«sequential»* Region müssen die Reihenfolge zueinander einhalten, wobei die Reihenfolge mit den Nachrichten, die sich außerhalb dieser Region aber innerhalb der umschließenden *Coregion* befinden, beliebig vertauschbar ist. Die Abbildung 3.7 auf Seite 67 zeigt die Definition der *«sequential»* Region durch eine äquivalente Menge zeitabstrakter Sequenzdiagramme ohne Verwendung dieser Region.

Minimale Verzögerungen von *n* Zeittakten zwischen zwei Nachrichten können in den zeitabstrakten AutoFocus Sequenzdiagrammen durch ein Δ Symbol, wie in Abbildung 3.8 auf Seite 68 definiert, spezifiziert werden.

Aus Gründen der Universalität der zeitabstrakten Sequenzdiagramme kann in diesen auch eine feste Anzahl von *n* Zeitschritten zwischen zwei Nachrichten spezifiziert werden. Hierzu wird neben einem Strich auf der Sequenzdiagrammachse, wie in Abbildung 3.9 auf Seite 68 gezeigt, die Anzahl der Zeitschritte spezifiziert.

Die in diesem Abschnitt definierte Variante von AutoFocus Sequenzdiagrammen lässt sich zur Betrachtung von zeitabstrahierten Schnittstellenverhalten einsetzen. Die in den folgenden Abschnitten definierten Zeitabstraktionen werden mit Hilfe der hier vorgestellten Sequenzdiagrammnotation erläutert. Darüber hinaus eignen sich die hier vor-

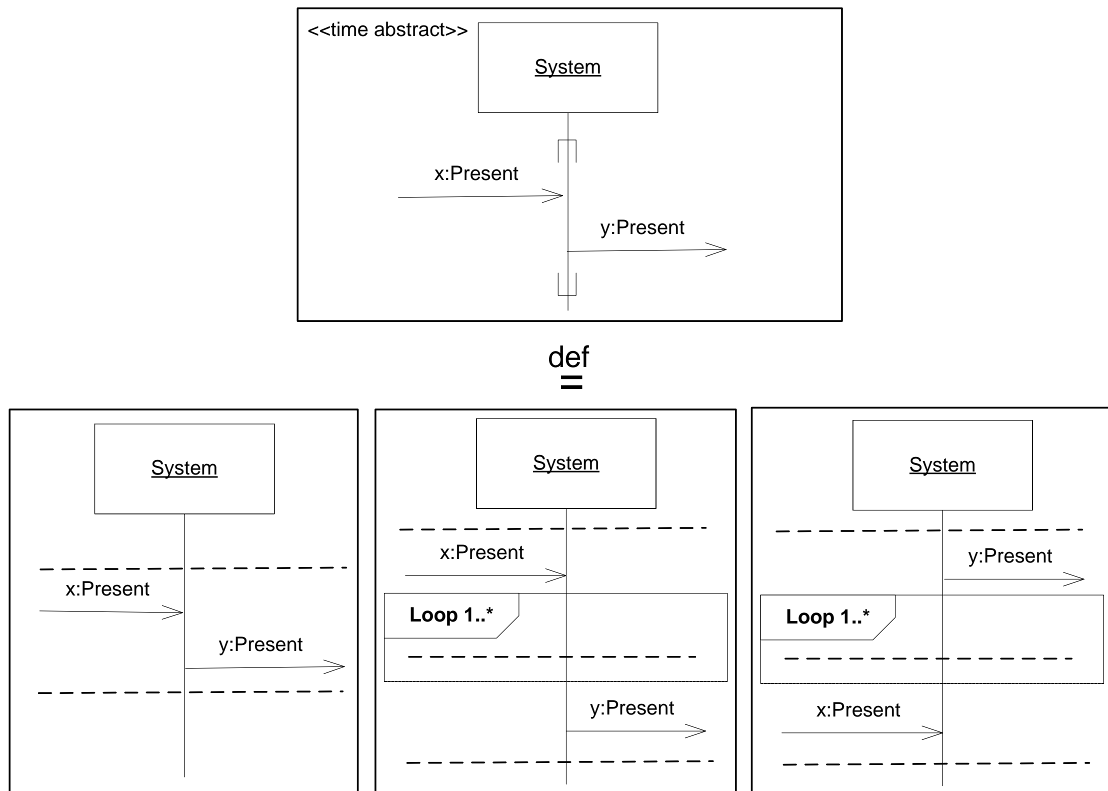


Abbildung 3.6.: Interpretation von *Coregions* in zeitabstrakten Sequenzdiagrammen.

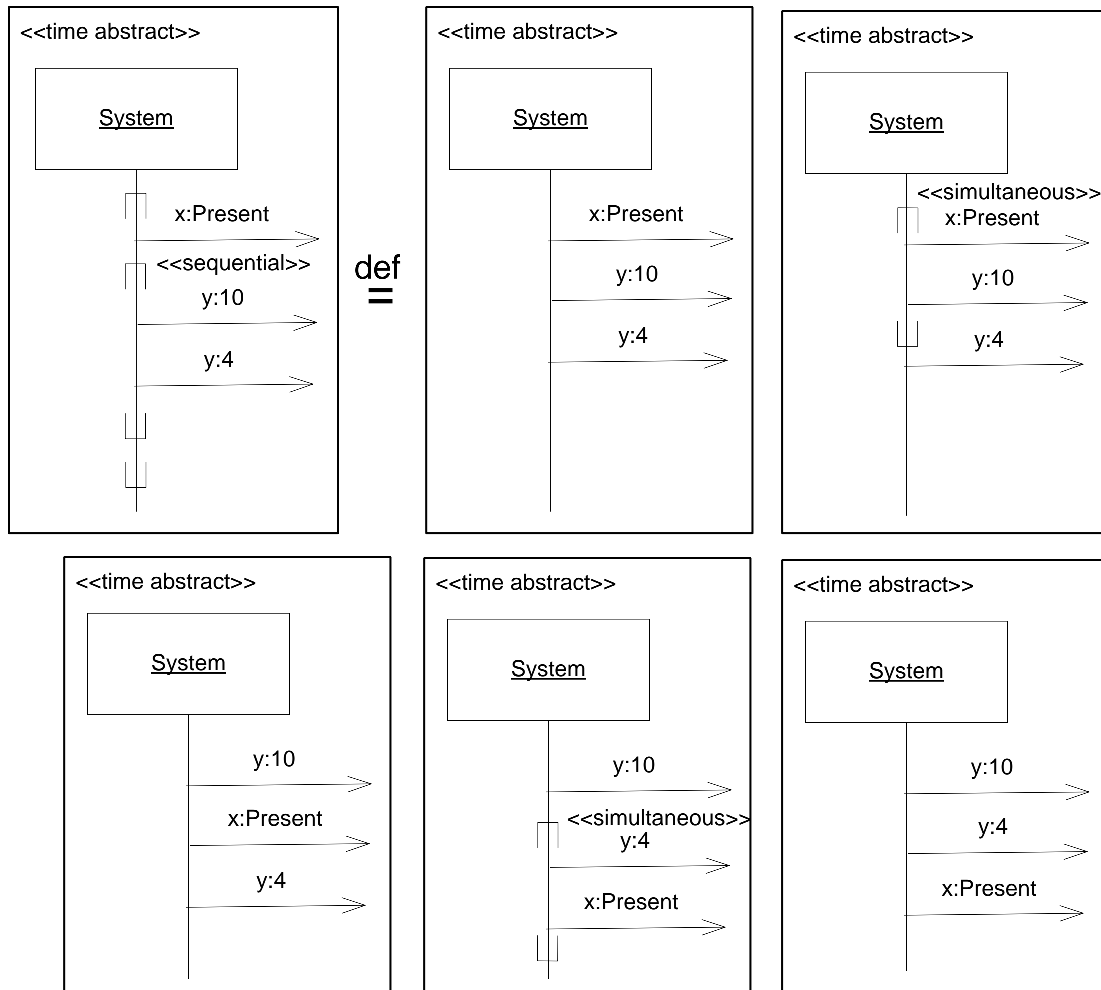


Abbildung 3.7.: *«sequential»* Region in zeitabstrakten Sequenzdiagrammen.

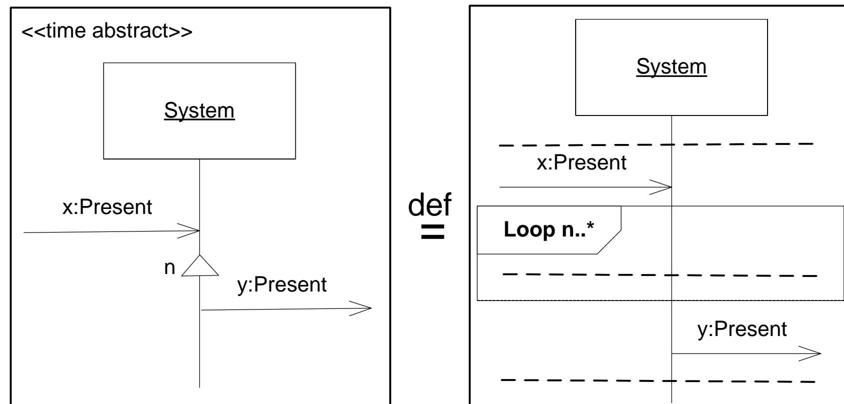


Abbildung 3.8.: Minimale Verzögerungen in zeitabstrakten Sequenzdiagrammen.

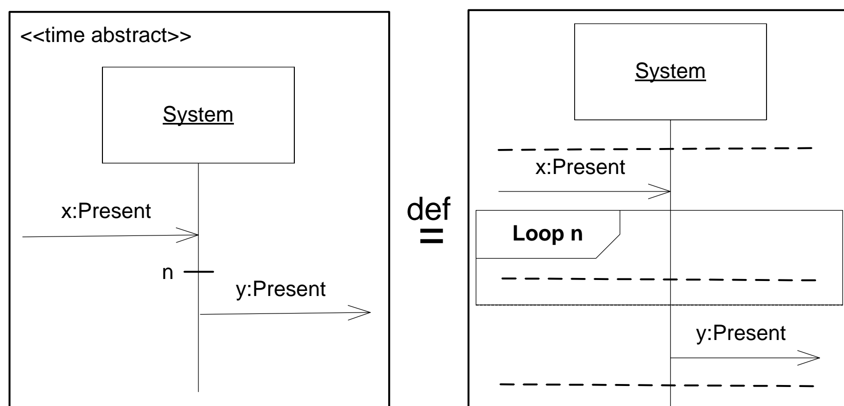


Abbildung 3.9.: Feste Anzahl von Zeitschritten zwischen zwei Nachrichten.

gestellten Sequenzdiagramme zur Betrachtung von zeitasynchronen AutoFocus Verhalten, deren Semantik in Kapitel 6 festgelegt wird. So wird in dem Abschnitt 7.5 diese Variante von Sequenzdiagrammen zum Nachweis der Verhaltensäquivalenz von Refactorings unter Zeitabstraktion eingesetzt.

3.7.2. Abstraktion von nicht verarbeiteten Eingaben

Nach der zeitsynchronen Semantik von AutoFocus werden Nachrichten, die zu einem bestimmten Zeitpunkt t auftreten, zum darauf folgenden Zeitpunkt $t+1$ wieder gelöscht beziehungsweise überschrieben. Tritt zum Zeitpunkt t eine Eingabe auf, dann hat diese entweder zu diesem Zeitpunkt eine Wirkung auf die Systemausführung oder sie hat zu diesem Zeitpunkt keine Wirkung und geht verloren. Wir bezeichnen Eingaben, die eine Wirkung auf die Systemausführung besitzen, als verarbeitete Eingaben und entsprechend Eingaben ohne Wirkung auf die Systemausführung als nicht verarbeitete Eingaben. Eine Eingabe wird verarbeitet, falls zu dem Zeitpunkt der Eingabe eine Transition gefeuert wird, deren Eingabebedingung von der getätigten Eingabe abhängt. Wird hingegen eine Transition ausgeführt, deren Eingabebedingung nicht von der betrachteten Eingabe abhängt, hat diese Eingabe keine Auswirkung auf die Systemausführung und wird somit als nicht verarbeitet angesehen.

Eingaben, die keine Auswirkung auf die Berechnungen beziehungsweise die erzeugten Ausgaben der betrachteten Modellspezifikation haben, sind für das Schnittstellenverhalten irrelevant. Von diesen Eingaben soll abstrahiert werden. Bei der Übersetzung von AutoFocus Modellen in Focus Spezifikationen wurden in Abschnitten 3.1 und 3.2 spezielle Verarbeitungskanäle eingeführt, die die Information der Verarbeitung von Eingaben auf der Ebene des Schnittstellenverhaltens sichtbar machen. Wenn eine Eingabe eines bestimmten Eingabe-Ports zum Zeitpunkt t verarbeitet wird, dann enthält der spezielle Verarbeitungsstrom dieses Eingabe-Ports an der Stelle des Zeitpunktes t den Wert *Present*. Keine Verarbeitung wird durch den Wert *void* repräsentiert.

Definition 3.16 (Abstraktion α_{NI})**.** Die Abstraktionsabbildung α_{NI} bildet einen konkreten AutoFocus Ablauf auf einen abstrakten Ablauf ab, wobei alle nicht verarbeiteten Eingaben aus den Eingabeströmen entfernt werden. Darüber hinaus werden die speziellen Verarbeitungsströme gelöscht. Die Abbildung liefert als Ergebnis Tupel von zeitsynchronen Focus Strömen.

Für ein in Focus übersetztes AutoFocus Modell, das die in Spezifikation 3.3 auf Seite 52 angegebene Form aufweist, ist die Zeitabstraktion α_{NI} wie folgt durch stromverarbeitende Funktionen in Focus definiert:[16]

$$M = \quad I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right)$$

[16] Der Ausdruck $\odot_{j=1}^{n} i_j$ bezeichnet die Konstruktion eines Tupels $(i_1, i_2, \ldots, i_n)$. Mit $\oplus_{Tup}$ ist die Konkatenation zweier Tupel zu einem Tupel bezeichnet (siehe Anhang A).

$$
\begin{aligned}
N &= I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \\
y &= \odot_{j=1}^{n+m+n} \checkmark \\
z &= \odot_{j=1}^{n+m+n} \langle\rangle \\
z' &= \odot_{j=1}^{n+m} \langle\rangle \\
\alpha_{NI} &\in M^{\underline{\omega},\mathsf{ts}} \rightarrow N^{\underline{\omega},\mathsf{ts}} \\
& \text{where } \alpha_{NI} \text{ so that } \forall\, x \in (M \cup y), s \in M^{\underline{\omega},\mathsf{ts}} : \\
\alpha_{NI}(x \& s) &= \Big((\odot_{j=1}^{n} istVerarbeitet(\pi_j.x, \pi_{n+m+j}.x)) \oplus_{Tup} (\odot_{k=n+1}^{n+m} \pi_k.x) \Big) \,\&\, \alpha_{NI}(s) \\
\alpha_{NI}(z) &= z'
\end{aligned}
\tag{3.45}
$$

Hierbei wird die Hilfsfunktion *istVerarbeitet* verwendet, die einzelne nichtverarbeitete Eingaben durch *void* maskiert.

$$
\begin{aligned}
R &= \bigcup_{j=1}^{n} I_j \\
istVerarbeitet &\in (R \cup \{\checkmark\}) \times (Signal \cup VoidDT \cup \{\checkmark\}) \rightarrow (R \cup \{\checkmark\}) \\
& \text{where } istVerarbeitet \text{ so that } \forall\, y \in (R \cup \{\checkmark\}) : \\
istVerarbeitet(y, Present) &= y \\
istVerarbeitet(y, void) &= void \\
istVerarbeitet(\checkmark, \checkmark) &= \checkmark
\end{aligned}
\tag{3.46}
$$

3.7.3. Zeitabstraktion von leeren Ausführungsschritten

Eine Zeitabstraktion von AutoFocus Schnittstellenverhalten besteht darin, Ausführungsschritte, in denen keine verarbeiteten Eingaben und keine Ausgaben auftreten, nicht zu berücksichtigen. Wir bezeichnen diese Abstraktion mit α_{ZA1}. Diese Abstraktion wird in der Focus Repräsentation durch das Löschen leerer Ausführungsschritte aus dem einen Ablauf repräsentierenden nach α_{NI} (Definition 3.16 auf der vorherigen Seite) abstrahierten Tupel von Ein- und Ausgabeströmen realisiert. Ein leerer Ausführungsschritt zum Zeitpunkt t ist hierbei dadurch charakterisiert, dass alle in dem Tupel enthaltenen Ströme zum Zeitpunkt t den Wert *void* enthalten.

Die Abbildung 3.10 auf der nächsten Seite zeigt den Übergang von zwei nach α_{NI} abstrahierten Abläufen zu einem nach α_{ZA1} abstrahierten Ablauf unter Verwendung von gezeiteten und zeitabstrakten AutoFocus Sequenzdiagrammen. In Sequenzdiagramm

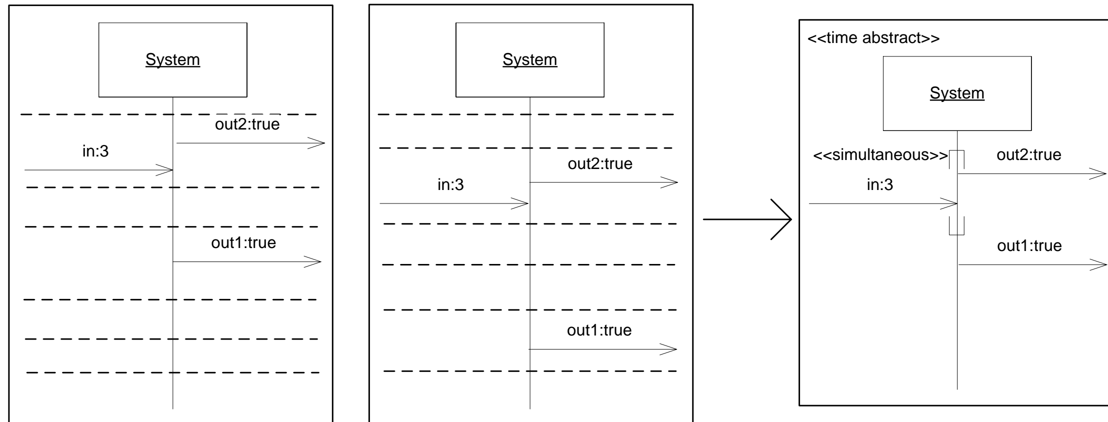


Abbildung 3.10.: Zeitabstraktion α_{ZA1}

1 und 2 treten die Eingabe $in : 3$ und die Ausgabe $out2 : true$ gleichzeitig auf. Die Gleichzeitigkeit von Nachrichten bleibt unter der Abstraktion α_{ZA1} erhalten. In dem zeitabstrakten Sequenzdiagramm ist die Gleichzeitigkeit durch die «*simultaneous*» Region dargestellt. In Sequenzdiagramm 1 vergehen von dem Empfang der Eingabe $in : 3$ bis zu dem Schreiben der Ausgabe $out1 : true$ zwei Zeitschritte. In dem Sequenzdiagramm 2 werden drei Ausführungsschritte bis zur Reaktion auf die Eingabe benötigt. Die Zeitabstraktion entfernt alle leeren Zeitschritte aus den Sequenzdiagrammen. Es ist unter der Zeitabstraktion α_{ZA1} nur noch erkennbar, dass auf die Eingabe irgendwann mit der Ausgabe reagiert wird. Somit verhalten sich beide Abläufe 1 und 2 unter Abstraktion gleich.

Definition 3.17 (Zeitabstraktion α_{ZA1})**.** Wir definieren die Abstraktionsabbildung α_{ZA1} als die Zeitabstraktion von leeren Ausführungsschritten. Die Abstraktion α_{ZA1} schließt die Abstraktion α_{NI} ein und entfernt alle Ausführungsschritte, in denen die Ein- und Ausgabekanäle ausnahmslos mit dem Wert *void* belegt sind. Die Abstraktion erhält hierbei die Reihenfolge und die Eigenschaft der Gleichzeitigkeit von Nachrichten. Als Ergebnis der Abstraktionsabbildung wird ein Tupel von zeitsynchronen Focus Strömen geliefert.

Die Abstraktionsabbildung für AutoFocus Schnittstellenabläufe ist als eine Focus stromverarbeitende Funktion gegeben.

$$
\begin{aligned}
M = \quad & I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right) \\
N = \quad & I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \\
\alpha_{ZA1} \in \quad & M^{\underline{\omega}, \mathsf{ts}} \to N^{\underline{\omega}, \mathsf{ts}} \\
& \text{where } \alpha_{ZA1} \text{ so that } \forall y \in M^{\underline{\omega}, \mathsf{ts}} :
\end{aligned}
$$

$$\alpha_{ZA1}(y) = ZA1(\alpha_{NI}(y)) \tag{3.47}$$

$$\begin{aligned}
x =\ & \odot_{i=1}^{n+m}\langle\, void, \checkmark \rangle \in N^{\underline{*},\mathsf{ts}} \\
z =\ & \odot_{i=1}^{n+m}\langle\rangle \\
Y =\ & \{a \in (N^{\underline{*},\mathsf{ts}} \setminus x) \text{ so that } \forall\, b \in [1;\ n+m] : \sharp(\Pi_b.a) = 2\} \\
ZA1 \in\ & N^{\underline{\omega},\mathsf{ts}} \rightarrow N^{\underline{\omega},\mathsf{ts}} \\
& \text{where } ZA1 \text{ so that } \forall\, y \in Y, s \in N^{\underline{\omega},\mathsf{ts}} : \\
& ZA1(x \frown s) = ZA1(s) \\
& ZA1(y \frown s) = y \frown ZA1(s) \\
& ZA1(z) = z
\end{aligned} \tag{3.48}$$

3.7.4. Zeitabstraktion von aufeinander folgenden Eingaben beziehungsweise Ausgaben

Die Abstraktion α_{ZA2} abstrahiert vollständig von AutoFocus Zeitschritten und schließt die Abstraktionen α_{ZA1} (Definition 3.17 auf der vorherigen Seite) und α_{NI} (Definition 3.16 auf Seite 69) mit ein. Es bleiben nur noch die zeitlichen Beziehungen zwischen den verarbeiteten Eingaben und den von ihnen bewirkten Ausgaben erhalten. Die Abstraktion erhält also die Reihenfolge von Ein- und Ausgaben zueinander, abstrahiert aber von der Reihenfolge aufeinander folgender Eingaben beziehungsweise aufeinander folgender Ausgaben. Die Reihenfolgen von Nachrichten auf einem Port bleiben durch die Abstraktion erhalten.

Die Abstraktion α_{ZA2} ist in Abbildung 3.11 auf der nächsten Seite an einem Beispiel unter Verwendung der zeitabstrakten AutoFocus Sequenzdiagramme (siehe Abschnitt 3.7.1) dargestellt. Unter der Zeitabstraktion sind in dem Beispiel folgende zeitliche Beziehungen sichtbar:

- Nach der Eingabe $q : 1$ erfolgen in beliebiger Reihenfolge zueinander die Ausgaben $s : 10$ und $z : 0$.
- Nach den Ausgaben $s : 10$ und $z : 0$ erfolgen die Eingaben $y : \mathit{Present}$, $x : 5$ und $x : 1$, wobei die $y : \mathit{Present}$ in beliebiger Reihenfolge zu den anderen beiden Eingaben stehen kann.
- Die Eingabe $x : 5$ erfolgt vor der Eingabe $x : 1$, da diese Eingaben auf einem gemeinsamen Port erfolgen.
- Nach den Eingaben $y : \mathit{Present}$, $x : 5$ und $x : 1$ erfolgt die Ausgabe $z : 6$.

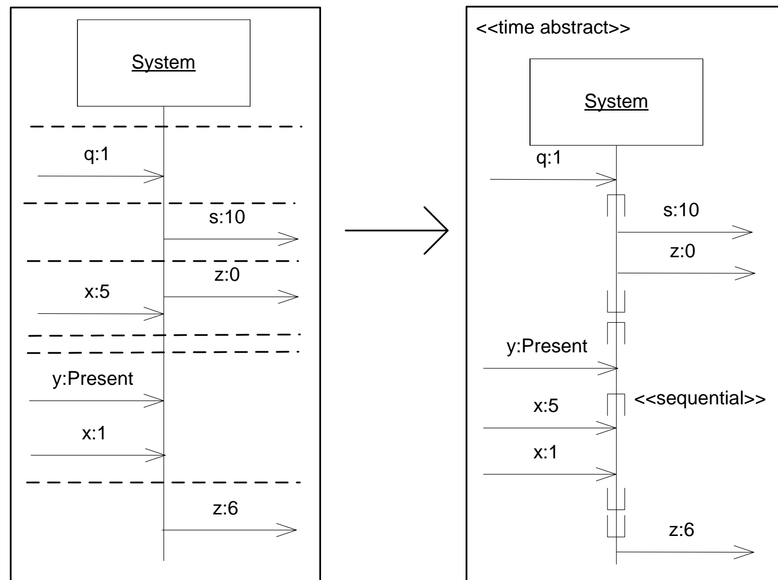


Abbildung 3.11.: Die Zeitabstraktion α_{ZA2}.

Zur Darstellung von nach α_{ZA2} abstrahierten Schnittstellenabläufen werden weiterhin Tupel von gezeiteten Focus Strömen verwendet, wobei die Tick Symbole nicht mehr als AutoFocus Zeitschritte interpretiert werden. Alle Nachrichten, die sich in den verschiedenen Strömen des Tuples zwischen dem Zeittick n und $n+1$ befinden, werden als nicht im zeitlichen Zusammenhang zueinander interpretiert und sind demnach in ihrer Reihenfolge zueinander beliebig vertauschbar. In der Abbildung 3.11 entspricht dies den im zeitabstrakten Sequenzdiagramm enthaltenen *Coregions*. Treten innerhalb eines Stroms zwischen dem Zeittick n und $n+1$ mehrere Nachrichten auf, dann stehen diese Nachrichten in einer festen Reihenfolge zueinander. Dieser Fall entspricht in AutoFocus sequentiellen Nachrichten auf einem Port. In der Abbildung 3.11 ist dieser Fall in dem abstrakten Sequenzdiagramm als *«sequential»* Region dargestellt. Die zur Notation von konkretem AutoFocus Schnittstellenverhalten verwendeten speziellen *void* Symbole sind in den nach α_{ZA2} abstrahierten Strömen nicht mehr enthalten. Treten in einem konkreten Schnittstellenablauf gleichzeitig Ein- und Ausgaben auf, dann werden diese in dem abstrakten Schnittstellenablauf zeitlich voneinander getrennt (die Ausgabe ist in der abstrakten Sicht zeitlich vor der Eingabe). Diese Trennung ist damit begründet, dass nach der zeitsynchronen AutoFocus Semantik eine Eingabe, die zeitgleich mit einer Ausgabe auftritt, auf diese Ausgabe keinen Einfluss nehmen kann, sondern erst auf Ausgaben zu einem späteren Zeitpunkt. Zwischen den Zeitticks n und $n+1$ sind in den verschiedenen Strömen entweder nur Eingabeströme oder nur Ausgabeströme mit Nachrichten belegt. Dies entspricht in dem abstrakten Sequenzdiagramm der zeitlichen Trennung von Ein- und Ausgaben durch getrennte *Coregions*.

Die formale Definition der Abstraktion α_{ZA2} verwendet die Abstraktion α_{ZA1} und erfor-

dert einige Hilfsfunktionen. Legen wir zunächst Mengen von Tupel von Nachrichten fest, die in der Definition von α_{ZA2} verwendet werden. A enthält die Menge aller Tupel von Nachrichten aus der abstrakten Schnittstellensignatur N, die ausschließlich Eingaben und keine Ausgaben enthalten. Die Menge B enthält alle Tupel von Nachrichten, die nur Ausgaben enthalten. Die Menge C ist die Menge aller Tupel von Nachrichten, die entweder nur Eingaben oder nur Ausgaben enthalten und die Menge D ist die Menge aller Tupel von Nachrichten, die gleichzeitig Ein- und Ausgaben enthalten.

$$N = I_1 \times ... \times I_n \times O_1 \times ... \times O_m$$

$$A = \{(i_1, ..., i_n, o_1, ..., o_m) \in N \text{ so that } \forall j \in [1, m] : o_j = void\} \tag{3.49}$$

$$B = \{(i_1, ..., i_n, o_1, ..., o_m) \in N \text{ so that } \forall j \in [1, n] : i_j = void\} \tag{3.50}$$

$$C = A \cup B \tag{3.51}$$

$$D = N \setminus C \tag{3.52}$$

Die Funktion *SplitIO* trennt gleichzeitig auftretende Ein- und Ausgaben durch das Einfügen von zusätzlichen Tick Symbolen zeitlich auf. Das Einfügen der Tick Symbole geschieht hierbei in allen Strömen des einen Ablauf beschreibenden Tupels von Strömen. Die Tick Symbole werden in allen diesen Strömen an der gleichen Stelle eingefügt, um den zeitlichen Bezug zwischen den einzelnen Strömen zu erhalten. Die Ausgaben werden zeitlich vor den Eingaben angeordnet, da Eingaben, die gleichzeitig mit Ausgaben auftreten, diese nach der zeitsynchronen AutoFocus Semantik nicht beeinflussen können.[17]

$$\begin{aligned}
q &= \odot_{i=1}^{n+m} \checkmark \\
z &= \odot_{i=1}^{n+m} \langle\rangle \\
SplitIO &\in N^{\underline{\omega},\text{ts}} \rightarrow N^{\underline{\omega},\text{ts}} \\
& \quad \text{where } SplitIO \text{ so that } \forall x \in C, y \in D \\
SplitIO(x \,\&\, s) &= x \,\&\, SplitIO(s) \\
SplitIO(y \,\&\, s) &= Output(y) \,\&\, q \,\&\, Input(y) \,\&\, SplitIO(s) \\
SplitIO(q \,\&\, s) &= q \,\&\, SplitIO(s) \\
SplitIO(z) &= z
\end{aligned} \tag{3.53}$$

[17] Bei Verwendung der speziellen Immediate Kommunikationssemantik können Eingaben gleichzeitig auftretende Ausgaben beeinflussen. Diese Art der Kommunikation wird an dieser Stelle nicht berücksichtigt.

Die Funktionen *Output* und *Input* liefern von einem Tupel von Nachrichten das Tupel von Nachrichten, dessen Eingaben beziehungsweise Ausgaben durch Maskierung mit *void* entfernt wurden.

$$\begin{aligned}
Output \in\ & N \rightarrow N \\
\text{where } Output \text{ so that}\ & \forall\, x \in N \\
Output(x) =\ & (\odot_{i=1}^{n} void) \oplus_{Tup} (\odot_{j=1}^{m} \pi_{n+j}.x) \qquad (3.54)\\
Input \in\ & N \rightarrow N \\
\text{where } Input \text{ so that}\ & \forall\, x \in N \\
Input(x) =\ & (\odot_{i=1}^{n} \pi_i.x) \oplus_{Tup} (\odot_{j=1}^{m} void) \qquad (3.55)
\end{aligned}$$

Die Hilfsfunktion *Combine* fügt in einem Tupel von Strömen von Nachrichten durch Entfernen von Ticksymbolen aufeinander folgende Zeitpunkte zusammen, in denen entweder nur Eingaben oder nur Ausgaben vorhanden sind.

$$\begin{aligned}
v =\ & \odot_{j=1}^{n+m} \checkmark \\
z =\ & \odot_{i=1}^{n+m} \langle\rangle \\
N^{\underline{\omega}} =\ & I_1^{\underline{\omega}} \times \ldots \times I_n^{\underline{\omega}} \times O_1^{\underline{\omega}} \times \ldots \times O_m^{\underline{\omega}} \\
Combine \in\ & N^{\underline{\omega},\text{ts}} \rightarrow N^{\underline{\omega}} \\
\text{where } Combine \text{ so that}\ & \forall\, a, b \in A : \forall\, c, d \in B : \forall\, s \in N^{\underline{\omega},\text{ts}} \\
Combine(a \,\&\, v \,\&\, b \,\&\, s) =\ & a \,\&\, Combine(b \,\&\, s) \\
Combine(c \,\&\, v \,\&\, d \,\&\, s) =\ & c \,\&\, Combine(d \,\&\, s) \\
Combine(a \,\&\, v \,\&\, c \,\&\, s) =\ & a \,\&\, v \,\&\, Combine(c \,\&\, s) \\
Combine(c \,\&\, v \,\&\, a \,\&\, s) =\ & c \,\&\, v \,\&\, Combine(a \,\&\, s) \\
Combine(v \,\&\, s) =\ & v \,\&\, Combine(s) \\
Combine(z) =\ & z \qquad (3.56)
\end{aligned}$$

Die Hilfsfunktion *RemoveVoid* entfernt aus den Tupel von Strömen alle vorkommenden *void* Nachrichten.

$$N = \quad I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m$$

$$\begin{aligned}
N_{noVoid} = \quad & (I_1 \setminus VoidDT \times \ldots \times I_n \setminus VoidDT \times \\
& O_1 \setminus VoidDT \times \ldots \times O_m \setminus VoidDT) \\
A = \quad & \left(\bigcup_{j=1}^{n} I_j \cup \bigcup_{k=1}^{m} O_k \right) \setminus VoidDT \\
RemoveVoid \in \quad & N^{\underline{\omega}} \rightarrow N^{\underline{\omega}}_{noVoid} \\
\text{where } RemoveVoid \quad & \text{so that } \forall\, s \in N^{\underline{\omega}} \\
RemoveVoid(s) = \quad & A \circledS s
\end{aligned} \tag{3.57}$$

Die Zeitabstraktion α_{ZA2} kann jetzt durch die kombinierte Anwendung der Zeitabstraktion α_{ZA1} und den vorangehend festgelegten Hilfsfunktionen definiert werden.

Definition 3.18 (Abstraktion von der Reihenfolge aufeinander folgender Ein- beziehungsweise Ausgaben α_{ZA2})**.** Die Abstraktionsabbildung bildet konkrete AutoFocus Schnittstellenabläufe, dargestellt durch Tupel von Focus Strömen, in abstrakte AutoFocus Schnittstellenabläufe ab. Es ist zu beachten, dass diese Abstraktion im Gegensatz zu den vorhergehend definierten Abstraktionen als abstrakte Abläufe keine Tupel von zeitsynchronen Focus Strömen, sondern Tupel von normalen gezeiteten Strömen liefert. Die Abstraktion ist in Focus wie folgt definiert:

$$\begin{aligned}
M = \quad & I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right) \\
N_{noVoid} = \quad & (I_1 \setminus VoidDT \times \ldots \times I_n \setminus VoidDT \times O_1 \setminus VoidDT \times \ldots \times O_m \setminus VoidDT) \\
\alpha_{ZA2} \in \quad & M^{\underline{\omega},\mathsf{ts}} \rightarrow N^{\underline{\omega}}_{noVoid} \\
\text{where } \alpha_{ZA2} \quad & \text{so that } \forall\, s \in M^{\underline{\omega},\mathsf{ts}} \\
\alpha_{ZA2}(s) = \quad & RemoveVoid(Combine(SplitIO(\alpha_{ZA1}(s))))
\end{aligned} \tag{3.58}$$

3.7.5. Die Focus Zeitabstraktion angewendet auf AutoFocus Schnittstellenabläufe

Die Focus Spezifikationssprache besitzt einen speziellen Zeitabstraktionsoperator $\bar{\ }$. Dieser Operator löscht aus einem Strom alle vorkommenden Zeitticksymbole $\checkmark$. Hierdurch kann der zeitliche Bezug zwischen verschiedenen Strömen aufgelöst werden.

Definition 3.19 (Focus Zeitabstraktion $\bar{\ }$ [BS01, S. 65])**.**

$$\begin{aligned}
N = \quad & (I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m) \\
\bar{\ } \in \quad & N^{\underline{\omega}} \rightarrow N^{\omega} \\
\bar{s} \overset{\mathsf{def}}{=} \quad & N \circledS s
\end{aligned} \tag{3.59}$$

Da bei der Übersetzung von AutoFocus nach Focus das Nichtvorhandensein von Nachrichten durch einen speziellen Wert *void* dargestellt wird, existiert in einem Strom zwischen zwei Zeitticksymbolen immer genau eine Nachricht. Diese spezielle Form von gezeiteten Focus Stömen wird als zeitsynchrones Focus bezeichnet. Werden die Zeitticksymbole entfernt, dann existieren weiterhin bei dieser speziellen Systemausprägung zeitliche Beziehungen zwischen den einzelnen Strömen. Durch das Entfernen aller *void* Werte werden diese zeitlichen Beziehungen aufgelöst.

Definition 3.20 (Zeitabstraktion α_{ZA3})**.** Die Zeitabstraktion α_{ZA3} bildet konkrete Schnittstellenabläufe auf abstrakte Schnittstellenabläufe, die von Reihenfolgen zwischen Nachrichten auf unterschiedlichen Kanälen abstrahieren, ab. Hierbei schließt diese Abstraktion die Abstraktion α_{NI} mit ein.

$$
\begin{aligned}
M = \quad & I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right) \\
N_{noVoid} = \quad & (I_1 \setminus VoidDT \times \ldots \times I_n \setminus VoidDT \times O_1 \setminus VoidDT \times \ldots \times O_m \setminus VoidDT) \\
\alpha_{ZA3} \in \quad & M^{\omega,\mathsf{ts}} \rightarrow N^{\omega}_{noVoid} \\
& \text{where } \alpha_{ZA3} \text{ so that } \forall s \in M^{\omega,\mathsf{ts}} \\
\alpha_{ZA3}(s) = \quad & Q \circledS \overline{\alpha_{NI}(s)} \qquad (3.60)
\end{aligned}
$$

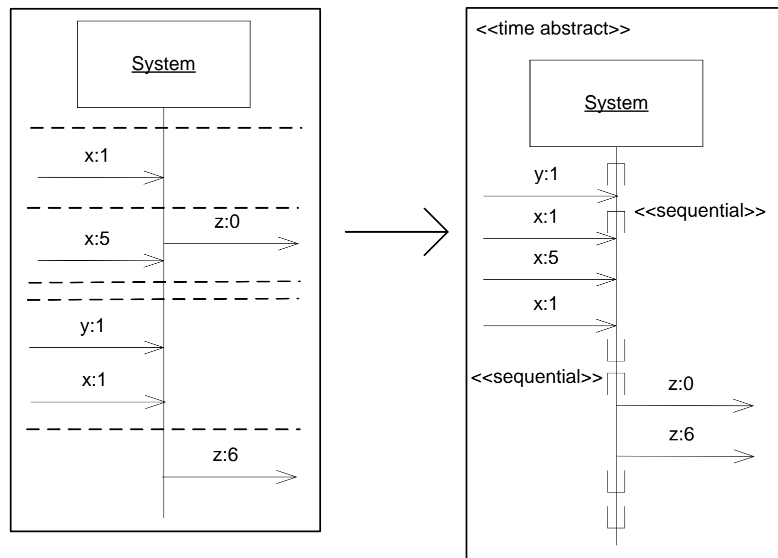


Abbildung 3.12.: Die Zeitabstraktion α_{ZA3}.

Die Abbildung 3.12 zeigt die Wirkung der Abstraktion α_{ZA3} anhand von Sequenzdiagrammen. Das gezeitete Sequenzdiagramm (links in der Abbildung) wird durch die

Abstraktion in das zeitabstrakte Sequenzdiagramm (rechts in der Abbildung) abgebildet. Alle Nachrichten, die auf ein und dem selben Port auftreten, werden in einer *«sequential»* Region dargestellt. Deren Reihenfolge ist nicht vertauschbar. Alle *«sequential»* Regionen sind in einer großen *Coregion* zusammengefasst, so dass die Reihenfolgen von Nachrichten auf verschiedenen Ports beliebig zueinander vertauscht werden können.

3.7.6. Zeitkonkretisierung von Schnittstellenverhalten

In den vorangehenden Abschnitten 3.7.2 bis 3.7.5 wurden Abstraktionsabbildungen auf Abläufen von AutoFocus Schnittstellenverhalten definiert. Die Zeitabstraktion eines Schnittstellenverhaltens kann nun durch die Anwendung der Abstraktion auf alle in einem Schnittstellenverhalten enthaltenen Abläufe definiert werden. Wir bezeichnen mit $\widetilde{\alpha}_x \in \mathcal{M} \rightarrow \mathcal{N}$ die entsprechend Definition 3.13 auf Seite 60 für die Anwendung auf Schnittstellenverhalten erweiterte Abstraktion. Mit $\underline{\widetilde{\alpha}_x} \in \underline{\mathcal{M}} \rightarrow \underline{\mathcal{N}}$ ist die Verallgemeinerung der Verhaltensabstraktion auf allgemeine Schnittstellensignaturen entsprechend Definition 3.12 auf Seite 59 bezeichnet.

Wir können nun in Abhängigkeit der allgemeinen Verhaltensabstraktion $\widetilde{\alpha}_x$ eine Konkretisierungsabbildung angeben.

Definition 3.21 (Konkretisierung von Schnittstellenverhalten $\widetilde{\gamma}_x$)**.** Die Konkretisierungsabbildung $\widetilde{\gamma}_x$ ist deklarativ durch die Eigenschaft angegeben, dass die sich ergebenden konkreten Schnittstellenverhalten durch die Abstraktion von Schnittstellenverhalten $\widetilde{\alpha}_x$ auf das ursprünglich abstrakte Verhalten abgebildet werden. Sei $\mathcal{N}$ die Menge der abstrakten Schnittstellenverhalten und $\mathcal{M}$ die Menge der konkreten Schnittstellenverhalten.

$$\begin{aligned}
&\widetilde{\gamma}_x \in \mathcal{N} \rightarrow \mathcal{P}(\mathcal{M}) \\
&\text{where } \widetilde{\gamma}_x \text{ so that } \forall s \in \mathcal{N} \\
&\widetilde{\gamma}_x(s) = \{t \in \mathcal{M} \text{ so that } \widetilde{\alpha}_x(t) = s\}
\end{aligned} \tag{3.61}$$

Mit $\underline{\widetilde{\gamma}_x} \in \underline{\mathcal{N}} \rightarrow \mathcal{P}(\underline{\mathcal{M}})$ ist die Verallgemeinerung der Konkretisierungsabbildung auf Verhalten mit beliebigen Schnittstellensignaturen bezeichnet. Die Konkretisierung liefert zu einem abstrakten Schnittstellenverhalten die Menge aller konkreten Schnittstellenverhalten, die unter der Abstraktion gleich dem gegebenen abstrakten Verhalten sind.

Mit Hilfe der Konkretisierungen sind die Semantiken der abstrakten Schnittstellenverhalten über die Semantik von konkretem AutoFocus Schnittstellenverhalten, die in Form von Focus Ein- / Ausgaberelationen gegeben sind, definiert.

Wird auf ein konkretes Schnittstellenverhalten $\mathcal{R}_K$ zunächst die Abstraktion $\widetilde{\alpha}_x$ und anschließend die Konkretisierung $\widetilde{\gamma}_x$ angewendet, dann ist das ursprüngliche Verhalten $\mathcal{R}_K$ in der resultierenden Menge von Verhalten enthalten.

$$\forall \mathcal{R}_K \in \mathcal{M} : \mathcal{R}_K \in \widetilde{\gamma}_x(\widetilde{\alpha}_x(\mathcal{R}_K)) \tag{3.62}$$

Wendet man auf ein abstraktes Schnittstellenverhalten $\mathcal{R}_x$ zunächst die Konkretisierung an, erhält man eine Menge von möglichen konkreten Verhalten. Wird anschließend auf alle in dieser Menge enthaltenen konkreten Verhalten die Abstraktion angewendet, dann erhält man für alle diese konkreten Verhalten wiederum $\mathcal{R}_x$ als abstraktes Verhalten.

$$\forall \mathcal{R}_x \in \mathcal{N}_x : \forall \mathcal{R}_K \in \widetilde{\gamma}_x(\mathcal{R}_x) : \widetilde{\alpha}_x(\mathcal{R}_K) = \mathcal{R}_x \tag{3.63}$$

3.7.7. Verhaltensäquivalenz unter Zeitabstraktion

In der Definition 3.14 auf Seite 60 wurde festgelegt, wann zwei Spezifikationen ein äquivalentes konkretes Schnittstellenverhalten aufweisen. Die vorangehend definierten Zeitabstraktionen des Schnittstellenverhaltens werden als Beobachtungsbegriffe für das Modell-Refactoring eingesetzt. Zu diesem Zweck wird ein Begriff von Verhaltensäquivalenz des Schnittstellenverhaltens unter Zeitabstraktion benötigt.

Definition 3.22 (Äquivalenz von zeitabstrahiertem AutoFocus Schnittstellenverhalten)**.** Zwei Schnittstellenverhalten zweier AutoFocus Modellspezifikationen S_1 und S_2 sind unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_x$ äquivalent, wenn beide Spezifikationen die gleiche syntaktische Schnittstelle besitzen (Definition 3.2 auf Seite 52 und Definition 3.6 auf Seite 54) und die zeitabstrahierten Schnittstellenverhalten (Definition 3.11 auf Seite 57) beider Spezifikationen identisch sind:

$$\begin{aligned} &\mathit{verhaltens\ddot{a}quivalentAF_\alpha_x}(S_1, S_2) \overset{\text{def}}{=} \\ &(Syn_{AF}(S_1) = Syn_{AF}(S_2)) \wedge \\ &(\underline{\widetilde{\alpha}}_x(\underline{SV}_{AF}(S_1)) = \underline{\widetilde{\alpha}}_x(\underline{SV}_{AF}(S_2))) \end{aligned} \tag{3.64}$$

3.7.8. Komposition abstrakten Schnittstellenverhaltens

In den vorangehenden Abschnitten wurden verschiedene Zeitabstraktionen von AutoFocus Schnittstellenverhalten definiert. Diese Abstraktionen können sowohl auf das Schnittstellenverhalten des Gesamtsystems als auch auf das Schnittstellenverhalten einzelner Komponenten angewendet werden. Um von den abstrakten Schnittstellenverhalten von Komponenten auf ein abstraktes Gesamtschnittstellenverhalten schließen zu können, wird ein abstrakter Kompositionsoperator benötigt.

Definition 3.23 (Abstrakte Komposition von Schnittstellenverhalten $\otimes_{\alpha_x}$)**.** Der Operator $\otimes_{\alpha_x}$ komponiert zwei nach $\underline{\widetilde{\alpha}}_x$ abstrahierte Schnittstellenverhalten $\mathcal{R}_{S_1}$ und $\mathcal{R}_{S_2}$ zusammen mit deren abstrakten syntaktischen Schnittstellen Syn_1 und Syn_2 zu einem abstrakten Gesamtverhalten. Die Definition erfolgt unter Verwendung des Kompositionsoperators $\otimes_K$ für konkretes Schnittstellenverhalten, der in Definition 3.15 auf Seite 62 festgelegt ist. Durch die Konkretisierungsabbildung $\underline{\widetilde{\gamma}}_x$ erhalten wir zu den zeitabstrahierten Schnittstellenverhalten $\mathcal{R}_{S_1}$ und $\mathcal{R}_{S_2}$ jeweils eine Menge von möglichen konkreten Verhalten. Die konkrete Komposition wird auf alle paarweisen Kombinationen der in den beiden Mengen enthaltenen konkreten Verhalten zusammen mit deren konkreten syntaktischen Schnittstellen angewendet. Es ergeben sich Mengen von möglichen konkreten Gesamtschnittstellenverhalten, die durch die Abstraktion $\underline{\widetilde{\alpha}}_x$ auf Mengen von möglichen abstrakten Gesamtschnittstellenverhalten abgebildet werden. Seinen $\underline{\mathcal{N}}$ die Menge entsprechend aller Schnittstellensignaturen möglichen abstrakten Schnittstellenverhalten und $KonSyn \in \mathcal{L}_{Syn} \rightarrow \mathcal{L}_{Syn}$ die Abbildung, die zu einer abstrakten syntaktischen Schnittstelle die entsprechende konkrete syntaktische Schnittstelle durch Hinzufügen der speziellen Verarbeitungskanäle liefert. Die allgemeine abstrakte Komposition ist definiert als:

$$\otimes_{\alpha_x} \in (\underline{\mathcal{N}} \times \mathcal{L}_{Syn}) \times (\underline{\mathcal{N}} \times \mathcal{L}_{Syn}) \rightarrow \mathcal{P}(\underline{\mathcal{N}})$$
$$(\mathcal{R}_{S_1}, Syn_1) \otimes_{\alpha_x} (\mathcal{R}_{S_2}, Syn_2) \stackrel{\text{def}}{=}$$
$$\Big\{\mathcal{R}_S \in \underline{\mathcal{N}} \text{ so that } \exists\, a \in \underline{\widetilde{\gamma}}_x(\mathcal{R}_{S_1}), b \in \underline{\widetilde{\gamma}}_x(\mathcal{R}_{S_2}) :$$
$$\mathcal{R}_S = \underline{\widetilde{\alpha}}_x((a, KonSyn(Syn_1)) \otimes_K (b, KonSyn(Syn_2))\} \tag{3.65}$$

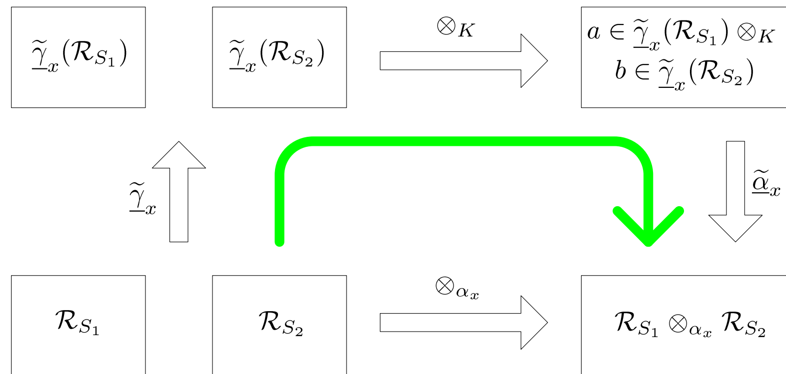


Abbildung 3.13.: Definition der abstrakten Komposition über Konkretisierung und konkrete Komposition.

Die abstrakte Komposition liefert Mengen von möglichen komponierten abstrakten Verhalten und nicht ein einzelnes abstraktes Verhalten. In Abbildung 3.13 ist die De-

finition des abstrakten Kompositionsoperators zum besseren Verständnis unter Weglassung der syntaktischen Schnittstellen graphisch dargestellt.

4. Formale Definition von Refactoring

In der Einleitung der Arbeit wurde die Definition von Refactoring nach Martin Fowler angegeben (Definition 1.6 auf Seite 21). Diese informelle Definition fordert die Verhaltensäquivalenz des beobachtbaren Verhaltens des Softwaresystems vor und nach Anwendung des Refactorings. Fowler definiert in [FBB$^+$99] jedoch nicht, was er genau unter beobachtbarem Verhalten versteht. Nach Fowler kann die Gleichheit des beobachtbaren Verhaltens durch ausgedehntes strukturiertes Testen gezeigt werden. Es bleibt aber weiterhin die Frage, wann durch das Testen beobachtbar gleiches Verhalten festgestellt wird und wann nicht.

Um an der für das Refactoring entscheidenden Frage der Gleichheit von beobachtbarem Verhalten präzise zu sein, liefert diese Arbeit eine formale Definition von Refactoring, die eine eindeutige Aussage der Verhaltensäquivalenz von Systemen vor und nach dem Refactoring bezüglich einer gewählten Abstraktion erlaubt. Zu diesem Zweck werden der vorangehend in der Definition 3.11 auf Seite 57 festgelegte Begriff des Schnittstellenverhaltens einer AutoFocus Modellspezifikation zusammen mit den in dem Abschnitt 3.7 eingeführten Begriffen von Zeitabstraktionen sowie der in den Definitionen 3.14 auf Seite 60 und 3.22 auf Seite 79 eingeführten Begriffen der Äquivalenz von konkreten und abstrakten Schnittstellenverhalten angewendet. Der Begriff des Refactoring wird an dieser Stelle sowohl allgemein als Programm- beziehungsweise Modelltransformation mit besonderen Eigenschaften definiert, als auch konkret als eine Transformation von AutoFocus Modellspezifikationen.

In Abschnitt 4.1 erfolgt zunächst die Definition des Begriffs des beobachtbaren Verhaltens und des Begriffs der Modelltransformation. Der Abschnitt 4.2 legt eine allgemeine formale Definition des Begriffs Refactoring unabhängig von der eingesetzten Modellierungs- beziehungsweise Programmiersprache fest. In dem Abschnitt 4.3 wird schließlich der Begriff des Modell-Refactorings speziell für die Modellierungssprache AutoFocus definiert.

4.1. Beobachtbares Verhalten und Modelltransformation

Definieren wir zunächst den von Fowler informell verwendeten Begriff des *beobachtbaren Verhaltens*.

Definition 4.1 (Beobachtbares Verhalten)**.** Das beobachtbare Verhalten eines Modells, einer Spezifikation beziehungsweise eines Programms S ist definiert als das unter einer Abstraktion sichtbare Schnittstellenverhalten. Sei $\mathcal{L}$ die Menge der syntaktisch korrek-

ten in einer Sprache darstellbaren Spezifikationen, $\underline{\mathcal{M}}$ die Menge aller möglichen konkreten Schnittstellenverhalten, $\underline{SV} \in \mathcal{L} \to \underline{\mathcal{M}}$ die Interpretationsabbildung, die zu einer Spezifikation das Schnittstellenverhalten der Spezifikation auf der Ebene der Systemschnittstelle liefert, $\underline{\mathcal{N}}$ die Menge aller möglichen abstrakten Schnittstellenverhalten, und $\underline{\widetilde{\alpha}}_x \in \underline{\mathcal{M}} \to \underline{\mathcal{N}}$ die Abstraktionsabbildung. Dann ist das beobachtbare Verhalten wie folgt definiert:[1]

$$BeobachtbaresVerhalten_x(S) \stackrel{\text{def}}{=} \underline{\widetilde{\alpha}}_x(\underline{SV}(S)) \tag{4.1}$$

Der allgemeinen Begriff der Programm- beziehungsweise Modelltransformation wird in dieser Arbeit wie folgt verwendet:

Definition 4.2 (Programm- beziehungsweise Modelltransformation)**.** Sei $\mathcal{L}$ die Menge aller durch eine Sprache L darstellbaren syntaktisch korrekten Spezifikationen. Die Sprache L kann hierbei sowohl eine herkömmliche Programmiersprache als auch eine graphische Sprache zur Modellierung von Software sein. Sei Par die Menge von Tupel aller möglichen Parameter, die der Transformationsabbildung übergeben werden. Eine Programm- beziehungsweise Modelltransformation ist eine Abbildung $T \in \mathcal{L} \times Par \to \mathcal{L}$ zur Änderung von Softwareprogrammen beziehungsweise Softwaremodellen. Sie bildet Spezifikationen in der Sprache L in Abhängigkeit der festgelegten Parameter auf abgeänderte Spezifikationen in der gleichen Sprache ab.

4.2. Allgemeine Definition von Refactoring

Die Definition des beobachtbaren Verhaltens ermöglicht nun die Definition von Refactoring als eine Programm- beziehungsweise Modelltransformation mit besonderen Eigenschaften.

Definition 4.3 (Refactoring (allgemein))**.** Sei $\mathcal{L}$ die Menge aller durch eine Sprache L darstellbaren syntaktisch korrekten Spezifikationen und Par die Menge von Tupel aller möglichen Parameter des Refactorings. Ein Refactoring $T \in \mathcal{L} \times Par \to \mathcal{L}$ ist eine Programm- beziehungsweise Modelltransformation, die die Qualität von Spezifikationen bezüglich Lesbarkeit, Verständlichkeit und Wartbarkeit erhöht und dabei das durch die Spezifikation festgelegte Schnittstellenverhalten unter Abstraktion nicht ändert.

Sei mit S eine syntaktisch korrekte Spezifikation eines Systems in der Sprache L bezeichnet. Des Weiteren sei $\underline{SV} \in \mathcal{L} \to \underline{\mathcal{M}}$ die Interpretationsabbildung, die zu einer Spezifikation das Schnittstellenverhalten der Spezifikation auf der Ebene der Systemschnittstelle liefert und $\underline{\widetilde{\alpha}}_x \in \underline{\mathcal{M}} \to \underline{\mathcal{N}}$ die Abstraktionsabbildung. Sei ferner $Q \in \mathcal{L} \to \mathbb{N}_0$ eine

[1] Als Abstraktion $\underline{\widetilde{\alpha}}_x$ kann auch die Identitätsabbildung $id \in \underline{\mathcal{M}} \to \underline{\mathcal{M}}$ mit $id(m) = m$ verwendet werden, falls das Verhalten ohne Abstraktion beobachtet werden soll.

Metrik zur Bewertung der Qualität einer Spezifikation.[2] Für das Refactoring T gilt:

$$\exists\, S \in \mathcal{L}, P \in Par :$$
$$\Big(verhaltensinvarianz(T,S,P) \overset{\text{def}}{=} \underset{\sim}{\widetilde{\alpha}}_x\,(\underline{SV}(S)) = \underset{\sim}{\widetilde{\alpha}}_x\,(\underline{SV}(T(S,P)))\Big) \wedge$$
$$\Big(h\ddot{o}hereQualit\ddot{a}t(T,S,P) \overset{\text{def}}{=} Q(S) < Q(T(S,P))\Big) \tag{4.2}$$

Die vorangehend aufgeführte Definition fordert nicht, dass ein Refactoring die Eigenschaften *verhaltensinvarianz* und *höhereQualität* für alle Spezifikationen, die in einer Programmier- oder Modellierungssprache ausdrückbar sind, erfüllen muss. Es reicht an dieser Stelle aus, dass eine Spezifikation S existiert, auf das das Refactoring angewendet werden kann und sich hierbei das abstrakte Schnittstellenverhalten der Spezifikation nicht ändert. Gleiches gilt für die Forderung der Steigerung der Qualität der Spezifikation.

Aus praktischer Sicht ist ein Refactoring eine Modelltransformation zur Qualitätsverbesserung, die eine Art *Best Practice* darstellt und daher in vielen Fällen anwendbar sein soll. In Erweiterung zu der vorgehend aufgeführten formalen Definition von Refactoring wird zusätzlich gefordert, dass die Verhaltensinvarianz- und Qualitätssteigerungseigenschaft nicht nur für eine einzelne Spezifikation gilt, sondern dass eine große Menge von Spezifikationen existiert, für die die Refactoring Eigenschaften gültig sind.

Es ist oft möglich eine syntaktische Vorbedingung über die Spezifikation und die Parameter der Refactoring-Operation festzulegen, die erfüllt sein muss, damit ein Refactoring ausgeführt werden darf. Refactoring-Operationen, die die Verhaltensinvarianzeigenschaft für alle syntaktisch korrekten durch die Sprache ausdrückbaren Spezifikationen und Parameter, für die die Vorbedingung gültig ist, erfüllen, werden in dieser Arbeit als *allgemeingültig* bezeichnet.

Definition 4.4 (Allgemeingültiges Refactoring)**.** Sei $Pre_T : \mathcal{L} \times Par \rightarrow \{wahr, falsch\}$ die syntaktische Vorbedingung, die von einer Spezifikation erfüllt sein muss, damit das Refactoring angewendet wird. Ein allgemeingültiges Refactoring $T : \mathcal{L} \times Par \rightarrow \mathcal{L}$ ist ein Refactoring für das gilt:

$$(\forall\, S \in \mathcal{L}, P \in Par : Pre_T(S,P) \Rightarrow verhaltensinvarianz(T,S,P)) \wedge$$
$$(\exists\, S \in \mathcal{L}, P \in Par : h\ddot{o}hereQualit\ddot{a}t(T,S,P)) \tag{4.3}$$

Selbst für allgemeingültige Refactorings wird nicht gefordert, dass die Qualität unabhängig von der betrachteten Spezifikation durch die Anwendung des Refactorings gesteigert wird. Es reicht hier weiterhin aus, dass Spezifikationen existieren, deren Quali-

[2] Die Metrik Q liefert als Ergebnis höhere Werte bei höherer Qualität. Die Qualitätsmetrik kann durch den Entwickler ausgewählt werden. Ein Beispiel für eine Qualitätsmetrik für AutoFocus Modellspezifikationen ist in Abschnitt 9.2 angeführt.

tät durch die Anwendung des Refactorings verbessert wird. Eine Forderung der Qualitätssteigerung aller Spezifikationen durch ein Refactoring ist praxisfremd. So lassen sich in der Praxis fast immer spezielle Beispiele für Spezifikationen finden, deren Qualität durch die Anwendung eines Refactorings verschlechtert wird.

Allgemeingültige Refactorings garantieren, die Erfüllung der Vorbedingung und eine korrekte Anwendung vorausgesetzt, automatisch die Verhaltensäquivalenz unter Abstraktion. Die Erfüllung der allgemeinen Verhaltensinvarianzeigenschaft kann bei diesen Refactorings unabhängig von konkreten Instanzmodellen gezeigt werden. Bei werkzeuggestütztem Refactoring kann die korrekte Anwendung der Refactoring-Operation garantiert werden. Es ist hier folglich keine Überprüfung der Verhaltensäquivalenz auf Basis der konkreten Instanzmodelle vor und nach Anwendung des Refactorings erforderlich. Dies stellt einen großen Vorteil allgemeingültiger Refactorings gegenüber nicht allgemeingültigen Refactorings dar.

Die vorangehend aufgeführte Definition von Refactoring betrachtet das Ändern von Spezifikationen von Gesamtsystemen und berücksichtigt dabei nicht die Änderung von Spezifikationen von Teilsystemen beziehungsweise Komponenten. Das Refactoring von Komponenten ist definiert als:

Definition 4.5 (Refactoring von Komponenten)**.** Ein Refactoring von Komponenten ist ein Refactoring angewendet auf die Spezifikation eines Teilsystems, dass das abstrahierte Schnittstellenverhalten des Gesamtsystems nicht verändert.

Sei $S \in \mathcal{L}$ eine syntaktisch korrekte Spezifikation eines Systems in der Sprache L, $K \subseteq S$ die Spezifikation einer Komponente des Systems, $S \setminus K$ die Spezifikation des Systems ohne diese Komponente und $\otimes$ die Komposition zweier Spezifikationen zu einer Gesamtspezifikation. Dann gilt für ein Refactoring $T : \mathcal{L} \times Par \rightarrow \mathcal{L}$, das auf die Spezifikation einer Komponente angewendet wird:

$$\begin{gathered} \exists\, S \in \mathcal{L}, K \subseteq S, P \in Par : \\ \Big(verhaltensinvarianzK(T,S,K,P) \stackrel{\text{def}}{=} \widetilde{\alpha}_x(\underline{SV}(S)) = \widetilde{\alpha}_x(\underline{SV}(T(K,P) \otimes (S \setminus K))) \Big) \wedge \\ h\ddot{o}hereQualit\ddot{a}t(T,K,P) \end{gathered} \tag{4.4}$$

In dieser Definition wird die lokale Änderung einer Komponente durch das Refactoring betrachtet. Obwohl nur ein Teil der Spezifikation von der Änderung betroffen ist, muss weiterhin die Verhaltensäquivalenz vor und nach Anwendung des Refactorings auf der Ebene des Systemverhaltens gezeigt werden. Der Grund hierfür ist die Möglichkeit, dass lokale Veränderungen des Verhaltens, die unter der Abstraktion auf Komponentenebene nicht sichtbar sind, eine Veränderung des unter Abstraktion sichtbaren Gesamtverhaltens bewirken können.

Bei der Qualitätsbewertung gehen wir in der Definition davon aus, dass die Metrik kompositional bezüglich der Bewertungsergebnisse von Teilspezifikationen ist, das

heißt eine Steigerung der Qualität von Teilspezifikationen ohne Änderung der Qualität der Restspezifikation zu einer Steigerung der Qualität der Gesamtspezifikation führt. Da das Refactoring von Komponenten nur die Teilspezifikation verändert und die Restspezifikation unverändert lässt, reicht es an dieser Stelle aus, nur die Qualitätssteigerung der Teilspezifikation zu fordern.

4.3. Refactoring von AutoFocus Modellspezifikationen

In den vorangehend vorgestellten Definitionen von Refactoring wird allgemein von Abstraktion und Schnittstellenverhalten gesprochen. Wir definieren jetzt den Begriff des Refactoring speziell für die Modellierungssprache AutoFocus. Als Beobachtungsbegriff wird das nicht abstrahierte AutoFocus Schnittstellenverhalten als auch das Verhalten unter den Zeitabstraktionen $\underline{\widetilde{\alpha}}_{ZA1}$, $\underline{\widetilde{\alpha}}_{ZA2}$ und $\underline{\widetilde{\alpha}}_{ZA3}$ (siehe Abschnitt 3.7) verwendet.

Definition 4.6 (Refactoring von AutoFocus Komponentenspezifikationen unter Zeitabstraktion)**.** Sei $\mathcal{L}_A$ die Menge aller syntaktisch korrekten Spezifikationen, die sich in der Modellierungssprache AutoFocus ausdrücken lassen und *Par* die Menge von Tupel aller Parameter des Refactorings.

Ein Refactoring von AutoFocus Modellen $T : \mathcal{L}_A \times Par \rightarrow \mathcal{L}_A$ ist eine Modelltransformation zur Änderung von Softwaremodellen, die in der Modellierungssprache AutoFocus vorliegen. Die Transformation wird angewendet auf eine syntaktisch korrekte Spezifikation einer Komponente innerhalb einer syntaktisch korrekten Spezifikation eines Gesamtsystems. Sie liefert als Ergebnis eine geänderte syntaktisch korrekte Spezifikation des Teilsystems in der Sprache AutoFocus. Durch Anwendung des AutoFocus Refactoring auf die Spezifikation einer Komponente wird die Qualität der Spezifikation des Gesamtsystems bezüglich Lesbarkeit, Verständlichkeit und Wartbarkeit verbessert und das Schnittstellenverhalten des Gesamtsystems unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_x$ bleibt unverändert.

Sei $S \in \mathcal{L}_A$ eine syntaktisch korrekte Spezifikation eines Systems in der Sprache AutoFocus, $K \subseteq S$ die Spezifikation einer Komponente des Systems, $S \setminus K$ die Spezifikation des Systems ohne die Spezifikation der Komponente K und $\otimes_A : \mathcal{L}_A \times \mathcal{L}_A \rightarrow \mathcal{L}_A$ die Komposition zweier AutoFocus Spezifikationen zu einer Gesamtspezifikation.

Die verwendete Eigenschaft *verhaltensäquivalentAF_*α_x ist in der Definition 3.22 auf Seite 79 festgelegt. Sei ferner $Q_A : \mathcal{L}_A \rightarrow \mathbb{N}_0$ eine kompositionale Metrik zur Bewertung der Qualität einer Spezifikation in der Sprache AutoFocus. Dann gilt für ein AutoFocus Refactoring T, das auf die Spezifikation einer Komponente in der Sprache AutoFocus angewendet wird:

$$\exists\, S \in \mathcal{L}_A, K \subseteq S, P \in Par :$$

$$\big(\mathit{verhaltensinvarianzAF_}\alpha_x(T, S, K, P) \overset{\text{def}}{=}$$

$$\mathit{verhaltensäquivalentAF_}\alpha_x(S, T(K,P) \otimes_A (S \setminus K))) \wedge$$
$$\Big(\mathit{höhereQualitätAF}(T,K,P) \stackrel{\text{def}}{=} Q_A(T(K,P)) > Q_A(K)\Big) \qquad (4.5)$$

Wie bei der Definition 4.5 auf Seite 86 muss auch beim AutoFocus Refactoring angewendet auf Komponentenspezifikationen das Verhalten des Gesamtsystems betrachtet werden. Zur Qualitätsbewertung reicht auf Grund der Kompositionalität der Bewertungsmetrik die Betrachtung der durch das Refactoring geänderten Teilspezifikation aus.

Die Definition 4.4 auf Seite 85 von allgemeingültigen Refactorings ist unter Verwendung der speziellen Eigenschaft $\mathit{verhaltensinvarianzAF_}\alpha_x$ ebenfalls für AutoFocus Refactorings gültig.

In der Definition von AutoFocus Refactorings wurde als Beobachtungsbegriff das Schnittstellenverhalten unter Zeitabstraktion eingesetzt. Es ist auch möglich, Verhaltensäquivalenz ohne Zeitabstraktion für das Refactoring zu fordern.

Definition 4.7 (Refactoring von AutoFocus Komponenten ohne Zeitabstraktion)**.** Ein Refactoring von AutoFocus Komponenten ohne Veränderung des zeitlichen Verhaltens der Komponenten ist ein AutoFocus Refactoring, für das statt der Eigenschaft $\mathit{verhaltensinvarianzAF_}\alpha_x$ die Eigenschaft *verhaltensinvarianzAF_ID* gefordert wird. Diese Eigenschaft verwendet die Eigenschaft *verhaltensäquivalentAF* (Definition 3.14 auf Seite 60) auf der AutoFocus Komponente vor und nach der Änderung.

$$\mathit{verhaltensinvarianzAF_ID}(T,K,P) \stackrel{\text{def}}{=}$$
$$\mathit{verhaltensäquivalentAF}(K, T(K,P)) \qquad (4.6)$$

Ist das nicht abstrahierte Schnittstellenverhalten einer Komponentenspezifikation vor und nach dem Refactoring identisch, dann impliziert die Verhaltensäquivalenz der Komponente auf Grund der Kompositionalität der Semantik von AutoFocus die Verhaltensäquivalenz des gesamten Systems. Es reicht der Nachweis der Äquivalenz der Schnittstellenverhalten der durch das Refactoring geänderten Komponenten aus, um die Verhaltensäquivalenz des Gesamtsystems zu zeigen.

Die in den Definitionen von Refactoring von AutoFocus Modellen verwendeten Eigenschaften *verhaltensäquivalentAF* und $\mathit{verhaltensäquivalentAF_}\alpha_x$, die den Begriff der Gleichheit des beobachteten Verhaltens festlegen, fordern beide neben der Äquivalenz des Schnittstellenverhaltens auch die Äquivalenz beider in Focus übersetzten syntaktischen Schnittstellen. Die Übersetzung legt fest, dass als Eingabekanalbezeichner nicht der AutoFocus Eingabe-Port-Bezeichner der betrachteten Komponente, sondern der AutoFocus Bezeichner des mit diesem Port über AutoFocus Kanäle verbunden Ausgabe-Ports der anderen Komponente verwendet wird. Durch die Forderung der Gleichheit der syntaktischen Schnittstellen ist auf Grund dieser Eigenschaft der Über-

setzung sichergestellt, dass unter Ausschluss von Änderungen von AutoFocus-Port-Bezeichnern zwei zu vergleichende Komponentenspezifikationen äquivalent mit der diese Spezifikationen umgebenden Restspezifikation über AutoFocus Kanäle verschaltet sind.

5. Strukturelles Refactoring ohne Änderung des zeitlichen Verhaltens

In Kapitel 4 wurden verschiedene Arten von AutoFocus Refactoring definiert. Eine besondere Klasse von Refactorings stellen die Refactorings von AutoFocus Komponenten dar, die das zeitliche Verhalten der Komponentenschnittstelle unverändert lassen (siehe Definition 4.7 auf Seite 88). Zum Verhaltensäquivalenznachweis reicht bei diesen Refactorings die Betrachtung des Schnittstellenverhaltens der veränderten Komponente aus.

Refactorings, die das nicht abstrahierte Verhalten unverändert lassen, sind in vielen Fällen rein strukturelle Spezifikationsänderungen in einer hierarchischen Modellierungssprache, wobei die Hierarchie hier keine semantische Bedeutung hat, sondern zur Unterteilung von Diagrammen in Unterdiagramme, mit dem Ziel der besseren Lesbarkeit der Spezifikation, dient. AutoFocus besitzt das Konzept der Hierarchie in den Systemstrukturdiagrammen (SSDs) und den Zustandsübergangsdiagrammen (STDs).

Die nachfolgend vorgestellten Refactorings werden nach einem festen Schema beschrieben. Das Schema orientiert sich an der von Fowler für Refactorings verwendeten Aufteilung [FBB^{+}99].

Name
: Ein eindeutiger Name in englischer Sprache zur Bezeichnung der Refactoring-Operation.

Problem
: Eine kurze Beschreibung des Problems, das durch die Anwendung des Refactorings gelöst wird.

Lösung
: Eine Kurzbeschreibung, wie das Refactoring das Problem löst.

Methode
: Eine ausführliche Schritt für Schritt Beschreibung der Refactoring-Operation.

In den folgenden Abschnitten werden AutoFocus Refactorings für die Veränderung der hierarchischen Beziehungen festgelegt. In Abschnitt 5.1 wird das Refactoring von hierarchischen Systemstrukturdiagrammen (SSDs) betrachtet. Der Abschnitt 5.2 beschreibt das Refactoring der Hierarchie in Zustandsmaschinen (STDs).

5.1. Refactoring von AutoFocus Systemstrukturdiagrammen

In AutoFocus Systemstrukturdiagrammen (SSDs) wird die Aufteilung der Software in Komponenten spezifiziert. Komponenten können wiederum aus Teilkomponenten zusammengesetzt werden. Hierdurch wird eine hierarchische Strukturierung des Systems ermöglicht. Neben der Festlegung von Komponenten lassen sich in SSDs auch Komponentenschnittstellen und die Kommunikationsbeziehungen zwischen Komponenten festlegen.

In AutoFocus SSDs wird die logische Architektur eines Systems spezifiziert. Soll ein bestehendes System erweitert werden, sind in den meisten Fällen Architekturänderungen erforderlich. Für das Verändern der hierarchischen Strukturierung eines Systems unter Beibehaltung der atomaren Komponenten, das heißt Komponenten mit zugewiesenem Verhalten, werden folgende Refactoring-Operationen definiert:

Push Down Component (Abschnitt 5.1.1).
: In einem SSD wird eine Komponente in eine andere nicht atomare Komponente des selben SSDs verschoben. Hierdurch wird die Komponente in der Systemstruktur um eine Hierarchieebene nach unten verschoben.

Pull Up Component (Abschnitt 5.1.2).
: Eine Komponente wird in der Systemstruktur um eine Hierarchieebene nach oben verschoben.

Wrap Components (Abschnitt 5.1.3).
: Eine Menge von Komponenten innerhalb eines SSDs wird in einer neuen Komponente zusammengefasst.

Move Component (Abschnitt 5.1.4).
: Eine Komponente wird an eine beliebige andere Stelle innerhalb der Systemstruktur verschoben.

Remove Single Component Hierarchy (Abschnitt 5.1.5).
: Eine komplette Hierarchieebene, die nur aus einer einzigen Komponente besteht, wird aus der Systemstruktur entfernt.

Durch eine kombinierte Anwendung der hier aufgeführten Refactorings lässt sich die hierarchische Strukturierung eines Systems, unter Beibehaltung der atomaren Komponenten mit deren Schnittstellen und deren Verhalten, beliebig abändern. Das nicht abstrahierte Systemverhalten vor und nach der Anwendung der hier aufgeführten Refactorings ist nach der zeitsynchronen AutoFocus Semantik, die in Kapitel 3 angegeben ist, und nach der in dieser Arbeit in Kapitel 6 festgelegten zeitasynchronen AutoFocus Semantik für alle möglichen Spezifikationen und Parameter, die die Vorbedingungen der Refactorings erfüllen, identisch. Somit sind diese Refactorings allgemeingültig (Definition 4.4 auf Seite 85). Der Nachweis dieser Eigenschaft erfolgt in Abschnitt 5.1.6.

Die hier vorgestellten Refactorings von Systemstrukturdiagrammen berücksichtigen Kommunikationsbeziehungen zwischen jeweils genau zwei Ports. In AutoFocus ist es

möglich, einen Ausgabe-Port über mehrere Kanäle mit mehreren Eingabe-Ports zu verbinden. Diese $1 \rightarrow n$ Kommunikationsbeziehung wird in den vorgestellten Refactorings nicht berücksichtigt.

5.1.1. Push Down Component Refactoring

Problem

Die Komponente *A* gehört funktional zu einer anderen Komponente *B*, wobei sich diese beiden Komponenten in dem selben SSD *C* befinden. Die Komponente *B* ist hierbei keine atomare Komponente, das heißt die Komponente *B* ist in weitere Unterkomponenten zerteilt.

Lösung

Verschieben der Komponente *A* um eine Hierarchieebene nach unten in das SSD der Komponente *B* (Abbildung 5.1).

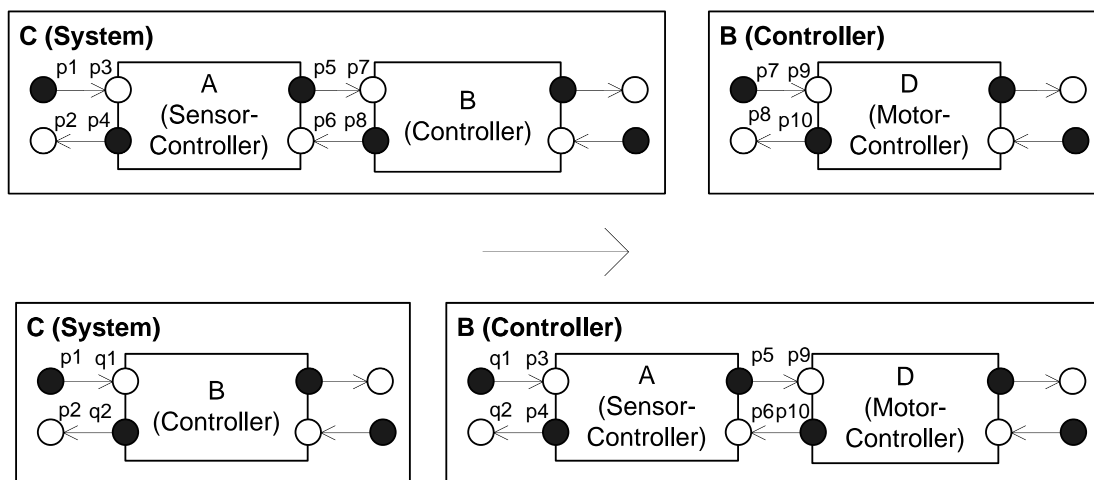


Abbildung 5.1.: Push Down Component Refactoring.

Motivation

Die hierarchische Dekomposition von Systemen in Komponenten und Unterkomponenten in AutoFocus SSDs ermöglicht es, Systeme in einer übersichtlichen und verständlichen Art zu strukturieren. Durch die Veränderung bestehender AutoFocus Modelle wird jedoch die Qualität der Systemarchitektur häufig verschlechtert. Neue Funk-

tionalität wird häufig an Stellen in der Architektur eingefügt, die eine möglichst schnelle und kostengünstige Realisierung erlauben. Weiterführende qualitätssteigernde Architekturveränderungen werden häufig zunächst vermieden.

Das *Push Down Component* Refactoring ist eine elementare Transformationsregel, die es ermöglicht, eine Komponente aus einem SSD in eine andere Komponente innerhalb des selben Diagramms zu verschieben. Falls Sie eine Komponente in einem SSD finden, die funktional zu einer anderen nicht atomaren Komponente innerhalb des gleichen Diagramms gehört, dann führen Sie das *Push Down Component* Refactoring aus. Durch die Anwendung des Refactorings wird die Komponente eine Hierarchieebene nach unten verschoben. Die Abbildung 5.1 auf der vorherigen Seite zeigt die Refactoring-Transformation, die die Komponente *A* in das SSD der Komponente *B* verschiebt.

Nachfolgend wird ein konkretes Beispiel für die Anwendung des *Push Down Component* Refactorings gezeigt. Angenommen es existieren in einem SSD die Komponenten *Controller* und *SensorController* (siehe Abbildung 5.1 auf der vorherigen Seite). Der Sensor Controller steuert die Messdatenverarbeitung. Die Controller Komponente übernimmt allgemeine Steueraufgaben einschließlich der Motorsteuerung. Die Funktionalität des Sensor Controllers ist aus logischer Sicht eine Unterfunktionalität der allgemeinen Controller Komponente. Es wird das *Push Down Component* Refactoring angewendet, um die Sensor Controller Komponente an die aus logischer Sicht passende Stelle als Teil der allgemeinen Controller Komponente zu verschieben.

Methode

Die Durchführung des *Push Down Component* Refactorings wird in einzelnen Teilschritten beschrieben. Darüber hinaus ist das Refactoring in Java unter Verwendung der AutoFocus Modell API im Anhang in Abschnitt B.1 definiert.

Nehmen wir an, es existiert eine Komponente *A* und eine Komponente *B* in einem SSD *C* (Abbildung 5.1 auf der vorherigen Seite). Die Komponente *B* ist keine atomare Komponente. Wir möchten die Komponente *A* in die Komponente *B* verschieben.

Benutzereingaben: Der Benutzer wählt eine Komponente *A*, die verschoben werden soll und eine nicht atomare Komponente *B* als Ziel der Verschiebeoperation.

Vorbedingungen:

1. Die gewählten Komponenten *A* und *B* müssen voneinander verschieden sein.
2. Beide Komponenten *A* und *B* müssen sich im selben SSD befinden.
3. Bei der Zielkomponente *B* darf es sich um keine atomare Komponente handeln.

Ausgangspunkt für die Beschreibung des Refactorings ist die in Abbildung 5.1 auf der vorherigen Seite oben gezeigte Komponentenstruktur.

Schritt 1: Zunächst wird eine Kopie *A*2 der Komponente *A* als Unterkomponente der Komponente *B* erzeugt. Hierbei wird die Unterstruktur von *A* als auch die darin enthaltenen Assoziationen von Komponenten zu Zustandsmaschinen mitkopiert (siehe Abbildung 5.2).

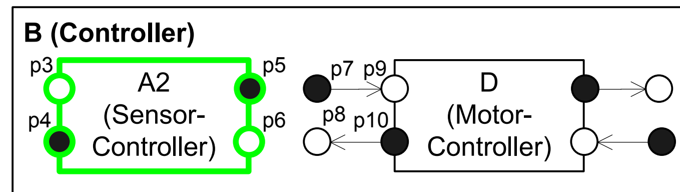


Abbildung 5.2.: Schritt 1: Komponente *A* als *A*2 in SSD *B* kopieren.

Schritt 2: Alle Kanäle, die zwischen Ports der Komponente *A* und den Komponenten des SSD *B* verlaufen, werden umgeleitet. Es entsteht eine Verbindung zwischen den Ports der neuen Komponente *A*2 und den anderen Komponenten in SSD *B*. Die durch die Umleitung nicht mehr benötigten Ports von *B* werden entfernt.

Schritt 2a: Die Kanäle zwischen den Komponenten *A* und *B* in dem SSD *C* werden entfernt (siehe Abbildung 5.3).

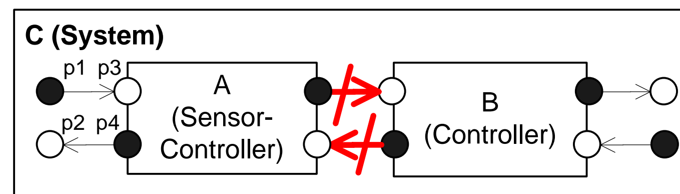


Abbildung 5.3.: Schritt 2a: Entfernen der Kanäle zwischen *A* und *B* in SSD *C*.

Schritt 2b: Die Kanäle, die mit nicht mehr benötigten Ports der Komponente *B* verbunden sind, werden zu Ports der neuen kopierten Komponente *A*2 umgeleitet. Die nicht mehr benötigten Ports der Komponente *B* werden gelöscht (siehe Abbildung 5.4 auf der nächsten Seite).

Schritt 3: Alle Kanäle, die mit den Ports der Komponente *A*, nicht aber mit den Ports der Komponente *B*, verbunden sind, werden umgeleitet. Die Kanäle werden hierbei von den Ports der Komponente *A* zu den Ports der kopierten Komponente *A*2 über die neu hinzugefügten Zwischen-Ports der Komponente *B* umgeleitet.

Schritt 3a: Die Schnittstelle der Komponente *B* wird um bestimmte Ports der Komponente *A* erweitert. Die Schnittstellenerweiterung wird benötigt, um die Kommunikation zwischen der kopierten Komponente *A*2 und den Komponenten außerhalb der Komponente *B* zu ermöglichen (siehe Abbildung 5.5 auf der nächsten Seite).

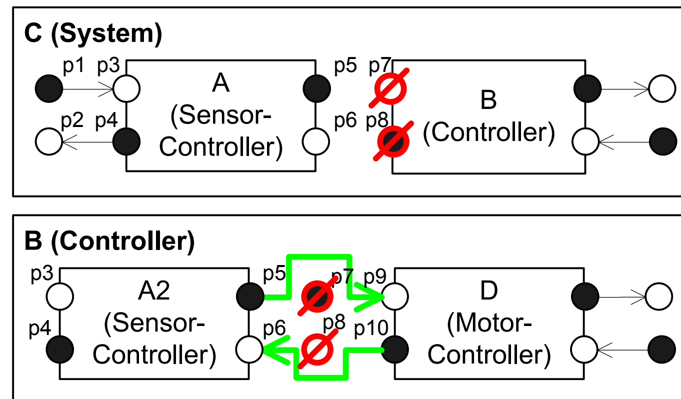


Abbildung 5.4.: Schritt 2b: Umleiten der Kanäle von nicht mehr benötigten Ports in *B*.

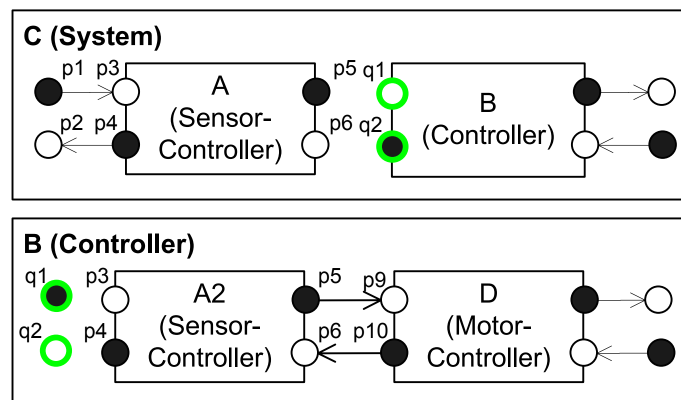


Abbildung 5.5.: Schritt 3a: Schnittstellenerweiterung der Komponente *B*.

Schritt 3b: Die betrachteten Kanäle werden von den Ports der Komponente A zu den neuen Zwischen-Ports der Komponente B in dem SSD C umgeleitet. Es werden neue Kanäle zwischen den neuen Ports der Komponente B und den Ports der kopierten Komponente $A2$ in dem SSD B eingefügt (siehe Abbildung 5.6).

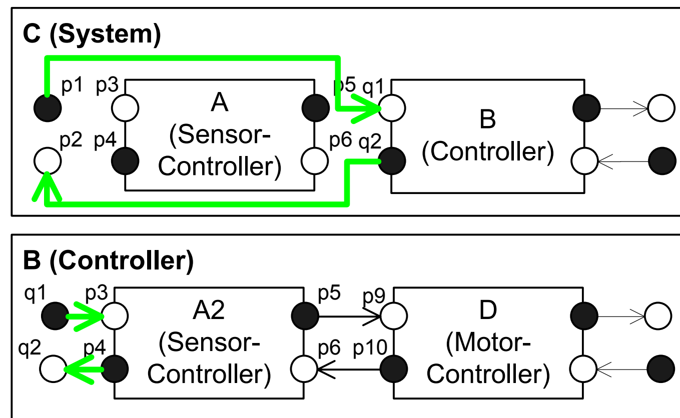


Abbildung 5.6.: Schritt 3b: Umleiten der Kanäle über neue Ports der Komponente B.

Schritt 4: Die originale Komponente A wird gelöscht (siehe Abbildung 5.7).

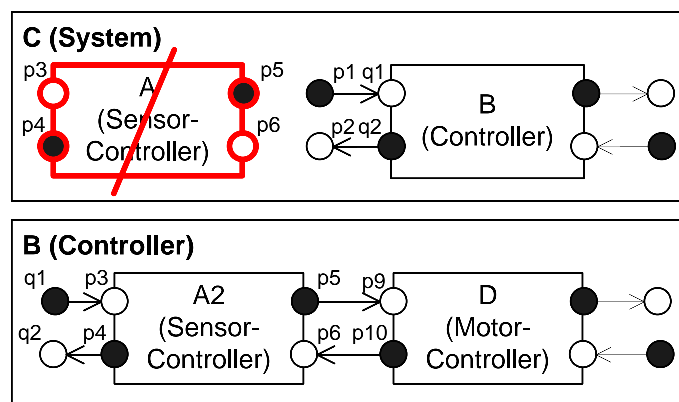


Abbildung 5.7.: Schritt 4: Löschen der originalen Komponente A.

Die Refactoring-Operation *Push Down Component* ist nach Schritt 4 abgeschlossen. Die Systemstruktur nach dem Refactoring ist in der Abbildung 5.1 auf Seite 93 unten dargestellt.

5.1.2. Pull Up Component Refactoring

Problem

Die Komponente *A* gehört funktional nicht zur Komponente *B*, in der *A* enthalten ist, sondern zur Komponente *C*, die die Super-Komponente von *B* ist.

Lösung

Verschieben der Komponente *A* um eine Hierarchieebene nach oben in das SSD der Komponente *C* (Abbildung 5.8).

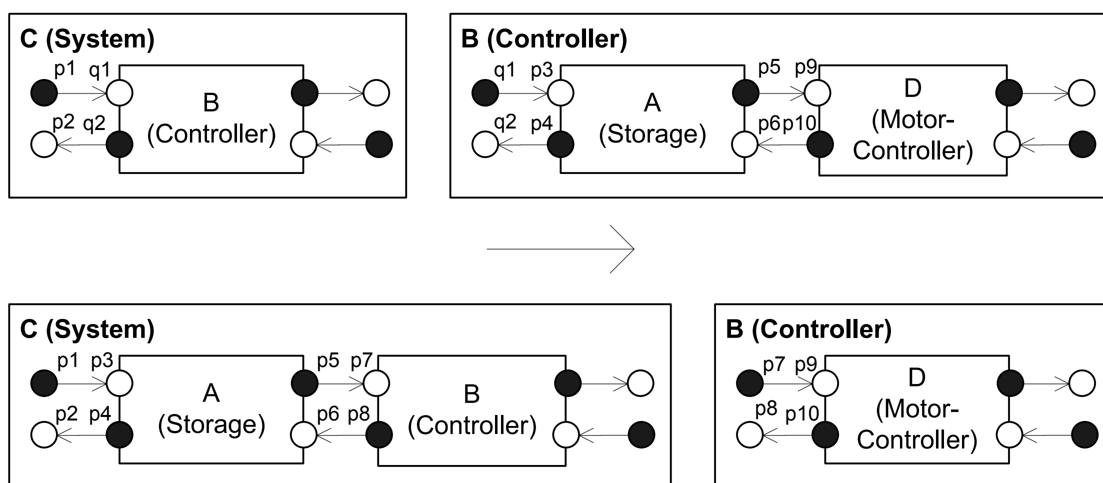


Abbildung 5.8.: Pull Up Component Refactoring.

Motivation

Das *Pull Up Component* Refactoring ist eine elementare Transformationsregel, die es ermöglicht, eine Komponente um eine Hierarchieebene nach oben in das SSD ihrer Super-Super-Komponente zu verschieben. Das Refactoring stellt die Umkehrabbildung des in Abschnitt 5.1.1 vorgestellten *Push Down Component* Refactorings dar. Falls in einem SSD eine Komponente existiert, die aus funktionaler Sicht nicht zu ihrer Super-Komponente passt, aber zu ihrer Super-Super-Komponente, dann wird das *Pull Up Component* Refactoring angewendet. Durch die Anwendung des Refactorings wird die betrachtete Komponente eine Hierarchieebene innerhalb der Systemstruktur nach oben in das SSD der Super-Super-Komponente verschoben.

Nachfolgend wird ein konkretes Beispiel für die Anwendung des *Pull Up Component* Refactorings angegeben: Die Komponente *Controller* besteht aus den Komponenten *Mo-*

torController und *Storage*. Die *Storage* Komponente übernimmt keine Kontrollaufgaben, sondern ist für die Datenspeicherung zuständig. Sie gehört aus funktionaler Sicht nicht zur *Controller* Komponente. Durch Anwendung des *Pull Up Component* Refactorings wird die *Storage* Komponente aus der *Controller* Komponente in die *System* Komponente verschoben. Dort ist aus funktionaler Sicht der korrekte Ort für diese Komponente.

Methode

Sei A die zu verschiebende Komponente in dem SSD B. Die Komponente B befindet sich wiederum im SSD C.

Benutzereingaben: Der Benutzer wählt eine Komponente A innerhalb eines SSDs B aus, die eine Hierarchieebene nach oben verschoben werden soll.

Vorbedingungen: Damit das Refactoring auf die Komponente A angewendet werden kann, muss zu dieser Komponente eine Super-Super-Komponente existieren. Die Komponente A darf sich folglich weder in dem SSD auf oberster Hierarchieebene befinden, noch selber die in der Hierarchie oberste Komponente sein.

Schritt 1: Die zu verschiebende Komponente A wird in das SSD C kopiert. Hierbei werden alle Unterkomponenten von A und deren Verhaltensbeschreibungen mit kopiert (siehe Abbildung 5.9).

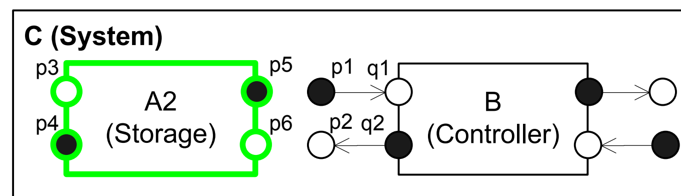


Abbildung 5.9.: Schritt 1: Komponente A in das SSD C kopieren.

Schritt 2: Alle Kanäle, die zwischen den Ports der Komponente A und den Ports des SSD B verlaufen, werden umgeleitet. Es entsteht eine direkte Verbindung zwischen den Ports der neuen Komponente $A2$ und den Ports der anderen Komponenten des SSDs C. Die durch die Umleitung nicht mehr benötigten Ports von B werden entfernt.

Schritt 2a: In dem SSD C werden alle Kanäle, die über Ports der Komponente B mit Ports der Unterkomponente A verbunden sind, zu der neuen Komponente $A2$ umgeleitet (siehe Abbildung 5.10 auf der nächsten Seite).

Schritt 2b: Die nicht mehr benötigten Ports von B, die zur Kommunikation mit A gedient haben, werden zusammen mit den mit ihnen verbundenen Kanälen gelöscht (siehe Abbildung 5.11 auf der nächsten Seite).

Schritt 3: Es werden alle Kanäle umgeleitet, die mit den Ports der Komponente A, nicht aber mit den Ports des SSD B, verbunden sind. Die Kanäle werden von den Ports der

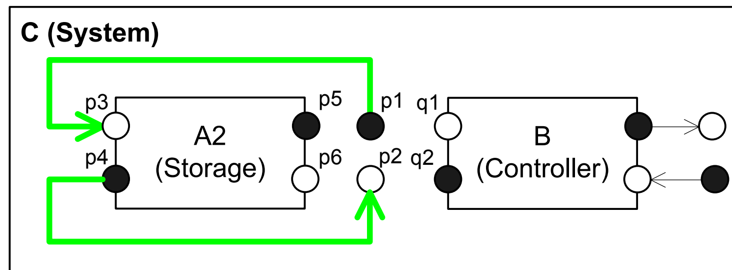


Abbildung 5.10.: Schritt 2a: Umleiten der Kanäle in dem SSD *C*.

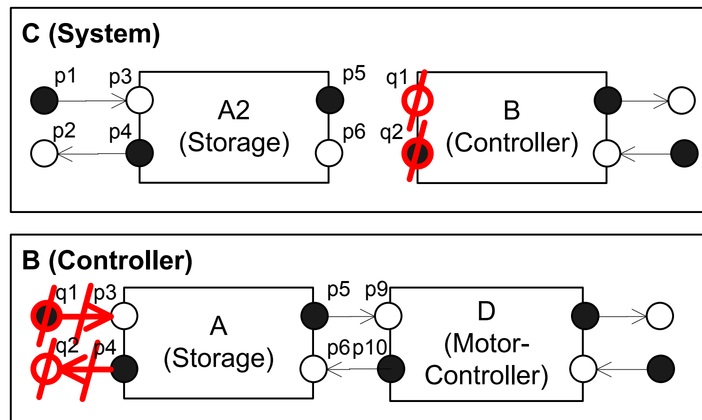


Abbildung 5.11.: Schritt 2b: Entfernen der Ports von *B*.

Komponente *A* über neu hinzugefügte Zwischen-Ports der Komponente *B* zu den Ports der kopierten Komponente *A*2 umgeleitet.

Schritt 3a: Die Schnittstelle von *B* wird um neue Zwischen-Ports, die zur Kommunikation zwischen der neuen Komponente *A*2 und den in *B* enthalten Komponenten benötigt werden, erweitert. Die Komponente *A*2 wird mit den neuen Zwischen-Ports verbunden (siehe Abbildung 5.12).

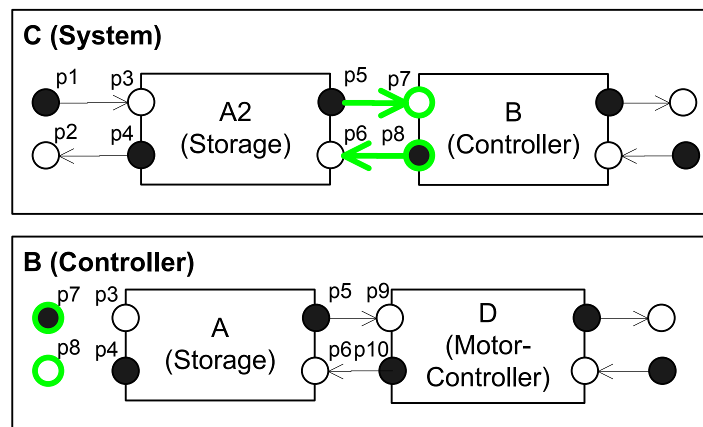


Abbildung 5.12.: Schritt 3a: Schnittstellenerweiterung von *B*.

Schritt 3b: Die Kommunikation zwischen der ursprünglichen Komponente *A* und den anderen Komponenten in dem SSD *B* wird zu den neuen Zwischen-Ports von *B* umgeleitet (siehe Abbildung 5.13).

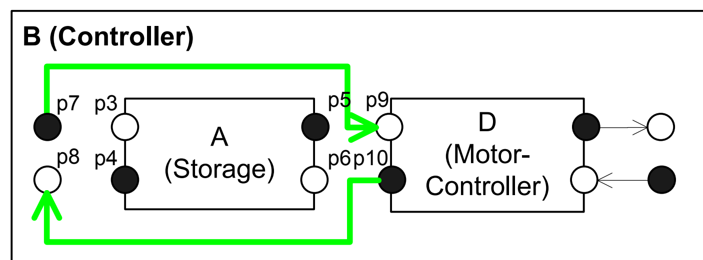


Abbildung 5.13.: Schritt 3b: Umleiten der Kanäle im SSD *B*.

Schritt 4: Die ursprüngliche Komponente *A* wird gelöscht (siehe Abbildung 5.14 auf der nächsten Seite).

Es entsteht die in Abbildung 5.8 auf Seite 98 unten gezeigte Systemstruktur.

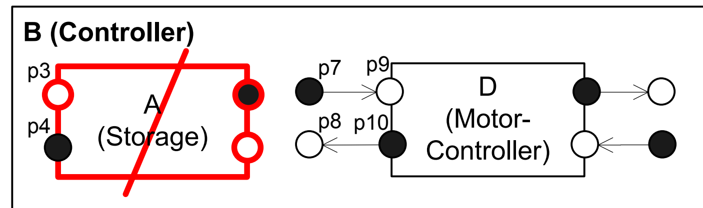


Abbildung 5.14.: Schritt 4: Löschen der ursprünglichen Komponente *A*.

5.1.3. Wrap Components Refactoring

Problem

Ein Systemstrukturdiagramm (SSD) besteht aus zu vielen Komponenten.

Lösung

Fassen Sie mehrere Komponenten des SSDs in einer neuen Unterkomponente zusammen (Abbildung 5.15).

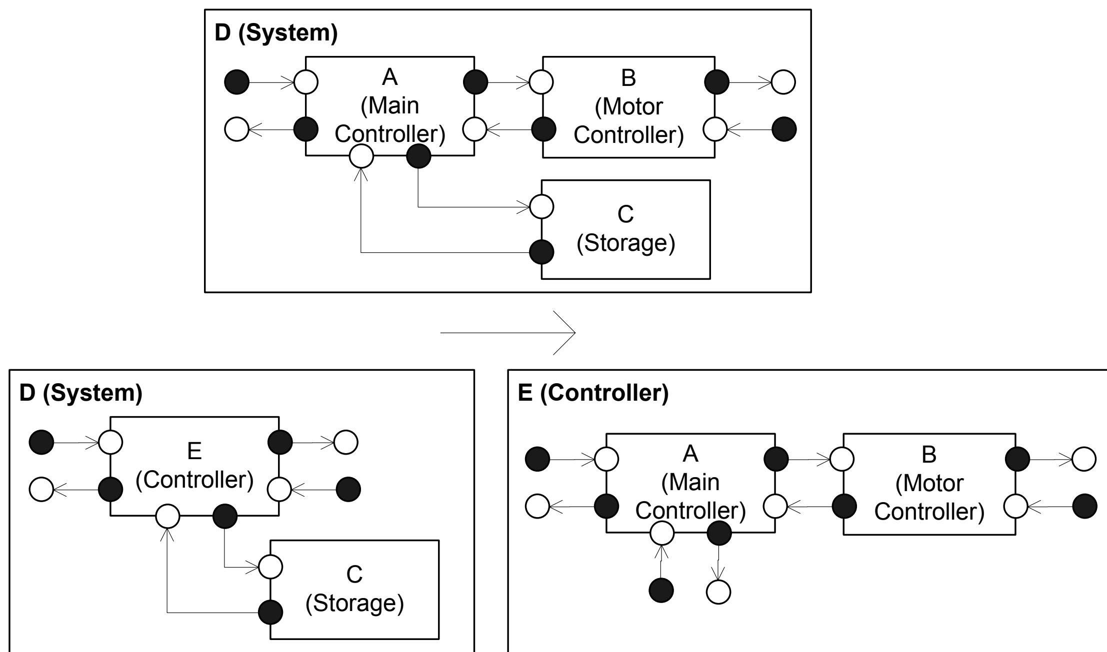


Abbildung 5.15.: Wrap Components Refactoring.

Motivation

Systemstrukturdiagramme mit vielen Komponenten, Ports und Kanälen werden sehr schnell unübersichtlich und dadurch unverständlich. Die hierarchische Dekomposition von Komponenten in Teilkomponenten ermöglicht eine übersichtliche Strukturierung des Systems. Um die Anzahl der graphischen Elemente pro Strukturdiagramm zu reduzieren, ist es vorteilhaft, zusammengehörende Komponenten in einer Super-Komponente zu kapseln. Hierdurch werden die Komponenten auf mehrere Hierarchieebenen von SSDs verteilt und die Anzahl der Komponenten pro SSD reduziert.

Ein Indiz zur Identifikation von zusammengehörenden Komponenten ist der Grad der Kopplung zwischen diesen Komponenten. Viele Kommunikationsbeziehungen beziehungsweise Kanäle zwischen zwei Komponenten deuten darauf hin, dass diese Komponenten sehr stark zusammenarbeiten. Falls in einem SSD sehr vielen Komponenten existieren, dann müssen funktional zusammengehörende Komponenten identifiziert werden. Auf diese Komponenten ist das *Wrap Components* Refactoring anzuwenden.

Das *Wrap Component* Refactoring lässt sich als ein Refactoring zur Realisierung eines Entwurfsmusters sehen. Durch das Zusammenfassen mehrerer Unterkomponenten in einer Komponente wird ähnlich zu dem *Fascade* Entwurfsmuster [GHJV95] für objektorientierte Sprachen eine gemeinsame Schnittstelle für den externen Zugriff auf die Unterkomponenten definiert.

Es existieren zwei Sonderfälle, in denen sich das *Wrap Components* Refactoring etwas zweckentfremdet einsetzen lässt. Das *Push Down Component* Refactoring erlaubt nicht die Wahl einer atomaren Komponente als Zielkomponente. Soll eine Komponente in eine atomare Zielkomponente verschoben werden, dann wird das *Wrap Components* Refactoring gemeinsam auf diese beide Komponenten angewendet. Als Name der durch das *Wrap Components* entstehenden Komponente wird der ursprüngliche Name der atomaren Komponente gewählt und die verschobene atomare Komponente wird entsprechend den Vorgaben des Entwicklers umbenannt.

Ein weiterer Sonderfall, bei dem das *Wrap Components* Refactoring eingesetzt werden kann, ist das Einfügen einer neuen *Top Level* Komponente. Zu diesem Zweck wird das *Wrap Components* Refactoring auf alle in dem obersten SSD enthaltenen Komponenten angewendet. Als Name für die neu entstehende Komponente wird der Name der ursprünglichen *Top Level* Komponente verwendet. Anschließend wird die *Top Level* Komponente entsprechend den Benutzervorgaben umbenannt.

Methode

In Abschnitt B.2 das Anhangs ist das *Wrap Components* Refactoring unter Verwendung der AutoFocus Modell API definiert.

Benutzereingaben: Der Benutzer wählt eine Menge von Komponenten innerhalb eines SSDs, die in einer neuen Komponente zusammengefasst werden sollen. Darüber hinaus

legt der Benutzer einen Namen für die neu entstehende Komponente fest.

Vorbedingungen: Der Benutzer muss mindestens eine Komponente für das Refactoring auswählen. Alle Komponenten, die in der neuen Komponente zusammengefasst werden sollen, müssen sich in dem selben SSD befinden.

Schritt 1: Eine neue leere Komponente wird in dem SSD, in denen sich die zusammenzufassenden Komponenten befinden, erzeugt. Der Name der neuen Komponente wird entsprechend der Benutzereingabe umbenannt.

Schritt 2: Jede vom Benutzer gewählte Komponente wird durch die Anwendung des *Push Down Component* Refactorings (siehe Abschnitt 5.1.1) in die neue Komponente verschoben.

Das *Wrap Components* Refactoring ist somit abgeschlossen.

5.1.4. Move Component Refactoring

Problem

Eine Komponente *A* befindet sich in einem falschen SSD innerhalb der Komponentenhierarchie.

Lösung

Die Komponente *A* wird in das richtige SSD *B* innerhalb der Komponentenhierarchie verschoben.

Motivation

Zum besseren Verständnis der Systemarchitektur werden jeweils funktional zusammengehörende Komponenten in Super-Komponenten zusammengefasst. Existiert eine Komponente in der Systemarchitektur, die nicht funktional zu ihrer Super-Komponente passt, dann muss diese an die passende Stelle in der Komponentenhierarchie verschoben werden. Die Refactorings *Push Down Component* und *Pull Up Component* ermöglichen das Verschieben einer Komponente um eine Hierarchieebene nach unten beziehungsweise nach oben. Ist der korrekte Ort für die zu verschiebende Komponente durch diese Refactorings nicht direkt erreichbar, dann wird das *Move Component* Refactoring angewendet, das eine Komponente an einen beliebigen anderen Ort innerhalb der Komponentenhierarchie verschiebt.

Methode

Das *Move Component* Refactoring lässt sich durch wiederholte Anwendung des *Pull Up Component* Refactorings (Abschnitt 5.1.2) und des *Push Down Component* Refactorings

(Abschnitt 5.1.1) realisieren. In Abschnitt B.3 des Anhangs ist das *Move Component* Refactoring auf Basis der AutoFocus Modell API definiert.

Benutzereingabe: Der Benutzer wählt eine Komponente *A* aus, die verschoben werden soll. Des Weiteren legt der Benutzer eine Komponente *B* als Ziel der Verschiebeoperation fest.

Vorbedingungen: Die Zielkomponente *B* darf nicht atomar sein und beide Komponenten *A* und *B* müssen sich in der selben Komponentenstruktur befinden, das heißt beide Komponenten müssen die selbe *Top Level* Komponente besitzen.

Schritt 1: Die Komponente *A* wird so lange durch Anwendung des *Pull Up Component* Refactorings (Abschnitt 5.1.2) in der Komponentenhierarchie nach oben verschoben, bis sich *A* in einem SSD befindet, in dem auch eine Komponente existiert, deren Unterkomponenten (auch über mehrere Hierarchieebenen hinweg) die Zielkomponente *B* enthält.

Schritt 2: Die Komponente *A* wird so lange durch Anwendung des *Push Down Component* Refactorings (Abschnitt 5.1.1) in der Komponentenhierarchie nach unten in Richtung der Zielkomponente *B* verschoben, bis sich *A* in dem SSD *B* befindet.

Das *Move Component* Refactoring ist abgeschlossen.

5.1.5. Remove Single Component Hierarchy Refactoring

Problem

In der Komponentenhierarchie existieren Komponenten, die aus nur einer Unterkomponente bestehen.

Lösung

Eine komplette Hierarchieebene bestehend aus nur einer Komponente wird aus der Systemstruktur entfernt (Abbildung 5.16 auf der nächsten Seite).

Motivation

Durch das Refactoring der Komponentenstruktur kann es passieren, dass Komponenten entstehen, die nur eine einzige Unterkomponente besitzen. Diese Komponenten verschlechtern die Verständlichkeit der Modellspezifikation, da diese die Verschachtelungstiefe der Komponentenstruktur erhöhen und dabei keinen Beitrag zur Reduzierung der Anzahl von Komponenten pro SSD leisten.

Komponenten, die nur aus einer Unterkomponente bestehen, sollen durch die Anwendung des *Remove Single Component Hierarchy* Refactorings aus der Komponentenhierarchie entfernt werden.

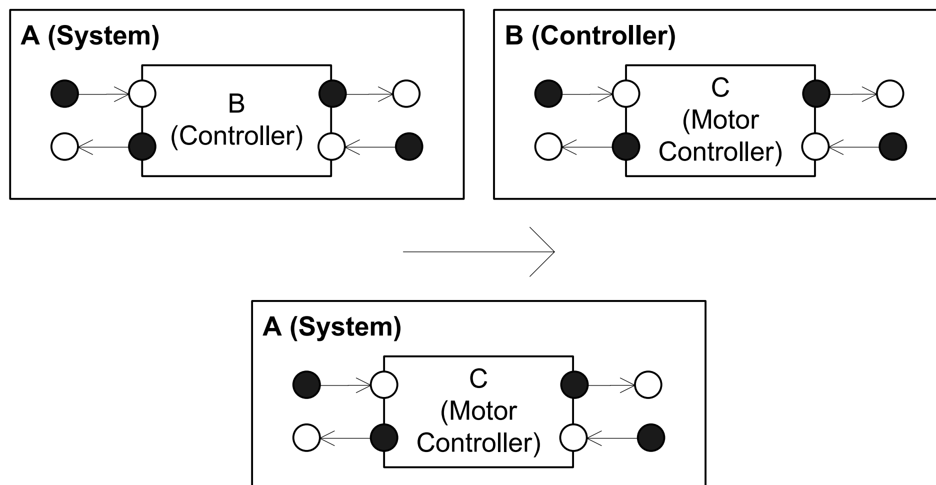


Abbildung 5.16.: Remove Single Component Hierarchy Refactoring

Methode

Das *Remove Single Component Hierarchy* Refactoring ist in Abschnitt B.4 das Anhangs auf Basis der AutoFocus Modell API definiert.

Benutzereingabe: Der Benutzer wählt eine zu entfernende Komponente A.

Vorbedingung: Die Komponente A besitzt nur eine Unterkomponente und A ist nicht die *Top Level* Komponente.

Schritt 1: Die Unterkomponente von A wird durch die Verwendung des *Pull Up Component* Refactorings (siehe Abschnitt 5.1.2) um eine Hierarchieebene nach oben verschoben.

Schritt 2: Die jetzt leere Komponente A wird gelöscht.

5.1.6. Verhaltensinvarianz der SSD Struktur-Refactorings

Die in den vorangehenden Abschnitten definierten Refactorings von Systemstrukturdiagrammen (SSDs) ermöglichen die Neuanordnung von Komponenten innerhalb der Komponentenhierarchie. Es handelt sich hierbei um Refactorings, die das Systemschnittstellenverhalten auch ohne Anwendung von Zeitabstraktion nicht verändern (siehe Definition 4.7 auf Seite 88). Bei dieser Art von Refactorings reicht zum Nachweis der Verhaltensinvarianz des Systems der Nachweis der Invarianz des Verhaltens der durch das Refactoring veränderten Komponente aus. Da es sich hier um allgemeingültige Refactorings (siehe Definition 4.4 auf Seite 85) handelt, erfolgt der Nachweis auf Basis der Semantik der Modellierungssprache.

Betrachten wir die Veränderungen, die durch das Refactoring *Push Down Component* (Abschnitt 5.1.1) bewirkt werden. Zum Verschieben wird die Komponente *A* einschließlich aller Unterkomponenten und zugewiesenen Zustandsmaschinen kopiert. Die kopierte Komponente *A*2 verhält sich somit identisch zur ursprünglichen Komponente.

Die Kommunikation zwischen *A* und der Zielkomponente *B* wird nach der Verschiebeoperation direkt innerhalb des SSDs *B* durch Umleitung der Kanäle zu *A*2 realisiert. Die nicht mehr benötigten Ports von *B* werden hierbei gelöscht. Die Kanäle zwischen der zu verschiebenden Komponente *A* und dem restlichen System werden über neue Ports der Zielkomponente *B* zu der Komponente *A*2 umgeleitet.

Es werden zu den Ports der Ursprungskomponente *A* jeweils die äquivalenten, das heißt gleich benannten, Ports der kopierten Komponente *A*2 ermittelt und die Umleitung der Kanäle erfolgt jeweils von Ports von *A* an die äquivalenten Ports von *A*2. Demnach ist sichergestellt, dass die neue Komponente *A*2 über Kanäle genau mit den gleichen Ports der die verschobenen Komponente umgebenden atomaren Komponenten, über eventuell eingefügte Zwischenports über mehrere Hierarchieebenen hinweg, verbunden ist, wie die ursprüngliche Komponente *A* vor Anwendung des Refactorings. Somit ist die von der Verhaltensäquivalenzeigenschaft (Definition 3.14 auf Seite 60) geforderte Gleichheit der syntaktischen Schnittstelle (Definitionen 3.2 auf Seite 52 und 3.6 auf Seite 54) der Komponentenspezifikation vor und nach dem Refactoring erfüllt.

Nach der Semantik der AutoFocus Modellierungssprache (siehe Kapitel 3 und [HSE97]) wird die Kommunikation zwischen zwei atomaren Komponenten jeweils genau um einen Systemtakt verzögert. Hierbei ist es irrelevant, über wie viele Zwischen-Ports nicht atomarer Komponenten beziehungsweise Hierarchieebenen die Kommunikation erfolgt. Die Abbildung 5.17 auf der nächsten Seite zeigt diese Äquivalenz. Beide Modelle besitzen das gleiche Systemverhalten, unabhängig davon, ob die Komponente *A* direkt mit *D* kommuniziert oder über einen Zwischen-Port der Komponente *B*.

In der in dieser Arbeit auf Basis von Focus festgelegten AutoFocus Semantik ergibt sich die Äquivalenz von hierarchischen und flachen AutoFocus Systemstrukturen aus der Struktur der Semantikdefinition. Die in dem Abschnitt 3.1 angegebene Übersetzung von Systemstrukturdiagrammen berücksichtigt lediglich atomare Komponenten und übersetzt AutoFocus Kommunikationsbeziehungen, die über Zwischen-Ports nicht atomarer Komponenten realisiert sind, in direkte Kommunikationskanäle zwischen den einzelnen atomaren Komponenten. Die Kommunikation mit der Systemumgebung erfolgt in der Übersetzung ebenfalls durch direkte Kommunikationskanäle. Die beiden in Abbildung 5.17 auf der nächsten Seite dargestellten AutoFocus Spezifikationen werden in ein und die selbe Focus Spezifikation übersetzt. Folglich sind beide AutoFocus Spezifikationen semantisch äquivalent.

Das *Push Down Component* Refactoring verändert die Kommunikation genau dahingehend, dass Zwischen-Ports eingefügt beziehungsweise Zwischen-Ports entfernt werden. Die Umleitung der Kommunikation hat folglich nach der AutoFocus Semantik keinen Einfluss auf das Komponentenverhalten der Super-Komponente. Das *Push Down Component* Refactoring ist ein allgemeingültiges Refactoring ohne Auswirkungen auf

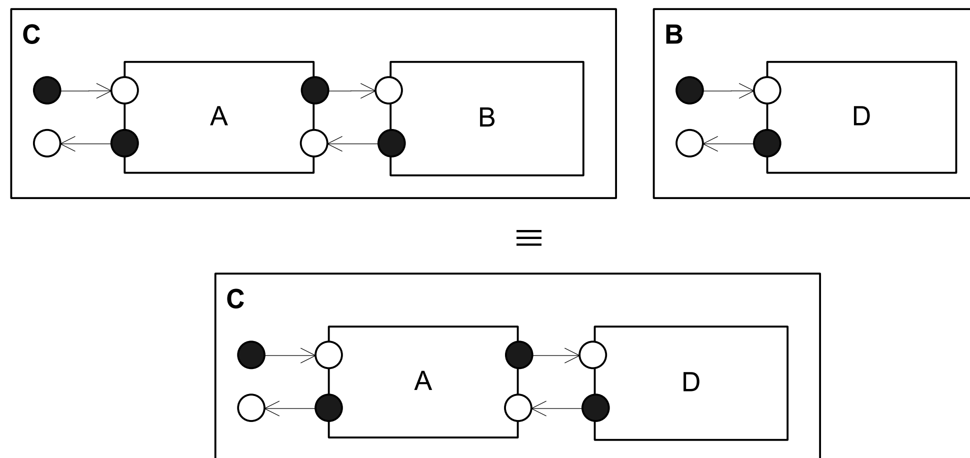


Abbildung 5.17.: Verhaltensäquivalenz bei Kommunikation über Zwischen-Ports.

das nicht zeitabstrahierte Systemverhalten.

Das Refactoring *Pull Up Component* stellt genau die Umkehrabbildung zu dem *Push Down Component* Refactoring dar und ist somit ebenfalls ein allgemeingültiges Refactoring. Das Refactoring *Wrap Components* fügt eine leere Komponente in ein SSD ein. Dies verändert nach der AutoFocus Semantik nicht das Schnittstellenverhalten des betrachteten SSDs. Anschließend wird zum Verschieben der Komponenten das *Push Down Component* Refactoring eingesetzt, das wiederum das Verhalten nicht ändert. Somit stellt *Wrap Components* ebenfalls ein allgemeingültiges Refactoring dar. Das *Move Component* Refactoring verändert Modellspezifikationen ausschließlich durch die Anwendung der *Pull Up Component* und *Push Down Component* Refactorings. Deshalb erfüllt dieses Refactoring ebenfalls die Eigenschaft der Allgemeingültigkeit. Das *Remove Single Component Hierarchy* Refactoring verwendet zunächst das *Pull Up Component* Refactoring und verändert dadurch nicht das Schnittstellenverhalten des betrachteten Modellausschnitts. Das anschließende Entfernen einer leeren nichtatomaren Komponente hat nach der AutoFocus Semantik keine Auswirkungen auf das Verhalten. Das *Remove Single Component Hierarchy* Refactoring ist folglich ebenfalls ein allgemeingültiges Refactoring, das das nicht zeitabstrahierte Systemverhalten nicht verändert.

5.2. Refactoring von hierarchischen AutoFocus Zustandsmaschinen

In der Modellierungssprache AutoFocus werden zur Verhaltensbeschreibung einzelner Komponenten spezielle Zustandsmaschinen (State Transition Diagrams / STDs) verwendet. Diese Zustandsmaschinen bieten, wie die zuvor betrachteten Systemstrukturdiagramme, das Konzept der Hierarchie. Ein Zustand einer Zustandsmaschine kann

einen Unterautomaten beinhalten. Die Unterteilung einer Zustandsmaschine in Unterautomaten hat nach der AutoFocus Semantik keinen Einfluss auf das Modellverhalten.[1]

Es lassen sich folgende Refactorings identifizieren, die es ermöglichen, Zustände in der Automatenhierarchie zu verschieben:

Push Down State (Abschnitt 5.2.1).
: Ein Zustand wird in einen Unterautomaten verschoben.

Pull Up State.
: Ein Zustand wird in der Automatenhierarchie um eine Hierarchieebene nach oben verschoben.

Wrap States (Abschnitt 5.2.2).
: Eine Menge von Zuständen innerhalb eines STDs wird in einem neuen Unterautomaten zusammengefasst.

Move State.
: Ein Zustand wird in der Automatenhierarchie an eine andere Stelle verschoben.

Remove Single State Hierarchy.
: Ein Unterautomat, der aus nur einem einzigen Zustand besteht, wird entfernt.

In hierarchischen AutoFocus Zustandsmaschinen werden die Transitionen zwischen Hierarchieebenen über Konnektoren realisiert. Hierbei werden in dem äußeren STD und dem STD des Unterautomaten Transitionen mit dem Konnektor verbunden. Die inneren und äußeren Transitionen können in AutoFocus unterschiedliche Bedingungen für das Feuern besitzen. Sie werden in der Ausführung durch ein logisches Und verknüpft. In AutoFocus ist es möglich, mehrere Transitionen in der äußeren beziehungsweise inneren Sicht mit einem Konnektor zu verbinden. Wir betrachten an dieser Stelle nur Unterautomaten, deren Konnektoren jeweils genau mit einer internen und einer externen Transition verbunden sind.

Die strukturellen Refactorings von hierarchischen AutoFocus Zustandsmaschinen sind sehr ähnlich zu den in Abschnitt 5.1 vorgestellten SSD Refactorings. Um eine Wiederholung der Konzepte zu vermeiden, wird darauf verzichtet, alle Refactorings im Einzelnen zu definieren. Wir beschränken uns an dieser Stelle exemplarisch auf die Definition des *Push Down State* Refactorings (Abschnitt 5.2.1) und des *Wrap States* Refactorings (Abschnitt 5.2.2). In Abschnitt 5.2.3 wird die Verhaltensinvarianz der betrachteten STD Struktur-Refactorings begründet.

[1] Bei vielen anderen Modellierungssprachen hat die Hierarchie in Zustandsautomaten einen Einfluss auf das Verhalten. So wird beispielsweise durch die Verwendung von Hierarchieebenen in *StateCharts* implizit eine Priorisierung der Transitionen vorgenommen.

5.2.1. Push Down State Refactoring

Problem

Der Zustand C[2] gehört funktional zu einer anderen Zustandsmaschine E. E ist ein Unterautomat des STD A, in dem sich auch der Zustand C befindet.

Lösung

Der Zustand C wird eine Hierarchieebene nach unten in das STD des Unterautomaten E verschoben (Abbildung 5.18).

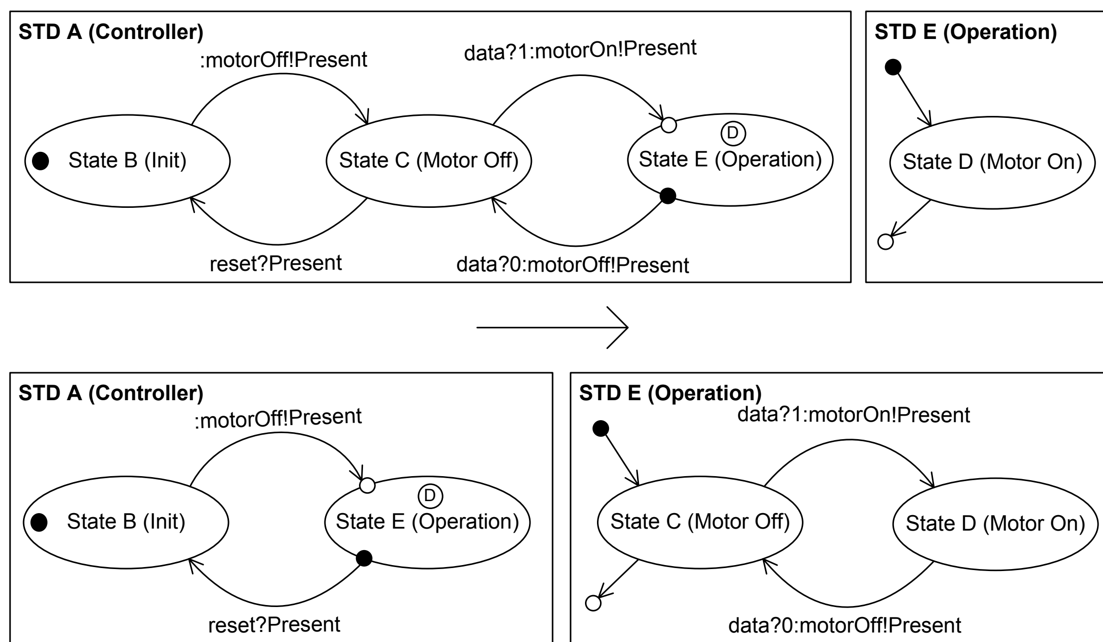


Abbildung 5.18.: Push Down State Refactoring.

Motivation

Die hierarchische Dekomposition von Zustandsmaschinen (*State Transition Diagrams* / STDs) in Unterautomaten ermöglicht es, Verhaltensbeschreibungen einzelner atomarer Komponenten in übersichtlicher und verständlicher Form zu strukturieren.

Das *Push Down State* Refactoring ist eine elementare Transformationsregel, welche es ermöglicht, einen Zustand aus einem STD in einen Unterautomaten des selben STDs

[2] Als Zustand kann auch ein Zustand gewählt werden, der einen Unterautomaten repräsentiert.

zu verschieben. Falls Sie einen Zustand in einem STD finden, der funktional zu einem Unterautomaten gehört, der in dem betrachteten STD enthalten ist, dann führen Sie das *Push Down State* Refactoring aus. Durch die Anwendung des Refactorings wird der Zustand eine Hierarchieebene nach unten verschoben. Die Abbildung 5.18 auf der vorherigen Seite zeigt die Refactoring-Transformation, die den Zustand *C* in den Unterautomaten *E* verschiebt.

Nachfolgend wird ein konkretes Beispiel für die Anwendung des *Push Down State* Refactorings gezeigt. Angenommen es existieren in einem STD der Zustand *Motor Off* zusammen mit dem Unterautomaten *Operation* (siehe Abbildung 5.18 auf der vorherigen Seite). Der Unterautomat besitzt einen Zustand *Motor On*. Der Zustand *Motor Off* gehört aus funktionaler Sicht zu dem Unterautomaten *Operation*. Es wird das *Push Down State* Refactoring angewendet, um den *Motor Off* Zustand in den Unterautomaten zu verschieben.

Methode

Das *Push Down State* Refactoring läuft vom Prinzip identisch zum *Push Down Component* Refactoring (siehe Abschnitt 5.1.1) ab. Im Unterschied zu dem *Push Down Component* Refactoring wird das *Push Down State* Refactoring statt auf Systemstrukturdiagramme (SSDs) auf Zustandsmaschinen (STDs) angewendet.

Nehmen wir an, es existiert ein Zustand *C* und ein Unterautomat *E* in einem STD *A* (Abbildung 5.18 auf der vorherigen Seite). Der Zustand *C* soll in den Unterautomaten *E* verschoben werden.

Benutzereingaben: Der Benutzer wählt einen Zustand *C*, der verschoben werden soll, und einen Zustand *E* mit assoziiertem Unterautomaten als Ziel der Verschiebeoperation. Der Zustand *C* darf auch einen assoziierten Unterautomaten besitzen.

Vorbedingungen:

- Die beiden Zustände *C* und *E* müssen voneinander verschieden sein.
- Beide Zustände *C* und *E* müssen sich in dem selben STD befinden.
- Der Zielzustand *E* muss einen assoziierten Unterautomaten besitzen.

Ausgangspunkt für das Refactoring ist die in Abbildung 5.18 auf der vorherigen Seite gezeigte Struktur von Zustandsmaschinen.

Schritt 1: Es wird eine Kopie *C*2 des Zustands *C* in dem STD *E* erzeugt. Bei dem Anlegen der Kopie werden auch Unterautomaten, die eventuell zu *C* assoziiert sind, mit kopiert (siehe Abbildung 5.19 auf der nächsten Seite).

Schritt 2: Alle indirekten Transitionen, die zwischen dem Zustand *C* im STD *A* und Zustand *D* im STD *E* über Konnektoren verlaufen, werden durch direkte Transitionen

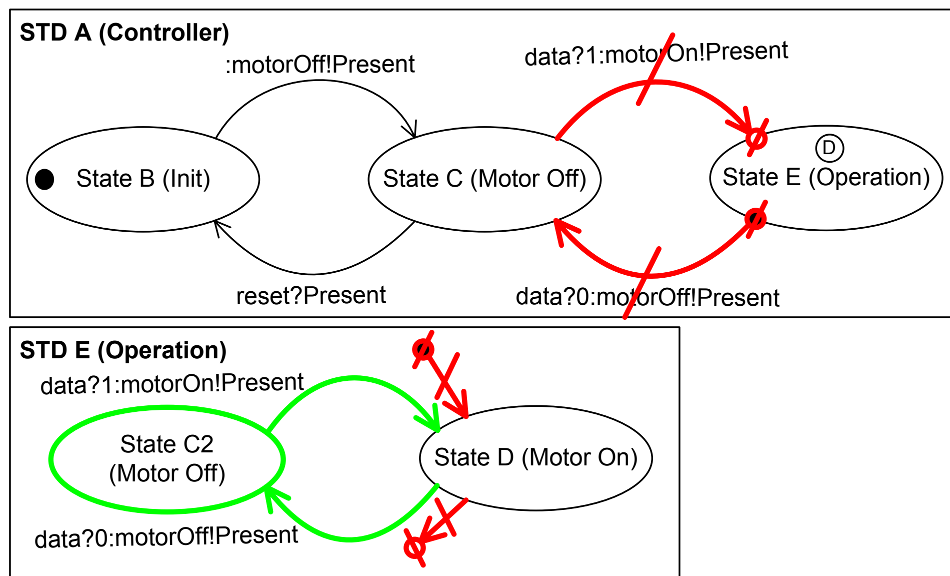


Abbildung 5.19.: Schritte 1 und 2 des Push Down State Refactorings.

zwischen dem neuen Zustand *C*2 und Zustand *D* im STD *E* ersetzt. Die Variablenvorbedingungen, Eingabebedingungen, Ausgabeterme und Variablenzuweisungsterme der neuen Transitionen werden durch Konjunktion der entsprechenden Terme der über Konnektoren verbundenen Transitionen gebildet. Die ursprünglichen Transitionen einschließlich der Konnektoren und der internen Transitionen, die mit den Konnektoren verbunden sind, werden gelöscht (siehe Abbildung 5.19).

Schritt 3: Alle Transitionen, die zwischen dem Zustand *C* und nicht dem Zustand *E* im STD *A* verlaufen, werden zu dem Unterautomaten *E* umgeleitet (siehe Abbildung 5.20 auf der nächsten Seite). Zu diesem Zweck werden für jede dieser Transitionen ein neuer Konnektor, entsprechend der Richtung der Transition, in dem Unterautomaten *E* eingefügt und dieser neue Konnektor in dem Unterautomaten wird über eine leere Transition[3] entsprechender Richtung mit dem neuen Zustand *C*2 verbunden. In dem STD *A* wird nun je nach Richtung der betrachteten Transition der Quellzustand beziehungsweise der Zielzustand der Transition von dem Zustand *C* auf den neuen Konnektor des Zustands *E* umgehängt (siehe Abbildung 5.20 auf der nächsten Seite).

Schritt 4: Der ursprüngliche Zustand *C* wird im STD *A* gelöscht (siehe Abbildung 5.20 auf der nächsten Seite).

[3]Unter einer leeren Transition wird in diesem Zusammenhang eine Transition verstanden, die weder eine Vorbedingung, eine Eingabebedingung, einen Ausgabeterm noch eine Variablenzuweisung beinhaltet.

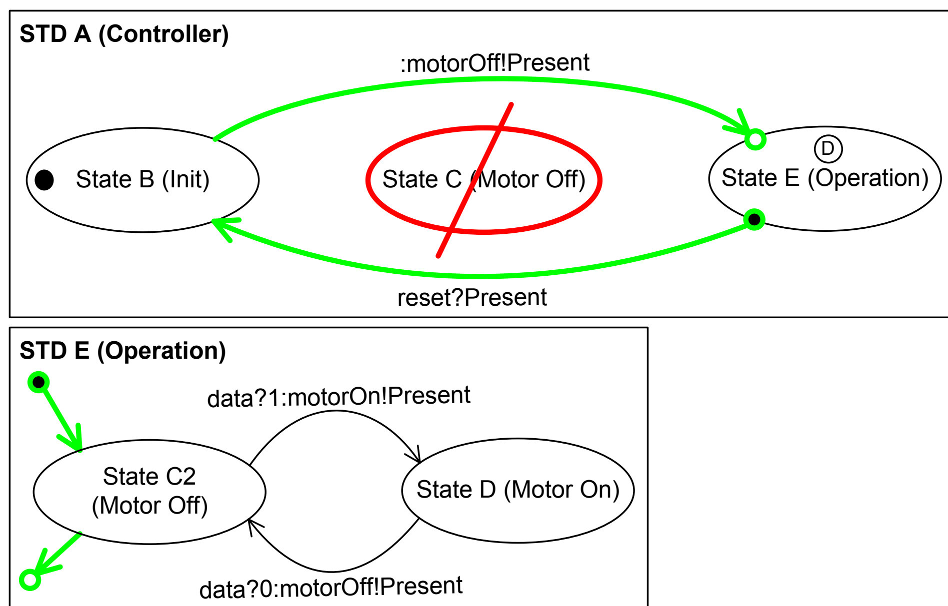


Abbildung 5.20.: Schritte 3 und 4 des Push Down State Refactorings.

5.2.2. Wrap States Refactoring

Problem

Ein *State Transition Diagram* (STD) besteht aus zu vielen Zuständen.

Lösung

Mehrere Zustände des STDs werden in einem neuen Unterautomaten zusammengefasst (Abbildung 5.21 auf der nächsten Seite).

Motivation

Zustandsübergangsdiagramme (STDs), die viele Zustände und Transitionen beinhalten, werden sehr schnell unübersichtlich. Die hierarchische Dekomposition von Zustandsmaschinen in Unterautomaten ermöglicht es, die Anzahl von Zuständen pro Diagramm zu beeinflussen. Es sollen aus funktionaler Sicht zusammengehörige Zustände in einem Unterautomaten gekapselt werden. Hierdurch werden die Zustände auf verschiedene Hierarchieebenen verteilt und die Anzahl der Zustände in der Hauptzustandsmaschine wird reduziert.

Sind in einem STD viele Zustände und Transitionen enthalten, dann werden zunächst

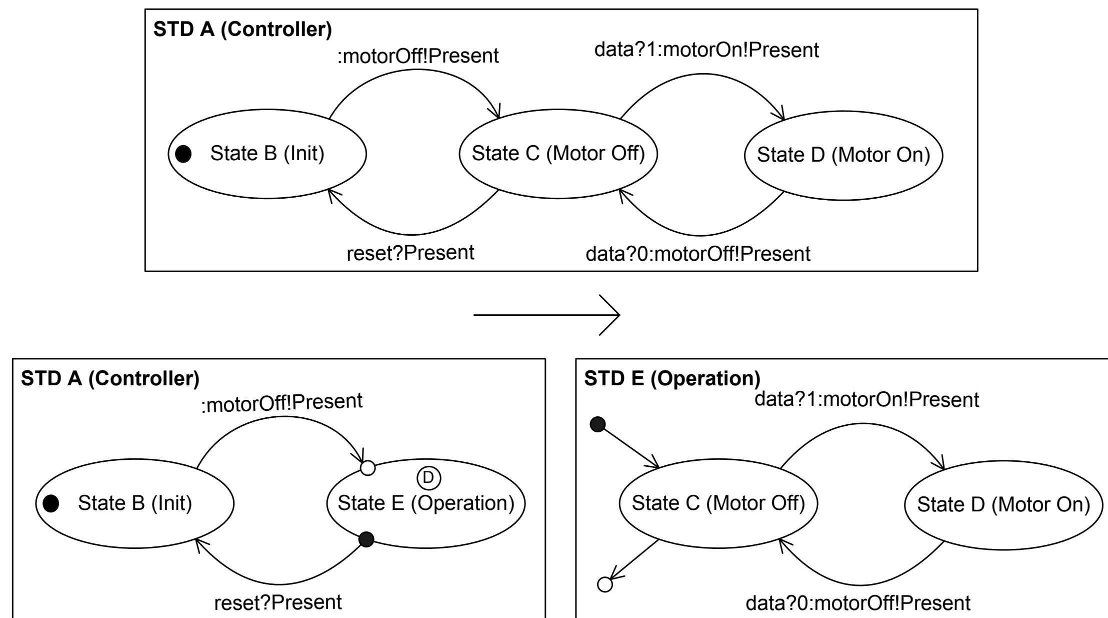


Abbildung 5.21.: Wrap States Refactoring.

funktional zusammengehörende Zustände identifiziert. Anschließend wird auf diese Zustände das *Wrap States* Refactoring angewendet.

Das *Wrap States* Refactoring, das auf STDs angewendet wird, ist das Pendant des *Wrap Components* SSD Refactorings (siehe Abschnitt 5.1.3).

Methode

Benutzereingabe: Der Benutzer wählt eine Menge von Zuständen innerhalb eines STDs *A* aus, die in einem neuen Unterautomaten gekapselt werden sollen. Zusätzlich wird der Name für den neuen Unterautomaten festgelegt.

Vorbedingungen: Mindestens ein Zustand muss für die Operation ausgewählt werden und alle gewählten Zustände müssen sich in dem selben STD *A* befinden.

Schritt 1: Zunächst wird ein neuer Zustand mit dem vom Benutzer vorgegebenen Namen und assoziierten leeren Unterautomaten in dem STD *A* erzeugt.

Schritt 2: Alle durch den Benutzer gewählten Zustände werden durch wiederholte Anwendung des *Push Down State* Refactorings (siehe Abschnitt 5.2.1) nacheinander verschoben. Als Zielunterautomat der Operation wird hierbei der neu eingefügte Unterautomat gewählt.

5.2.3. Verhaltensinvarianz der STD Struktur-Refactorings

Wie die SSD Refactorings (vergleiche Abschnitt 5.1.6) verändern die in den vorangehenden Abschnitten vorgestellten Struktur-Refactorings von Zustandsmaschinen nicht das Systemverhalten ohne Einsatz von Zeitabstraktionen (siehe Definition 4.7 auf Seite 88). Die Struktur-Refactorings von Zustandsmaschinen sind allgemeingültig (siehe Definition 4.4 auf Seite 85).

Die Allgemeingültigkeit dieser Refactorings lässt sich über die AutoFocus Semantik zeigen. Betrachten wir zunächst das *Push Down State* Refactoring (siehe Abschnitt 5.2.1). Das Refactoring erstellt eine Kopie des zu verschiebenden Zustands in dem Zielunterautomaten. Bei der Kopieroperation werden eventuell assoziierte Unterautomaten des Zustands mit kopiert, die Kopie ist also vollständig identisch zu dem zu verschiebenden Zustand. Das Refactoring leitet nun alle Transitionen, von denen der zu verschiebende Zustand entweder Quell- oder Zielzustand ist, zu dem neu kopierten Zustand um. Bei dem Umleiten der Transitionen werden zwei Fälle unterschieden. Entweder wird ein neuer Konnektor in dem Unterautomaten eingefügt und dieser Konnektor in dem Unterautomaten mit einer leeren Transition mit dem kopierten Zustand verbunden. Oder ein nicht mehr benötigter Konnektor wird gelöscht und die Transition erfolgt direkt innerhalb des Unterautomaten.

Nach der AutoFocus Semantik (siehe Kapitel 3 und [HSE97]) sind eine direkte Transition zwischen zwei Zuständen innerhalb eines STDs und zwei indirekte über einen Konnektor verbundene Transitionen äquivalent, falls sie äquivalente Quell- und Zielzustände besitzen[4] und die Variablenvorbedingung der direkten Transition gleich der Konjunktion der Variablenvorbedingungen der indirekten Transitionen sind, die Eingabebedingung der direkten Transition gleich der Konjunktion der Eingabebedingungen der indirekten Transitionen sind, der Ausgabeterm der direkten Transition gleich der Konjunktion der Ausgabeterme der indirekten Transitionen sind und der Variablenzuweisungsterm der direkten Transition gleich der Konjunktion der Variablenzuweisungsterme der indirekten Transitionen sind.

In der Semantikdefinition von AutoFocus STDs auf Basis von Focus, die in Abschnitt 3.2 aufgeführt ist, lässt sich diese Äquivalenz von direkten und indirekten Transitionen aus der Struktur der Übersetzung ableiten. Die hierarchischen Zustandsmaschinen werden in der Übersetzung von AutoFocus nach Focus zunächst in Zustandsmaschinen abgebildet, die keine hierarchischen Zustände enthalten. Diese Übersetzung erfolgt durch Einsetzen der Unterautomaten in den Hauptautomaten und durch Ersetzen von indirekten Transitionen durch direkte Transitionen, deren Inhalte aus der Konjunktion der Variablenvorbedingungen, Eingabebedingungen, Ausgabeterme und Variablenzuweisungsterme der indirekten Transitionen gebildet werden.

Für den Sonderfall, dass eine der beiden über einen Konnektor verbundenen indirek-

[4]Der Quellzustand der direkten Transition muss äquivalent zu dem Quellzustand der Transition, die als Ziel den Konnektor besitzt, sein. Darüber hinaus muss der Zielzustand der direkten Transition äquivalent zu dem Zielzustand der Transition, die als Quelle den Konnektor besitzt, sein.

ten Transitionen leer ist, das heißt weder eine Variablenvorbedingung, eine Eingabebedingung, einen Ausgabeterm noch einen Variablenzuweisungsterm besitzt, ergibt sich demnach aus der Struktur der Semantikdefinition, dass aus der Äquivalenz der Variablenvorbedingungen, der Eingabebedingungen, der Ausgabeterme und der Variablenzuweisungsterme der direkten Transition und der nicht leeren indirekten Transition die Äquivalenz der direkten Transition und der indirekten Transition folgt. Diese Äquivalenz ist in der Abbildung 5.22 auf der nächsten Seite dargestellt. Beide in der Abbildung dargestellten Verhaltensspezifikationen sind auf semantischer Ebene äquivalent.

Das *Push Down State* Refactoring ersetzt direkte Transitionen durch indirekte Transitionen mit Konnektoren, von denen eine der beiden indirekten Transitionen leer ist und die Terme der anderen indirekten Transition identisch zu den Termen der direkten Transition sind. Entsprechend der vorangehend beschriebenen Äquivalenz von direkten und indirekten Transitionen verändert diese Transformation das Verhalten der Spezifikation nicht. Des Weiteren werden durch das *Push Down State* Refactoring indirekte Transitionen durch direkte Transitionen ersetzt, deren Terme durch Konjunktion der entsprechenden Terme der über Konnektoren verbundenen indirekten Transitionen gebildet werden, und somit ebenfalls aus Sicht des Verhaltens äquivalent sind. Das *Push Down State* Refactoring ist folglich ein allgemeingültiges Refactoring.

Das in Abschnitt 5.2.2 definierte *Wrap States* Refactoring fügt einen leeren Unterautomaten in ein STD ein. Diese Operation hat keinen Einfluss auf die Interpretation des Verhaltens des STDs. Anschließend wird wiederholt das allgemeingültige Refactoring *Push Down State* eingesetzt. Das *Wrap States* ist folglich ebenfalls ein allgemeingültiges Refactoring. Die weiteren Struktur-Refactorings *Pull Up State*, *Move State* und *Remove Single State Hierarchy*, die zu Beginn des Abschnitts 5.2 erwähnt aber nicht detailliert besprochen wurden, stellen ebenfalls allgemeingültige Refactorings dar, die das nicht abstrahierte Schnittstellenverhalten nicht verändern.

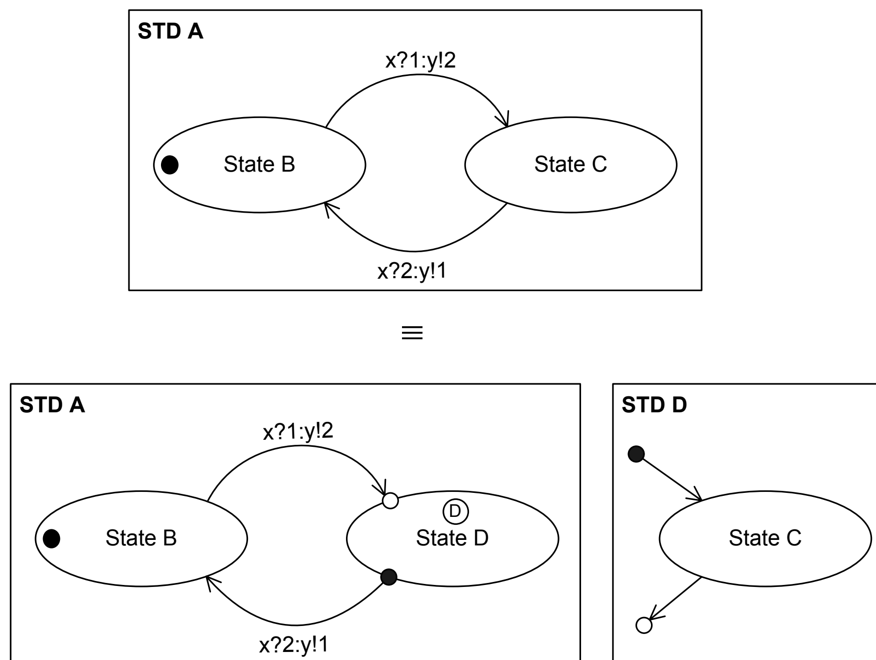


Abbildung 5.22.: Verhaltensäquivalenz von Unterautomaten-Transitionen.

6. Zeitsynchronität versus Zeitasynchronität

Diese Arbeit definiert Refactorings für die AutoFocus Modellierungssprache. Neben rein strukturellen Refactorings, die das zeitliche Verhalten des Modells und seiner Teile nicht verändern, werden auch Operationen betrachtet, die das zeitliche Verhalten einzelner Komponenten abändern. Nach der formalen Definition von Refactoring von AutoFocus Modellen (Definition 4.6 auf Seite 87) wird für das zeitabstrahierte Schnittstellenverhalten des Gesamtsystems vor und nach der Durchführung des Refactorings Äquivalenz gefordert. Die Betrachtung des zeitabstrahierten Schnittstellenverhaltens des durch das Refactoring geänderten Teils der Spezifikation reicht im allgemeinen nicht aus, um auf das zeitabstrakte Schnittstellenverhalten des Gesamtsystems schließen zu können.

In Abschnitt 6.1 werden die Auswirkungen lokaler zeitlicher Veränderungen auf das nicht zeitliche Gesamtverhalten nach der bestehenden zeitsynchronen Semantik von AutoFocus betrachtet. Die bestehende Semantik von AutoFocus erweist sich als labil gegenüber lokalen zeitlichen Veränderungen der Spezifikationen. Aus diesem Grund werden in dem Abschnitt 6.2 Modelleigenschaften festgelegt, die die Durchführung von komplexen Refactorings auf Spezifikationen, die diese Eigenschaften erfüllen, erleichtern. Über diese Eigenschaften wird eine besondere Unterklasse von Spezifikationen in der Sprache AutoFocus identifiziert, die zeitrobusten Spezifikationen. Auf diesen speziellen AutoFocus Modellen lassen sich lokale zeitliche Veränderungen des Verhaltens durchführen, ohne dabei das nicht zeitliche Systemverhalten unkontrolliert zu verändern. In Abschnitt 6.3 werden Anforderungen an eine neue Semantik der Modellierungssprache AutoFocus unter Berücksichtigung der Refactoring-Methode festgelegt. Die Erfüllung der vorangehend erwähnten Zeitrobustheitseigenschaft spielt hierbei eine zentrale Rolle. In Abschnitt 6.4 werden die in verschiedenen existierenden Programmier- und Modellierungssprachen eingesetzten semantischen Konzepte auf ihre Eignung für das Refactoring hin betrachtet. Auf Basis dieser Betrachtung wird entschieden, eine zeitasynchrone Semantik unter Verwendung von gepufferter Kommunikation für AutoFocus zu entwickeln. In dem Abschnitt 6.5 werden geeignete Zeitabstraktionen, bezüglich derer die Zeitrobustheitseigenschaft durch die neue Semantik erfüllt werden soll, ausgewählt. In Abschnitt 6.6 wird die neue zeitasynchrone Semantik für die AutoFocus Modellierungssprache durch eine Abbildung in das ursprüngliche zeitsynchrone AutoFocus definiert. Anschließend wird in Abschnitt 6.7 nachgewiesen, dass die neue zeitasynchrone AutoFocus Modellierungssprache die Eigenschaft der Zeitrobustheit für alle in ihr ausdrückbaren Spezifikationen erfüllt.

6.1. Zeitsynchrones AutoFocus und zeitliche lokale Verhaltensänderungen

Die Semantik der AutoFocus Modellierungssprache wird entsprechend der Klassifikation in [KS03] als nachrichtenasynchron und zeitsynchron bezeichnet. Alle in einem System komponierten Zustandsmaschinen arbeiten nebenläufig. Pro Systemtakt wird in jeder Zustandsmaschine genau eine Transition gefeuert.[1] Die Ausgaben der einzelnen nebenläufigen Komponenten werden jeweils um einen Systemtakt verzögert, das heißt die Komponenten lesen zum Zeitpunkt t die Eingaben und die daraus resultierenden Ausgaben werden zum Zeitpunkt $t+1$ getätigt.[2] Wird eine Eingabe durch eine Zustandsmaschine zum Zeitpunkt ihres Auftretens nicht verarbeitet, dann wird sie zum darauf folgenden Zeitpunkt gelöscht beziehungsweise überschrieben und geht somit verloren.

Die zeitsynchrone AutoFocus Semantik impliziert einen globalen Zeitbegriff, der in den einzelnen Zustandsmaschinen sichtbar ist. Dieser Zeitbegriff in AutoFocus bedingt eine Labilität des Systemverhaltens gegenüber kleinen lokalen zeitlichen Veränderungen der Verhalten von Teilsystemen. Diese Tatsache erschwert in hohem Maße sowohl die Erweiterung bestehender AutoFocus Modelle um neue Funktionalitäten als auch das Refactoring von AutoFocus Modellen.

Die Abbildung 6.1 auf der nächsten Seite zeigt ein kleines AutoFocus Modell, das als Beispiel zur Verdeutlichung der angesprochenen Problematik dienen soll. Das Modell verhält sich nach der herkömmlichen zeitsynchronen AutoFocus Semantik wie in Tabelle 6.1 auf der nächsten Seite angegeben.

Die Komponente *Controller* sendet zum Zeitpunkt $t = 2$ eine Nachricht über den Kanal *Error*. Diese Nachricht wird zum selben Zeitpunkt von der *Controller* Komponente empfangen, welche daraufhin zum Zeitpunkt $t = 3$ eine Nachricht über den Kanal *Failure* an die Systemumgebung sendet.

Wir möchten nun die Zustandsmaschine der Komponente *Controller* erweitern und fügen einen neuen Zustand *Init2* für die Durchführung erweiterter Initialisierungsarbeiten zwischen den Zuständen *Init* und *Operation* ein (siehe Abbildung 6.2 auf der nächsten Seite). Das aus dieser Änderung resultierende Verhalten der Spezifikation ist in der Tabelle 6.2 auf Seite 122 angegeben.

Die neue veränderte Komponente *Controller* sendet über den Kanal *Failure* keine Nachricht mehr, obwohl von der *Watchdog* Komponente ein Alarm gesendet wird. Diese Veränderung des Systemverhaltens war nicht die Intention des Entwicklers bei der Erweiterung der Zustandsmaschine der *Controller* Komponente.

Die *Controller* Komponente empfängt zum Zeitpunkt $t = 2$ die *Alarm* Nachricht, kann

[1] Ist keine der spezifizierten Transitionen, die vom aktuellen Kontrollzustand ausgehen, schaltbereit, dann wird in AutoFocus eine implizite *Idle* Transition ausgeführt.

[2] Eine Ausnahme hiervon stellt die spezielle *Immediate* Kommunikationssemantik dar, die es ermöglicht, zum selben Zeitpunkt auf Eingaben mit Ausgaben zu reagieren.

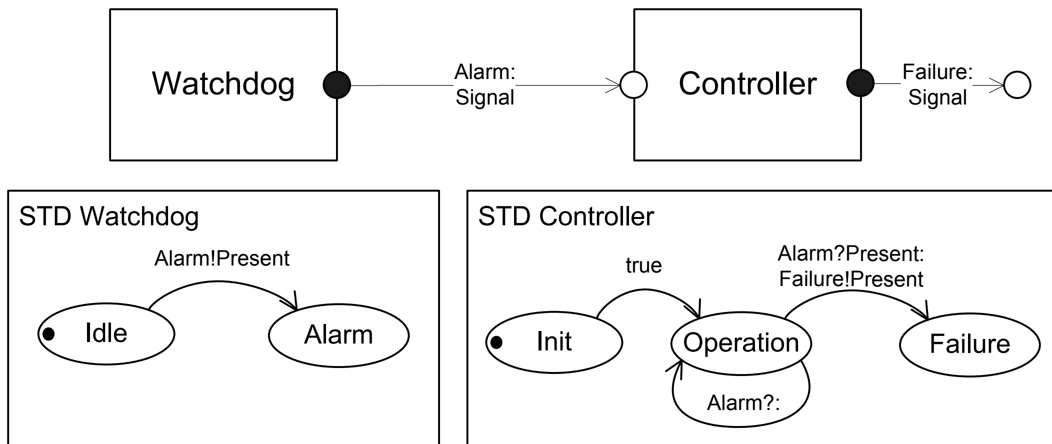


Abbildung 6.1.: Beispielmodell zur Verdeutlichung des Zeitproblematik.

Zeit t	Zustand Watchdog	Ausgabe Watchdog	Zustand Controller	Eingabe Controller	Ausgabe Controller
1	Idle	-	Init	-	-
2	Alarm	Present	Operation	Present	-
3	Alarm	-	Failure	-	Present

(a) Verhalten als Tabelle dargestellt

$$\{(\langle void, \checkmark, Present, \checkmark, void, \checkmark\rangle)\}$$

$$\{(\langle void, \checkmark, Present, \checkmark, void, \checkmark\rangle, \langle void, \checkmark, void, \checkmark, Present, \checkmark\rangle)\}$$

(b) Verhalten in Focus durch Mengen von Tupel von Strömen dargestellt.

Tabelle 6.1.: Verhalten des Beispielmodells vor Durchführung der Änderung.

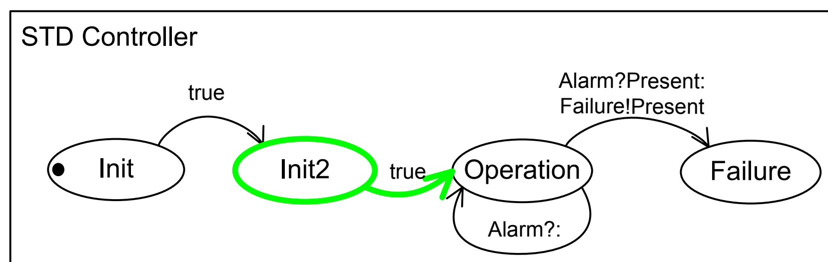


Abbildung 6.2.: Abgeänderte Zustandsmaschine der Komponente *Controller*.

Zeit t	Zustand Watchdog	Ausgabe Watchdog	Zustand Controller	Eingabe Controller	Ausgabe Controller
1	Idle	-	Init	-	-
2	Alarm	Present	Init2	Present	-
3	Alarm	-	Operation	-	-

(a) Verhalten als Tabelle dargestellt

$$\{(\langle void, \checkmark, Present, \checkmark, void, \checkmark\rangle)\}$$

$$\{(\langle void, \checkmark, Present, \checkmark, void, \checkmark\rangle, \langle void, \checkmark, void, \checkmark, void, \checkmark\rangle)\}$$

(b) Verhalten in Focus durch Mengen von Tupel von Strömen dargestellt.

Tabelle 6.2.: Verhalten des Beispielmodells nach Durchführung der Änderung.

diese Nachricht jedoch nicht verarbeiten, da sie sich noch in der Initialisierungsphase (Zustand *Init2*) befindet. In der ursprünglichen Zustandsmaschine aus Abbildung 6.1 auf der vorherigen Seite ist implizit modelliert, dass eine Reaktion auf eine *Alarm* Nachricht ab dem Zeitpunkt $t = 2$ erfolgt. Die erweiterte Zustandsmaschine kann hingegen erst ab dem Zeitpunkt $t = 3$ auf diese Nachricht reagieren. Dieser feste Zeitbezug in der zeitsynchronen AutoFocus Semantik kann, wie in dem gezeigten Beispiel geschehen, bei dem Versuch der lokalen Erweiterung von AutoFocus Modellen zu unbeabsichtigten durch den Entwickler nur schwer nachvollziehbaren Verhaltensänderungen des Gesamtsystems führen.

Ziehen wir ein konkretes Beispiel für ein Refactoring einer Modellspezifikation in AutoFocus heran. Wir betrachten eine Operation zum Aufteilen einer Komponente in zwei Teilkomponenten, wie sie in Abbildung 7.3 auf Seite 162 dargestellt ist. Diese Modelltransformation führt sowohl zu Änderungen der strukturellen Sicht als auch zu Veränderungen der Verhaltensbeschreibung der atomaren Komponenten. Die Modelländerung soll durchführbar sein, ohne dabei das Systemverhalten unter Zeitabstraktion zu verändern.

Betrachten wir die gewünschte Modelländerung unter Verwendung der bestehenden AutoFocus Semantik. Durch die Aufspaltung einer Komponente in zwei Komponenten ist im Allgemeinen eine Kommunikation zwischen diesen neuen Komponenten erforderlich. Diese Kommunikation benötigt nach der herkömmlichen zeitsynchronen AutoFocus Semantik Zeit.[3] In Abbildung 6.3 auf der nächsten Seite ist ein Beispiel für das Verhalten bei Aufteilung einer Komponente in zwei Komponenten in Form von

[3]Das Konzept der AutoFocus *Immediate Ports* wird an dieser Stelle nicht betrachtet. Die Erweiterung ist für eine allgemeine Lösung des hier vorgestellten Problems ungeeignet, da diese Art der Kommunikation nicht zyklisch sein darf. Eine bidirektionale Kommunikation zwischen den beiden neuen Komponenten, die im Allgemeinen erforderlich ist, ist unter Verwendung der *Immediate Ports* nicht möglich.

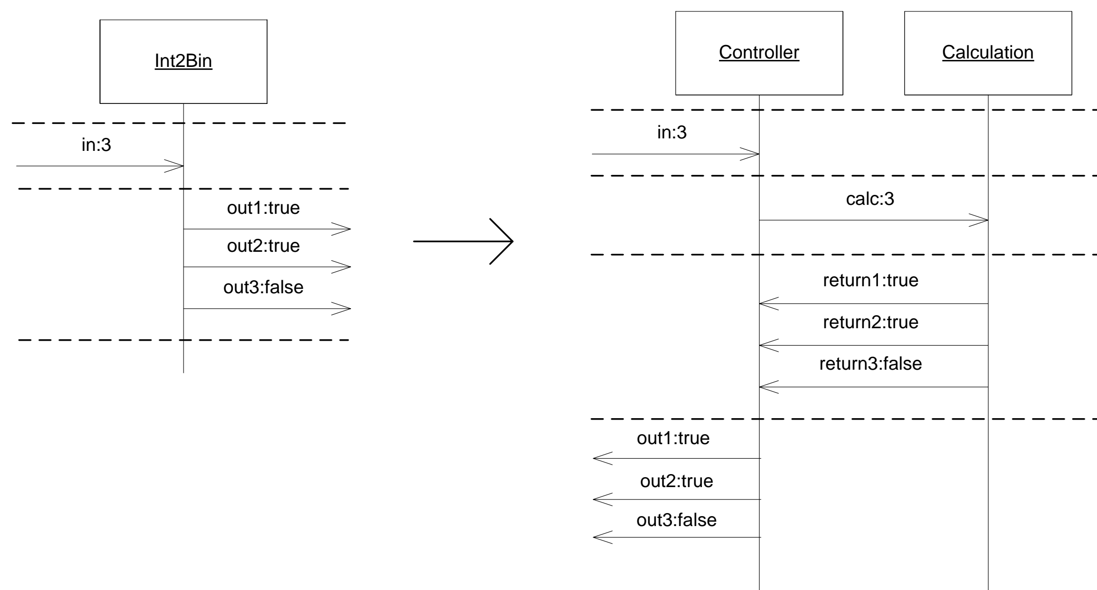


Abbildung 6.3.: Aufteilen einer AutoFocus Komponente.

AutoFocus Sequenzdiagrammen aufgeführt. Durch das Einfügen von Kommunikation zwischen den beiden neuen Komponenten wird das zeitliche Verhalten verändert. Im Beispiel vergehen in der aufgeteilten Variante drei Zeitschritte von dem Lesen der Eingabe bis zum Schreiben der Ausgabe. Die ursprüngliche Komponente benötigt hierfür lediglich einen Zeitschritt. Diese lokale Veränderung des zeitlichen Verhaltens kann, wie bereits zu Beginn dieses Abschnittes erläutert, unter Anwendung der zeitsynchronen AutoFocus Semantik zu weitgehenden Veränderungen des nicht zeitlichen Verhaltens des Gesamtsystems führen.

6.2. Modelleigenschaften zur Vereinfachung von Refactoring

Im vorangehenden Abschnitt wurde die Inflexibilität von AutoFocus Modellen gegenüber Modelländerungen gezeigt. In diesem Abschnitt definieren wir Eigenschaften von AutoFocus Modellen, die die Durchführung von Refactoring vereinfachen.

Entsprechend der Definition von Refactoring (siehe Definition 4.6 auf Seite 87) soll das zeitabstrahierte Schnittstellenverhalten des Systems durch die Refactoring-Operation unverändert bleiben. Eine sehr wichtige Eigenschaft ist in diesem Zusammenhang die Möglichkeit, aus der Beobachtung des zeitabstrahierten Schnittstellenverhaltens einer Komponente auf das Schnittstellenverhalten des Gesamtsystems unter Zeitabstraktion schließen zu können. Diese Eigenschaft wird als Kompositionalität der Abstraktion bezeichnet. In Abschnitt 6.2.1 wird die Eigenschaft der Kompositionalität der Abstrak-

tion als die Homomorphismuseigenschaft der konkreten und abstrakten Komposition definiert.

Tatsächlich stellen wir fest, dass sich die Eigenschaft der Kompositionalität für die in dieser Arbeit in Abschnitt 3.7 definierten AutoFocus Zeitabstraktionen nur für eine sehr eingeschränkte Menge von AutoFocus Spezifikationen erfüllen lässt. Aus diesem Grund definieren wir in Abschnitt 6.2.2 die schwächere Eigenschaft der Zeitrobustheit, die durch eine größere Menge von AutoFocus Spezifikationen erfüllt wird. In Modellspezifikationen, die die Zeitrobustheitseigenschaft bezüglich einer auf Komponentenebene und einer auf Systemebene angewendeten Zeitabstraktion erfüllen, haben lokale zeitliche Veränderungen des Verhaltens, die unter der gewählten lokalen Zeitabstraktion nicht sichtbar sind, keine Wirkung auf das globale zeitabstrakte Verhalten.

Mit Hilfe dieser Eigenschaft lässt sich der Aufwand für die Überprüfung der Korrektheit von Refactoring stark reduzieren. Ist die Zeitrobustheitseigenschaft gegeben, dann reicht statt der Betrachtung des Gesamtsystems die Betrachtung der durch das Refactoring geänderten Teile des Systems aus, um die Verhaltensinvarianz der Refactoring-Operation zu zeigen. In dem Abschnitt 6.2.3 wird schließlich eine Definition von Refactoring zeitrobuster AutoFocus Spezifikationen angegeben.

6.2.1. Kompositionalität der Abstraktion

Die Eigenschaft der Kompositionalität der Abstraktion besagt, dass es unerheblich ist, ob zuerst die Abstraktion auf das Schnittstellenverhalten von Teilsystemen angewendet wird und anschließend diese abstrakten Verhalten zu einem abstrakten Gesamtverhalten komponiert werden, oder ob die konkreten Teilverhalten komponiert werden und anschließend die Abstraktion durchgeführt wird.

Definition 6.1 (Kompositionalität der Abstraktion)**.** Die Eigenschaft der Kompositionalität der Abstraktion ist mathematisch als die Homomorphismus Eigenschaft der Abstraktionsabbildung $\underline{\widetilde{\alpha}}_x$ bezüglich des konkreten Kompositionsoperators $\otimes_K$ (siehe Definition 3.15 auf Seite 62) und eines abstrakten Kompositionsoperators $\otimes'_{\alpha_x}$ (vergleiche Definition 3.23 auf Seite 80) angegeben:

$$
\begin{aligned}
&\otimes'_{\alpha_x} \in \underline{\mathcal{N}} \times \mathcal{L}_{Syn} \times \underline{\mathcal{N}} \times \mathcal{L}_{Syn} \to \underline{\mathcal{N}} \\
&\mathit{kompositionalit\ddot{a}tAbstraktion}(\underline{\widetilde{\alpha}}_x, \otimes_K, \otimes'_{\alpha_x}) \stackrel{\text{def}}{=} \\
&\forall\, S_1, S_2 \in \mathcal{L}_A : \\
&\exists\, Syn_{S_1}, Syn_{S_2} \in \mathcal{L}_{Syn} : \\
&(Syn_{S_1} = Syn_{AF}(S_1)) \wedge (Syn_{S_2} = Syn_{AF}(S_2)) \wedge \\
&\exists\, \mathcal{R}_{S_1}, \mathcal{R}_{S_2} \in \underline{\mathcal{M}} : (\mathcal{R}_{S_1} = \underline{SV}_{AF}(S_1)) \wedge (\mathcal{R}_{S_2} = \underline{SV}_{AF}(S_2))
\end{aligned}
$$

$$\underline{\widetilde{\alpha}}_x((\mathcal{R}_{S_1}, Syn_{S_1}) \otimes_K (\mathcal{R}_{S_2}, Syn_{S_2})) =$$
$$(\underline{\widetilde{\alpha}}_x(\mathcal{R}_{S_1}), AbsSyn(Syn_{S_1})) \otimes'_{\alpha_x} (\underline{\widetilde{\alpha}}_x(\mathcal{R}_{S_2}), AbsSyn(Syn_{S_2})) \tag{6.1}$$

Die Abbildung 6.4 auf der nächsten Seite stellt die Eigenschaft der Kompositionalität der Abstraktion unter Weglassung der syntaktischen Schnittstellen graphisch dar.

Durch die Erfüllung dieser Eigenschaft folgt aus der Verhaltensäquivalenz der veränderten Teilspezifikation unter Abstraktion automatisch die Äquivalenz des abstrakten Verhaltens des Gesamtsystems.

Sei $T : \mathcal{L}_A \times Par \rightarrow \mathcal{L}_A$ eine Refactoring Modelltransformation, die auf eine AutoFocus Teilspezifikation S_1 mit den Parametern P angewendet wird, die sich innerhalb einer Gesamtspezifikation $S = S_1 \cup S_2$ befindet. Das Schnittstellenverhalten des Gesamtsystems vor dem Refactoring $\mathcal{R}_S$ und das Schnittstellenverhalten des Gesamtsystems nach der Anwendung des Refactorings $\mathcal{R}'_S$ ergeben sich zu folgenden Ausdrücken:

$$S'_1 = T(S_1, P)$$
$$Syn_{S_1} = Syn_{AF}(S_1) = Syn_{AF}(S'_1)$$
$$Syn_{S_2} = Syn_{AF}(S_2)$$
$$\mathcal{R}_{S_1} = \underline{SV}_{AF}(S_1)$$
$$\mathcal{R}_{S_2} = \underline{SV}_{AF}(S_2)$$
$$\mathcal{R}'_{S_1} = \underline{SV}_{AF}(S'_1)$$
$$\mathcal{R}_S = (\mathcal{R}_{S_1}, Syn_{S_1}) \otimes_K (\mathcal{R}_{S_2}, Syn_{S_2}) \tag{6.2}$$
$$\mathcal{R}'_S = (\mathcal{R}'_{S_1}, Syn_{S_1}) \otimes_K (\mathcal{R}_{S_2}, Syn_{S_2}) \tag{6.3}$$

Aus Definition 4.6 auf Seite 87 folgt aus der Eigenschaft *verhaltensinvarianzAF_*α_x:

$$\underline{\widetilde{\alpha}}_x(\mathcal{R}_S) = \underline{\widetilde{\alpha}}_x(\mathcal{R}'_S) \tag{6.4}$$
$$\underline{\widetilde{\alpha}}_x((\mathcal{R}_{S_1}, Syn_{S_1}) \otimes_K (\mathcal{R}_{S_2}, Syn_{S_2})) =$$
$$\underline{\widetilde{\alpha}}_x((\mathcal{R}'_{S_1}, Syn_{S_1}) \otimes_K (\mathcal{R}_{S_2}, Syn_{S_2})) \tag{6.5}$$

Mit der Definition der Kompositionalität der Abstraktion (Definition 6.1 auf der vorherigen Seite) folgt:

$$((\underline{\widetilde{\alpha}}_x(\mathcal{R}_{S_1}), AbsSyn(Syn_{S_1})) \otimes'_{\alpha_x} (\underline{\widetilde{\alpha}}_x(\mathcal{R}_{S_2}), AbsSyn(Syn_{S_2})) =$$

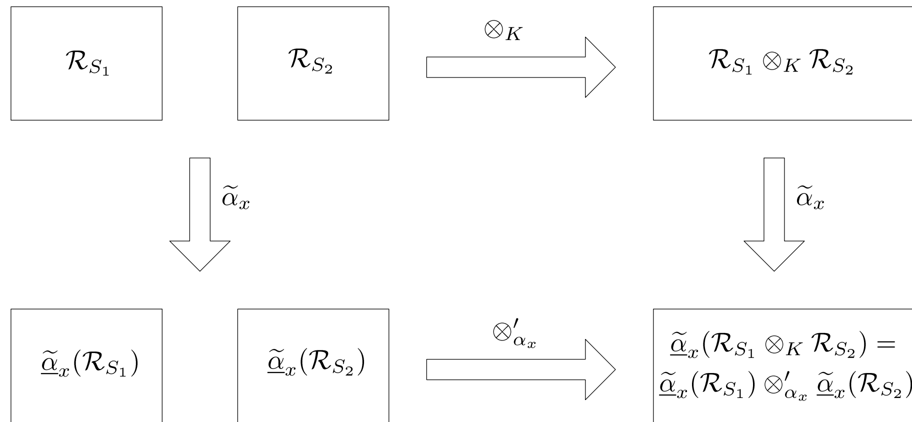


Abbildung 6.4.: Kompositionalität der Abstraktion.

$$(\widetilde{\underline{\alpha}}_x(\mathcal{R}'_{S_1}), AbsSyn(Syn_{S_1})) \otimes'_{\alpha_x} (\widetilde{\underline{\alpha}}_x(\mathcal{R}_{S_2})), AbsSyn(Syn_{S_2})) \Leftarrow$$
$$(\widetilde{\underline{\alpha}}_x(\mathcal{R}_{S_1}) = \widetilde{\underline{\alpha}}_x(\mathcal{R}'_{S_1})) \ \square \qquad (6.6)$$

Für den Nachweis der Äquivalenz muss folglich bei Erfüllung der Eigenschaft der Kompositionalität der Abstraktion nur das abstrakte Schnittstellenverhalten der veränderten Teilspezifikation und nicht das Gesamtverhalten betrachtet werden, wodurch der Aufwand hierfür beträchtlich reduziert wird. Darüber hinaus lassen sich für Modelle, die die Eigenschaft der Kompositionalität der Abstraktion erfüllen, in vielen Fällen allgemeingültige Refactorings definieren, deren Gültigkeit auf der Ebene der Modellierungssprache gezeigt werden kann und nicht auf der Ebene der konkreten Modellspezifikationen nachgewiesen werden muss.

6.2.2. Zeitrobustheit

Im vorangehenden Abschnitt 6.2.1 wurde die Kompositionalität einer Abstraktion als eine zentrale Eigenschaft zur Ermöglichung von komplexen Refactorings herausgestellt. Die Erfüllung dieser Eigenschaft ist sowohl von der Komponierbarkeit abstrakten Verhaltens als auch von der Tolerierung von zeitlichen Veränderungen durch die Semantik der Modellierungssprache abhängig. In der Praxis erweist sich die Erfüllung der Kompositionalität der Abstraktion auf Basis von AutoFocus als nur unter sehr starken Einschränkungen realisierbar (vergleiche hierzu den nachfolgenden Abschnitt 6.5).

Aus diesem Grund wird in diesem Abschnitt eine schwächere Eigenschaft festgelegt, die sicherstellt, dass lokale Änderungen des Verhaltens, die unter der Abstraktion nicht sichtbar sind, nicht zu Veränderungen des abstrakten Systemverhaltens führen. Das Modellverhalten wird hierbei unter verschiedenen Abstraktionen auf Komponenten- und Systemebene betrachtet. Die Eigenschaft der Zeitrobustheit bezüglich $\widetilde{\underline{\alpha}}_x$ auf Kom-

ponentenebene und $\underline{\widetilde{\alpha}}_y$ auf Gesamtsystemebene ist wie folgt definiert:

Definition 6.2 (Zeitrobustheit einer atomaren Komponente)**.** Eine atomare Komponente ist zeitrobust bezüglich der Abstraktion $\underline{\widetilde{\alpha}}_x$ auf Komponentenebene, der Abstraktion $\underline{\widetilde{\alpha}}_y$ auf Systemebene sowie einer entsprechend der Semantik der Modellierungssprache eingeschränkten Menge von komponierbaren Schnittstellenverhalten von Restsystemen $\underline{\mathcal{M}}_R \subseteq \underline{\mathcal{M}}$ und einer Menge der entsprechend der Semantik zulässigen Schnittstellenverhalten von lokal betrachteten Komponenten $\underline{\mathcal{M}}_K \subseteq \underline{\mathcal{M}}$, falls alle lokalen Änderungen des zeitlichen Verhaltens der Komponente, die unter der Abstraktion $\underline{\widetilde{\alpha}}_x$ gleich sind und entsprechend $\underline{\mathcal{M}}_K$ zulässiges lokales Schnittstellenverhalten darstellen, keine Veränderung eines nach $\underline{\widetilde{\alpha}}_y$ abstrahierten Schnittstellenverhaltens eines Gesamtsystems, das aus der Komposition des veränderten Komponentenverhaltens und einem frei aus der Menge der komponierbaren Verhalten $\underline{\mathcal{M}}_R$ gewählten Schnittstellenverhalten des Restsystems gebildet wird, bewirkt. Eine atomare AutoFocus Komponentenspezifikation S_K ist zeitrobust bezüglich $\underline{\widetilde{\alpha}}_x$, $\underline{\widetilde{\alpha}}_y$, $\underline{\mathcal{M}}_R$ und $\underline{\mathcal{M}}_K$, falls folgenden Eigenschaft gilt:[4]

$$
\begin{aligned}
& \mathit{zeitrobustheit}_{\alpha_x,\alpha_y,\underline{\mathcal{M}}_K,\underline{\mathcal{M}}_R}(S_K) \stackrel{\text{def}}{=} \\
& \exists\, \mathcal{R}_K \in \underline{\mathcal{M}}_K, \mathit{Syn}_K \in \mathcal{L}_{Syn} : \\
& \mathcal{R}_K = \underline{SV}_{AF}(S_K) \wedge C_{Focus}(S_K) \in \mathit{Syn}_K \wedge \\
& \forall\, \mathcal{R}_R \in \underline{\mathcal{M}}_R : \\
& \forall\, \mathit{Syn}_R \in \{a \in \mathcal{L}_{Syn} \text{ so that } \exists\, S \in \mathcal{L}_F : (\mathit{Bez}(a) \in \mathcal{R}_R \Leftrightarrow [\![\, S \,]\!]) \wedge S \in a\} : \\
& \exists\, t_{\mathcal{R}_R} \in \underline{\mathcal{N}} : \\
& \forall\, s \in \left\{ z \in \underline{\widetilde{\gamma}}_x(\underline{\widetilde{\alpha}}_x(\mathcal{R}_K)) \text{ so that } z \in \underline{\mathcal{M}}_K \right\} : \\
& t_{\mathcal{R}_R} = \underline{\widetilde{\alpha}}_y((s, \mathit{Syn}_K) \otimes_K (\mathcal{R}_R, \mathit{Syn}_R))
\end{aligned}
\qquad (6.7)
$$

Die Zeitrobustheitseigenschaft legt für Spezifikationen, die zeitliche Bedingungen enthalten, fest, ob diese zeitliche Verschiebungen der Ein- und Ausgaben, innerhalb der durch die gewählte Zeitabstraktion vorgegebenen Zeitfenster, tolerieren und deren nicht zeitliche Verhalten hierbei gleich bleibt. Die Zeitunabhängigkeit einer Spezifika-

[4] In den Zeilen 2-3 der Eigenschaftsdefinition wird zur gegebenen Komponentenspezifikation S_K deren Schnittstellenverhalten $\mathcal{R}_K$ und deren syntaktische Schnittstelle Syn_K gebunden. In der Zeile 5 werden alle möglichen syntaktischen Schnittstellen des Schnittstellenverhaltens des Restsystems $\mathcal{R}_R$ an die Variable Syn_R gebunden. Die eigentliche Zeitrobustheitseigenschaft ist in den Zeilen 6-8 dargestellt. An die Variable s werden alle Verhalten, die sich aus der Abstraktion $\underline{\widetilde{\alpha}}_x$ mit anschließender Konkretisierung $\underline{\widetilde{\gamma}}_x$ des lokalen Verhaltens $\mathcal{R}_K$ ergeben und somit alle konkreten lokalen Verhalten darstellen, die unter der Abstraktion $\underline{\widetilde{\alpha}}_x$ gleich dem abstrahierten Verhalten von $\mathcal{R}_K$ sind, gebunden. Für alle diese Verhalten s fordern wir nun, dass sich aus der konkreten Komposition von s mit einem bestimmten zulässigen Restsystemverhalten aus $\mathcal{R}_R$ genau ein nach $\underline{\widetilde{\alpha}}_y$ abstrahiertes Gesamtverhalten $t_{\mathcal{R}_R}$ ergibt.

tion ist die Eigenschaft, dass die Ein- / Ausgaberelation der Spezifikation völlig unabhängig von der Zeit ist, das heißt der Zeitpunkt der Eingaben beeinflusst nicht die Ausgaben modulo deren Zeitpunkte. Zeitunabhängige Modelle können hierbei Zeitbedingungen enthalten, die keine Wirkung auf die Ein- / Ausgaberelation besitzen oder die Modelle enthalten gar keine Zeitbedingungen und sind hierdurch per Konstruktion zeitunabhängig. Durch die Verwendung einer Abstraktion, die vollständig von zeitlichen Beziehung abstrahiert, lässt sich die Zeitrobustheitseigenschaft auch im Sinne der Eigenschaft der Zeitunabhängigkeit für gezeitete Modelle einsetzen.

Definition 6.3 (Zeitrobustheit einer Gesamtspezifikation)**.** Eine Gesamtspezifikation ist zeitrobust, falls alle in ihr enthaltenen atomaren Komponentenspezifikationen, die Gegenstand von Veränderungen durch das Refactoring sind,[5] die Zeitrobustheitseigenschaft erfüllen. Sei S die Menge der in einer Gesamtspezifikation enthaltenen atomaren Komponenten und $\mathit{veränderbar} \in \mathcal{L}_{AF} \rightarrow \{\mathit{wahr}, \mathit{falsch}\}$ eine Abbildung, die für alle entsprechend der Semantik der Modellierungssprache durch den Entwickler veränderbaren atomaren Komponenten den Wahrheitswert *wahr* und für alle nicht veränderbaren Komponenten *falsch* liefert. Dann gilt:

$$\mathit{zeitrobustheitGesamt}_{\alpha_x,\alpha_y,\underline{\mathcal{M}_K},\underline{\mathcal{M}_R}}(S) \stackrel{\text{def}}{=} \forall k \in S : \mathit{veränderbar}(k) \Rightarrow \mathit{zeitrobustheit}_{\alpha_x,\alpha_y,\underline{\mathcal{M}_K},\underline{\mathcal{M}_R}}(k) \tag{6.8}$$

Die Menge aller zeitrobusten AutoFocus Spezifikationen wird mit $\mathcal{L}_{ZR}$ und die Menge aller zeitrobusten atomaren AutoFocus Spezifikationen, die durch den Entwickler verändert werden können, wird mit $\mathcal{L}_V$ bezeichnet.

$$\mathcal{L}_{ZR} = \{a \in \mathcal{L}_A \text{ so that } \mathit{zeitrobustheitGesamt}_{\alpha_x,\alpha_y,\underline{\mathcal{M}_K},\underline{\mathcal{M}_R}}(a)\} \tag{6.9}$$

$$\mathcal{L}_V = \{a \in \mathcal{L}_{ZR} \text{ so that } \mathit{veränderbar}(a)\} \tag{6.10}$$

Ist ein gesamtes System zeitrobust, dann kann lokal das zeitliche Verhalten aller atomaren Komponenten, die Gegenstand von Veränderungen sind, innerhalb der Grenzen dessen, was unter der Abstraktion $\underline{\widetilde{\alpha}_x}$ gleich erscheint, geändert werden, ohne das nach $\underline{\widetilde{\alpha}_y}$ zeitabstrahierte Systemverhalten zu verändern. Wird die Veränderung des lokalen Verhaltens unter Erhaltung der Zeitrobustheitseigenschaft durchgeführt, lassen sich lokale Änderungen kombiniert durchführen, ohne dabei das zeitabstrahierte Systemverhalten zu verändern. Für die Menge der zeitrobusten Spezifikationen ist es somit möglich, komplexe allgemeingültige Refactorings zu definieren, die lokale zeitliche Verhaltensänderungen bewirken. Diese Refactorings sollen ihrerseits die Zeitrobustheitseigenschaft erhalten.

[5] Die Einschränkung auf atomare Komponenten, die Veränderungen unterliegen, ist dadurch begründet, dass eine Modellspezifikation bestimmte Komponenten, wie beispielsweise Puffer, enthalten kann, die ein festes Verhalten aufweisen und deren Verhalten nicht verändert werden soll.

6.2.3. Refactoring von zeitrobusten AutoFocus Modellen

In Kapitel 4 wurde bei der Definition von Refactoring gefordert, dass das zeitabstrahierte Schnittstellenverhalten des Gesamtsystems vor und nach Anwendung des Refactorings gleich bleibt. Durch die Beschränkung auf zeitrobuste AutoFocus Modelle kann jetzt eine Definition von Refactoring angegeben werden, die statt der Äquivalenz des abstrakten Schnittstellenverhaltens der Gesamtspezifikation lediglich die Äquivalenz des abstrakten Verhaltens der durch das Refactoring veränderten Teilspezifikation fordert.

Definition 6.4 (Allgemeingültiges Refactoring von zeitrobusten AutoFocus Modellen)**.** Sei $S \in \mathcal{L}_{ZR}$ eine syntaktisch korrekte Spezifikation eines Gesamtsystems in der Sprache AutoFocus, die zeitrobust bezüglich der Zeitabstraktionen $\underline{\widetilde{\alpha}}_x$ auf Komponentenebene und $\underline{\widetilde{\alpha}}_y$ auf Systemebene und der Menge von komponierbaren Schnittstellenverhalten von Restsystemen $\underline{\mathcal{M}}_R$ sowie der Menge von zulässigen Komponentenschnittstellenverhalten $\underline{\mathcal{M}}_K$ ist. Diese Spezifikation S ist die Menge der in ihr enthaltenen atomaren Komponentenspezifikationen. Sei $K \in (S \cap \mathcal{L}_V)$ die Spezifikation einer atomaren Komponente des Systems, die durch den Entwickler verändert werden kann und *Par* die Menge aller möglichen Tupel von Parametern des Refactorings. Sei ferner $Q \in \mathcal{L}_A \rightarrow \mathbb{N}_0$ eine kompositionale Metrik zur Bewertung der Qualität von Spezifikationen beziehungsweise Teilspezifikationen und sei $Pre_T \in \mathcal{L}_V \times Par \rightarrow \{wahr, falsch\}$ die Vorbedingung, die für die Anwendbarkeit des Refactorings erfüllt sein muss. Die verwendete Eigenschaft *verhaltensäquivalentAF_*α_x ist in der Definition 3.22 auf Seite 79 festgelegt. Dann gilt für ein allgemeingültiges zeitrobustes AutoFocus Refactoring $T \in \mathcal{L}_V \times Par \rightarrow \mathcal{L}_{ZR}$, das auf eine atomaren Komponentenspezifikation innerhalb einer zeitrobusten Gesamtspezifikation angewendet wird:

$$\begin{aligned} &\forall\, S \in \mathcal{L}_{ZR}, P \in Par : \forall\, K \in (S \cap \mathcal{L}_V) : Pre_T(K,P) \Rightarrow \\ &\Big(verhaltensinvarianzZR(T,K,P) \stackrel{\text{def}}{=} verhaltens\ddot{a}quivalentAF_\alpha_x(K, T(K,P)) \Big) \wedge \\ &h\ddot{o}hereQualit\ddot{a}t(T,K,P) \end{aligned} \tag{6.11}$$

Hierbei ist zu beachten, dass die Definition von zeitrobustem Refactoring die Erhaltung der Zeitrobustheitseigenschaft durch die Refactoring Transformation fordert. Diese Forderung ermöglicht die kombinierte Anwendung von zeitrobusten Refactorings. Entsprechend der Definition darf das Refactoring nur auf die atomare Komponenten einer zeitrobusten Gesamtspezifikation angewendet werden, für die eine Veränderbarkeit durch den Entwickler entsprechend der Semantik der Modellierungssprache vorgesehen ist.

6.3. Anforderungen an die neue Semantik der AutoFocus Modellierungssprache

In Abschnitt 6.1 wurde gezeigt, dass sich die Systemverhalten von AutoFocus Instanzmodellen labil gegenüber lokalen Spezifikationsänderungen verhalten und sich damit die Modellierungssprache als problematisch für die Durchführung von Refactoring erweist. Wir legen jetzt Anforderungen an eine abgewandelte Semantik von AutoFocus fest, die lokale zeitliche Verhaltensänderungen unter Beibehaltung eines zeitabstrakten Systemverhaltens ermöglicht und somit das Refactoring besser unterstützt.

Betrachten wir zunächst allgemeine Anforderungen an die Semantik einer Modellierungssprache im Softwareentwurf. Die Semantik soll möglichst einfach, intuitiv und verständlich sein, damit der Entwickler die entworfenen Modellspezifikationen besser verstehen kann und auf diese Weise Spezifikationsfehler vermieden werden. Hierbei ist es sinnvoll, dass sich die Semantik der Modellierungssprache an einem Maschinenmodell orientiert, das die Eigenschaften der Plattform, für die die Software entwickelt wird, wiederspiegelt. In dieser Arbeit wird die Anwendungsdomäne der Reaktiven Systeme und Eingebetteten Systeme betrachtet. In dieser Domäne werden Funktionalitäten über verschiedene Mikrocontroller verteilt realisiert. Die Ausdrückbarkeit der Verteilung von Funktionalitäten unter Verwendung des Konzepts der parallelen Ausführung ist somit eine zentrale Anforderung an die Modellierungssprache und ihre Semantik. Eine weitere allgemeine Anforderung an eine Modellierungssprache und damit auch an ihre Semantik ist die Ausdrucksstärke. Modellspezifikationen sollen in der Sprache kompakt ausdrückbar sein.

Zu diesen allgemeinen Anforderungen kommen nun spezielle Anforderungen an die Veränderbarkeit von Modellspezifikationen hinzu, die Refactoring vereinfachen und die eine Erweiterbarkeit von Spezifikationen gewährleisten. Die in der Modellierungssprache AutoFocus ausdrückbaren Spezifikationen, interpretiert nach der neu zu entwickelnden Semantik, sollen nach Abbildung in das zeitsynchrone AutoFocus zeitrobust (Definition 6.2 auf Seite 127 und 6.3 auf Seite 128) bezüglich geeignet gewählter Zeitabstraktionen sein. Die Zeitrobustheit ermöglicht es, aus der Gleichheit von abstrakten Komponentenverhalten auf die Gleichheit des abstrakten Systemverhaltens schließen zu können. Hierdurch lässt sich der Nachweis der Korrektheit der Verhaltensinvarianzeigenschaft des Refactorings statt auf der Ebene des Gesamtsystems auf der Ebene des veränderten Teilsystems führen. Dies vereinfacht weitgehend die Definition und Durchführung von Refactorings. Die gewählte Zeitabstraktion soll von möglichst vielen zeitlichen Beziehungen abstrahieren, um möglichst weitgehendes Refactoring zu ermöglichen.

Für die Definition einer neuen Semantik von AutoFocus fordern wir darüber hinaus, dass diese auf einfache Weise auf die herkömmliche zeitsynchrone AutoFocus Semantik abgebildet werden kann. Durch diese Anforderung ist sichergestellt, dass Modellspezifikationen, die entsprechend der neuen Semantik interpretiert werden, in herkömmliche zeitsynchrone AutoFocus Spezifikationen übersetzt werden können und

somit die Möglichkeit besteht, zwischen den verschiedenen semantischen Modellen zu wechseln. Zusätzlich bietet die Abbildbarkeit der neuen Semantik auf die alte Semantik den Vorteil, dass die bestehende Werkzeuginfrastruktur von AutoFocus zum großen Teil unverändert weiterverwendet werden kann und sich die Integration der neuen Semantik in das bestehende CASE Werkzeug auf die Realisierung der Übersetzungsabbildung zwischen den zwei Semantiken beschränkt.

6.4. Semantikansätze und deren Auswirkungen auf das Refactoring

In diesem Abschnitt werden verschiedene existierende Ansätze von Semantiken von Modellierungs- und Programmiersprachen betrachtet und deren Eignung für die Durchführung von Refactoring beurteilt. Betrachten wird zunächst die heute vorwiegende Anwendungsdomäne von Refactoring, die objektorientierten Programmiersprachen. Tatsächlich wird in diesem Umfeld die Refactoring-Technik mit großem Erfolg eingesetzt. Die Problematiken der Zeitlabilität, wie sie in AutoFocus Instanzmodellen auftreten können, treten hier nicht beziehungsweise nur in Sonderfällen auf. Das Refactoring in der heutigen Form wird auf Programme angewendet, die dem Paradigma der Funktionsaufrufe folgen. Die Aufteilung der Funktionalität erfolgt durch die Realisierung verschiedener Methoden, die sich gegenseitig aufrufen und über Übergabeparameter und Rückgabewerte miteinander kommunizieren. Ruft eine Hauptmethode eine Untermethode auf, dann wird die Hauptmethode so lange angehalten, bis die Untermethode fertig abgearbeitet ist. Das Refactoring der objektorientierten Sprachen betrachtet rein sequentielle Programme. Daher treten hier nicht die von AutoFocus bekannten Zeitlabilitätsprobleme auf, die das Refactoring erschweren. Leider steht dieses einfache Programmiermodell im Widerspruch zu unserer zentralen Anforderung der Unterstützung von Parallelverarbeitung durch die Modellierungssprache, die sich aus der Anwendungsdomäne der Reaktiven Systeme und Einbetteten Systeme ergibt. Zwar unterstützen die heutigen objektorientierten Sprachen auch Konzepte zur Realisierung von parallelen Programmen, wie beispielsweise die Thread-Programmierung, für diese Art der Programme wird jedoch derzeit keine Refactoring Unterstützung geboten. In einem solchen parallelen Programmiermodell sind ähnliche Problematiken beim Refactoring zu erwarten, wie sie in der jetzigen AutoFocus Semantik auftreten.

Betrachten wir nun verschiedene existierende Modellierungs- und Spezifikationssprachen und deren Semantiken in Hinblick auf ihre Eignung für das Refactoring. Zeitsynchrones Focus mit strenger Kausalität [BS01] besitzt wie AutoFocus einen globalen Systemtakt mit einem Takt Verzögerung der Ausgaben. Hier ergeben sich die gleichen Probleme bezüglich der Veränderbarkeit von Spezifikationen.

In den synchronen Sprachen, wie beispielsweise in Esterel [Ber98], existiert der Begriff des *Perfect Synchrony*. Demnach benötigen Berechnungen keine Zeit und auf Eingaben kann ohne Verzögerung mit Ausgaben reagiert werden. Das Systemmodell beinhaltet

eine Zeitabstraktion und wäre demnach eigentlich sehr gut für Refactoring geeignet. In den synchronen Sprachen sind jedoch keine Rückkopplungen von Werten beziehungsweise Nachrichten erlaubt, die in null Zeit auftreten und Abhängigkeiten zwischen den in Rückkopplung stehenden Ein- und Ausgaben hervorrufen. Diese Sprachen bieten spezielle Synchronisierungsausdrücke an, die eine Rückkopplung von Werten ermöglichen. Diese Synchronisierungsausdrücke bewirken explizit einen Zeitschritt. Durch die Einführung dieses Zeitschritts treten in den synchronen Sprachen die gleichen Zeitprobleme auf, wie bei AutoFocus. Synchrone Sprachen unter Verwendung von Rückkopplungen erweisen sich demnach als ungeeignet für das Refactoring.

Die in dem Werkzeug StateMate verwendete Semantik von *StateCharts* unterscheidet Mikro und Makro Schritte (dort *Steps* und *Super Steps* genannt). Mikro Schritte sind sehr ähnlich zu den Berechnungen, die in synchronen Sprachen in null Zeit durchgeführt werden können. Ein Makro Schritt entspricht dementsprechend in den synchronen Sprachen dem Synchronisationsausdruck. Refactoring innerhalb eines Mikro Schrittes wird durch die *StateCharts* unterstützt, beim Refactoring über einen Makroschritt hinweg tritt jedoch wiederum die gleiche zeitliche Inflexibilität gegenüber Änderungen auf, wie sie bei AutoFocus zu beobachten ist. Des Weiteren wird in *StateCharts* das Refactoring durch das implizite Vorhandensein von Prioritäten von Transitionen in den hierarchischen Zustandsmaschinen erschwert.

Ein weiteres Beispiel für eine Modellierungssprache im Bereich der Reaktiven Systeme ist die *Specification and Description Language (SDL)* [ITU02]. Sie besitzt eine nachrichtenasynchrone und zeitasynchrone Semantik unter Verwendung von unendlich großen Puffern. Es existiert keine direkte zeitliche Beziehung zwischen den verschiedenen Komponenten des modellierten Systems. Aus diesem Grund tritt bei Modellen in dieser Sprache nicht das Phänomen der Zeitlabilität auf. Für Sprachen mit dieser Art von Semantik lassen sich sehr gut Refactoring-Operationen definieren. Die Modellierungssprachen *Real-Time Object-Oriented Modeling (ROOM)* [SGW94] und dessen Nachfolger *UML Real-Time* (UML-RT), die in der *UML 2.0* [OMG04] integriert wurden, verwenden zur Kommunikation ebenfalls Puffer unendlicher Länge. Das ungezeitete Focus [BS01] verwendet ebenfalls ein gepuffertes Kommunikationsmodell.

Durch die unendlich großen Puffer und die zeitliche Entkopplung besitzen Modelle in zeitasynchronen Sprachen jedoch ein sehr großes Maß an Nichtdeterminismus und es besteht die Gefahr, dass in der Ausführung die Puffer immer weiter anwachsen. Diese Nachteile werden jedoch in dieser Arbeit bewusst in Kauf genommen, um ein hohes Maß an Flexibilität der Modelle hinsichtlich der Veränderbarkeit durch Refactoring zu erzielen. In Abschnitt 6.6 wird eine zeitasynchrone Semantik für die Modellierungssprache AutoFocus definiert, die die Zeitrobustheitseigenschaft bezüglich geeigneter Abstraktionen erfüllt und somit durch die Refactoring-Methode gut unterstützt werden kann.

6.5. Wahl von geeigneten Zeitabstraktionen für das AutoFocus Refactoring

In den Abschnitten 3.7.3 bis 3.7.5 sind verschiedene Zeitabstraktionen von AutoFocus Schnittstellenverhalten definiert. Wir wählen jetzt geeignete Abstraktionen als Beobachtungsbegriff für das AutoFocus Modell-Refactoring aus. Um möglichst weitgehende Veränderungen des zeitlichen Verhaltens zu ermöglichen, liegt es nahe, die stärkste Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ zur Beobachtung des Verhaltens einzusetzen. Diese Abstraktion abstrahiert von jeglichen zeitlichen Beziehungen zwischen Nachrichten auf verschiedenen Ports beziehungsweise Kanälen.

Die Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ erfüllt jedoch nicht die Eigenschaft der Kompositionalität (Definition 6.1 auf Seite 124) in Systemen, die Feedback Kommunikation enthalten. Die zeitliche Verschiebung der Ein- und Ausgaben einer Komponente, die unter der Abstraktion nicht sichtbar ist, kann zu einem Deadlock des Gesamtsystems und somit zu einer Verhaltensänderung führen.

Betrachten wir das Beispiel in Abbildung 6.5 auf der nächsten Seite. Die Komponente A zeigt das in dem linken Sequenzdiagramm dargestellte Verhalten. Nach dem Empfang der Nachricht auf dem Port a wird auf den Port d eine Nachricht gesendet, die über einen Feedback Kanal wieder an Port b als Eingabe empfangen wird. Diese Eingabe bewirkt die Ausgabe einer Nachricht auf Port c. Unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ zeigt die Komponente A das in dem zeitabstrakten Sequenzdiagramm in der Mitte der Abbildung dargestellte Verhalten. Die Reihenfolge der vier Nachrichten ist beliebig vertauschbar, da die Nachrichten auf unterschiedlichen Ports auftreten. Eine Komponente, die zuerst auf beide Eingaben auf den Ports a und b wartet und anschließend beide Ausgaben auf den Ports c und d tätigt, verhält sich somit unter dieser Abstraktion gleich zur Komponente A (siehe Sequenzdiagramm auf der rechten Seite der Abbildung). Auf Grund des Feedback Kanals kann diese Komponente jedoch nie eine Eingabe auf dem Port b erhalten, da diese selber hierzu eine Ausgabe auf den Port d tätigen müsste. Dieses System befindet sich in einem Deadlock.

Diese Problematik ergibt sich nicht nur durch direkte Feedback Kommunikation, sondern auch durch indirektes Feedback über mehrere Zwischenkomponenten hinweg. Die Beschränkung des Refactorings auf Systeme, die kein Feedback enthalten, erscheint als nicht sehr praxisorientiert, da diese Art der Kommunikation in sehr vielen Systemen vorhanden ist. Daher wird auf den Einsatz der Abstraktion $\widetilde{\alpha}_{ZA3}$ zur Beobachtung von Komponentenverhalten verzichtet.

Zur Vermeidung der vorangehend beschriebenen Feedback Problematik muss die Reihenfolge zwischen Ein- und Ausgaben unter Abstraktion erhalten bleiben. Die Abstraktion α_{ZA2} (siehe Abschnitt 3.7.4) leistet genau dies. Sie erhält den kausalen Zusammenhang zwischen Eingaben, die verarbeitet werden, und die durch diese bewirkten Ausgaben. Die Anwendung von α_{ZA2} ist auf der Ebene der atomaren Komponenten sinnvoll, da hier der unmittelbare Zusammenhang zwischen Ein- und Ausgaben sichtbar ist. Wendet man die Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ auf komponiertes Verhalten an, kann durch die

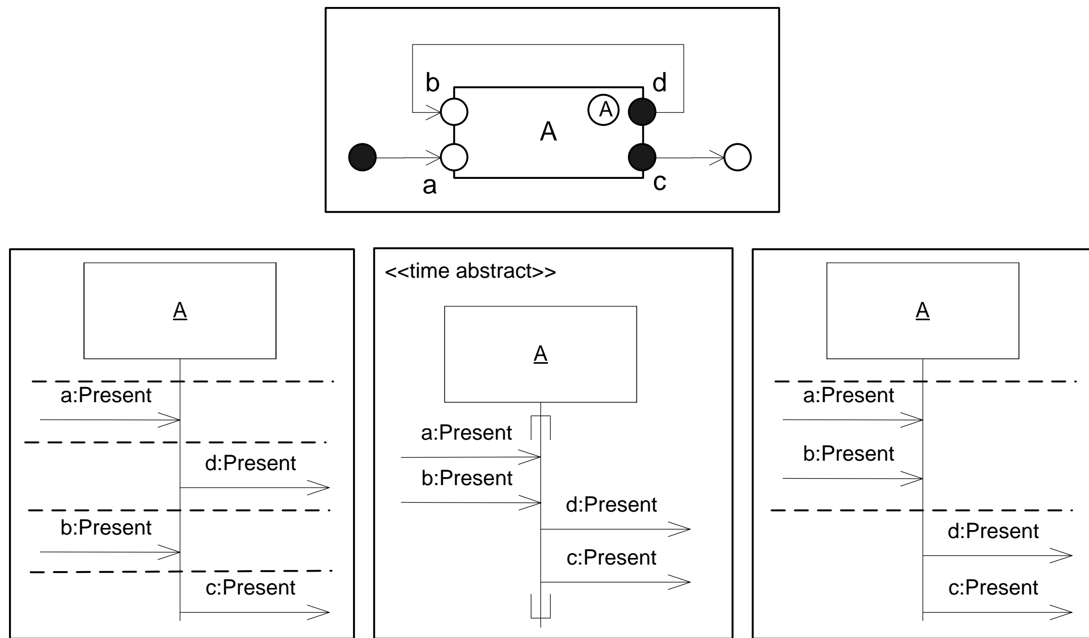


Abbildung 6.5.: Feedback und die Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$.

Abstraktion nicht zwischen tatsächlichen Beziehungen zwischen Ein- und Ausgaben, die im Modell in den Transitionen der Zustandsmaschinen ausgedruckt sind, und zufälligen zeitlichen Beziehungen, die sich aus der parallelen Ausführung von Komponenten ergeben und nicht eine Beziehung im Modell darstellen, unterschieden werden. Die Abstraktion berücksichtigt also alle für sie sichtbaren potentiellen Abhängigkeiten und liefert gegebenenfalls ein abstraktes Verhalten, das stärkeren Einschränkungen unterliegt, als das abstrakte Verhalten, das sich aus der Komposition der abstrakten Verhalten der atomaren Komponenten ergibt. Die Abstraktion ist somit nicht kompositional und die Abstraktion erscheint ungeeignet zur Betrachtung des Verhaltens komplex zusammengesetzter Komponentenstrukturen.

Um möglichst weitgehendes Refactoring zu ermöglichen, entscheiden wir uns für den kombinierten Einsatz der Abstraktionen $\underline{\widetilde{\alpha}}_{ZA2}$ und $\underline{\widetilde{\alpha}}_{ZA3}$. Zur Beobachtung des Systemverhaltens wird die Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ verwendet. Auf der Systemebene spielt die vorangehend angeführte Feedback Problematik keine Rolle, da es sich auf dieser Ebene um die äußerste Schnittstelle des Modells handelt, die in keiner Kommunikationsbeziehung mit anderen Systemen beziehungsweise mit sich selber steht. Die Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ wird zur Betrachtung des Verhaltens von atomaren Komponenten als auch von wenigen zusammengesetzten atomaren Komponenten eingesetzt.

Ziel ist es im Folgenden, eine Semantik in Form einer Übersetzung in das zeitsynchrone AutoFocus zu entwerfen, so dass alle in der neuen Sprache ausdrückbaren Spezifikationen die Zeitrobustheitseigenschaft (siehe Definition 6.2 auf Seite 127) bezüglich $\underline{\widetilde{\alpha}}_{ZA2}$

auf der Ebene der geänderten Komponente und der Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ auf der Ebene des Gesamtsystems erfüllen. Die Menge der lokal veränderbaren Verhalten $\underline{\mathcal{M}}_K$ soll hierbei alle Verhalten atomarer Komponenten enthalten, die durch den Entwickler entsprechend der zu entwickelnden neuen Semantik definiert werden können. Die Menge der mit lokalem Verhalten komponierbaren Restverhalten $\underline{\mathcal{M}}_R$ enthält somit alle Verhalten, mit denen entsprechend der neuen Semantik der Modellierungsprache atomare Komponenten komponiert werden können.

Ist die Eigenschaft der Zeitrobustheit erfüllt, dann ist sichergestellt, dass Änderungen des Verhaltens von Komponenten, die unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ nicht sichtbar sind, auch keine Veränderungen des nach $\underline{\widetilde{\alpha}}_{ZA3}$ abstrahierten Systemverhaltens bewirken. Somit reicht zum Nachweis der Verhaltensäquivalenz der Refactorings dieser Spezifikationen die Betrachtung des Verhaltens auf Komponentenebene unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ aus, um daraus auf die Verhaltensäquivalenz des Gesamtsystems unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ schließen zu können.

6.6. Eine zeitasynchrone AutoFocus Semantik

In diesem Abschnitt wird eine nachrichtenasynchrone und zeitasynchrone Semantik für die Modellierungssprache AutoFocus entwickelt. Die neue Semantik wird durch eine Abbildung von zeitasynchrone AutoFocus Modellspezifikationen auf herkömmliche zeitsynchrone AutoFocus Modellspezifikationen definiert.

Die neue Semantik stellt sicher, dass alle syntaktisch korrekten AutoFocus Modellspezifikationen, die nach der neuen zeitasynchronen Semantik interpretiert werden, die Eigenschaft der Zeitrobustheit (siehe Definition 6.4 auf Seite 129) in Bezug auf die Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ auf Komponentenebene und $\underline{\widetilde{\alpha}}_{ZA3}$ auf Systemebene erfüllen. Modelle, die nach der neuen Semantik interpretiert werden, sind folglich wesentlich flexibler gegenüber zeitlichen Veränderungen und eignen sich sehr gut für die Durchführung von Refactorings, die lokales Komponentenverhalten unter Beibehaltung des Systemverhaltens verändern, als Modelle, die nach der herkömmlichen zeitsynchronen AutoFocus Semantik interpretiert werden.

6.6.1. Notationserweiterung

Die Notation von AutoFocus wird zur Kennzeichnung von zeitasynchronen Modellen erweitert. Eine AutoFocus Modellspezifikation kann entweder nach der herkömmlichen zeitsynchronen AutoFocus Semantik oder nach der hier entworfenen zeitasynchronen Semantik interpretiert werden. Bei der Interpretation wird immer eines der beiden Systemmodelle für die gesamte Modellspezifikation angewendet. Die gemischte Anwendung beider Semantiken ist nicht erlaubt. Wir definieren das Stereotyp *«timeAsync»* zur Kennzeichnung von Spezifikationen, die nach der neuen Semantik interpretiert werden. Um Missverständnissen vorzubeugen, werden alle in einer Spezifika-

tion enthaltenen Diagramme einer nach der zeitasynchronen Semantik interpretierten Spezifikation mit dem Stereotyp *«timeAsync»* versehen. Ist dieses Stereotyp nicht vorhanden, dann wird die herkömmliche zeitsynchrone AutoFocus Semantik angewendet.

In der Notation zeitasynchroner Modellspezifikationen werden die Sprachkonstrukte von AutoFocus für Systemstrukturdiagramme (SSDs) und Zustandsübergangsdiagramme (STDs) weiterhin verwendet. Eine Ausnahme hiervon stellen die Notationserweiterungen von AutoFocus zur Kennzeichnung spezieller Kommunikationssemantiken, wie beispielsweise das *«Immediate»* Stereotyp, dar. Diese speziellen Stereotypen und deren assoziierten speziellen Kommunikationssemantiken dürfen in zeitasynchronen Modellspezifikationen nicht verwendet werden.

In dem zeitsynchronen AutoFocus ist es möglich, das Nichtvorhandensein einer Eingabe durch den Ausdruck *<Portname>?* ohne gefolgten Wert oder Variable als Teil der Eingabebedingung einer Transition zu überprüfen und damit das Feuern der Transition von dem Nichtvorhandensein von Nachrichten abhängig zu machen. Durch die in dieser Arbeit festgelegten Zeitabstraktionen wird genau von diesem Nichtvorhandensein von Nachrichten abstrahiert. Um weiterhin alle Ereignisse, die das Verhalten der Modellspezifikation beeinflussen, beobachten zu können, ist in dem zeitasynchronen AutoFocus die Auswertung des Nichtvorhandenseins von Nachrichten nicht gestattet. Es dürfen in den Eingabebedingungen der Transitionen in den zeitasynchronen Zustandsübergangsdiagrammen nicht Ausdrücke der Form *<Portname>?* ohne nachfolgendem Wert beziehungsweise nachfolgender Variable verwendet werden.

In der zeitasynchronen AutoFocus Notation ist im Gegensatz zu der klassischen zeitsynchronen AutoFocus Notation die Festlegung von zwei zu eins Kommunikationsbeziehungen möglich. Dies geschieht durch das Verbinden zweier Kanäle, die auf der Quellseite mit zwei Ausgabe-Ports von atomaren Komponenten verbunden sind, mit einem gemeinsamen Eingabe-Port (Abbildung 6.6).

Eine weitere notationelle Erweiterung des zeitasynchronen AutoFocus gegenüber dem zeitsynchronen AutoFocus besteht in der Möglichkeit, mehrere Nachrichten auf einen Port innerhalb einer Transition auszugeben. Hierzu werden in dem Ausgabeterm

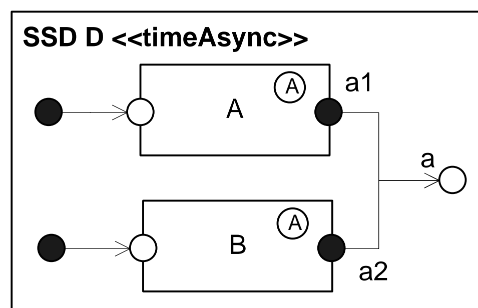


Abbildung 6.6.: Zwei zu eins Kommunikation.

der Transition eines zeitasynchronen Zustandübergangsdiagramms einem gleichen Ausgabe-Port mehrfach Werte zugewiesen. So ist beispielsweise der Ausgabeterm $a!1;\ a!4;\ a!3$ in dem zeitasynchronen AutoFocus zulässig.

6.6.2. Zeitliche Entkopplung

Durch die zeitsynchrone Semantik von AutoFocus besteht eine implizite zeitliche Beziehung zwischen den einzelnen Komponenten des Systems. Jede Komponente feuert pro systemweitem Zeitschritt genau eine Transition.[6] In AutoFocus ist es möglich, Verhalten in Abhängigkeit der impliziten Systemzeit auszudrücken.

Diese implizite Auswertung der Systemzeit steht im Widerspruch zur Zeitrobustheitseigenschaft. Das neue Systemmodell muss folglich genau die Möglichkeit zur Auswertung der Systemzeit und den impliziten Zeitbezug zwischen den verschiedenen Zustandsmaschinen auflösen.

Wir erreichen dies durch die Verwendung von *Stuttering Steps*. *Stuttering Steps* sind ein Konzept zur zeitlichen Entkopplung der Ausführung nebenläufiger Teilsysteme. Die Ausführung der Teilsysteme wird nichtdeterministisch verzögert. *Stuttering Steps* werden beispielsweise in der *Temporal Logic of Actions*(TLA) [Lam93] eingesetzt. In Zustandsmaschinenmodellen können *Stuttering Steps* durch spezielle nichtdeterministische Transitionen in den einzelnen Zustandsmaschinen realisiert werden. Durch die nichtdeterministische von einander unabhängige Verzögerung der Ausführung in den einzelnen Zustandsmaschinen ist die globale Systemzeit in den einzelnen Komponenten nicht mehr sichtbar und kann somit die Ausführung nicht beeinflussen.

Zur Realisierung der *Stuttering Steps* in AutoFocus werden alle Zustände aller Zustandsautomaten einer nach der zeitasynchronen Semantik interpretierten Modellspezifikation um eine Transition zur Verzögerung der Ausführung erweitert.[7] Die Variablenbedingung und die Eingabebedingung der *Stuttering Step* Transition sind leer und damit immer erfüllt. Die Transition tätigt keine Ausgaben, verändert keine lokalen Variablen und ändert auch den aktuellen Kontrollzustand nicht. Die Abbildung 6.7 auf der nächsten Seite zeigt die Transformation einer zeitasynchronen AutoFocus Zustandsmaschine in eine zeitsynchrone AutoFocus Zustandsmaschine durch Einfügen von *Stuttering Steps*.

Die eingefügten *Stuttering Steps* stellen nach der zeitsynchronen AutoFocus Semantik einen expliziten Nichtdeterminismus dar. In jedem Systemtakt kann entweder ein *Stuttering Step* ausgeführt und somit die Verarbeitung verzögert werden oder es wird eine Transition gefeuert, die einen Fortschritt der Verarbeitung beziehungsweise der Berechnung darstellt. Die AutoFocus Semantik besitzt keinen Fairnessbegriff für das Feuern

[6] Ist keine der vom aktuellen Kontrollzustand ausgehenden Transitionen schaltbereit, dann wird in AutoFocus die implizite *Idle* Transition ausgeführt.

[7] Die Zustandsmaschinen der in den folgenden Abschnitten festgelegten speziellen Puffer- und Multiplex-Komponenten sind von dem Hinzufügen der *Stuttering Steps* ausgenommen, da diese nicht Bestandteil des durch den Entwickler spezifizierten Modells sind.

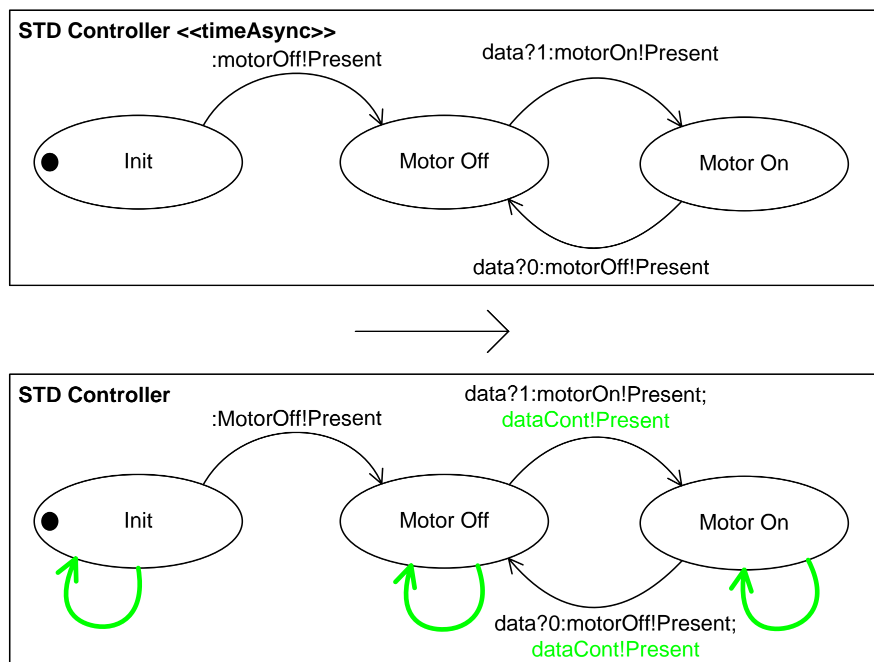


Abbildung 6.7.: Zeitliche Entkopplung durch *Stuttering Steps* und Einfügen der Verarbeitungsnachrichten.

von explizit spezifizierten nichtdeterministischen Transitionen. Es ist folglich möglich, dass in einer unendlich langen Ausführung des Systems eine Komponenten nie arbeitet, sondern immer nur die Verarbeitung durch *Stuttering Steps* verzögert. Somit sind ohne Existenz eines Fairnessbegriffs auf Grund der *Stuttering Steps* keine *Liveness* Eigenschaften erfüllbar.

Um trotzdem in der hier vorgestellten zeitasynchronen Semantik über *Liveness* Eigenschaften Aussagen machen zu können, führen wir an dieser Stelle einen Fairness Begriff speziell für die Ausführung von *Stuttering Steps* ein. Es ist demnach in keiner unendlich langen Ausführung einer Zustandsmaschine von zeitasynchronen AutoFocus Modellspezifikationen zu jedem beliebigen Zeitpunkt t möglich, dass ab diesem Zeitpunkt t nur noch *Stuttering Step* Transitionen gefeuert werden, vorausgesetzt andere Transitionen sind ebenfalls schaltbereit, das heißt die Zustandsmaschine befindet sich nicht in einem *Dead Lock*.

6.6.3. Pufferkomponenten

Entsprechend der zeitsynchronen Semantik von AutoFocus können Nachrichten zu dem Zeitpunkt ihres Auftretens entweder verarbeitet werden oder sie gehen verloren. Das Verlorengehen von Nachrichten ist einer der Gründe, die zur Labilität von AutoFocus gegenüber lokalen Änderungen des zeitlichen Verhaltens (siehe Abschnitt 6.1) führen. In der neuen zeitasynchronen Semantik wird durch unendliche Eingabepuffer sichergestellt, dass Nachrichten bis zu ihrer Verarbeitung zwischengespeichert werden und somit nicht verloren gehen können.

Eine Modellspezifikation, die mit dem Stereotyp *«timeAsync»* versehen ist, lässt sich in eine herkömmliche zeitsynchrone AutoFocus Modellspezifikation übersetzen. Hierzu wird jedem Eingabe-Port aller atomaren Komponenten der Spezifikation eine Pufferkomponente vorgeschaltet.

Die Abbildung 6.8 auf der nächsten Seite zeigt das Einfügen der Pufferkomponente Buffer in einem Systemstrukturdiagramm. Über einen zusätzlichen Kommunikationskanal zwischen Empfänger und Puffer (Port *name*Cont) wird der Puffer über die Verarbeitung der gepufferten Nachricht informiert. Für die Benachrichtigung über die Verarbeitung wird die *Immediate* Kommunikationssemantik eingesetzt. Die Verwendung der *Immediate* Kommunikationssemantik ist erforderlich, um unmittelbar nach der Verarbeitung der Nutznachricht in der Empfängerkomponente noch innerhalb des selben AutoFocus Taktes die entsprechende Nachricht aus dem Puffer zu löschen und im darauf folgenden Takt der Empfängerkomponente die selbe Nachricht nicht ein zweites Mal zur Verfügung zu stellen. Dadurch wird eine Doppelauswertung der Nutznachricht verhindert.[8]

[8]Es ist auch möglich, ohne die Verwendung von *Immediate* Kommunikation die Puffer in der AutoFocus Semantik darzustellen. Statt der Verwendung der *Immediate* Semantik auf dem *name*Cont Port wird bei diesem Ansatz die Zustandsmaschine der Empfängerkomponente um Zwischenzustände erweitert. Hierdurch wird erreicht, dass die Empfängerkomponente nach Verarbeitung einer Nachricht einen

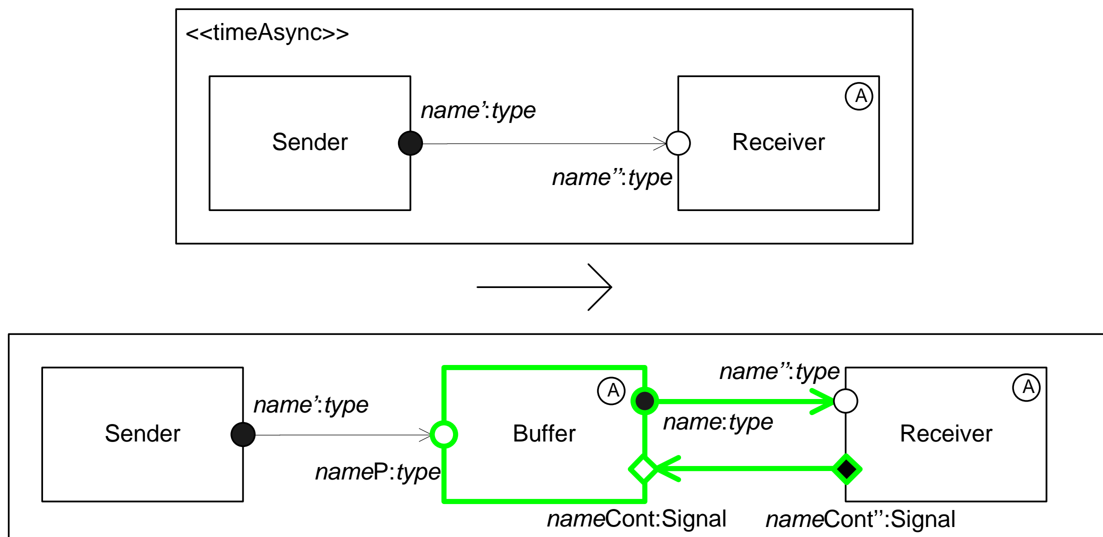


Abbildung 6.8.: Einfügen einer Pufferkomponente in ein SSD.

6.6.4. Das Verhalten der Pufferkomponente

Das Verhalten der Pufferkomponente ist durch ein AutoFocus STD nach der herkömmlichen zeitsynchronen Semantik beschrieben. Diese Komponente arbeitet folglich nicht mit *Stuttering Steps*. Alle eingehenden Nachrichten werden zwischengespeichert und nach dem FIFO Prinzip wieder ausgegeben. Um eine zeitliche Entkopplung von Nachrichten auf verschiedenen Ports zu erreichen, kann die Pufferkomponente das Weiterleiten von Nachrichten nichtdeterministisch verzögern.

Die Ausgabe der zwischengespeicherten Nachrichten an die empfangende Komponente erfolgt durch Dauersignale.[9] Wird eine Nachricht von der Empfängerkomponente verarbeitet, so sendet diese eine Benachrichtigung an die Pufferkomponente (Port *name*Cont′) unter Verwendung der Immediate Kommunikationssemantik. Die Pufferkomponente löscht daraufhin die verarbeitete Nachricht und sendet über den Port *name* die in dem Puffer als nächstes gespeicherte Nachricht. Das Senden der nächsten gespeicherten Nachricht kann an dieser Stelle des Ablaufs verzögert werden. Während der Verzögerung wird von dem Puffer keine Nachricht ausgegeben. Ist in dem Puffer keine Nachricht zwischengespeichert, dann wird ebenfalls keine Nachricht ausgegeben.

Zur Realisierung des Puffers werden rekursiv definierte Listen verwendet. Da AutoFocus keine polymorphen Datentypen unterstützt, ist für jeden Datentyp, der gepuffert wird, ein separater Listen Datentyp erforderlich. Im Anhang C.1 ist der für die Rea-

Systemtakt wartet, während die Pufferkomponente die verarbeitete Nachricht löscht und dem Empfänger die nächste Nachricht zur Verfügung stellt.

[9] Unter einem Dauersignal wird in diesem Zusammenhang ein wiederholtes Senden der gleichen Nachrichten innerhalb eines Zeitintervalls verstanden.

lisierung eines *Integer* Puffers nötige *Integer* Listen Datentyp zusammen mit von dem Puffer benötigten Funktionen in der AutoFocus Datentypsprache Quest/F aufgeführt.

Die Pufferkomponente besitzt eine lokale Variable *bufferVar* vom Typ *IntList*. Die Konstruktorfunktion $Cons(Int, IntList) \rightarrow IntList$ fügt ein neues Element vorne in die Liste ein. Die Funktion $lastIntElement(IntList) \rightarrow Int$ liefert das letzte Element der Liste und mit $deleteLastInt(IntList) \rightarrow IntList$ lässt sich das letzte Element der Liste löschen. Die Funktion $intListLength(IntList) \rightarrow Int$ liefert die Anzahl der Elemente in der Liste.

Die AutoFocus Zustandsmaschine des Puffers ist in Abbildung 6.9 dargestellt und besteht aus den Zuständen *Empty*, *Send* und *Delay*. Die Transitionen der Zustandsmaschine sind im Anhang C.2 aufgeführt.

Wie bei den vorangehend verwendeten *Stuttering Steps* legen wir auch für die Verzögerung der Kommunikation durch die Puffer einen Fairness Begriff fest. Zu jedem beliebigen Zeitpunkt *t* innerhalb einer unendlich langen Ausführung ist es nicht möglich, dass sich ab diesem Zeitpunkt *t* der Puffer nur noch in dem *Delay* Zustand befindet.

6.6.5. Verarbeitung von Nachrichten

Beim Einfügen der Pufferkomponenten wurde ein zusätzlicher Kanal zwischen der Empfängerkomponente und der Pufferkomponente eingefügt, der zur Benachrichtigung über die Verarbeitung einer Nachricht dient (siehe Abschnitt 6.6.3). Die Zustandsmaschine der Empfängerkomponente muss nun um das Senden der Verarbeitungsbenachrichtigung erweitert werden.

Zu diesem Zweck wird in jeder Transition der Zustandsmaschine für jeden in dem Eingabemuster der Transition vorkommenden Port-Bezeichner die Transitionsausgabe um eine entsprechende *name*Cont!Present Nachricht erweitert (siehe Abbildung 6.7 auf Seite 138).

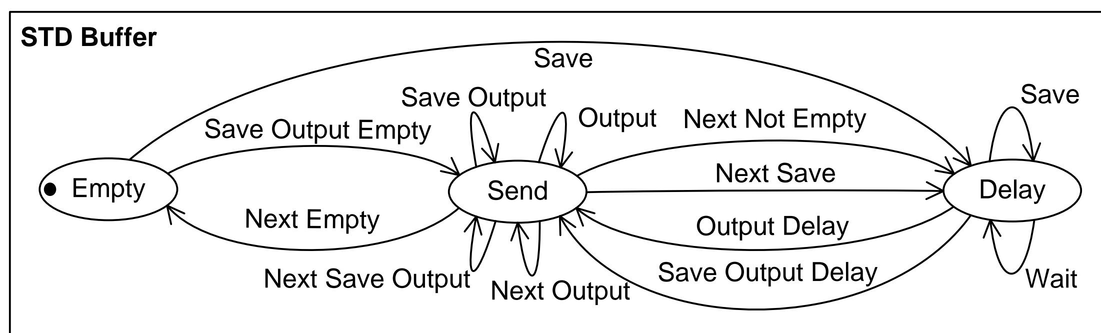


Abbildung 6.9.: STD der Pufferkomponente.

6.6.6. Zwei zu eins Kommunikationsbeziehungen

In zeitsynchronen AutoFocus Systemstrukturdiagrammen (SSDs) besteht nicht die Möglichkeit, einen Eingabe-Port gleichzeitig mit zwei eingehenden Kanälen zu verbinden. Nach der zeitsynchronen Semantik erscheint diese Art der Kommunikation nicht sehr sinnvoll, da in dieser Semantik pro Zeitschritt genau eine Nachricht über einen Port geschickt werden kann und nicht verarbeitete Nachrichten verloren gehen. In dem Fall, dass über beide in einen Port eingehende Kanäle gleichzeitig eine Nachricht gesendet wird, müsste nichtdeterministisch eine der beiden Nachrichten weitergeleitet und die andere Nachricht ignoriert werden. Um Spezifikationsfehler auf Grund dieses Nichtdeterminismus zu umgehen, sind zwei zu eins Kommunikationsbeziehungen in dem zeitsynchronen AutoFocus nicht erlaubt.

Nach der neuen zeitasynchronen AutoFocus Semantik ist eine zwei zu eins Kommunikationsbeziehung jedoch durchaus sinnvoll, da in dieser Semantik Nachrichten nicht verloren gehen und der Zeitpunkt von Eingaben variabel ist. Für den Fall, dass zwei Nachrichten über zwei Kanäle gleichzeitig von einem Port empfangen werden, können diese beiden Nachrichten in beliebiger Reihenfolge zueinander weitergeleitet werden. Eine zwei zu eins Kommunikationsbeziehung in der zeitasynchronen AutoFocus Semantik ist durch eine Transformation in das zeitsynchrone AutoFocus unter Hinzufügung einer Multiplexing Komponente definiert (siehe Abbildung 6.10 auf der nächsten Seite).

Die Komponente *Mux* agiert als ein Puffer, der alle auf den Ports $o1$ und $o2$ eingehenden Nachrichten zwischenspeichert und nach dem FIFO Prinzip an den Port o ausgibt. Die Reihenfolge der Nachrichten auf den Ports $o1$ und $o2$ zueinander bleibt durch die Pufferung erhalten. Treten gleichzeitig Nachrichten auf den Ports $o1$ und $o2$ auf, dann wird nichtdeterministisch eine Reihenfolge zwischen diesen beiden Nachrichten festgelegt. Durch die Multiplexing Komponente erfolgt keine nichtdeterministische Verzögerung, Nachrichten werden nicht bis zu ihrer Verarbeitung zwischengespeichert, sondern frühestmöglich weitergeleitet und die Komponente verwendet keine *Stuttering Steps* Transitionen. Für die Kommunikation zwischen den sendenden Komponenten und der Multiplexing Komponente wird das *Immediate* Kommunikationsmodell verwendet. Dies hat den Vorteil, dass, vorgesetzt keine gleichzeitigen Nachrichten werden von der Multiplexing Komponente empfangen, diese das zeitliche Verhalten der Spezifikation nicht durch eine zusätzliche Verzögerung verändert.

Analog zu den in den Abschnitten 6.6.3 und 6.6.4 definierten Eingabepuffern besitzt die *Mux* Komponente eine lokale Variable mit dem Namen *bufferVar*, in der Listen von Werten von dem Datentyp der Eingabe-Ports $o1$ und $o2$ gespeichert werden. Die Datentyp- und Funktionsdefinition erfolgt identisch zu der im Anhang C.1 aufgeführten Definition, die in Abschnitt 6.6.4 bereits näher erläutert wurde. Die Zustandsmaschine der *Mux* Komponente besitzt einen Zustand. Die Transitionen der Zustandsmaschine sind im Einzelnen im Anhang C.3 aufgeführt.

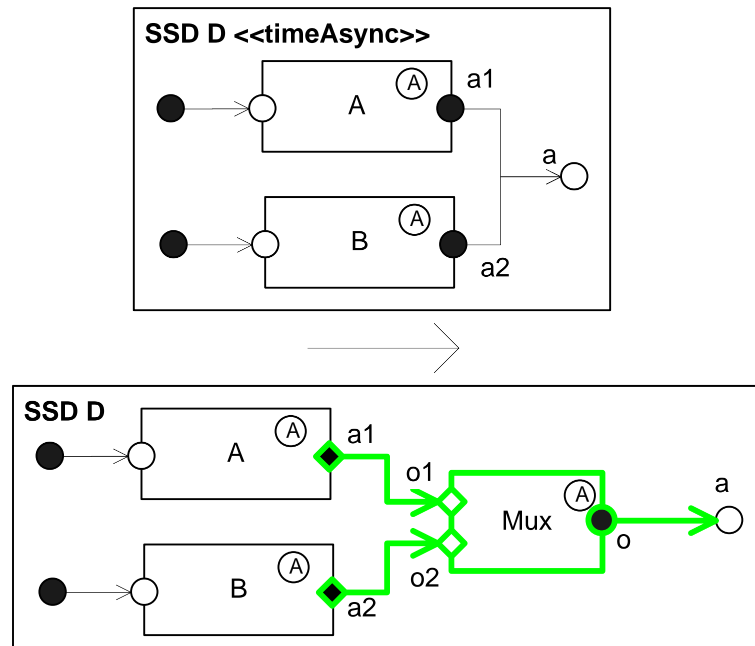


Abbildung 6.10.: Darstellung von zwei zu eins Kommunikation durch eine Multiplex Komponente.

6.6.7. Sequentielles Senden von Nachrichten

In der zeitsynchronen AutoFocus Modellierungssprache kann in dem Ausgabeterm einer Transition auf einen bestimmten Port immer nur eine einzige Nachricht ausgegeben werden. In der zeitasynchronen Variante ist auf Grund der zeitlichen Entkopplung von Verarbeitung und Kommunikation ein sequentielles Senden mehrerer Nachrichten auf einen Port jedoch durchaus sinnvoll. Das sequentielle Senden lässt sich durch eine Abbildung von zeitasynchronen AutoFocus Spezifikationen in wiederum zeitasynchrone AutoFocus Spezifikationen äquivalenten Verhaltens als rein notationelle beziehungsweise syntaktische Abkürzung formulieren.

Ist in dem Ausgabeterm einer Transition der gleiche Ausgabe-Port über dessen Bezeichner mehrmals referenziert, dann wird dies als sequentielles Senden der dort angegebenen Nachrichten über diesen einen Port unter Beibehaltung der im Term verwendeten Reihenfolge interpretiert. Die Abbildung 6.11 auf der nächsten Seite zeigt die Definition des sequentiellen Sendens von Nachrichten auf einen Port innerhalb einer Transition. Eine Transition, die n Nachrichten sequentiell auf einen gleichen Port sendet, wird interpretiert als n Transitionen, die sequentiell über $n-1$ Zwischenzustände abgearbeitet werden. Die erste dieser Transitionen besitzt hierbei zur ursprünglichen Transition den gleichen Variablenvorbedingungsterm, den gleichen Eingabebedingungsterm und den gleichen Variablenzuweisungsterm. Der Ausgabeterm enthält die Zuweisung der ers-

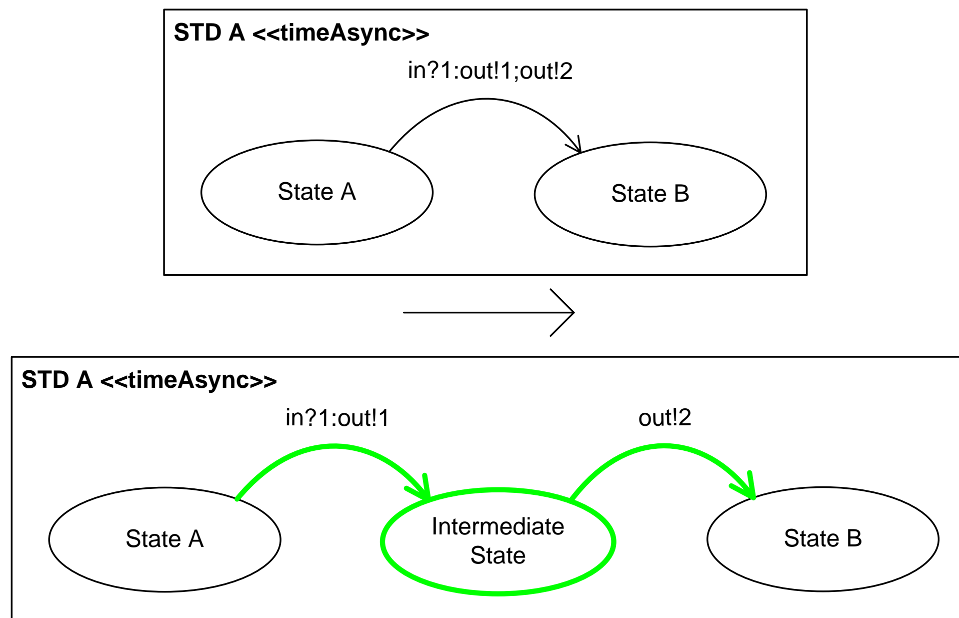


Abbildung 6.11.: Darstellung von sequentiellem Senden innerhalb einer Transition.

ten Nachricht der im Term der ursprünglichen Transition vorkommenden Nachrichten, die auf den gleichen Port gesendet werden sollen. Alle Weiteren sequentiell angeordneten Transitionen besitzen leere Variablenvorbedingungsterme, Eingabebedingungsterme und Variablenzuweisungsterme. In den Ausgabetermen dieser Transitionen wird jeweils eine der im Ausgabeterm der ursprünglichen Transition vorkommenden Nachrichten unter Einhaltung der dort festgelegten Reihenfolge gesendet.

6.7. Erfüllung der Zeitrobustheitseigenschaft

In dem vorangehenden Abschnitt wurde eine zeitasynchrone Semantik durch deren Abbildung in das zeitsynchrone AutoFocus definiert. Ziel der Semantikdefinition ist die Erfüllung der Zeitrobustheitseigenschaft (Definition 6.2 auf Seite 127) bezüglich der Abstraktionen $\underline{\widetilde{\alpha}}_{ZA2}$ angewendet auf Komponentenverhalten und $\underline{\widetilde{\alpha}}_{ZA3}$ angewendet auf Systemverhalten und einer Menge von durch die Semantik der Sprache festgelegten zulässigen komponierbaren Verhalten von Restsystemen $\underline{\mathcal{M}}_R$ sowie einer durch die Semantik festgelegten Menge von zulässigen Schnittstellenverhalten von Komponenten $\underline{\mathcal{M}}_K$ für alle in der Modellierungssprache des zeitasynchronen AutoFocus ausdrückbaren Modellspezifikationen auf der Ebene der konkreten Schnittstellenverhalten.

Im Folgenden werden die in dem zeitasynchronen AutoFocus explizit durch den Entwickler spezifizierbaren atomaren Komponenten als Verarbeitungskomponenten

bezeichnet. Die zeitasynchrone Semantik verwendet Pufferkomponenten zur zeitlichen Entkopplung der einzelnen Verarbeitungskomponenten. Diese Pufferkomponenten sind implizit für jeden Eingabe-Port in dem zeitasynchronen AutoFocus vorhanden und deren Verhalten ist fest durch die Semantik vorgegeben und kann nicht durch Änderungen in der Modellspezifikation verändert werden. Das Verhalten der Puffer ist somit nicht Gegenstand der Veränderungen durch das Refactoring. Diese Pufferkomponenten müssen somit nicht die Zeitrobustheitseigenschaft erfüllen (vergleiche Eigenschaft *veränderbar* in Definition 6.3 auf Seite 128). Zur Erfüllung der Zeitrobustheitseigenschaft werden die einzelnen Verarbeitungskomponenten isoliert von den vorgeschalteten Eingabepuffern betrachtet. Als die Menge der zulässigen Komponentenverhalten $\underline{\mathcal{M}}_K$ wird die Menge aller konkreten gezeiteten Schnittstellenverhalten, die eine Verarbeitungskomponente nach der zeitasynchronen AutoFocus Semantik aufweisen kann, eingesetzt. Diese Menge umfasst alle Verhalten beliebiger atomarer AutoFocus Komponenten, deren Zustandsmaschinen *Stuttering Step* Transitionen in jedem Zustand besitzen. Die den atomaren Komponenten vorgeschalteten Eingabepuffer werden bei den in dieser Menge enthaltenen Verhalten wiederum nicht berücksichtigt.

Als die Menge des komponierbaren Verhaltens von Restsystemen $\underline{\mathcal{M}}_R$ wird die Menge aller konkreten gezeiteten Schnittstellenverhalten, mit dem eine Verarbeitungskomponente in dem zeitasynchronen AutoFocus komponiert werden kann, angenommen. Diese Menge umfasst die Verhalten aller möglichen zeitasynchronen AutoFocus Spezifikationen, deren Ausgabe-Ports jeweils Eingabe-Puffer entsprechend den Abschnitten 6.6.3 und 6.6.4 nachgeschaltet sind.

Die speziellen *Continue* Nachrichten, die zur Kommunikation zwischen den Verarbeitungskomponenten und den Eingabepuffern eingesetzt werden, um das nächste im Puffer gespeicherte Datum bereitzustellen, werden in der nachfolgenden Betrachtung der Verhalten aus Gründen der Übersichtlichkeit nicht dargestellt. Diese Informationen sind bereits redundant über die Verarbeitungsnachrichten auf den in der Übersetzung von AutoFocus nach Focus eingeführten Verarbeitungskanälen beziehungsweise der Berücksichtigung von nur verarbeiteten Eingaben (Abstraktion $\underline{\widetilde{\alpha}}_{NI}$) in dem Verhalten sichtbar.

Die Zeitrobustheitseigenschaft (Definition 6.2 auf Seite 127) fordert, dass alle zeitlichen Veränderungen der Verarbeitungskomponenten, die unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ nicht sichtbar sind und in der Menge $\underline{\mathcal{M}}_K$ enthalten und somit entsprechend der Semantik zulässig sind, nicht zu Veränderungen der Gesamtverhalten unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$, die aus der Komposition des lokal geänderten und aller beliebigen in der Menge der zulässigen mit dem lokalen Verhalten komponierbaren Verhalten $\underline{\mathcal{M}}_R$ enthaltenen Verhalten gebildet werden, führen.

6.7.1. Wirkung von Stuttering Steps und Verzögerung der Kommunikation

Betrachten wir zunächst die direkten Wirkungen von *Stuttering Steps* und Verzögerung der Kommunikation, die zur Realisierung der zeitasynchronen AutoFocus Semantik eingesetzt werden. Die verzögernden Puffer bewirken eine zeitliche Entkopplung zwischen dem Zeitpunkt des Empfangs einer Eingabe durch den Puffer und deren Bereitstellung zur Verarbeitung in der Verarbeitungskomponente. Somit können Eingaben zu einem beliebig späteren Zeitpunkt verarbeitet werden.

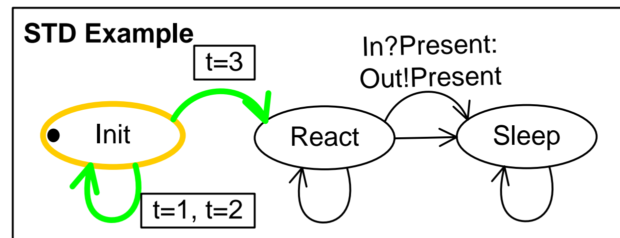
Die Wirkung der *Stuttering Steps* soll an einem konkreten Beispiel verdeutlicht werden. Betrachten wir die Verarbeitung einer Eingabe in einer Verarbeitungskomponente ohne Berücksichtigung ihrer Eingabepuffer. Die Abbildung 6.12 auf der nächsten Seite zeigt verschiedene alternative Abläufe einer Zustandsmaschine, die *Stuttering Step* Transitionen enthält und zum Zeitpunkt $t = 3$ die Eingabe *Present* auf dem Port *In* erhält. Im ersten Fall, der in Abbildung 6.12a gezeigt wird, wird zu den Zeitpunkten $t = 1$ und $t = 2$ zunächst die *Stuttering Step* Transition des Zustands *Init* ausgeführt. Zum Zeitpunkt $t = 3$, zu dem die Eingabe zur Verfügung steht, befindet sich die Zustandsmaschine noch immer im Zustand *Init* und kann auf diese nicht reagieren, da eine Reaktion auf die Eingabe nur im Zustand *React* vorgesehen ist. Die Zustandsmaschine ist in dem hier gezeigten Fall noch nicht bereit zur Verarbeitung der Eingabe und die Eingabe geht verloren.

Die Abbildung 6.12b zeigt einen Ablauf, bei dem zum Zeitpunkt $t = 1$ kein *Stuttering Step* ausgeführt wird, sondern ein Zustandswechsel hin zu dem Zustand *React*. Anschließend wird die *Stuttering Step* Transition des Zustands *React* gefeuert, so dass sich die Zustandsmaschine zum Zeitpunkt der Eingabe ($t = 3$) weiterhin in diesem Zustand befindet und durch Feuern der Transition *In?Present:Out!Present* die Eingabe verarbeitet und eine Ausgabe erzeugt. In dieser Ablaufalternative wird also die Eingabe verarbeitet.

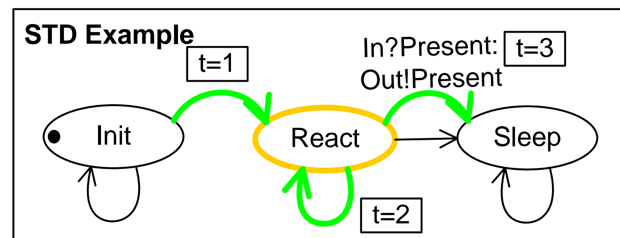
In Abbildung 6.12c ist ein Ablauf zu sehen, in dem die Zustandsmaschine zum Zeitpunkt $t = 3$ bereits in den Zustand *Sleep* übergegangen ist. Die Eingabe kann hier nicht mehr verarbeitet werden.

Die hier gezeigten drei alternativen Abläufe lassen sich in dem Beispiel erst beobachten, wenn die Eingabe frühestens zum Zeitpunkt $t = 3$ erfolgt. Für eine Eingabe zum Zeitpunkt $t = 2$ lässt sich der Fall des nicht mehr Verarbeitens nicht mehr konstruieren. Für eine Eingabe zum Zeitpunkt $t = 1$ fällt ebenfalls der Fall der Verarbeitung der Eingabe in dem hier gezeigten Beispiel weg.

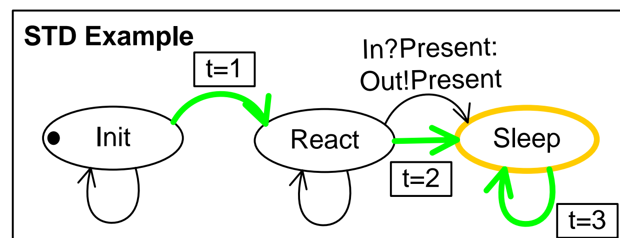
Verallgemeinern wir nun das Beispiel auf eine beliebige Verarbeitungskomponente, deren Verhalten durch eine Zustandsmaschine mit *Stuttering Step* Transitionen beschrieben ist, und beliebige Sequenzen von Eingaben. Als Konsequenz der *Stuttering Steps* enthält das Verhalten der Verarbeitungskomponente für alle entsprechend der Datentypen der Eingabe-Ports der Komponente möglichen Tupel von Eingabeströmen (Sequenzen von Eingaben für die verschiedenen Eingabe-Ports) Abläufe, so dass ein Zeit-



(a) Komponente noch nicht zur Verarbeitung bereit.



(b) Komponente zur Verarbeitung bereit.



(c) Komponente nicht mehr zur Verarbeitung bereit.

Abbildung 6.12.: Ausführungsalternativen eines STDs unter Verwendung von *Stuttering Steps*.

punkt existiert, an dem alle durch die Verarbeitung eines gewählten Tupels von Eingabeströmen und deren Präfixe erreichbaren Zustände (einschließlich Variablenzustände) in den verschiedenen Abläufen aktiv sind. Eine neue Eingabe zu diesem Zeitpunkt kann folglich in Abhängigkeit dieser verschiedenen aktiven Zustände zu unterschiedlichen Reaktionen in Form der Ausführung unterschiedlicher Transitionen, die von diesen Zuständen ausgehen, mit unterschiedlichen Folgezuständen und unterschiedlichen Ausgaben führen. Auf Grund der verzögernden Eingabepuffer, die der Verarbeitungskomponente vorgeschalteten sind, kann jede Eingabe so lange verzögert werden, bevor sie an die Verarbeitungskomponente weitergeleitet wird, bis der Zeitpunkt erreicht ist, an dem alle diese vorangehend beschriebenen Zustände aktiv sein können und die Eingabe zu allen beschriebenen Ausgabealternativen führen kann. Die Zeitpunkte von Eingaben an eine betrachtete Einheit aus Pufferkomponenten und Verarbeitungskomponente haben folglich keinen Einfluss auf die resultierenden zeitabstrahierten Ausgaben, sondern lediglich auf die frühest möglichen Zeitpunkte von Ausgaben. Das Verhalten einer Einheit aus Eingabepuffern und Verarbeitungskomponente toleriert folglich beliebige zeitliche Verschiebungen der Eingaben unter Beibehaltung der Reihenfolge von Eingaben auf gleiche Ports und enthält in den Abläufen alle vorangehend angesprochenen Ausgabealternativen modulo Zeitverschiebungen. Eine Verschiebung der Zeitpunkte der Eingaben an die Puffer unter Beibehaltung der Reihenfolge von Eingaben auf einzelnen Ports bewirkt keine Veränderung des Verhaltens der Verarbeitungskomponente unter der Abstraktion $\widetilde{\underline{\alpha}}_{ZA3}$.

6.7.2. Lokale Wirkung zeitlicher Veränderungen

In dem Verhalten einer Verarbeitungskomponente einer zeitasynchronen AutoFocus Spezifikation existiert zu jedem Ablauf ein Ablauf minimaler zeitlicher Verzögerung. Das ist der Ablauf, in dem nur *Stuttering Step* Transitionen ausgeführt werden, falls keine andere Transition ausgehend vom aktuellen Zustand schaltbereit ist. Zu jedem dieser minimal zeitlich verzögerten Abläufe existieren auf Grund der *Stuttering Steps* in dem Verhalten Abläufe, die alle Kombinationen von Verzögerungen unter Einhaltung der Fairness Bedingung aufweisen. Diese zusätzlichen Verzögerungen verschieben die Zeitpunkte der Verarbeitung von Eingaben und die Zeitpunkte der daraus resultierenden Ausgaben, sind jedoch unter der Zeitabstraktion $\underline{\alpha}_{ZA2}$ identisch.

Wir betrachten im Folgenden die Wirkungen von zeitlichen Spezifikationsänderungen atomarer Komponenten, deren Verhaltensänderungen unter $\widetilde{\underline{\alpha}}_{ZA2}$ nicht sichtbar sind. Es werden vier Fälle der lokalen zeitlichen Veränderungen unterschieden:

Fall 1: Hinzufügen eines Leertaktes.

Die Spezifikation eines zusätzlichen Leertaktes in dem Verhalten der Komponente, die durch das Einfügen eines zusätzlichen Zwischenzustands mit anschließender leerer Transition erreicht werden kann, führt in einem minimal verzögerten Ablauf des Verhaltens zu einer zusätzlichen Verzögerung um einen Systemtakt der Verarbeitung der nachfolgenden Eingaben und der Erzeugung der nachfol-

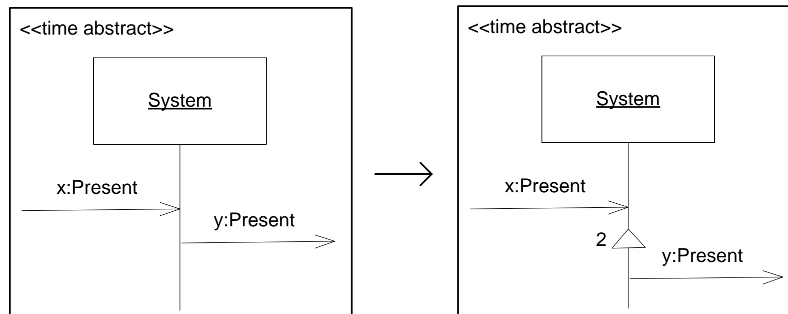


Abbildung 6.13.: Hinzufügen eines Leertaktes.

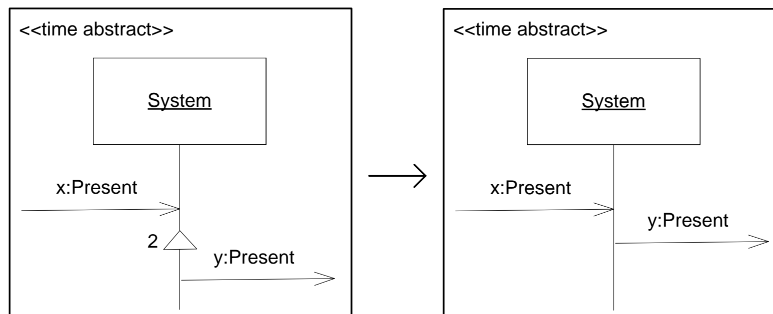


Abbildung 6.14.: Entfernen eines Leertaktes.

genden Ausgaben. Der minimale Ablauf der Modellspezifikation vor der Änderung ist auf Grund der erzwungenen Verzögerung in dem neuen Verhalten nicht mehr enthalten. Die hier beschriebene zeitliche Veränderung ist in Abbildung 6.13 auf der vorherigen Seite mit Hilfe eines zeitabstrakten AutoFocus Sequenzdiagramms dargestellt.

Fall 2: Entfernen von Leertakten.
Analog hierzu hat das Entfernen einer leeren Transition einschließlich eines Zwischenzustands auf einen minimal verzögerten Ablauf die Wirkung des Entfernens eines Leertaktes und somit das Vorziehen der nachfolgenden Ein- und Ausgaben um einen Systemtakt (siehe Abbildung 6.14 auf der vorherigen Seite). Der ursprüngliche minimal verzögerte Ablauf ist weiterhin in dem neuen Verhalten auf Grund der *Stuttering Steps* enthalten, erfüllt jedoch nicht mehr die Eigenschaft der minimalen Verzögerung.

Fall 3: Vertauschen der Reihenfolge aufeinander folgender Eingaben.
In diesem Fall wird die Reihenfolge der Verarbeitung zweier unmittelbar aufeinander folgender Eingaben auf unterschiedlichen Ports vertauscht, wobei keine Ausgaben zwischen den beiden Eingaben auftreten. Dies kann in einer Spezifikation durch das Vertauschen der Eingabebedingungen zweier aufeinander folgender Transitionen erfolgen, wobei die erste der beiden Transitionen keinen Ausgabeterm besitzen darf. Hierbei ändern sich in einem minimal verzögerten Ablauf lediglich die Zeitpunkte der Verarbeitung der beider Eingaben und das restliche zeitliche Verhalten bleibt gleich.

Fall 4: Vertauschen der Reihenfolge aufeinander folgender Ausgaben.
Analog hierzu hat das Vertauschen der Reihenfolge zweier unmittelbar aufeinander folgender Ausgaben die Wirkung der Vertauschung der Zeitpunkte, an denen die beiden Ausgaben innerhalb des minimal verzögerten Ablaufs getätigt werden. Auf die anderen Ein- und Ausgaben des minimal verzögerten Ablaufs hat diese Vertauschung der Reihenfolge von Ausgaben keine zeitlichen Auswirkungen.

6.7.3. Auswirkungen lokaler zeitlicher Verhaltensänderungen auf komponiertes Verhalten

Betrachten wir nun die Wirkungen der vorangehend unterschiedenen lokalen zeitlichen Veränderungen auf ein komponiertes Gesamtverhalten entsprechend der zeitasynchronen AutoFocus Semantik. Es wird ein lokales Verhalten einer Verarbeitungskomponente, deren Verhalten zeitlich verändert wird, mit anderen Verhalten von Verarbeitungskomponenten unter Zwischenschalten der in der zeitasynchronen Semantikdefinition verwendeten Pufferkomponenten komponiert.

Im Abschnitt 6.7.1 wurde festgestellt, dass bei kombinierter Betrachtung von Eingabepuffern und Verarbeitungskomponente eine Verschiebung der Zeitpunkte der Ein-

gaben an die Puffer unter Beibehaltung der Reihenfolge von Eingaben auf einzelnen Ports keine Veränderung des zeitabstrakten Verhaltens der Verarbeitungskomponente unter den Abstraktionen $\underline{\tilde{\alpha}}_{ZA3}$ bewirken.

Die in dieser Arbeit betrachtete Art der Komposition von konkreten Schnittstellenverhalten (siehe Definition 3.15 auf Seite 62) berücksichtigt Feedback zwischen den einzelnen Komponenten. In Abschnitt 6.5 wurde bereits diskutiert, dass durch die Möglichkeit von Feedback implizit zeitliche Zusammenhänge zwischen den Komponenten bestehen, die bei Änderung der Reihenfolgen zwischen Ein- und Ausgaben, die unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA3}$ gleich sind, zu Dead Locks in den Abläufen des komponierten Verhaltens und somit zu einer Verhaltensänderung des Gesamtsystems unter Zeitabstraktion führen können. Die vorangehend im Anschnitt 6.7.2 betrachteten zeitlichen Veränderungen von Verarbeitungskomponenten bei isolierter Betrachtung von ihren Eingabepuffern sind alle unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA2}$ nicht sichtbar. Unter dieser Abstraktion bleiben die kausalen Zusammenhänge zwischen den Ein- und Ausgaben erhalten und die Vertauschungen der Reihenfolgen zwischen Ein- und Ausgaben sind somit unter der Abstraktion sichtbar. Durch die Verwendung der schwächeren Abstraktion $\underline{\tilde{\alpha}}_{ZA2}$ auf lokaler Ebene anstelle von $\underline{\tilde{\alpha}}_{ZA3}$ wird verhindert, dass durch Feedback Kommunikation Veränderungen des abstrakten Gesamtverhaltens auftreten können, ohne dass diese auf lokaler Ebene unter Abstraktion sichtbar sind.

Die vorangehend in Abschnitt 6.7.2 aufgeführten vier Fälle der zeitlichen Veränderung lokalen Verhaltens der Verarbeitungskomponente, die alle unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA2}$ nicht sichtbar sind, führen alle zu Verschiebungen der Zeitpunkte, an denen bestimmte Eingaben verarbeitet werden und bestimmte Ausgaben getätigt werden. Diese zeitlichen Verschiebungen haben alle, bei der Komposition mit den Verhalten der Eingabepuffer der veränderten Verarbeitungskomponente zusammen mit den Verhalten anderer Verarbeitungskomponenten einschließlich deren vorgeschalteten Eingabepuffern, auf Grund der Tolerierung zeitlicher Verschiebungen von Eingaben der Kombination aus Eingabepuffern und Verarbeitungskomponenten, wie in Abschnitt 6.7.1 erläutert, lediglich die Wirkung der zeitlichen Verschiebung von Komponentenausgaben. Die zeitliche Verschiebung der Komponentenausgaben wird von den diese Ausgaben empfangenden Komponenten auf Grund der diesen vorgeschalteten Puffer wiederum toleriert. Alle lokalen zeitliche Verschiebungen von Ein- und Ausgaben der Verarbeitungskomponente, die unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA2}$ nicht sichtbar sind, haben folglich auf der Ebene der Gesamtsystemschnittstelle lediglich die zeitliche Verschiebung von Systemausgaben zur Folge. Diese zeitlichen Verschiebungen sind unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA3}$, die für die Beobachtung von Gesamtsystemverhalten eingesetzt wird, nicht sichtbar. Somit ist gezeigt, dass lokale zeitliche Veränderungen von Verarbeitungskomponenten, die unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA2}$ nicht sichtbar sind, nicht zu einer Veränderung des Verhaltens der Gesamtspezifikation unter der Abstraktion $\underline{\tilde{\alpha}}_{ZA3}$ führen. Die Menge aller zeitasynchronen AutoFocus Komponentenspezifikationen erfüllen die Zeitrobustheitseigenschaft (Definition 6.2 auf Seite 127) bezüglich der Abstraktionen $\underline{\tilde{\alpha}}_{ZA2}$ angewendet auf der Ebene der Verarbeitungskomponenten und $\underline{\tilde{\alpha}}_{ZA3}$ angewendet auf Gesamtsystemebene unter Ausschluss der Veränderung der speziellen Pufferkomponenten und der

speziellen Multiplexing Komponenten[10] und unter Komposition mit externem gepufferten Verhalten[11]. Durch die implizite Existenz der Pufferkomponenten in der Definition der zeitasynchronen Semantik ist für alle zeitasynchronen Spezifikationen sichergestellt, dass die Pufferkomponenten nicht Gegenstand von Änderungen durch den Entwickler sein können und das jedes mit einer Verarbeitungskomponente komponierbare Verhalten gepufferte Kommunikation aufweist.

Da jede zeitasynchrone AutoFocus Spezifikation die Zeitrobustheitseigenschaft erfüllt und die in Kapitel 7 betrachteten Refactorings, die zeitliche Veränderungen bewirken, auf der Ebene der Spezifikation wiederum zu zeitasynchronen AutoFocus Spezifikationen führen, bleibt die Eigenschaft der Zeitrobustheit durch diese Refactorings erhalten. Dies ermöglicht die kombinierte Anwendung von zeitlichen Refactorings, die die Zeitrobustheit als Vorbedingung voraussetzen.

Es bleibt Anzumerken, dass zur Erfüllung der Zeitrobustheitseigenschaft sowohl das Konzept der *Stuttering Steps* als auch das Konzept der Verzögerung der Kommunikation erforderlich sind. Auf Grund der *Stuttering Steps* können Ausgaben, die anderen Komponenten als Eingaben zur Verfügung gestellt werden, natürlich beliebig verzögert werden. Somit haben *Stuttering Steps* eine ähnliche Wirkung wie die Verzögerung der Kommunikation. Durch die *Stuttering Steps* bleibt jedoch der zeitliche Zusammenhang zwischen gleichzeitig von einer atomaren Komponente gesendeten Ausgaben erhalten. Eine atomare Komponente würde also diese beide Nachrichten ohne Verzögerung der Kommunikation ebenfalls gleichzeitig empfangen und kann gezielt auf diese Gleichzeitigkeit reagieren. Wird das gleichzeitige Senden der Nachrichten in ein sequentielles Senden abgeändert, was unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ verhaltensäquivalent ist, würde die empfangende Komponente diese Nachrichten nicht mehr gleichzeitig empfangen. Die empfangende Komponente kann auf die nicht mehr gleichzeitig empfangenen Nachrichten jetzt anders reagieren, als ursprünglich auf die gleichzeitig empfangenen Nachrichten. Dies würde eine nicht zeitliche Verhaltensänderung des Gesamtsystems bewirken. Zur Erfüllung der Zeitrobustheitseigenschaft werden folglich sowohl das Konzept der *Stuttering Steps* als auch das Konzept der Verzögerung der Kommunikation benötigt.

6.7.4. Praktische Anwendbarkeit der zeitasynchronen Semantik

Der Nachweis der Eigenschaft der Zeitrobustheit zeigt, dass zu ihrer Erfüllung die Funktionalität der Puffer die Kommunikation nichtdeterministisch zu verzögern ausreicht und dass die Funktionalität der Puffer, Nachrichten bis zu ihrer Verarbeitung zwischenzuspeichern, zur Erfüllung der Zeitrobustheit nicht erforderlich ist. Aus Sicht der praktischen Anwendbarkeit der Modellierungssprache erscheint es jedoch sinnvoll,

[10] Die Menge $\underline{\mathcal{M}}_K$ enthält alle entsprechend der zeitasynchronen AutoFocus Semantik möglichen Verhalten von Verarbeitungskomponenten.

[11] Die Menge $\underline{\mathcal{M}}_R$ enthält alle entsprechend der zeitasynchronen AutoFocus Semantik möglichen mit einer Verarbeitungskomponente komponierbaren externen Verhalten, welche die Verhalten der Eingabepufferkomponenten der Verarbeitungskomponente auf Seiten der Ausgaben des Restsystems enthält.

die Funktionalität des Zwischenspeicherns von Nachrichten bis zu ihrer Konsumierung in der zeitasynchronen Semantik zu integrieren, da dieses Kommunikationsmodell für den Entwickler wesentlich besser verständlich ist als ein Kommunikationsmodell, das einfach nur nichtdeterministisch verzögert. Der Entwickler muss bei dieser Art der Kommunikation in der Spezifikation nicht berücksichtigen, dass zu jedem beliebigen Zeitpunkt auf eine Eingabe reagiert werden muss, um keine Eingabe zu verpassen. Das Zwischenspeichern der Eingaben bis zu ihrer Verarbeitung bietet dem Entwickler die Möglichkeit, zyklisch auf neue Eingaben prüfen und darauf reagieren zu können.

7. Refactoring mit Veränderung des zeitlichen lokalen Verhaltens

In Kapitel 5 wurden Refactorings von Systemstrukturdiagrammen (SSDs) und Zustandsübergangsdiagrammen (STDs) festgelegt, die verhaltensinvariant bezüglich des konkreten Schnittstellenverhaltens der durch das Refactoring veränderten Komponente ohne Anwendung von Zeitabstraktionen sind. In diesem Kapitel werden jetzt Refactorings von Zustandsmaschinen und Systemstrukturdiagrammen definiert, die das zeitliche Verhalten von Komponenten ändern, aber dabei das nicht zeitliche Verhalten des Gesamtsystems unverändert lassen. Für das klassische zeitsynchrone AutoFocus können diese Art von erweiterten Refactorings auf Grund der festen zeitlichen Beziehungen nicht allgemeingültig angegeben werden. Wir beschränken uns daher auf die Unterklasse der nach der Zeitabstraktion $\underset{\sim}{\widetilde{\alpha}}_{ZA2}$ auf Komponentenebene und nach der Abstraktion $\underset{\sim}{\widetilde{\alpha}}_{ZA3}$ auf Systemebene zeitrobusten AutoFocus Modelle. In Kapitel 6 wurde die Frage, welche Systeme zeitliche Änderungen tolerieren, ausgiebig besprochen. Es wurde eine zeitasynchrone Semantik für AutoFocus konstruiert (Abschnitt 6.6), die die Erfüllung der Zeitrobustheitseigenschaft für alle ausdrückbaren, syntaktisch korrekten Modellspezifikationen garantiert.

Die nachfolgend aufgeführten Refactorings sind für Modellspezifikationen in der neuen zeitasynchronen AutoFocus Modellierungssprache definiert und sind für diese allgemeingültig. Es ist ebenfalls möglich, diese Refactorings in leicht abgewandelter Form als allgemeingültige Refactorings für die Menge der zeitsynchronen AutoFocus Modelle, die die Zeitrobustheitseigenschaft (Definition 6.2 auf Seite 127) in Bezug auf die Zeitabstraktionen $\underset{\sim}{\widetilde{\alpha}}_{ZA2}$ lokal und $\underset{\sim}{\widetilde{\alpha}}_{ZA3}$ global erfüllen, zu definieren. Darüber hinaus sind die nachfolgenden Refactorings in abgewandelter Form auch auf beliebige zeitsynchrone AutoFocus Modellspezifikationen anwendbar. Für die Klasse von Modellen ist jedoch nicht die Eigenschaft der Allgemeingültigkeit dieser Refactorings erfüllt. Die Verhaltensinvarianz muss bei diesen Modellen nach Durchführung des Refactorings gesondert an Hand der Modellspezifikation auf der Ebene des Schnittstellenverhaltens des Gesamtsystems nachgewiesen werden. Hierzu können die in Abschnitt 9.3 angeführten Verifikationstechniken eingesetzt werden.

In den anschließenden Abschnitten werden folgende Refactorings definiert, die das zeitliche Komponentenverhalten verändern:

Remove Intermediate State Refactoring (Abschnitt 7.1).
    Entfernen eines nicht benötigten Zwischenzustands.

Insert Intermediate State Refactoring (Abschnitt 7.2).
: Einfügen eines neuen Zwischenzustands, der für eine Erweiterung des Verhaltens benötigt wird.

Split Component Refactoring (Abschnitt 7.3).
: Aufspalten einer atomaren Komponente in zwei atomare Komponenten einschließlich der assoziierten Zustandsmaschine.

Remove Inactive States Refactoring (Abschnitt 7.4).
: Entfernen des Aktivierungsmechanismus zwischen zwei durch das *Split Component* Refactoring aufgespaltenen Komponenten. Dieses Refactoring ist nicht allgemeingültig.

In Abschnitt 7.5 wird schließlich die Verhaltensinvarianzeigenschaft dieser Refactorings gezeigt.

7.1. Remove Intermediate State Refactoring

Problem:

Ein Zustandsübergangsdiagramm (STD) enthält nicht erforderliche Zwischenzustände. Durch die erhöhte Anzahl von Zuständen und Transitionen ist das Zustandsübergangsdiagramm unübersichtlich und unverständlich.

Lösung:

Reduzierung der Anzahl der Zustände und Transitionen durch das Entfernen nicht erforderlicher Zwischenzustände (Abbildung 7.1 auf der nächsten Seite).

Motivation:

Zustandsübergangsdiagramme (STDs) mit vielen Zuständen und Transitionen sind unübersichtlich und unverständlich. Wird eine Zustandsmaschine mit einer großen Anzahl von Zuständen identifiziert, dann soll versucht werden, die Anzahl der Zustände und Transitionen zu reduzieren. In Zustandsmaschinen sind häufig Zustände enthalten, die keine richtigen Systemzustände darstellen, sondern eine Art Zwischenzustand. Solche Zwischenzustände besitzen genau eine eingehende und eine ausgehende Transition und sind häufig nach der Aktion einer Transition bezeichnet. Besitzt die von einem Zwischenzustand ausgehende Transition weder eine Variablenvorbedingung noch eine Eingabebedingung und werden in dem Variablenzuweisungsterm der ein- und ausgehenden Transition nicht die selben Variablen referenziert, dann lässt sich der Zwischenzustand ohne Veränderung des abstrakten Komponentenverhaltens, unter Zusammenfassung der ein- und ausgehenden Transition zu einer einzigen Transition, entfernen.

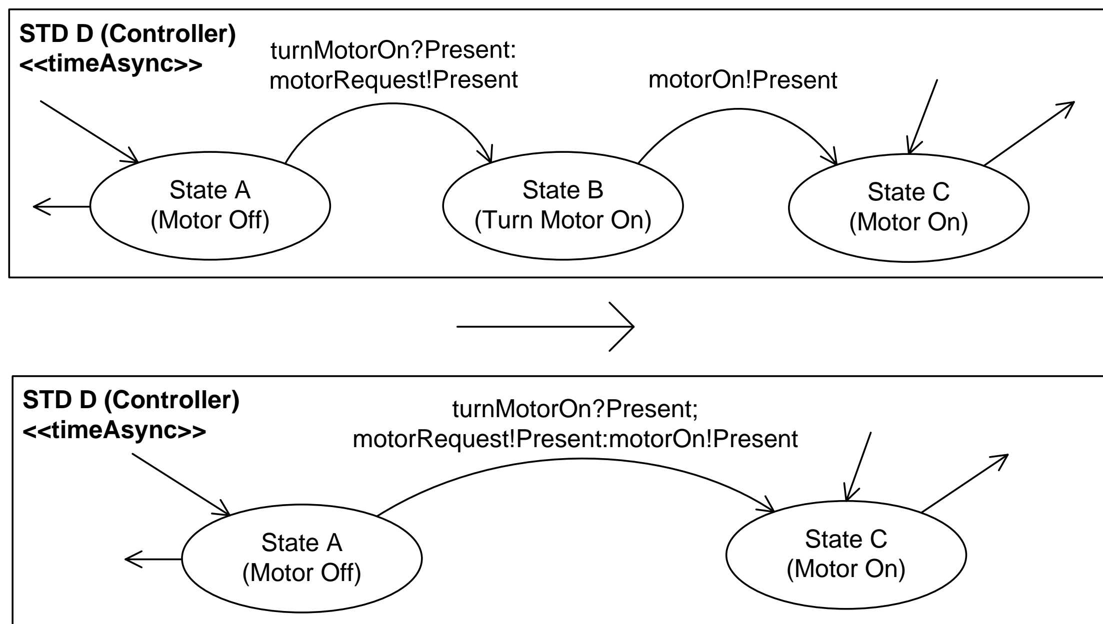


Abbildung 7.1.: Remove Intermediate State Refactoring.

Werden Zwischenzustände in einer Zustandsmaschine identifiziert, dann sind diese durch die Anwendung des *Remove Intermediate State* Refactorings zu entfernen.

Die Reduktion von Zwischenzuständen in Zustandsmaschinen ist ein allgemein bekanntes Problem in der Automatentheorie. Dort existieren Konstruktionsverfahren von Automaten mit minimaler Anzahl von Zuständen, die das Verhalten aufweisen beziehungsweise die gleiche Sprache akzeptieren, wie ein gegebener Automat. In Abschnitt 8.3 ist ein solches Minimierungsverfahren erwähnt. Bei dem Refactoring von Modellspezifikationen steht im Gegensatz zu einer minimalen Anzahl von Zuständen die Qualität der Beschreibung in Hinblick auf Lesbarkeit und Verständlichkeit im Vordergrund. In vielen Fällen ist eine automatisch abgeleitete minimale Zustandsmaschine wesentlich schlechter verständlich als deren äquivalente nicht minimale durch den Entwickler konstruierte Zustandsmaschine, da der Minimierungsalgorithmus die Struktur der Zustandsmaschine weitgehend verändert. Das *Remove Intermediate State* Refactoring hat nicht die Konstruktion einer minimalen Zustandsmaschine zum Ziel. Es sollen vielmehr offensichtlich nicht notwendige Zwischenzustände entfernt werden, ohne dabei die Gesamtstruktur der Zustandsmaschine zu verändern.

Ein Beispiel für einen Zwischenzustand ist in Abbildung 7.1 aufgeführt. In der Zustandsmaschine existiert ein Zustand *Turn Motor On*. In der eingehenden Transition wird auf die Nachricht *Present* auf dem Eingabe-Port *turnMotorOn* geprüft und eine Ausgabe über den Ausgabe-Port *motorRequest* gesendet. Die ausgehende Transition tätigt mit *motorOn!Present* eine Ausgabe zum Einschalten des Motors. Beide Transitionen

können zu einer Transition zusammengefasst und der nicht mehr benötigte Zwischenzustand *Turn Motor On* kann gelöscht werden.

Sind in einer Zustandsmaschine nach wiederholter Anwendung des *Remove Intermediate State* Refactorings keine entfernbaren Zwischenzustände mehr vorhanden, das resultierende Zustandsübergangsdiagramm aber weiterhin zu groß, dann kann durch die Anwendung des *Wrap States* Refactorings (siehe Abschnitt 5.2.2) die Zustandsmaschine in Unterautomaten unterteilt werden.

Methode:

Ausgangspunkt für die Beschreibung der Operation ist das in Abbildung 7.1 auf der vorherigen Seite oben gezeigte Fragment einer Zustandsmaschine. Ein Zwischenzustand B befindet sich zwischen den Zuständen A und C.

Benutzereingabe: Der Benutzer wählt einen zu entfernenden Zwischenzustand B aus. Alternativ kann die Identifikation von Zwischenzuständen auch automatisiert werden. Hierzu wird jeder Zustand eines STDs betrachtet, ob dieser die nachfolgend aufgeführten Vorbedingungen für dieses Refactoring erfüllt. Dem Entwickler bleibt weiterhin die Entscheidung, ob er bestimmte Zwischenzustände entfernen möchte oder nicht.

Vorbedingungen: Ein Zwischenzustand B besitzt genau eine eingehende und eine ausgehende Transition. Ferner müssen die eingehende Transition e und die ausgehende Transition a dieses Zustands zu einer Transition zusammenfassbar sein. Dies ist genau dann der Fall, wenn die folgenden Bedingungen erfüllt sind:

- Die Transition a besitzt weder eine Variablenvorbedingung noch eine Eingabebedingung.[1]
- Die Variablenzuweisungen von e und a erfolgen auf voneinander verschiedene Variablen.

Schritt 1: Zwischen dem Quellzustand der Transition e und dem Zielzustand der Transition a wird eine leere Transition n eingefügt.

Schritt 2: Aus den in den Transitionen e und a enthaltenen Termen werden die Terme der neuen Transition n abgeleitet.

Schritt 2a: In der neuen Transition n wird die Variablenvorbedingung der Transition e verwendet.

Schritt 2b: Als Eingabebedingung der neuen Transition n wird die Eingabebedingung der Transition e verwendet.

[1] Diese starke Einschränkung der Anwendbarkeit dieses Refactorings ist erforderlich, um zu verhindern, dass es sich bei dem zu entfernenden Zwischenzustand um einen lokalen Dead Lock Zustand handelt. Durch Entfernen eines Dead Lock Zustandes wird das lokale zeitabstrahierte Verhalten verändert.

Schritt 2c: Der Ausgabeterm der neuen Transition n wird durch eine logische Konjunktion der Ausgabeterme der Transitionen e und a gebildet. Werden in den Ausgabetermen der Transitionen e und a gleiche Ausgabe-Ports referenziert, dann enthält der Ausgabeterm der neuen Transition beide Wertzuweisungen an den gemeinsamen Ausgabe-Port, wobei die Ausgabezuweisungen der Transition e in dem Ausgabeterm vor den Ausgabezuweisungen der Transition a angeordnet werden.[2]

Schritt 2d: Auch der Variablenzuweisungsterm der neuen Transition n ergibt sich durch eine logische Konjunktion der Variablenzuweisungsterme der Transitionen e und a.

Schritt 3: Der Zwischenzustand B wird zusammen mit den mit ihm verbundenen Transitionen e und a gelöscht.

7.2. Insert Intermediate State Refactoring

Problem:

Ein Automat soll um zusätzliche Funktionalität, deren Realisierung zusätzliche Kontrollzustände erfordert, erweitert werden.

Lösung:

Einfügen eines Zwischenzustands mit anschließender leerer Transition. Die leere Transition wird später zur Erweiterung der Funktionalität der Zustandsmaschine genutzt (Abbildung 7.2 auf der nächsten Seite).

Motivation:

Das *Remove Intermediate State* Refactoring (siehe Abschnitt 7.1) besagt, dass Zwischenzustände, die keinen richtigen Systemzustand darstellen, unnötig sind und entfernt werden sollten. Es existiert jedoch eine Situation in der die temporäre Existenz von Zwischenzuständen durchaus gewollt ist. Bei der Erweiterung von Zustandsmaschinen um neue Funktionalitäten kann es durchaus sinnvoll sein, zunächst Zwischenzustände mit zusätzlichen leeren Transitionen einzufügen. Diese Operation verändert das zeitabstrahierte Verhalten der Zustandsmaschine nicht. In einem zweiten Schritt, in dem die eigentliche Erweiterung des Verhaltens durchgeführt wird, werden dann die leeren Transitionen mit Bedingungen und Zuweisungen versehen. Auf diese Weise werden nach Abschluss der Erweiterung die Zwischenzustände zu richtigen Systemzuständen.

[2] Das sequentielle Senden von mehreren Nachrichten auf einen Ausgabe-Port innerhalb einer Transition ist in Abschnitt 6.6.7 definiert.

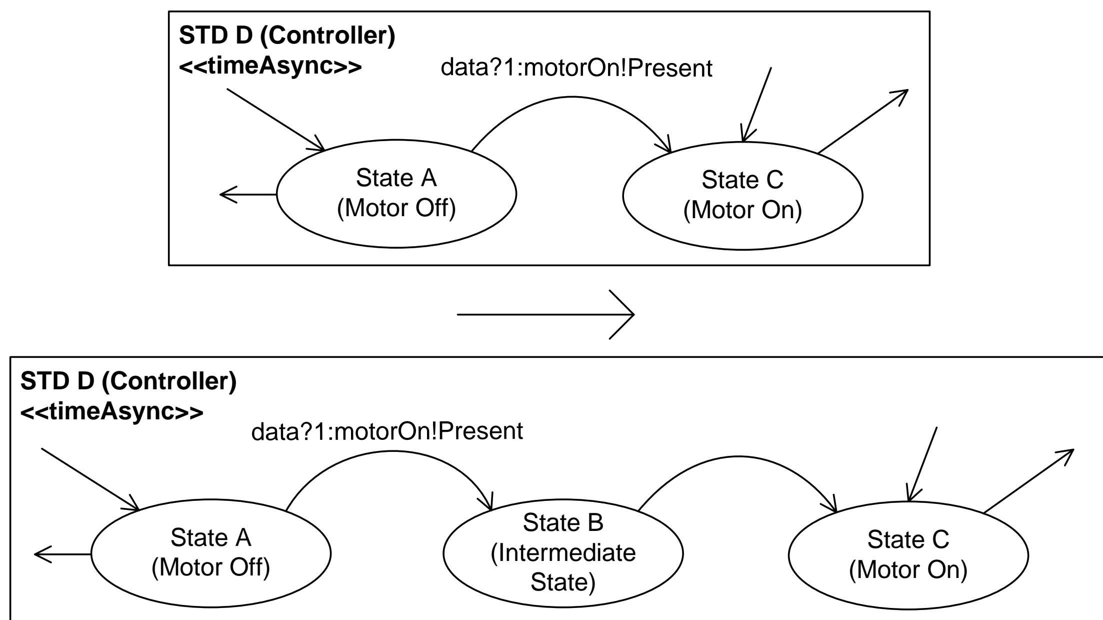


Abbildung 7.2.: Insert Intermediate State Refactoring.

Methode:

Benutzereingabe: Der Benutzer wählt eine Transition *e* aus, die um einen Zwischenzustand mit anschließender leerer Transition erweitert werden soll.

Vorbedingungen: Zu dieser Operation existieren keine Vorbedingungen.

Schritt 1: Ein neuer Zustand *B* wird in das STD eingefügt.

Schritt 2: Eine leere Transition mit Quellzustand *B* und dem Zielzustand der Transition *e* wird eingefügt.

Schritt 3: Die Transition *e* wird zu dem Zwischenzustand *B* umgeleitet, das heißt dem Zielzustand der Transition *e* wird der Zustand *B* zugewiesen.

7.3. Split Component Refactoring

Die vorangehend in diesem Kapitel vorgestellten Refactorings wurden ausschließlich auf Zustandsübergangsdiagramme (STDs) angewendet.

Das *Split Component* Refactoring verändert gleichzeitig Systemstrukturdiagramme (SSDs) und Zustandsübergangsdiagramme (STDs). Das Refactoring führt weitgehende Veränderungen der Diagramme durch und ist wesentlich komplexer als die anderen in dieser Arbeit vorgestellten Refactorings.

Problem

Atomare Komponenten mit vielen Ports und zugewiesenen komplexen Zustandsmaschinen, das heißt Zustandsmaschinen bestehend aus vielen Zuständen und Transitionen, sind schwer verständlich.

Lösung

Teilen Sie die atomare Komponente in zwei atomare Teilkomponenten auf. Hierbei wird neben der Komponentenstruktur auch eine Aufteilung der zugeordneten Zustandsmaschine in zwei Zustandsmaschinen vorgenommen (siehe Abbildung 7.3 auf der nächsten Seite).

Motivation

Atomare Komponenten, die viel Funktionalität realisieren, sind meist in ihrer Beschreibung sehr komplex. In dem Systemstrukturdiagramm besitzt eine solche atomare Komponente häufig sehr viele Ports zur Kommunikation. Die Zustandsmaschine einer solchen atomaren Komponente benötigt zur Realisierung der komplexen Funktionalität viele Zustände, Transitionen und Variablen.

Diese Charakteristika führen häufig dazu, dass Modellspezifikationen von atomaren Komponenten, die komplexe Funktionalität realisieren, sehr unübersichtlich und nur sehr schwer verständlich sind. In AutoFocus ist es möglich, Komponenten in Unterkomponenten aufzuteilen, die miteinander kommunizieren und jeweils eine Teilfunktionalität realisieren. Hierdurch wird die Funktionalität auf mehrere Teilsysteme verteilt. Die Funktionalität der Teilsysteme ist weniger komplex als die Gesamtfunktionalität. Es ergeben sich mehrere kompaktere Verhaltensbeschreibungen der Teilsysteme, die in den meisten Fällen besser verständlich sind als die Gesamtverhaltensbeschreibung.

Das *Split Component* Refactoring ermöglicht die Aufteilung einer komplexen atomaren Komponente in zwei weniger komplexe atomare Teilkomponenten innerhalb eines Systemstrukturdiagramms (SSDs). Die assoziierte Zustandsmaschine wird in zwei Zustandsmaschinen zerteilt. Wird eine atomare Komponente identifiziert, die viele Ports besitzt oder besitzt deren assoziierte Zustandsmaschine viele Zuständen und Transitionen, dann kann das *Split Component* Refactoring auf diese Komponente angewendet werden.

Durch die Durchführung des Refactorings wird die Anzahl der Komponenten des SSDs, in dem sich die zerteilte Komponente befindet, erhöht. Neben der Aufteilung der Komponente in zwei Komponenten werden gegebenenfalls zusätzliche Multiplexing Komponenten zur Realisierung von gemeinsamen Ausgaben auf einen Port in das Systemstrukturdiagramm eingefügt. Um die Anzahl der Komponenten des SSDs nach Anwendung des *Split Component* Refactorings wieder zu reduzieren, wird empfohlen,

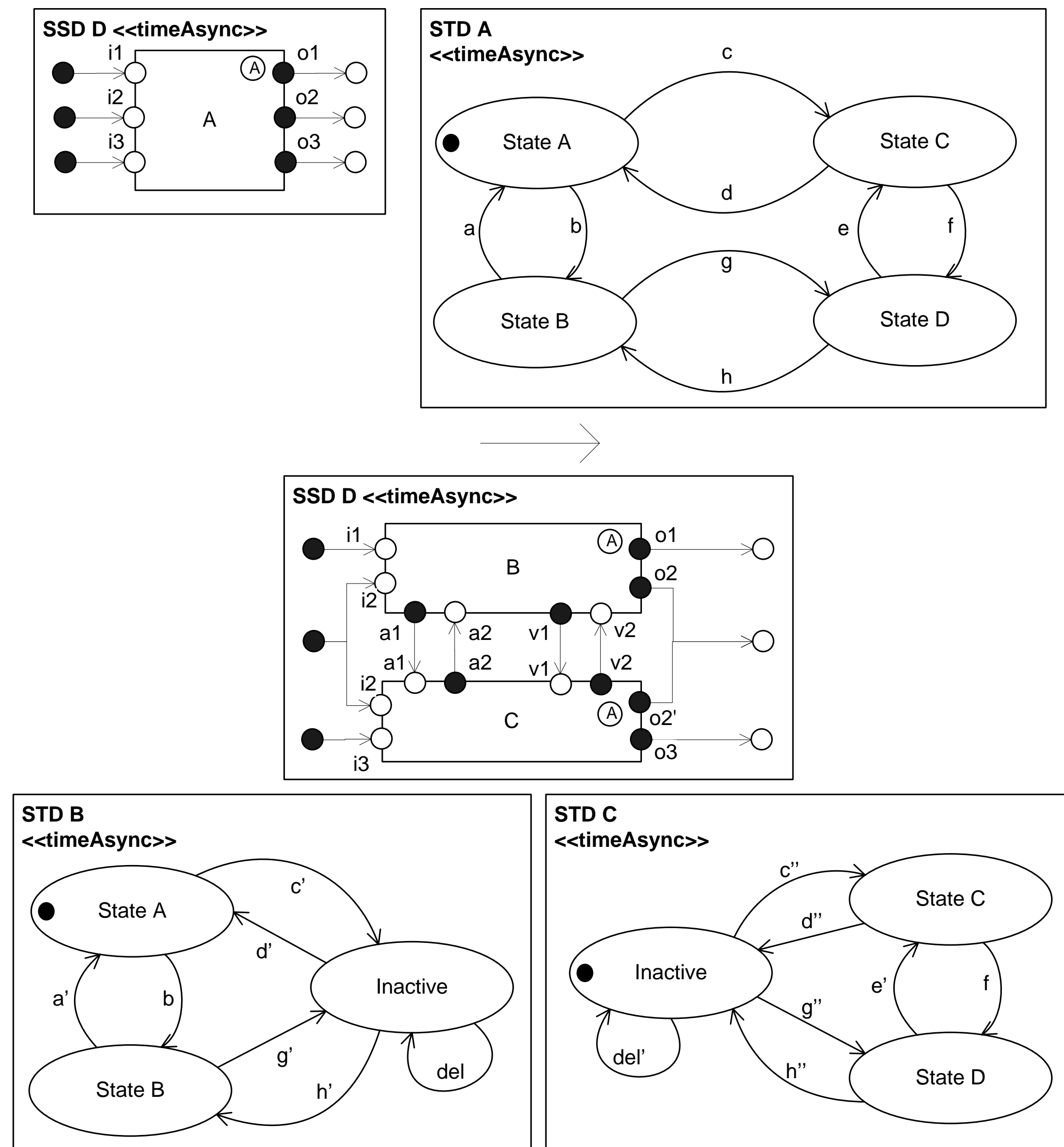


Abbildung 7.3.: Split Component Refactoring.

das *Wrap Components* Refactoring (siehe Abschnitt 5.1.3) auf die durch das *Split Component* Refactoring eingefügten Komponenten anzuwenden. Hierdurch werden diese zusammengehörenden Komponenten in einer Super-Komponente gekapselt.

Nach der Durchführung des *Split Component* Refactorings arbeiten die sich durch die Zerteilung ergebenden beiden Komponenten jeweils wechselweise, das heißt immer nur eine der beiden Komponenten ist zu einem gegebenen Zeitpunkt aktiv. Die Komplexität der Modellspezifikation kann weiter reduziert werden, wenn beide Komponenten unabhängig voneinander immer aktiv sein können. Das *Remove Inactive States* Refactoring (siehe Abschnitt 7.4) ist ein nicht allgemeingültiges Refactoring, mit dem versucht werden kann, den Aktivierungsmechanismus zwischen den zerteilten Komponenten zu entfernen. Dieses Refactoring funktioniert jedoch nicht bei allen durch das *Split Component* Refactoring zerteilten Komponenten.

Es bleibt anzumerken, dass sich durch das *Split Component* Refactoring nicht in allen Fällen die Qualität der Modellspezifikation steigern lässt. Die Qualitätsveränderung durch dieses Refactoring ist von der Wahl der Partitionierung der zu zerteilenden Zustandsmaschine, die von dem Entwickler vorgenommen wird, und von der Struktur der Zustandsmaschine, abhängig. Sind durch die gewählte Partitionierung die Abhängigkeiten von gemeinsamen Eingaben und Ausgaben sowie von gemeinsamen Variablen gering und ist eine Aktivierung zwischen den zerteilten Komponenten selten erforderlich, dann lässt sich die Qualität einer Modellspezifikation, die komplexe Zustandsmaschinen besitzt, in Hinblick auf Lesbarkeit und Verständlichkeit durch die Anwendung des *Split Component* Refactorings deutlich steigern.

Methode

Benutzereingaben: Der Entwickler wählt eine atomare Komponente A in einem SSD aus, die zu komplex ist und zerteilt werden soll. Zusätzlich gibt er zwei Namen für die beiden resultierenden zerteilten Komponenten an.

Darüber hinaus muss der Entwickler in der der Komponente A assoziierten Zustandsmaschine eine Partitionierung der Zustände in zwei Teilmengen vornehmen. Entsprechend dieser Partitionierung erfolgt die Aufteilung der Schnittstelle der Komponente, die Verteilung von lokalen Variablen und das Zerschneiden der Zustandsmaschine in zwei Zustandsmaschinen. Der Benutzer wählt eine Teilmenge der Zustände in dem STD A aus. Die Menge der ausgewählten Zustände wird mit Z_C bezeichnet. Diese Zustände werden durch das Refactoring in einer neuen Zustandsmaschine C abgespalten. Mit Z_B bezeichnen wir die Menge aller Zustände des STDs A, die vom Benutzer nicht markiert wurden. Diese Zustände werden durch das Refactoring in der neuen Zustandsmaschine B zusammengefasst.

Es sein angemerkt, dass alternativ zur Festlegung einer Partitionierung der Zustandsmenge auch mit einer Partitionierung der Ports der Komponentenschnittstelle innerhalb des Systemstrukturdiagramms begonnen werden kann. Diese Schnittstellenpartitionierung legt jedoch stark einschränkende Bedingungen an die Möglichkeiten der

anschließend erforderlichen Zerteilung der Zustandsmaschine fest. Da bei dem *Split Component* Refactoring die Verbesserung der Qualität der Zustandsübergangsdiagramme im Vordergrund steht, wird hier, wie Eingangs beschrieben, zunächst die Partitionierung der Zustandsmenge vorgenommen. Bei der Festlegung der Partitionierung der Zustandsmenge wird deren Wirkung auf die Schnittstellenaufteilung berücksichtigt.

Die Wahl der Partitionierung ist entscheidend dafür, inwieweit sich durch die Anwendung des *Split Component* Refactorings die Qualität der Modellspezifikation steigern lässt. Ziel der Partitionierung ist, Zustände, deren ausgehende Transitionen ein zusammengehörendes Teilverhalten realisieren, zusammenzufassen und Zustände, deren ausgehende Transitionen unterschiedliches Teilverhalten realisieren, zu trennen.

An dieser Stelle werden einige Heuristiken für die Identifikation zusammengehörender Zustände angegeben. Diese Heuristiken stellen keine absoluten Kriterien für die Auswahl der Zustände dar. Letztendlich bleibt die Auswahl der abzuspaltenden Zustände eine Entscheidung des Entwicklers. Als Indiz für Zustände, die eine zusammengehörige Funktionalität realisieren, kann der Kopplungsgrad der Zustände durch Transitionen herangezogen werden. Starke Vernetzung von Zuständen durch Transitionen deutet auf zusammengehörende Zustände hin.[3] Umgekehrt deutet ein niedriger Kopplungsgrad zwischen Zuständen darauf hin, dass diese Zustände unterschiedliche Funktionalitäten realisieren und getrennt werden sollten.

Ein weitere Heuristik zur Identifikation zusammengehörender Zustände ist die Verwendung der lokalen Variablen innerhalb der Transitionen der Zustandsmaschine. Es soll möglichst eine Menge von Zuständen gewählt werden, deren ausgehenden Transitionen die gleichen lokalen Variablen verwenden.[4] Umgekehrt soll darauf geachtet werden, dass diese Variablen von den Zuständen der anderen Partition nicht verwendet werden. Wie bei dem Kopplungsgrad handelt es sich auch bei den Kriterien zur Wahl der Partitionierung entsprechend der Verwendung von lokalen Variablen lediglich um eine Empfehlung. Die Anzahl der Variablen, die gemeinsam in beiden Partitionen verwendet werden, soll möglichst klein gehalten werden. Diese gemeinsamen Variablen stellen Abhängigkeiten zwischen den Partitionen dar, die sich später nach der Aufteilung in zwei Komponenten in dem SSD in Form von zusätzlichen Kanälen und Ports und in den STDs in Form von zusätzlichen Mechanismen zur Synchronisierung der Variablen ausdrücken.

Bei der Wahl der Partitionierung der Zustände sollte darüber hinaus auch die Verwendung der Ein- und Ausgabe-Ports in den Transitionen der Zustandsmaschine berücksichtigt werden. In den resultierenden aufgespaltenen Komponenten sollten möglichst wenige von beiden Komponenten gemeinsam benutzte Eingabe-Ports und möglichst wenige gemeinsam benutzte Ausgabe-Ports auftreten, da diese eins zu zwei und zwei

[3] Der Kopplungsgrad der Zustände kann automatisiert durch Graphen-Algorithmen ermittelt werden. So lassen sich beispielsweise starke Zusammenhangskomponenten in dem Zustandsübergangsgraphen identifizieren.

[4] Unter Verwendung von Variablen wird das Prüfen von Belegungen der Variablen und die Wertzuweisung verstanden.

zu eins Kommunikationsbeziehungen erfordern, die die Komplexität der Interaktion zwischen den Komponenten erhöhen. Durch die Wahl der Partitionierung in der Art, dass die von den Zuständen ausgehenden Transitionen einer Partition möglichst wenige gemeinsame Ports mit den ausgehenden Transitionen der Zustände der anderen Partition teilen, entstehen bei der Aufteilung der Komponente möglichst wenige eins zu zwei und zwei zu eins Kommunikationsbeziehungen.

Vorbedingungen: Die zu zerteilende Komponente *A* muss atomar sein, das heißt sie besitzt keine Unterkomponente und es existiert ein ihr assoziiertes Zustandsübergangsdiagramm (STD). Darüber hinaus darf es sich bei *A* nicht um die *Top Level* Komponente der Modellspezifikation handeln, da zu dieser Komponente in AutoFocus kein SSD existiert.

Die ausgewählten Zustände müssen sich alle in dem STD *A* befinden, das der Komponente *A* assoziiert ist. Es muss mindestens ein Zustand für die Abspaltungsoperation gewählt werden und es dürfen nicht alle Zustände des STDs *A* abgespalten werden.

Das *Split Component* Refactoring verändert das Systemstrukturdiagramm (SSD) in dem sich die Komponente *A* befindet. Darüber hinaus werden zwei neue Zustandsübergangsdiagramme (STDs) auf Basis der der Komponente *A* assoziierten Zustandsmaschine konstruiert. Zunächst werden die Veränderungen des Systemstrukturdiagramms beschrieben.

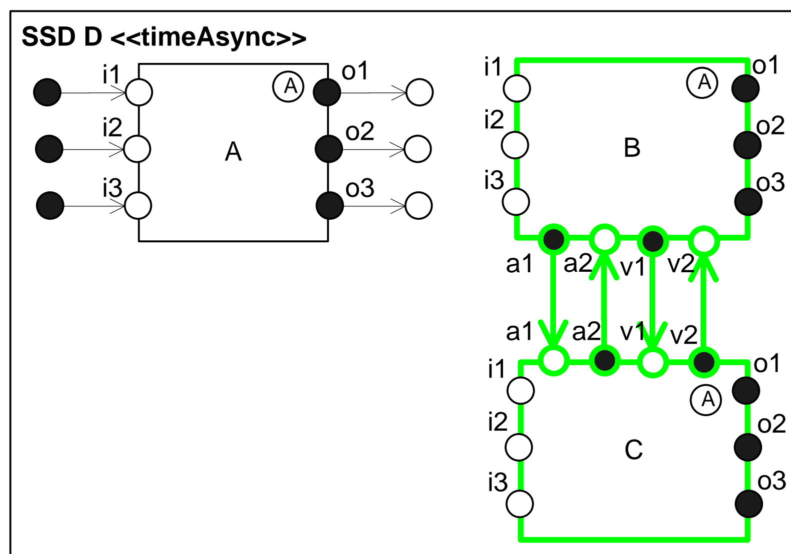


Abbildung 7.4.: Schritte 1 bis 3 des Split Component Refactorings.

Schritt 1: Die Komponente *A* wird einschließlich aller Ports, lokaler Variablen und der assoziierten Zustandsmaschine zweimal kopiert. Die Kopien heißen *B* und *C*, befinden sich in dem SSD *D*, in dem auch die Komponente *A* enthalten ist, und besitzen die assoziierten STDs *B* und *C* (siehe Abbildung 7.4).

Schritt 2: Nach der Zerteilung der Komponente in zwei Komponenten, müssen diese miteinander kommunizieren, um die Abhängigkeiten zwischen den Zuständen, die dann auf zwei Komponenten verteilt sind, weiterhin zu erhalten. Diese Abhängigkeiten werden durch gegenseitige Aktivierung der Komponenten realisiert. Das Ereignis, dass eine Transition zwischen Zuständen der Mengen Z_B und Z_C gefeuert wird, muss zwischen diesen Komponenten kommuniziert werden. Zu diesem Zweck werden die Komponentenschnittstellen von B und C um jeweils einen Ein- und einen Ausgabe-Port erweitert. Der neue Ausgabe-Port $a1$ von B und der neue Eingabe-Port $a1$ von C sind vom Datentyp *activateC* und der Ausgabe-Port $a2$ von C und der Eingabe-Port $a2$ von B sind vom Datentyp *activateB*. Der Ausgabe-Port $a1$ von B wird über einen neuen Kanal mit dem neuen Eingabe-Port $a1$ von C in dem SSD D verbunden. Die Verbindung zwischen dem Ausgabe-Port $a2$ von C und dem Eingabe-Port $a2$ von B erfolgt ebenfalls über einen neuen Kanal (Abbildung 7.4 auf der vorherigen Seite).

Schritt 3: Werden in den ausgehenden Transitionen der Zustände in Z_B und Z_C gemeinsame Variablen benutzt, dann stellen diese Variablen eine zusätzliche Abhängigkeit zwischen den neu aufgeteilten Komponenten dar. Es wird pro gemeinsam benutzter Variable eine zusätzliche Kommunikation zur Synchronisierung benötigt. Falls eine der beiden geteilten Komponenten nur lesend auf die Variable zugreift, reicht eine unidirektionale Kommunikation aus. Andernfalls ist eine bidirektionale Kommunikation zur Synchronisierung der Variable erforderlich. Für jede Variable, auf die in beiden aufgeteilten Komponenten Wertzuweisungen erfolgen, fügen wir in dem SSD D zu der Komponente B einen Ausgabe-Port $v1$ und einen Eingabe-Port $v2$ und in der Komponente C einen Eingabe-Port $v1$ und einen Ausgabe-Port $v2$ von dem Datentyp der Variable ein. Die Ports $v1$ beider Komponenten werden durch einen Kanal und die Ports $v2$ beider Komponenten werden durch einen Kanal verbunden (siehe Abbildung 7.4 auf der vorherigen Seite). Wird in den ausgehenden Transitionen von Z_C nur lesend auf die Variable zugegriffen, dann wird auf das Einfügen der Ports $v2$ und des diese Ports verbindenden Kanals verzichtet.

Schritt 4: Die Komponenten B und C wurden von der Komponente A kopiert und besitzen zunächst die gleichen lokalen Variablen. Alle Variablen, die von den ausgehenden Transitionen der Zustände in Z_B nicht benutzt werden, werden von der späteren Komponente B nicht benötigt. Sie werden gelöscht. Analog hierzu werden alle Variablen der Komponente C gelöscht, die in den ausgehenden Transitionen der Zustände in der Zustandsmenge Z_C nicht benutzt werden.

Schritt 5: Die Komponenten B und C wurden von der Ursprungskomponente A kopiert und besitzen zunächst die gleichen Ports wie diese. Die neuen aufgeteilten Komponenten benötigen jedoch nicht beide zwingender Weise alle Ports der Ursprungskomponente. Es kann sein, dass bestimmte Ein- und Ausgaben nur von der Komponente B und bestimmte Ein- und Ausgaben nur von C verarbeitet und getätigt werden. Um die Zugehörigkeit der Ports zu identifizieren, müssen die Transitionen der ursprünglichen Zustandsmaschine entsprechend der durchgeführten Partitionierung betrachtet werden. Wir betrachten jeweils alle ausgehenden Transitionen der Zustände, die in Z_B

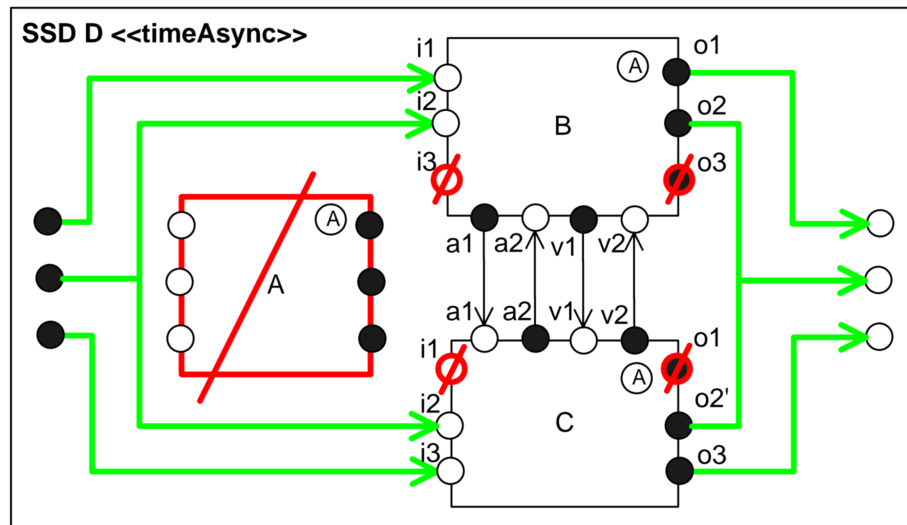


Abbildung 7.5.: Schritte 5 bis 8 des Split Component Refactorings.

und Z_C enthalten sind. Ist in einer Eingabebedingung einer Transition die Abfrage eines Eingabe-Ports enthalten, dann wird der Port, je nachdem, ob der Quellzustand der betrachteten Transition in der Zustandsmenge Z_B oder Z_C enthalten ist, als zur Komponente *B* beziehungsweise *C* gehörend, markiert. Analog hierzu werden Ausgabe-Ports als zu den Komponenten *B* beziehungsweise *C* gehörend markiert, wenn eine Transition mit Quellzustand in den Mengen Z_B beziehungsweise Z_C eine Zuweisung auf einen Ausgabe-Port tätigt. Nach der Betrachtung aller Transitionen der ursprünglichen Zustandsmaschine werden alle nicht benötigten Ports, das heißt alle Ports, die nicht zur entsprechenden Komponente gehörend markiert wurden, aus den Komponenten *B* und *C* in dem SSD *D* gelöscht (siehe Abbildung 7.5).

Schritt 6: In dem nächsten Schritt werden die Kanäle in dem SSD *D* von der ursprünglichen Komponente *A* zu den neuen Komponenten *B* und *C* umgeleitet. Betrachten wir zunächst die Eingabe-Ports der Ursprungskomponente *A*. Falls zu einem Eingabe-Port *s* der Komponente *A* ein äquivalenter, das heißt gleich bezeichneter, Port *t* der Komponente *B*, nicht aber ein äquivalenter Port der Komponente *C*, existiert, dann wird der mit *s* verbundene Kanal von dem Port *s* zu dem Port *t* umgeleitet. Analog hierzu werden Kanäle, die mit Eingabe-Ports von *A* verbunden sind, zu äquivalenten Ports von *C* umgeleitet, falls die Komponente *B* keinen äquivalenten Port besitzt.

Besitzen beide Komponenten *B* und *C* einen zu dem Port *s* der Komponente *A* äquivalenten Eingabe-Port, dann wird der ursprüngliche mit *s* verbundene Kanal in zwei Kanäle aufgespalten, die zu den beiden Ports der Komponenten *B* und *C* umgeleitet werden. Es entsteht in diesem Fall eine eins zu zwei Kommunikationsbeziehung (siehe Ports *i*2 der Komponenten *B* und *C* in Abbildung 7.5). Entsprechend der Definition der zeitasynchronen AutoFocus Semantik wird für jeden der beiden Eingabe-Ports inner-

halb einer eins zu zwei Kommunikationsbeziehung ein eigener Eingabepuffer verwendet. Beide Puffer erhalten die gleichen Eingaben zur Zwischenspeicherung. Es muss sichergestellt werden, dass alle Eingaben, die von einer der aufgeteilten Komponenten verarbeitet werden, aus dem Puffer der anderen Komponente auch entfernt werden, um eine doppelte Verarbeitung der Eingabe zu vermeiden. Der zuvor erwähnte Aktivierungsmechanismus übernimmt diese Aufgabe.

Schritt 7: Nachdem die Kanäle, die mit Eingabe-Ports der Ursprungskomponente A verbunden waren, umgeleitet wurden, erfolgt jetzt die Umleitung der Ausgaben von A. Wir betrachten die einzelnen Ausgabe-Ports u der Komponente A. Falls zu einem Port u ein äquivalenter, das heißt gleich benannter, Ausgabe-Port w der Komponente B existiert, aber kein äquivalenter Port der Komponente C, dann wird der mit u verbundene Kanal an den Port w umgehängt. Analog hierzu erfolgt die Umleitung des Kanals zu einem äquivalenten Port der Komponente C, falls in B kein äquivalenter Port existiert. Existieren zu einem Ausgabe-Port u der Komponente A in beiden Komponenten B und C äquivalente Ports w und x, dann müssen die Ausgaben beider Ports w und x berücksichtigt werden. Der ursprüngliche Kanal, dessen Quelle mit Port u verbunden ist, wird zu Port w umgehängt und ein neuer Kanal mit Quell-Port x und gleichem Ziel-Port, wie der zuvor betrachtete Kanal, wird in das SSD eingefügt. Es entsteht eine zwei zu eins Kommunikationsbeziehung, die in der zeitasynchronen AutoFocus Notation im Gegensatz zu der herkömmlichen zeitsynchronen AutoFocus Notation erlaubt ist (siehe Abschnitt 6.6.6).

Zur Vereinfachung der in Abschnitt 3.3 festgelegten Semantik von zeitsynchronem AutoFocus wurde dort gefordert, dass Ausgabe-Ports atomarer Komponenten eindeutig bezeichnet sein müssen. Diese Vereinfachung gilt auch für die zeitasynchrone Variante von AutoFocus. Aus diesem Grund muss an dieser Stelle einer der beiden in zwei zu eins Kommunikationsbeziehung stehenden Ausgabe-Ports umbenannt werden. In der Abbildung 7.5 auf der vorherigen Seite ist zu sehen, dass der Ausgabe-Port $o2$ in der Komponente C in $o2'$ umbenannt wird. Die Ausgabe-Port-Bezeichner in den Ausgabetermen der Transitionen der der Komponente C assoziierten Zustandsmaschine C sind entsprechend dieser Umbenennung ebenfalls umzubenennen.

Schritt 8: Die ursprüngliche Komponente A ist jetzt nicht mehr über Kanäle mit anderen Ports des SSDs D verbunden. Sie wird gelöscht (Abbildung 7.5 auf der vorherigen Seite). Gleichzeitig wird auch das zur Komponente A assoziierte STD A aus der Modellspezifikation gelöscht. Die Änderungen innerhalb des SSDs sind somit abgeschlossen.

Nachfolgend werden die Veränderungen der Zustandsübergangsdiagramme durch das *Split Component* Refactoring betrachtet. In Schritt 1 wurde die assoziierte Zustandsmaschine der Komponente A durch das Kopieren der Komponente A in die Komponenten B und C mit kopiert, das heißt die Komponenten B und C sind zu den STDs B und C assoziiert, die momentan beide eine eins zu eins Kopie des STDs A der ursprünglichen Komponente A sind.

Schritt 9: In die STDs B und C wird jeweils ein Zustand mit dem Namen *Inactive* eingefügt (Abbildung 7.6 auf der nächsten Seite). Falls in der Zustandsmenge Z_B kein Start-

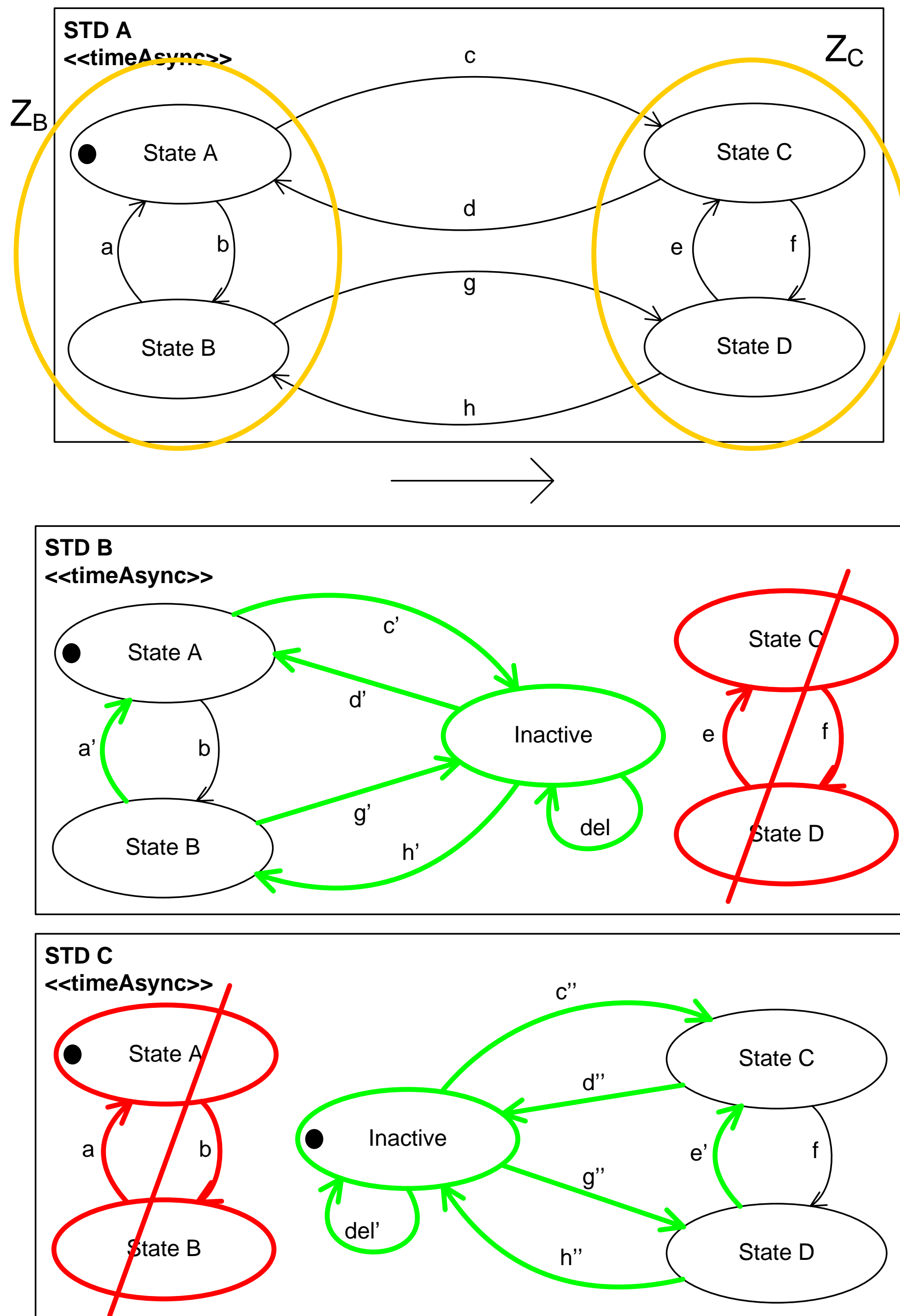


Abbildung 7.6.: Schritte 9 bis 14 des Split Component Refactorings.

zustand enthalten ist, dann wird der *Inactive* Zustand im STD *B* als ein Startzustand markiert. Gleiches gilt für das STD *C* falls kein Startzustand in Z_C enthalten ist.

Schritt 10: Stehen Eingabe-Ports der aufgespaltenen Komponenten in eins zu zwei Kommunikationsbeziehungen, dann muss sichergestellt werden, dass Eingaben, die von einer der beiden Komponenten verarbeitet wurden, aus dem Eingabepuffer der anderen Komponente gelöscht werden. Diese Aufgabe wird über den Aktivierungsmechanismus realisiert und in den folgenden Teilschritten erläutert.

Schritt 10a: Die Informationen über die Verarbeitung von Eingaben in eins zu zwei Kommunikationsbeziehungen werden über die Aktivierungskanäle zwischen den beiden aufgespaltenen Komponenten kommuniziert. Zu diesem Zweck müssen zunächst in den Datentypen der Aktivierungskanäle entsprechende Nachrichten hinzugefügt werden. Für jedes in eins zu zwei Kommunikationsbeziehung stehende Eingabe-Port-Paar werden beide Datentypen, die zur Komponentenaktivierung verwendet werden (hier *activateC* und *activateB*), um eine Nachricht, deren Name aus dem Port-Namen und dem Schlüsselwort *Delete* zusammengesetzt wird, erweitert.

Für die Spezifikation aus Abbildung 7.3 auf Seite 162 ergeben sich die zur Kommunikation der Verarbeitung verwendeten Nachrichten wie folgt:

```
data activateC = i2Delete;
data activateB = i2Delete;
```

In den Schritten 11a und 12a werden diese Datentypen um die eigentlichen Aktivierungsnachrichten ergänzt.

Schritt 10b: Die STDs B und C werden nun um das Senden der Benachrichtigung von der Verarbeitung von Eingaben, die über Ports in eins zu zwei Kommunikationsbeziehungen empfangen werden, erweitert. Wir bezeichnen im Folgenden diese speziellen Nachrichten als Löschnachrichten. Alle Transitionen dieser STDs, deren Eingabebedingungen Eingabe-Ports referenzieren, die in einer eins zu zwei Kommunikationsbeziehung stehen, werden um das Senden entsprechender Löschnachrichten an die andere momentan inaktive Komponente über den Aktivierungskanal ergänzt.

In der Spezifikation aus Abbildung 7.3 auf Seite 162 wird in dem STD *B* eine Transition, die den Port *i*2 in der Eingabebedingung referenziert, um die folgende Ausgabe ergänzt:

```
activateC!i2Delete
```

Ebenso wird in dem STD *C* eine Transition, die den Port *i*2 in der Eingabebedingung referenziert, um die folgende Ausgabe erweitert:

```
activateB!i2Delete
```

In der Abbildung 7.6 auf der vorherigen Seite ist das Einfügen der speziellen

Löschnachrichten durch die Umbenennung der Transitionen a und e in a' und e' angedeutet.

Werden in einer Transition mehrere Eingabe-Ports in eins zu zwei Kommunikationsbeziehung referenziert, dann werden für jede dieser Referenzierungen eine den entsprechenden Port referenzierende Löschnachricht über den Aktivierungs-Port gesendet. Das Senden von mehreren Nachrichten auf einen Port innerhalb einer Transition wird in dem zeitasynchronen AutoFocus unterstützt und ist in Abschnitt 6.6.7 definiert.

Schritt 10c: Im vorhergehenden Teilschritt wurden die Zustandsmaschinen um das Senden von Löschnachrichten zur Synchronisation von eins zu zwei Kommunikationsbeziehungen erweitert. In diesem Teilschritt wird nun das Löschen von Eingaben aus den Eingabepuffern, die von der anderen Komponente verarbeitet wurden, realisiert. Hierzu wird für jede in Schritt 10a in den Datentypen der Aktivierungskanäle hinzugefügten Löschnachrichten eine Transition zum Löschen der Eingabe mit Start- und Zielzustand *Inactive* in die STDs B und C eingefügt. Die Eingabebedingung der Löschtransition prüft auf das Vorhandensein der Löschnachricht sowie auf eine Nachricht auf dem durch die Löschnachricht referenzierten Eingabe-Port, der in einer eins zu zwei Kommunikationsbeziehung steht. Die Transition tätigt keine Ausgaben und Variablenzuweisungen.

In der Abbildung 7.6 auf Seite 169 sind die Löschtransitionen mit den Transitionsbezeichnern *del* und *del'* gekennzeichnet. Diese Transitionen sind für die hier betrachtete Spezifikation wie folgt festgelegt:

```
Bezeichnung: del
Eingabe:     a2?i2Delete; i2?x

Bezeichnung: del'
Eingabe:     a1?i2Delete; i2?x
```

Durch den hier eingefügten Mechanismus wird erreicht, dass, bevor eine Aktivierung einer Komponente erfolgen kann, alle Eingaben über Eingabe-Ports in eins zu zwei Kommunikationsbeziehungen, die von der anderen Komponente verarbeitet wurden, aus ihren Eingabepuffern gelöscht werden. Die Eingabepuffer beider Komponenten werden also synchronisiert.

Schritt 11: Wir betrachten nun alle Transitionen s des STDs B, die über die Partitionierungsgrenze hinweg von Z_B nach Z_C verlaufen, das heißt deren Quellzustand in der Zustandsmenge Z_B und deren Zielzustand in der Menge Z_C enthalten sind.

Schritt 11a: Für jede Transition s wird der Datentyp *activateC* um ein zusätzliches eindeutiges Aufzählungselement erweitert. Sinnvollerweise sollten hierfür die Transitionsnamen (Labels) von s verwendet werden. Sind keine Transitionsnamen vorhanden oder sind diese nicht eindeutig, dann sind sie durch entsprechende eindeutige Namen zu ersetzen.

Zu der Zustandsmaschine aus Abbildung 7.6 auf Seite 169 ergibt sich bei der Partitionierung $Z_B = \{A, B\}, Z_C = \{C, D\}$ folgende Datentypdefinition von *activateC*:

```
data activateC = i2Delete | c | g ;
```

Schritt 11b: In dem STD B werden die Zielzustände aller Transitionen s, die von Z_B nach Z_C verlaufen, umgeleitet. Neuer Zielzustand ist der neu eingefügte *Inactive* Zustand. In Abbildung 7.6 auf Seite 169 werden die ursprünglichen Transitionen c und g umgeleitet, die nach der Umleitung mit c' und g' bezeichnet sind. Die umgeleitete Transition s wird um das Senden einer Nachricht zur Aktivierung der Komponente C erweitert. Hierzu wird über den Kanal $a1$ der Name der Transition gesendet (siehe Datentyp *activateC*). Hierbei ist darauf zu achten, dass die Aktivierungsnachricht nach allen in der Transition eventuell vorhandenen Löschnachrichten, die ebenfalls über den Port $a2$ gesendet werden, gesendet wird. Dies wird dadurch erreicht, dass die Aktivierungsnachricht in dem Ausgabeterm der Transition als letzte Nachricht erscheint.

Zusätzlich zum Senden der Aktivierungsnachricht muss die Belegung von gemeinsam verwendeten Variablen synchronisiert werden. Zu diesem Zweck wird zu jeder gemeinsam benutzten Variablen, die durch ausgehende Transitionen der Zustände in Z_B verändert wird, über den entsprechenden Synchronisationskanal (in dem Beispiel $v1$) der in Schritt 3 eingefügt wurde, der aktuelle Wert der Variablenbelegung gesendet. So wird in dem Beispiel die Transition mit dem Namen c um folgenden Inhalt erweitert:

```
Ausgabe: a1!c; v1!v
```

Tätigt die betrachtete Transition s selber eine Variablenzuweisung, dann muss der neu zugewiesene Wert bei der Variablensynchronisierung gesendet werden. Da entsprechend der AutoFocus Semantik bei der Ausgabe die in der aktuellen Transition getätigten Variablenzuweisungen nicht berücksichtigt werden, muss der Variablenzuweisungsterm in den Ausgabeterm der Transition integriert werden. Zu diesem Zweck wird in dem Ausgabeterm statt dem Variablenbezeichner der in der Variablenzuweisung rechts von dem $=$ Zeichen stehende Term eingesetzt.

Schritt 11c: In dem STD C wird die zur Transition s äquivalente Transition t ebenfalls auf den *Inactive* Zustand umgeleitet (In Abbildung 7.6 auf Seite 169 sind dies die Transitionen c'' und g''). In dieser Zustandsmaschine wird der Quellzustand von t auf den *Inactive* Zustand umgesetzt. Die Variablenvorbedingung, Eingabebedingung, Ausgabeterm und Variablenzuweisung der Transition t werden gelöscht. Statt dessen wird eine neue Eingabebedingung in t eingefügt, die auf eine entsprechende Aktivierungsnachricht mit dem Namen der Transition s von der Komponente B, die über den Kanal $a1$ gesendet wird, prüft.

Zusätzlich muss in der Transition t die Synchronisation gemeinsamer Variablen realisiert werden. Für jede gemeinsam benutze Variable, die durch ausgehende Transitionen der Zustände in Z_B verändert wird, muss der von der Komponente B zum Zeitpunkt der Aktivierung von C gesendete Variablenwert in der lokalen Variable gleichen Namens in der Komponente C gespeichert werden. Der Wert der gesendeten Nachricht

wird in der Eingabebedingung zunächst in einer Transitionsvariablen gespeichert und anschließend in der Variablenzuweisung der lokalen Variable zugewiesen.

In dem Beispiel ergibt sich folgender Inhalt für die Transition c'':

```
Eingabe: a1?c; v1?x
Variablenzuweisung: v=x
```

Schritt 12: Als nächstes werden alle Transitionen u des STDs C betrachtet, die über die Partitionierungsgrenze hinweg von Z_C nach Z_B verlaufen, das heißt deren Quellzustand in der Zustandsmenge Z_C und deren Zielzustand in der Menge Z_B enthalten sind. Die Änderungen erfolgen analog zu Schritt 11 aber in entgegengesetzter Richtung.

Schritt 12a: Für jede Transition u wird der Datentyp *activateB* um ein zusätzliches eindeutiges Aufzählungselement erweitert. Zu der Zustandsmaschine aus Abbildung 7.6 auf Seite 169 ergibt sich bei der Partitionierung $Z_B = \{A, B\}, Z_C = \{C, D\}$ folgende Datentypdefinition von *activateB*:

```
data activateB = i2Delete | d | h ;
```

Schritt 12b: In dem STD C werden die Zielzustände aller Transitionen u, die von Z_C nach Z_B verlaufen, umgeleitet. Neuer Zielzustand ist der neu eingefügte *Inactive* Zustand (siehe Abbildung 7.6 auf Seite 169 Transitionen d'' und h''). Die umgeleitete Transition u wird um das Senden einer Nachricht zur Aktivierung der Komponente B erweitert. Hierzu wird über den Port $a2$ der Name der Transition gesendet (siehe Datentyp *activateB*). Hierbei ist darauf zu achten, dass die Aktivierungsnachricht nach allen in der Transition eventuell vorhandenen Löschnachrichten, die ebenfalls über den Port $a2$ gesendet werden, gesendet wird. Dies wird dadurch erreicht, dass die Aktivierungsnachricht in dem Ausgabeterm der Transition als letzte Nachricht erscheint.

Zu jeder gemeinsam benutzten Variablen, die durch ausgehende Transitionen der Zustände in Z_C verändert wird, wird über den entsprechenden Synchronisationskanal (in dem Beispiel $v2$) der aktuelle Wert der Variablenbelegung gesendet. Hierbei sind auch die Variablenzuweisungen, die von der betrachteten Transition u selber getätigt werden, zu berücksichtigen. In dem Beispiel aus Abbildung 7.6 auf Seite 169 wird die Transition d'' um folgenden Inhalt erweitert:

```
Ausgabe: a2!d; v2!v
```

Schritt 12c: In dem STD B wird die zur Transition u äquivalente Transition w ebenfalls auf den *Inactive* Zustand umgeleitet (Transitionen d' und h' in Abbildung 7.6 auf Seite 169). In dieser Zustandsmaschine wird der Quellzustand von w auf den *Inactive* Zustand umgesetzt. Die Variablenvorbedingung, Eingabebedingung, Ausgabeterm und Variablenzuweisung der Transition w werden gelöscht. Statt dessen wird eine neue Eingabebedingung in w eingefügt, die auf eine entsprechende Aktivierungsnachricht mit dem Namen der Transition u von der Komponente C, die über den Kanal $a2$ gesendet wird, prüft.

Für jede gemeinsam benutze Variable, die durch ausgehende Transitionen der Zustände in Z_C verändert wird, muss der von der Komponente C zum Zeitpunkt der Aktivierung von B gesendete Variablenwert in der lokalen Variable gleichen Namens in der Komponente B gespeichert werden.

In dem Beispiel ergibt sich folgender Inhalt für die Transition mit dem Namen d':

```
Eingabe: a2?d; v2?x
Variablenzuweisung: v=x
```

Schritt 13: In dem STD B werden alle Zustände einschließlich verbundener Transitionen gelöscht, die nicht in Z_B enthalten sind (Abbildung 7.6 auf Seite 169). Ausgenommen hiervon ist der neu eingefügte *Inactive* Zustand.

Schritt 14: In dem STD C werden alle Zustände einschließlich verbundener Transitionen gelöscht, die nicht in Z_C enthalten sind (siehe Abbildung 7.6 auf Seite 169). Ausgenommen hiervon ist der neu eingefügte *Inactive* Zustand.

Das *Split Component* Refactoring ist hiermit abgeschlossen.

7.4. Remove Inactive States Refactoring

Problem

Das *Split Component* Refactoring (Abschnitt 7.3) fügt einen Aktivierungsmechanismus zwischen den beiden zerteilten Komponenten ein, der die Komplexität der Spezifikation erhöht.

Lösung

Falls der Aktivierungsmechanismus nicht erforderlich ist, wird dieser entfernt (siehe Abbildung 7.7 auf der nächsten Seite).

Motivation

Das in Abschnitt 7.3 beschriebene *Split Component* Refactoring fügt einen Aktivierungsmechanismus in die beiden aufgespaltenen Komponenten ein. Dieser Mechanismus stellt sicher, dass zu jedem Zeitpunkt immer nur eine der beiden Komponenten aktiv ist. Die andere Komponente befindet sich dann immer in dem *Inactive* Zustand. Bei einer Transition über die Partitionierungsgrenze des *Split Component* Refactorings hinweg wird die jeweils andere Komponente über die aufgetretene Transition informiert und damit auch aktiviert. Durch die Aktivierung erfolgt eine Synchronisation von gemeinsam benutzten Variablen und eine Synchronisation der Eingabepuffer von gemeinsam verwendeten Eingaben.

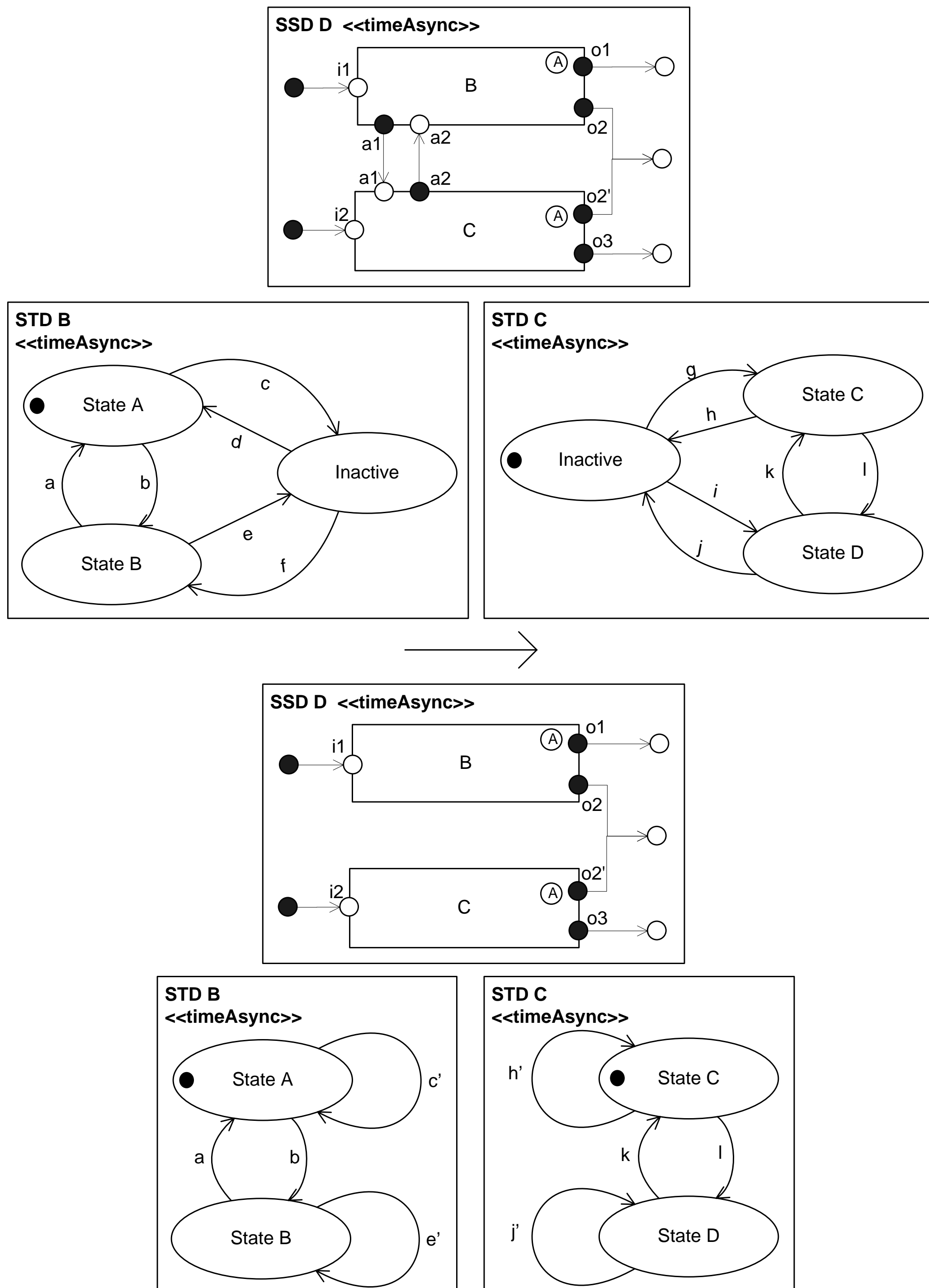


Abbildung 7.7.: Remove Inactive States Refactoring.

Der Aktivierungsmechanismus erfordert zur Realisierung zusätzliche Ports und Kanäle in dem Systemstrukturdiagramm und einen zusätzlichen Zustand sowie zusätzliche Transitionen in den Zustandsübergangsdiagrammen (STDs) beider aufgespaltener Komponenten. Der Mechanismus führt folglich zu einer komplexeren Modellspezifikation.

In manchen Fällen hat der Aktivierungsmechanismus jedoch keine Auswirkungen auf das zeitabstrahierte Verhalten dieser beiden Komponenten. In diesen Fällen sollte der Aktivierungsmechanismus entfernt und dadurch die Modellspezifikation vereinfacht werden. Die Bedingungen dafür, dass das *Remove Inactive States* Refactoring das zeitabstrahierte Systemverhalten nicht verändert, sind sehr komplex und lassen sich nur schwer von der Verhaltensinvarianzeigenschaft ableiten. Wir entschließen uns das *Remove Inactive States* Refactoring als ein nicht allgemeingültiges Refactoring[5] zu definieren, das eine Überprüfung der Verhaltensäquivalenz der Modellspezifikation nach Anwendung des Refactorings erfordert.

Nach der Anwendung des *Split Component* Refactorings sollte grundsätzlich betrachtet werden, ob die Anwendung des *Remove Inactive States* Refactorings möglich und sinnvoll ist. Das *Remove Inactive States* Refactoring kann angewendet werden, wenn die beiden aufgespaltenen Komponenten keine gemeinsamen Variablen besitzen, das heißt keine Variablen über Kanäle, die durch das *Split Component* Refactoring eingefügt wurden, synchronisiert werden. Darüber hinaus dürfen die aufgespaltenen Komponenten keine Eingabe-Ports in eins zu zwei Kommunikationsbeziehung besitzen, deren Puffer durch den Aktivierungsmechanismus, der durch das *Split Component* Refactoring eingefügt wurde, synchronisiert werden.

Weisen die aufgeteilten Komponenten, die aus der Anwendung des *Split Component* Refactorings resultieren, keine gemeinsamen Variablen auf und besitzen diese keine synchronisierten Eingabe-Ports, dann soll das *Remove Inactive States* Refactoring angewendet werden. Anschließend ist die zeitabstrahierte Verhaltensäquivalenz der Modellspezifikationen vor und nach Durchführung des Refactorings zu zeigen (siehe Abschnitt 9.3). Wird festgestellt, dass das *Remove Inactive States* Refactoring zu einer Verhaltensänderung unter Zeitabstraktion geführt hat, dann muss das Refactoring rückgängig gemacht werden.

Methode

Benutzereingaben:

Der Entwickler wählt zwei Komponenten B und C in einem SSD D aus. Die Komponente B besitzt ein assoziiertes STD B und das STD C ist der Komponente C assoziiert.

Vorbedingungen:

Damit das Refactoring angewendet werden kann, müssen die Komponenten B und C die aus der Anwendung des *Split Component* Refactorings resultierenden aufgespalte-

[5] Die Definition allgemeingültiger Refactorings ist in Definition 4.4 auf Seite 85 aufgeführt.

nen Komponenten sein. Beide Komponenten dürfen nicht gemeinsame Variablen verwenden, das heißt es existieren keine durch die Anwendung des *Split Component* Refactorings eingefügten Kanäle und Ports zur Variablensynchronisierung zwischen den beiden Komponenten. Darüber hinaus dürfen beide Komponenten nicht Eingabe-Ports in eins zu zwei Kommunikationsbeziehung, die über den Aktivierungsmechanismus synchronisiert werden, besitzen.

Schritt 1: In dem SSD D werden die Aktivierungs-Ports $a1$ und $a2$ der Komponenten B und C einschließlich der verbundenen Kanäle gelöscht (siehe Abbildung 7.7 auf Seite 175 oben).

Schritt 2: In dem STD (STD B oder STD C), in dem der *Inactive* Zustand Startzustand ist, wird ein beliebiger anderer Zustand als Startzustand gewählt (Abbildung 7.7 auf Seite 175 unten).

Schritt 3: In den STDs B und C werden alle Transitionen, deren Zielzustand der *Inactive* Zustand ist, zu deren Quellzustand umgeleitet. Es entstehen also Transitionen mit gleichem Quell- und Zielzustand (Abbildung 7.7 auf Seite 175 unten). In dem Ausgabeterm dieser Transition wird das Senden der Aktivierungsnachricht entfernt. In dem in der Abbildung angegeben Beispiel wird beispielsweise in der Transition c des STD B der folgende Ausdruck entfernt und anschließend als c' bezeichnet:

```
Ausgabe: a1!c
```

Schritt 4: In beiden Zustandsmaschinen wird der *Inactive* Zustand einschließlich aller ausgehenden Transitionen gelöscht (Abbildung 7.7 auf Seite 175 unten).

Im Anschluss an dieses Refactoring muss die Verhaltensäquivalenzeigenschaft auf Komponentenebene unter Verwendung der Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ oder auf Systemebene unter Verwendung der Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ überprüft werden.

7.5. Nachweis der Verhaltensinvarianzeigenschaft unter Zeitabstraktion

Bei den Refactorings *Remove Intermediate State*, *Insert Intermediate State*, *Split Component* und *Combine Components* handelt es sich um allgemeingültige Refactorings, die die Verhaltensinvarianzeigenschaft bezüglich der lokal geänderten Komponentenspezifikation unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ für alle zeitasynchronen AutoFocus Modellspezifikationen erfüllen. Auf Grund der Erfüllung der Zeitrobustheitseigenschaft durch alle in der zeitasynchronen AutoFocus Modellierungssprache ausdrückbaren Spezifikationen wird hierdurch die Verhaltensäquivalenz einer Gesamtspezifikation vor und nach dem Refactoring, die die veränderte Komponentenspezifikation enthält, unter der Abstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ sichergestellt (vergleiche Definition 6.4 auf Seite 129 von zeitrobustem Refactoring).

Die hier vorgestellten Refactorings lassen sich in leicht abgewandelter Form auch auf

zeitsynchrone AutoFocus Modelle anwenden, die die Zeitrobustheitseigenschaft in Bezug auf die Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ auf der Ebene von Komponentenverhalten und die Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA3}$ auf der Ebene von Systemverhalten erfüllen.

Wir betrachten im Folgenden die einzelnen vorangehend definierten Refactorings in Hinblick auf die Veränderung des Schnittstellenverhaltens der veränderten Verarbeitungskomponente ohne Berücksichtigung der vorgeschalteten Eingabepuffer unter Anwendung der Abstraktion $\underline{\widetilde{\alpha}}_{ZA2}$.

7.5.1. Verhaltensinvarianz des Remove Intermediate State Refactorings

Das *Remove Intermediate State* Refactoring fasst zwei Transitionen zu einer Transition zusammen. Der Zwischenzustand stellt keinen richtigen Systemzustand dar, da er genau eine eingehende und eine ausgehende Transition besitzt. In dem Zwischenzustand kann folglich „nur“ gewartet werden. Das Zusammenfassen der Transitionen ist nur erlaubt, falls die vom Zwischenzustand ausgehende Transition weder eine Variablenvorbedingung noch eine Eingabebedingung besitzt. Durch das Refactoring wird das nicht abstrahierte Schnittstellenverhalten der Komponente dahingehend verändert, dass Ausgaben auf verschiedene Ports, die vorher nacheinander erfolgt sind, jetzt gleichzeitig auftreten. Durch das Zusammenfassen zweier Transitionen zu einer Transition ergibt sich für das Verhalten ohne Anwendung von Zeitabstraktion die Änderung, dass auf Eingaben schneller mit Ausgaben reagiert werden kann. In der Spezifikation vor dem Refactoring wird nach dem Feuern der zwei Transitionen ein Kontrollzustand erreicht, der zu dem Kontrollzustand in der Spezifikation nach dem Refactoring, der nach Feuern der einen zusammengefassten Transition erreicht wird, äquivalent ist.

Der Variablenzustand, der in der Spezifikation vor dem Refactoring nach dem Feuern beider Transitionen erreicht wird, ist ebenfalls zu dem Variablenzustand, der nach dem Feuern der einen Transition in der Spezifikation nach dem Refactoring erreicht wird, äquivalent, da der Variablenzuweisungsterm der durch das Refactoring neu entstandenen Transition aus der Konjunktion der beiden Variablenzuweisungsterme der beiden ursprünglichen Transitionen gebildet wird, und auf Grund der Vorbedingung des Refactorings ein Zusammenfassen nur möglich ist, wenn die Variablenzuweisungsterme beider Transitionen unterschiedliche Variablen referenzieren.

Die Vorbedingung des *Remove Intermediate State* Refactorings legt fest, dass die aus dem Zwischenzustand ausgehende Transition keine Variablenvorbedingung und keine Eingabebedingung besitzen darf. Diese Vorbedingung verhindert eine lokale nicht zeitliche Verhaltensänderung, die durch das Zusammenfassen der zwei Transitionen entstehen würde, falls es sich bei dem Zwischenzustand um einen lokalen *Dead Lock* Zustand handelt. Da entsprechend der Vorbedingung die ausgehende Transition keine Bedingungen an das Feuern festlegt, ist diese jederzeit schaltbereit und bei dem betrachteten Zwischenzustand kann es sich daher nicht um einen lokalen *Dead Lock* Zustand handeln.

Betrachten wir nun die Wirkung des Refactorings auf das Schnittstellenverhalten der

veränderten Komponente. Wir betrachten hierbei die Verarbeitungskomponente isoliert von den in der zeitasynchronen Semantik verwendeten Eingabepuffern.

Die Abbildung 7.8 auf der nächsten Seite zeigt die Veränderung des Verhaltens durch das Refactoring des Beispielsystems aus Abbildung 7.1 auf Seite 157 in den zeitabstrakten AutoFocus Sequenzdiagrammen. Der in der Abbildung auf der linken Seite aufgeführte Ablauf beschreibt das Verhalten vor dem Refactoring. Zunächst wird die Eingabe auf dem Port *turnOnMotor* empfangen und verarbeitet. Anschließend wird im nächsten Takt die Ausgabe auf dem Port *motorRequest* gesendet. Anschließend erfolgt mit mindestens einem Takt Verzögerung die Ausgabe *motorOn:Present*.

In der Abbildung 7.8 auf der nächsten Seite auf der rechten Seite ist das Verhalten der Beispielkomponente nach Anwendung des *Remove Intermediate State* Refactorings dargestellt. Auf die Eingaben wird jetzt mit einem Takt Verzögerung mit beiden Ausgaben gleichzeitig reagiert.

Die Zeitabstraktion $\widetilde{\underline{\alpha}}_{ZA2}$ (Definition 3.18 auf Seite 76) abstrahiert von AutoFocus Zeitschritten und von der Reihenfolge aufeinander folgender Eingaben beziehungsweise Ausgaben auf unterschiedlichen Ports. Die Reihenfolge zwischen Ein- und Ausgaben bleibt jedoch erhalten. Unter dieser Abstraktion sind beide in der Abbildung gezeigten Abläufe gleich. Die Veränderungen des Refactorings auf das Schnittstellenverhalten sind also unter der Zeitabstraktion nicht sichtbar. Nach der Definition 6.4 auf Seite 129 von Refactoring zeitrobuster Spezifikationen reicht für den Nachweis der Verhaltensäquivalenz eines Gesamtsystems der Nachweis der Verhaltensäquivalenz der veränderten Komponente aus. Die aus dem Refactoring resultierende Modellspezifikation ist, wie in der Definition von zeitrobustem Refactoring gefordert, ebenfalls zeitrobust, da diese Spezifikation weiterhin nach der zeitasynchronen Semantik interpretiert wird, die per Konstruktion die Eigenschaft der Zeitrobustheit erfüllt.

Da dieses Refactoring keine Veränderungen der Systemstrukturdiagramme vornimmt, ist sichergestellt, dass die betrachtete Komponente vor und nach dem Refactoring, wie von der Verhaltensäquivalenzeigenschaft für das Refactoring gefordert, die gleiche syntaktische Schnittstelle besitzt. Das Refactoring ist somit ein allgemeingültiges Refactoring für AutoFocus Spezifikationen, die nach der zeitasynchronen Semantik interpretiert werden.

7.5.2. Verhaltensinvarianz des Insert Intermediate State Refactorings

Das *Insert Intermediate State* Refactoring stellt genau die Umkehrabbildung zu dem Spezialfall des *Remove Intermediate State* Refactorings, bei dem eine beliebige Transition mit einer leeren Transition zusammengefasst wird, dar. Für die Umkehrabbildung eines Refactorings gilt ebenfalls die Verhaltensinvarianzeigenschaft der Spezifikationen unter Abstraktion vor und nach Anwendung der Transformation. Das *Insert Intermediate State* Refactoring ist somit ebenfalls ein allgemeingültiges Refactoring für zeitasynchrone AutoFocus Modellspezifikationen.

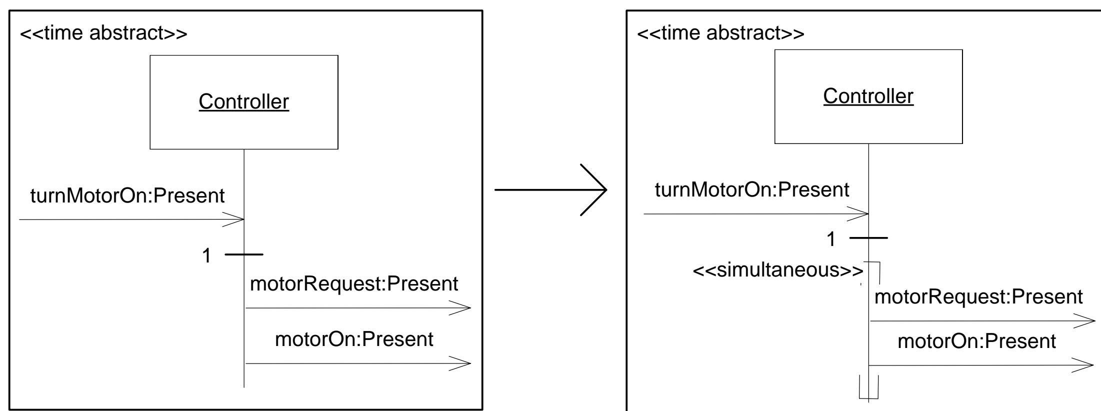


Abbildung 7.8.: Veränderung des Verhaltens durch das *Remove Intermediate State* Refactoring.

7.5.3. Verhaltensinvarianz des Split Component Refactorings

Das *Split Component* Refactoring verändert, wie in Abbildung 7.3 auf Seite 162 dargestellt, sowohl das Systemstrukturdiagramm (SSD) als auch Zustandsübergangsdiagramme (STDs). Betrachten wir zunächst die Änderungen, die in dem Systemstrukturdiagramm durchgeführt werden.

Veränderungen in dem Systemstrukturdiagramm. Die Ursprungskomponente A wird zweimal kopiert. Die Kopien heißen B und C, enthalten alle Ports von A und sind assoziiert zu Kopien der Zustandsmaschine von A. Durch eine Analyse der Transitionen entsprechend der gewählten Partitionierung werden in beiden Komponenten gemeinsam verwendete Ports identifiziert. Gemeinsam verwendete Eingabe-Ports der Komponenten B und C erhalten durch eine eins zu zwei Kommunikationsbeziehung die gleichen Eingaben. Ausgaben auf gemeinsame Ports werden durch zwei zu eins Kommunikationsbeziehungen, die in dem zeitasynchronen AutoFocus zulässig sind, zusammengeschaltet. Ports, die von einer der aufgeteilten Komponenten entsprechend der gewählten Partitionierung der ursprünglichen Zustandsmaschine nicht verwendet werden, werden gelöscht. Die Kommunikationskanäle, die mit Ports der Komponente A verbunden waren, werden an die äquivalenten Ports der neuen Komponenten B und C umgeleitet.

Die resultierende Kommunikationsinfrastruktur leitet Nachrichten, die ursprünglich von der Komponente A empfangen wurden, an die äquivalenten Ports der Komponenten B und / oder C um. Es ist sichergestellt, dass die neuen Komponenten B und C mit den von ihnen benötigten Eingaben versorgt werden. Auf der Ausgabeseite sind die Ausgabe-Ports von B und C mit den Kanälen verbunden, die ursprünglich mit äquivalenten Ports von A verbunden waren. Die Ausgaben der aufgeteilten Komponenten

werden folglich über die korrekten Kanäle weitergeleitet.

Beobachtete Schnittstelle. Die lokalen Verhaltensänderungen der vorangehend besprochenen Refactorings wurden unter Ausschluss der durch die zeitasynchrone Semantik definierten vorangestellten Eingabepuffer betrachtet. Der Grund hierfür liegt in einer Einschränkung der angewendeten Abstraktion $\widetilde{\underline{\alpha}}_{ZA2}$. Diese Abstraktion kann bei Anwendung auf komponiertes Verhalten nicht zwischen tatsächlich in der Verhaltensspezifikation festgelegten kausalen Abhängigkeiten zwischen Ein- und Ausgaben und zufälligen zeitlichen Beziehungen zwischen Ein- und Ausgaben verschiedener parallel arbeitender Komponenten unterscheiden (vergleiche Abschnitt 6.5).

Auf Grund dieser Einschränkung der anzuwendenden Abstraktion werden auch beim Nachweis der Verhaltensäquivalenz des *Split Component* Refactorings nicht die an der Grenze der Schnittstelle der zerteilten Komponente vorhandenen Eingabepuffer mit betrachtet. In der Abbildung 7.9 auf der nächsten Seite sind die beobachteten Schnittstellen für den Nachweis der Verhaltensinvarianz des Refactorings durch eine gestrichelte Linie mit dem Wort *Observe* gekennzeichnet. Die Abbildung oben zeigt die Komponentenspezifikation vor der Anwendung des Refactorings und unten ist die Komponentenspezifikation nach dem Refactoring dargestellt. Bei den in der Abbildung gezeigten Spezifikationen handelt es sich um zeitasynchrone AutoFocus Modelle, die entsprechend der Semantikdefinition in Kapitel 6 in zeitsynchrone AutoFocus Modelle übersetzt sind. Die Beobachtung schließt, wie in der Abbildung dargestellt, die nach außen sichtbaren Eingabepuffer nicht mit ein. Die zur Realisierung von zwei zu eins Kommunikationsbeziehungen in der zeitasynchronen Semantikdefinition festgelegten Mux Komponenten (siehe Abschnitt 6.6.6) werden hingegen bei der Betrachtung des lokalen Verhaltens mit eingeschlossen.[6]

Ein Vergleich der syntaktischen Schnittstellen der markierten beobachteten Schnittstellen der beiden Modelle in Abbildung 7.9 auf der nächsten Seite zeigt, dass in dem oberen Modell ein Eingabe-Port *i*2 existiert, das untere Modell hingegen zwei verschiedene Eingabe-Ports *i*2 besitzt. Entsprechend der Semantikdefinition der zeitasynchronen AutoFocus Semantik werden eins zu zwei Kommunikationsbeziehungen mit Hilfe von zwei getrennten Eingabe-Puffern realisiert.[7]

[6] Die Verwendung von Immediate Kommunikation zwischen den Verarbeitungskomponenten und den Mux Komponenten stellt sicher, dass das zeitliche Verhalten durch das Zwischenschalten der Mux Komponenten nicht verändert wird, solange von den mit der Mux Komponente verbundenen Verarbeitungskomponenten keine gleichzeitigen Ausgaben auftreten.

[7] Die hier auftretende Problematik kann alternativ durch eine abgewandelte Semantik der eins zu zwei Kommunikationsbeziehung gelöst werden. Wird die Semantik so definiert, dass bei eins zu zwei Kommunikation nicht zwei unabhängige Eingabe-Puffer, sondern nur ein Eingabe-Puffer eingefügt wird, und die Verzweigung der Kommunikation erst nach diesem Puffer erfolgt, dann sind die Signaturen beider zu vergleichenden Schnittstellen identisch. Tatsächlich würde diese Art der Kommunikationssemantik in dem hier betrachteten Refactoring die Verhaltensäquivalenz gewährleisten. Aus praktischer Sicht ist diese Art der Kommunikation jedoch problematisch, da der Fall auftreten kann, dass beide mit dem Puffer verbundenen Verarbeitungskomponenten eine bestimmte Eingabe gleichzeitig verarbeiten. Bei nicht gleichzeitiger Verarbeitung wird hingegen die Eingabe immer nur von einer Verar-

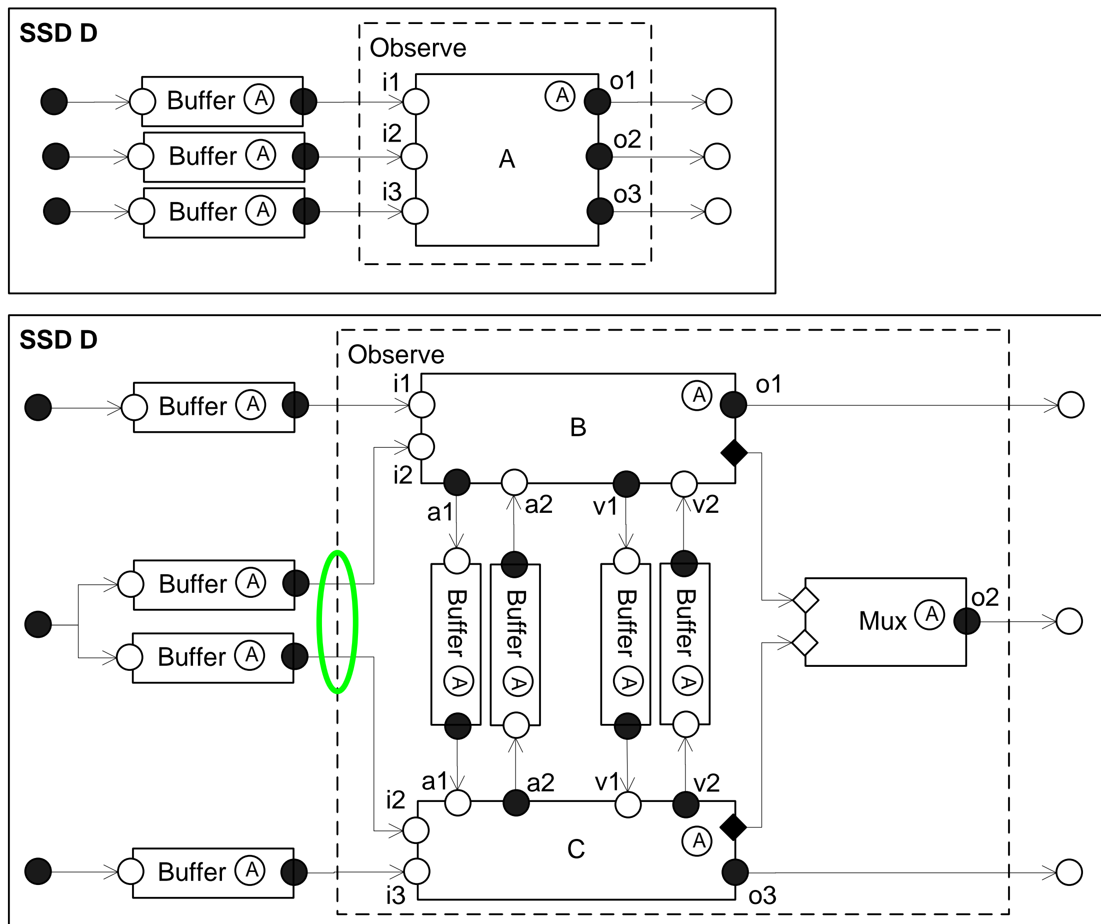


Abbildung 7.9.: Beobachtete Schnittstelle beim Nachweis der Verhaltensäquivalenz des Split Component Refactorings.

Erläuterungen zur Abbildung: Oben ist ein Modell vor der Anwendung des Split Component Refactorings dargestellt. Der untere Teil der Abbildung zeigt dieses Modell nach der Anwendung des Refactorings. Beide dargestellten Modelle sind zeitasynchrone AutoFocus Modelle, die entsprechend der Semantikdefinition in zeitsynchrone AutoFocus Modelle übersetzt sind. Die gestrichelte Line mit dem Wort Observe legt in beiden Modellen die Schnittstelle fest, deren Verhalten für den Nachweis der Verhaltensäquivalenz betrachtet wird.

Die syntaktischen Schnittstellen beider betrachteter Verhalten sind also verschieden voneinander. Der direkte Vergleich der Schnittstellenverhalten ist somit nicht möglich. Es kann jedoch eine stromverarbeitende Funktion definiert werden, die beide in eins zu zwei Kommunikationsbeziehung stehenden Ströme zu einem Strom zusammenfasst, die Signatur der Schnittstellen also aneinander anpasst und somit den Vergleich der Schnittstellenverhalten der beiden betrachteten Modelle ermöglicht. Auf Grund der Eigenschaft der eins zu zwei Kommunikationsbeziehung, eingehende Nachrichten dupliziert in beiden Puffern zur Verfügung zu stellen, sind beide betrachteten nach $\underline{\alpha}_{NI}$ abstrahierten[8] Eingabeströme modulo zeitlicher Verschiebungen gleich, falls in den Ausführungen der beiden diese Eingabeströme empfangenden Komponenten keine *Progress Locks*, das heißt keine dauerhaften Stillstände in der Verarbeitung bezüglich der betrachteten Eingaben, auftreten. In solchen Eingabeströmen sind also für alle *n* die *n*-ten in den beiden Strömen vorkommenden verarbeiteten Nutznachrichten[9] identisch. Wir definieren eine stromverarbeitende Focus Funktion *combine12Comm*, die zwei Eingabeströme zu einem Eingabestrom kombiniert, wobei jeweils für alle *n* nur die zeitlich früher auftretende *n*-te verarbeitete Nutznachricht beider Ströme in dem resultierenden gezeiteten Strom berücksichtigt wird. Die *combine12Comm* Funktion ist im Anhang D formal definiert. Die Funktion ist auch in Fällen anwendbar, in denen die vorangehend erwähnten *Progress Locks* auftreten. Tritt in dem Ablauf einer der verarbeitenden Komponenten ein *Progress Lock* bezüglich des betrachteten Eingabestroms auf, dann werden ab diesem Zeitpunkt nur noch die verarbeiteten Eingaben in dem Ablauf der anderen verarbeitenden Komponente berücksichtigt.

Ist ein zeitasynchrones AutoFocus Schnittstellenverhalten, dessen Eingabeströme, die in eins zu zwei Kommunikationsbeziehungen stehen, mit Hilfe der *combine12Comm* Funktion zu jeweils einem Eingabestrom zusammengefasst wurden, identisch zu einem anderen AutoFocus Verhalten, dessen Eingabeströme nicht in einer eins zu zwei Kommunikationsbeziehung stehen, dann haben beide Verhalten die gleichen Wirkungen auf ein diese Verhalten umgebendes Restsystemverhalten. Somit lässt sich die Verhaltensäquivalenzeigenschaft bezüglich der Abstraktion $\widetilde{\underline{\alpha}}_{ZA2}$, die für zeitrobustes AutoFocus Refactoring (Definition 6.4 auf Seite 129) gefordert wird, unter Verwendung der *combine12Comm* Funktion für das *Split Component* Refactoring erfüllen. Des Weiteren wird bei kombinierter Betrachtung von Eingabeströmen, die in eins zu zwei Kommunikationsbeziehung stehen, auch davon ausgegangen, dass in der syntaktischen Schnittstelle des zu beobachtenden Verhaltens ebenfalls die entsprechenden Eingabe-Ports beziehungsweise Eingabe-Kanäle zu einem Focus Eingabe-Kanal zusammengefasst werden. Auf Grund der Argumentation im vorangehenden Abschnitt „Verände-

beitungskomponente verarbeitet und anschließend aus dem Puffer entfernt. Diese Anomalie zwischen gleichzeitiger und nicht gleichzeitiger Verarbeitung von Eingaben ist für einen Entwickler nur sehr schwer nachvollziehbar. Aus diesem Grund wird auf den Einsatz der hier vorgestellten alternativen Semantik von eins zu zwei Kommunikationsbeziehungen verzichtet.

[8] Bei der Betrachtung der Eingabeströme werden hier nur verarbeitete Eingaben berücksichtigt.

[9] Als Nutznachrichten werden alle entsprechend dem für den Eingabekanal verwendeten Datentyp erlaubten Nachrichten, ausgenommen der speziellen *void* Nachricht und dem Focus Tick Symbol ✓, bezeichnet.

rungen in dem Strukturdiagramm“ sind unter dieser kombinierten Betrachtung von Eingabe-Ports in eins zu zwei Kommunikationsbeziehungen unter der gewählten Beobachtungsgrenze die syntaktischen Schnittstellen einer Modellspezifikation vor und nach Anwendung des *Split Component* Refactorings identisch.

Veränderungen in den Zustandsübergangsdiagrammen. Betrachten wir nun die Änderungen der Zustandsmaschinen durch das Refactoring und deren Wirkung auf das Schnittstellenverhalten der neu aufgeteilten Komponenten. Wie bei der Betrachtung der anderen Refactorings wird das Schnittstellenverhalten der Verarbeitungskomponenten ohne die vorgeschalteten Eingabepuffer betrachtet.

Zunächst besitzen die Komponenten B und C eine exakte Kopie des STDs A der Komponente A. Alle Transitionen und Zustände, die innerhalb der Partitionierung, also zwischen Zuständen aus der Menge Z_B beziehungsweise Z_C verlaufen, sind in dem STD B beziehungsweise STD C modulo der Erweiterung der Ausgabeterme der Transitionen um das Senden zusätzlicher Löschnachrichten, die nach außen nicht sichtbar sind, unverändert zur ursprünglichen Zustandsmaschine. Das dort spezifizierte nach außen sichtbare Verhalten bleibt gleich.

Interessant bezüglich einer Verhaltensänderung sind die Transitionen, die die Partitionierungsgrenze überschreiten. Betrachten wir ein konkretes Beispiel. Seien die Modellspezifikationen aus Abbildung 7.3 auf Seite 162 gegeben und betrachten wir den Wechsel von Zustand B über die Zustände A und C nach Zustand D in dem STD A und vergleichen den daraus resultierenden Ablauf mit dem äquivalenten Ablauf der durch das Refactoring veränderten Spezifikation (STDs B und C). Als konkrete Zustandsübergänge sind die in der Tabelle 7.1 auf der nächsten Seite aufgeführten Transitionen angenommen.

Die ursprüngliche Transition a wird durch das Refactoring um das Senden der Löschnachricht an die inaktive Komponente erweitert (neuer Transitionsnamen a') und die ursprüngliche Transition c wird durch das Refactoring in die zwei Transitionen c' und c'' aufgespalten, die auf die Zustandsmaschinen der zwei aufgeteilten Komponenten verteilt sind. Diese Transitionen sind in den resultierenden Zustandsmaschinen mit den neu eingefügten *Inactive* Zuständen verbunden. Jede Transition c', deren Zielzustand der *Inactive* Zustand ist, besitzt weiterhin die Variablenbedingung, die Eingabebedingung, den Ausgabeterm und die Variablenzuweisung der ursprünglichen Transition c des ursprünglichen STDs A. Zusätzlich sendet diese neue Transition c' eine Aktivierungsnachricht, die eindeutig das Feuern der Transition c' repräsentiert, an die andere zerteilte Komponente. Gleichzeitig werden auch die aktuellen Belegungen aller in den beiden zerteilten Komponenten gemeinsam benutzten Variablen an die andere Komponente übermittelt. In der anderen Komponente existiert nun eine Transition c'' mit Quellzustand *Inactive* und dem zu der ursprünglichen Transition c äquivalenten Zielzustand. Die Transition c'' wird genau dann gefeuert, wenn die entsprechende Aktivierungsnachricht, mit der Information, dass dort die Transition c' gefeuert wurde, der anderen Komponente empfangen wird. Die Transition c'' enthält nicht die Va-

Name	Vorbedingung	Eingabe	Ausgabe	Zuweisung
a		i2?Present	o1!locVar	
c	locVar=10	i1?Present	o2!Present	locVar=locVar+1
e		i3?Present	o3!locVar	

(a) Transitionen vor dem Refactoring.

Name	Vorbedingung	Eingabe	Ausgabe	Zuweisung
a′		i2?Present	o1!locVar; a1!i2Delete	
del′		a1?i2Delete, i2?x		
c′	locVar=10	i1?Present	o2!Present; a1!c; v1!locVar+1	locVar=locVar+1
c″		a1?c; v1?x		locVar=x
e		i3?Present	o3!locVar	

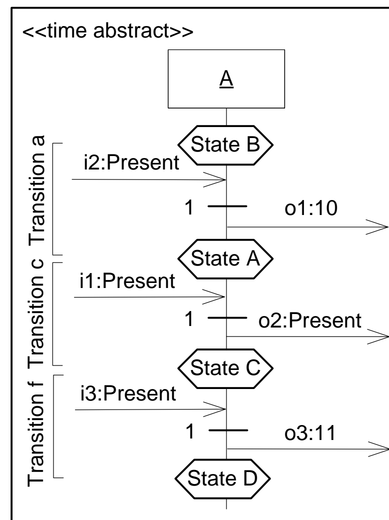
(b) Transitionen nach dem Refactoring.

Tabelle 7.1.: Transitionen des Beispiels vor und nach Anwendung des *Split Component* Refactorings.

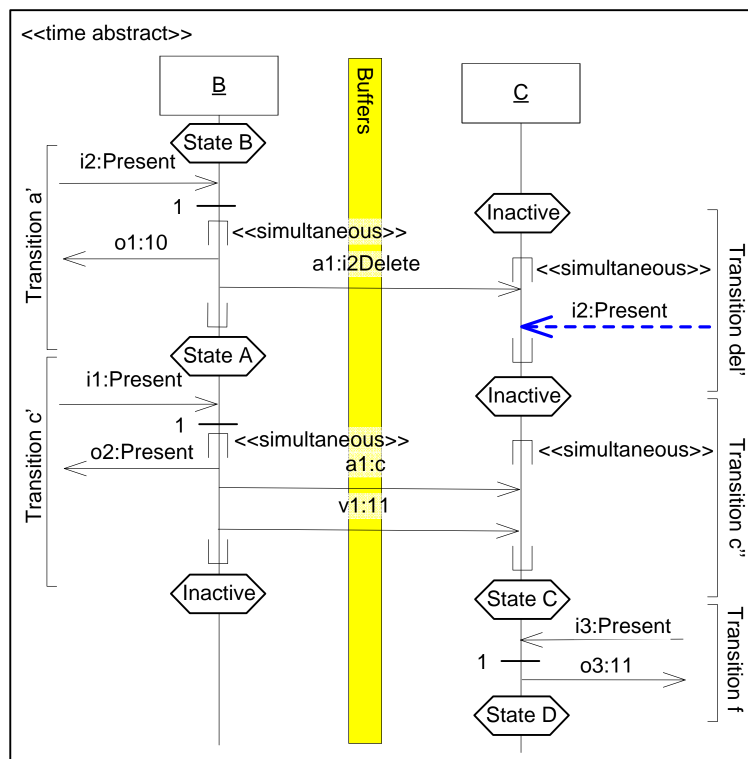
riablenbedingung, die Eingabebedingung, den Ausgabeterm und die Variablenzuweisung der ursprünglichen Transition c, da diese bereits zuvor von der Transition c' der anderen zerteilten Komponente ausgewertet wurde. Die Transition c'' frägt lediglich die Aktivierungsnachricht ab und aktualisiert beim Feuern alle gemeinsam benutzten Variablen, deren Werte von der anderen zerteilten Komponente zusammen mit der Aktivierungsnachricht gesendet wurden.

Die Abbildung 7.10 auf der nächsten Seite zeigt den im Beispiel betrachteten Ablauf der Zustandsfolge $B \rightarrow A \rightarrow C \rightarrow D$ vor und nach Durchführung des Refactorings unter Verwendung der zeitabastrakten AutoFocus Sequenzdiagramme. Aus Übersichtlichkeitsgründen wurden die zwischen den Komponenten B und C vorhandenen Puffer in den Sequenzdiagrammen nicht explizit dargestellt. Darüber hinaus werden nur verarbeitete Eingaben (Abstraktion $\underline{\alpha}_{NI}$) in den Abläufen berücksichtigt. Betrachten wir nun die Unterschiede zwischen beiden Abläufen.

In der Abbildung 7.10a ist der Ablauf vor Durchführung des Refactorings gegeben. Die Abbildung 7.10b zeigt den Ablauf nach Zerteilung der Komponente. Zu Beginn dieses Ablaufs wird beim Feuern der Transition a' eine Nachricht von dem Eingabe-Port $i2$ der Komponente B gelesen und auf Grund der für diesen Port bestehenden eins zu zwei Kommunikationsbeziehung eine Löschnachricht an die inaktive Komponente C gesendet, die daraufhin durch Feuern der Transition del' die bereits in der Komponente B verarbeitete Eingabe auf dem Port $i2$ der Komponente C löscht. Das Löschen der Einga-



(a) Sequenzdiagramm eines Ablaufs vor dem Split Component Refactoring.



(b) Sequenzdiagramm eines Ablaufs nach dem Split Component Refactoring.

Abbildung 7.10.: Verhaltensänderung durch das *Split Component* Refactoring.

be ist ebenfalls als eine Verarbeitung von Eingaben unter der Abstraktion $\underline{\alpha}_{NI}$ sichtbar. Durch die zur kombinierten Betrachtung von Eingabeströmen in eins zu zwei Kommunikationsbeziehungen verwendete *combine12Comm* Funktion wird jedoch diese zeitlich spätere Verarbeitung der Eingabe zu einer bereits zuvor durch die andere Komponente verarbeiteten äquivalenten Eingabe nicht berücksichtigt. In dem Sequenzdiagramm ist daher diese Eingabe durch einen gestrichelten Pfeil dargestellt.

Mit den Transitionen c' und c'' erfolgt die Deaktivierung der Komponente B, die Synchronisierung der lokalen Variablen und die Aktivierung der Komponente C. Die hierfür benötigte Kommunikation ist nach außen nicht sichtbar.

Beide Abläufe in Abbildung 7.10 auf der vorherigen Seite unterscheiden sich in den nach außen sichtbaren Ein- / Ausgabesequenzen unter Anwendung der kombinierten Betrachtung von eins zu zwei Kommunikation lediglich in unterschiedlichen zeitlichen Verzögerungen, der durch den Aktivierungsmechanismus zwischen den Komponenten B und C bedingt ist. Unter der Abstraktion $\underline{\alpha}_{ZA2}$ sind beide Abläufe äquivalent. Darüber hinaus wird nach der betrachteten Ausführung in der am Schluss aktiven Komponente C ein zur ursprünglichen Ausführung der Komponente A äquivalenter Kontrollzustand erreicht. Der Variablenzustand der aktiven Komponente C nach der Ausführung des betrachteten Ablaufs ist ebenfalls äquivalent zum Variablenzustand der Ursprungskomponente A nach der Ausführung des hier betrachteten Ablaufs.[10]

Wird in der einen gerade aktiven zerteilten Komponente eine Transition gefeuert, die die Partitionierungsgrenze überschreitet, dann folgt daraus, dass diese Komponente deaktiviert wird und danach in der anderen Komponente, die bisher deaktiviert war, die Aktivierungstransition gefeuert wird und somit gemeinsam benutzte lokale Variablen synchronisiert werden und in den zum Zielzustand der ursprünglichen nicht zerteilten Transition äquivalenten Zustand der zerteilten Komponente gewechselt wird.

Verarbeitet die momentan aktive Komponente Eingaben, die in eins zu zwei Kommunikationsbeziehungen stehen, dann wird mit jeder Verarbeitung eine entsprechende Löschnachricht über den Aktivierungskanal gesendet. Somit wird sichergestellt, dass vor der Aktivierung der anderen Komponente zunächst diese alle verarbeiteten Eingaben aus ihrem Eingabepuffer löscht. Der Zustand der Eingabepuffer von eins zu zwei Kommunikationsbeziehungen wird somit immer vor der Aktivierung der anderen Komponente synchronisiert.

Da durch den Aktivierungsmechanismus zu jedem Zeitpunkt sichergestellt ist, dass nur eine der beiden zerteilten Komponenten gleichzeitig aktiv ist, können nicht von beiden Komponenten gleichzeitig Ausgaben getätigt werden. Die zur Realisierung von zwei zu eins Kommunikationsbeziehungen verwendete Multiplexing (Mux) Komponente (siehe Abschnitt 6.6.6) stellt genau für diesen Fall, dass keine gleichzeitigen Aus-

[10] Der Variablenzustand der Komponente A kann nach dem Refactoring auf beide Komponenten B und C verteilt sein. Nicht gemeinsam verwendete Variablen, die auf beide Komponenten verteilt sind, können einfach zusammen betrachtet werden. Bei Variablen, die gemeinsam von den Komponenten B und C verwendet werden, wird jeweils die aktuelle Belegung der momentan aktiven Komponente bei der Ermittlung des Variablenzustands berücksichtigt.

gaben auftreten, durch Verwendung der *Immediate* Kommunikationssemantik sicher, dass durch das Zwischenschalten dieser Komponente die Ausgaben nicht verändert und keine zeitlichen Verschiebungen der Ausgaben bewirkt werden.

Aus der vorangehend aufgeführten Argumentation leiten wir ab, dass das Schnittstellenverhalten von durch das *Split Component* Refactoring aufgespaltenen Komponenten unter kombinierter Betrachtung von Eingabeströmen, die in eins zu zwei Kommunikationsbeziehungen stehen, abgesehen von durch den Aktivierungsmechanismus bedingten zeitlichen Verzögerungen gleich dem ursprünglichen Schnittstellenverhalten vor Anwendung des Refactorings ist. Dies gilt für alle zeitasynchronen AutoFocus Modellspezifikationen. Von dieser zeitlichen Verzögerung wird unter der Zeitabstraktion $\underline{\widetilde{\alpha}}_{ZA2}$ abstrahiert. Die Verhaltensäquivalenzeigenschaft, die für zeitrobustes AutoFocus Refactoring gefordert wird (siehe Definition 6.4 auf Seite 129), wird somit durch das *Split Component* Refactoring erfüllt. Das *Split Component* Refactoring ist ein allgemeingültiges Refactoring für zeitasynchrone AutoFocus Modellspezifikationen.

8. Weitere Modelloptimierungen

Nachfolgend sind weitere Optimierungsverfahren aufgeführt, die auf AutoFocus Modellspezifikationen, die nach der klassischen zeitsynchronen als auch nach der neuen zeitasynchronen Semantik interpretiert werden, angewendet werden können. Diese Optimierungsverfahren entsprechen der Definition 4.3 auf Seite 84 von Refactoring und können daher als solche verstanden werden. Im Unterschied zu den vorangehend in dieser Arbeit in den Kapiteln 5 und 7 aufgeführten AutoFocus Refactorings basieren die hier vorgestellten Modelloptimierungen auf klassischen Term-, Graph- und Automatenoptimierungsverfahren, haben das konkrete Ziel, nicht benötigte Modellelemente zu eliminieren und können voll automatisiert auf Spezifikationen angewendet werden.

In Anschnitt 8.1 wird ein Verfahren zum automatisierten Entfernen nicht benutzter AutoFocus Ports, Kanäle und Variablen angesprochen. Das Entfernen von nicht erreichbaren Zuständen und nicht ausführbaren Transitionen wird im Abschnitt 8.2 diskutiert. Das Thema Minimalisierung von Zustandsautomaten wird in dem Abschnitt 8.3 angesprochen und in dem Abschnitt 8.4 wird die Konstruktion eines Produktautomaten, der das Zusammenfassen zweier atomarer Komponenten zu einer atomaren Komponente ermöglicht, skizziert.

8.1. Entfernen unbenutzter Ports, Kanäle und Variablen.

Durch eine statische Analyse werden in der einer atomaren Komponente assoziierten Zustandsmaschine (STD) alle in deren Transitionen verwendeten Bezeichner von Eingabe- und Ausgabe-Ports sowie von lokalen Variablen ermittelt. Alle Ports und lokalen Variablen der atomaren Komponente, deren Bezeichner nicht in den Transitionen der assoziierten Zustandsmaschine vorkommen, werden in dem entsprechenden Systemstrukturdiagramm (SSD) gelöscht. Zusätzlich werden die Kanäle, die mit den entfernten Ports verbunden waren, ebenfalls entfernt. Ist der zu entfernende Kanal auf der gegenüberliegenden Seite nicht direkt mit einem Port der Umgebung oder einem Port einer atomaren Komponente verbunden, handelt es sich also bei diesem Port um einen Zwischen-Port, dann wird dieser Zwischen-Port einschließlich des auf anderer Hierarchieebene wiederum mit ihm verbundenen Kanals gelöscht. Dieser Vorgang wird auf den verschiedenen Hierarchieebenen so lange wiederholt, bis der gelöschte Kanal auf der gegenüberliegenden Seite mit einem Port der Umgebung beziehungsweise einem Port einer atomaren Komponente verbunden ist.

Das hier beschriebene Verfahren kann auf alle atomaren Komponenten einer Modellspezifikation angewendet werden. Nach der Anwendung existieren in der Modellspezifikation keine Ports und Variablen, deren Bezeichner nicht in den assoziierten Zustandsmaschinen verwendet werden. Darüber hinaus enthält die Modellspezifikation nach der Optimierung auch keine Kanäle und Zwischen-Ports mehr, die der Kommunikation mit den nicht verwendeten Ports gedient haben.

8.2. Entfernen nicht erreichbarer Zustände und nicht ausführbarer Transitionen.

Zustandsmaschinen könne Zustände besitzen, die auf Grund der in deren eingehenden Transitionen festgelegten Bedingungen nicht erreichbar sind. Ferner können Zustände existieren, die überhaupt keine eingehende Transition besitzen und nicht Startzustand sind und somit ebenfalls nicht erreicht werden können. Eine Zustandsmaschine kann Transitionen beinhalten, die nie ausführbar sind. Diese nicht erreichbaren Zustände und nicht ausführbaren Transitionen haben keine Auswirkungen auf Verhalten und sollten entfernt werden, um dadurch die Größe der Zustandsmaschine zu reduzieren.

Durch eine dynamische Analyse der Zustandsmaschine werden alle nicht erreichbaren Zustände und nicht ausführbaren Transitionen identifiziert. Ähnlich zur Model Checking Technik [Hol03, CGP00] werden alle möglichen Abläufe der Zustandsmaschine betrachtet und die Zustände und Transitionen identifiziert, die in keinem der Abläufe erreicht beziehungsweise ausgeführt werden.

Es bleibt anzumerken, dass nicht erreichbare Zustände und nicht ausführbare Transitionen, die bewusst von dem Entwickler in die Spezifikation eingefügt wurden, häufig auf einen Spezifikationsfehler hindeuten, da in der Regel der Entwickler Zustände und Transitionen mit der Absicht in die Spezifikation einfügt, ein Verhalten zu beschreiben.

8.3. Minimalisierung von Zustandsmaschinen.

Zu jeder Zustandsmaschine existiert eine verhaltensäquivalente Zustandsmaschine, die eine minimale Anzahl von Zuständen und Transitionen aufweist. In der Automatentheorie wurde das Problem des Findens von minimalen Zustandsmaschinen beziehungsweise minimalen Automaten eingehend behandelt. So lässt sich durch Anwendung des Minimierungsalgorithmus von Hopcroft zu einem beliebigen deterministischen endlichen Automaten ein endlicher Automat äquivalenter Akzeptanzmenge von Wörtern, also gleicher Sprache, konstruieren, der eine minimale Anzahl von Zuständen besitzt [HU94, S. 69 ff].

Auf Basis dieses Minimierungsverfahrens lässt sich auch ein Minimierungsverfahren für AutoFocus Zustandsmaschinen nach zeitsynchroner oder zeitasynchroner Semantik entwickeln. Der Nutzen eines solchen Verfahrens in Hinblick auf das Refactoring

erscheint jedoch fragwürdig. Die Eigenschaft der minimalen Anzahl von Transitionen und Zuständen in einer Zustandsmaschine ist nicht gleichbedeutend mit besserer Lesbarkeit und Verständlichkeit. Es ist vielmehr so, dass eine Transition einer minimalen Zustandsmaschine mehr Verhalten realisiert und dadurch sehr schnell unverständlich wird. Durch die Anwendung eines Minimierungsalgorithmus wird die Struktur der ganzen Zustandsmaschine geändert. Die resultierende Zustandsmaschine ist in vielen Fällen für den Entwickler, der die ursprüngliche Zustandsmaschine per Hand konstruiert hat, nur sehr schwer zu verstehen, da diese völlig anders strukturiert sein kann.

8.4. Berechnung eines Produktautomaten

Existieren in einer AutoFocus Modellspezifikation viele atomare Komponenten, die jeweils nur sehr wenig Funktionalität bieten und sehr stark miteinander vernetzt sind, dann kann diese große Zahl von Abhängigkeiten zwischen Komponenten zu einer sehr schwer verständlichen Modellspezifikation führen. In einem solchen Fall kann überlegt werden, die Anzahl der atomaren Komponenten zu reduzieren, in dem jeweils zwei atomare Komponenten zu einer neuen atomaren Komponente zusammengefasst werden. Die neu entstehende atomare Komponente besitzt die Ein- und Ausgabe-Ports beider zusammenzufassender atomaren Komponenten und muss auf äquivalente Weise mit der Restspezifikation über Kanäle verbunden sein. Kommunikation, die zwischen den beiden ursprünglichen Komponenten verlaufen ist, ist in der zusammengefassten Komponente weiterhin als eine direkte Feedback Kommunikation sichtbar.

Das Verhalten der resultierenden atomaren Komponente wird durch eine Zustandsmaschine festgelegt, deren Verhalten ohne Anwendung von Zeitabstraktionen äquivalent zu dem Verhalten der Zustandsmaschinen der beiden zusammenzufassenden Komponenten in paralleler Ausführung ist. Hierzu lässt sich ein Konstruktionsverfahren angeben, das ähnlich zur Konstruktion eines Produktautomaten zweier endlicher Automaten [HU94, S. 63] ist.

9. Entwicklungsprozess und Werkzeuge

Die in dieser Arbeit vorgestellte Modell-Refactoring-Technik dient in erster Linie der Verbesserung der Qualität von Entwurfsmodellen in der Softwareentwicklung und lässt sich als fester Bestandteil in einen Entwicklungsprozess integrieren. In Abschnitt 9.1 wird das Modell-Refactoring in einen iterativen Entwicklungsprozess eingeordnet. Zur Bewertung der Qualität von Entwurfsmodellen lassen sich Metriken einsetzen, auf deren Basis die Auswahl geeigneter Refactorings erfolgen kann (Abschnitt 9.2). Für nicht allgemeingültige Refactorings muss der Entwickler auf Basis der veränderten Modellspezifikation die Verhaltensinvarianzeigenschaft nachweisen. Hierzu lassen sich die in Abschnitt 9.3 aufgeführten zum Teil automatisierten Techniken einsetzen. Der Abschnitt 9.4 zeigt die Anforderungen an eine umfassende Werkzeugunterstützung für das Modell-Refactoring auf.

9.1. Modell-Refactoring im Entwicklungsprozess

Nach Fowler wird die Refactoring-Technik zur Verbesserung der Qualität eines Softwareprogramms in Hinblick auf Lesbarkeit und Verständlichkeit vor der Durchführung einer Erweiterung des Programms eingesetzt, um diese Erweiterung zu vereinfachen und Fehler in der Erweiterung zu vermeiden [FBB+99]. Wir adaptieren diese Idee auf die Modell-Refactoring-Technik.

In Abschnitt 9.1.1 wird ein iterativer Entwicklungsprozess unter Integration der Modell-Refactoring-Technik definiert. Der Prozess der Anwendung eines einzelnen Refactorings ist in Abschnitt 9.1.2 festgelegt. In dem Abschnitt 9.1.3 wird schließlich der Einsatz von Modell-Refactoring innerhalb inkrementeller Entwicklungsprozesse diskutiert.

9.1.1. Iterativer modellbasierter Entwicklungsprozess

Wir gehen von einem vollständig modellbasierten Entwicklungsprozess aus, das heißt bei allen in der Entwicklung durch den Entwickler bearbeiteten Artefakten handelt es sich um Softwaremodelle. Die Aufgabe der Programmierung des eigentlichen Softwareprogramms wird in einem solchen Entwicklungsprozess voll automatisch durch einen Code Generator durchgeführt. Die von dem Entwickler bereitgestellten Entwurfsmodelle werden durch diesen Code Generator in eine Zielprogrammiersprache übersetzt. Diese Art eines modellbasierten Entwicklungsprozesses wird von der *Model*

Driven Architecture (MDA) [OMG03a] propagiert. Im Bereich der Softwareentwicklung für Reaktive Systeme ist die Codegenerierungstechnik bereits heute in vielen CASE Werkzeugen integriert und wird sich in absehbarer Zeit auch in der Praxis durchsetzen. Ein solcher modellbasierter Entwicklungsprozess ist die ideale Einsatzumgebung für die Modell-Refactoring-Technik, da alle Veränderungen durch den Entwickler auf der Modellebene vorgenommen werden und sich jede Modelländerung wiederum automatisch auf eine Änderung der Implementierung auswirkt. Somit werden auch Änderungen, die durch Modell-Refactoring bewirkt sind, automatisch in die Implementierung übernommen.

Alternativ kann das Modell-Refactoring natürlich auch in herkömmlichen Entwicklungsprozessen, die sowohl Änderungen auf der Ebene der Modelle als auch Änderungen auf der Ebene des Programmcodes durch den Entwickler vorsehen, eingesetzt werden. In einem solchen Szenario ist jedoch darauf zu achten, dass Änderungen auf einer der beiden Ebenen jeweils auch äquivalente Änderungen der anderen Ebene nach sich ziehen. Die gegenseitige Synchronisierung der Modell- und Codeebene wird als *Round Trip Engineering* bezeichnet.

In der Abbildung 9.1 auf der nächsten Seite ist ein iterativer modellbasierter Entwicklungsprozess dargestellt, der die Modell-Refactoring-Technik integriert. Als Darstellungsform ist ein informelles UML Aktivitätsdiagramm gewählt, wobei die *Swimlanes* statt bestimmter Rollen die einzelnen Phasen des Entwicklungsprozesses repräsentieren. Der skizzierte Entwicklungsprozess startet mit der Anforderungsanalyse für eine erste Version der zu entwickelnden Software. Nach Abschluss der Anforderungsanalyse existiert zunächst noch kein Entwurfsmodell, es wird also ein neues Entwurfsmodell entwickelt, das die Anforderungen an die erste Version der Software erfüllen soll.

In einem nächsten Schritt wird die Konformität zwischen der Anforderungsspezifikation und dem entwickelten Entwurfsmodell überprüft. Es wird überprüft, ob das Entwurfsmodell die festgelegten Anforderungen erfüllt. Für diese Überprüfung lassen sich eine Vielzahl von verschiedenen Techniken einsetzen. Durch *Requirements Tracing* [SPW04, SFGP05a] lassen sich die Modellelemente des Entwurfsmodells identifizieren, die bestimmte Anforderungen umsetzen. Durch das Testen der Entwurfsspezifikation in einem Simulator, ähnlich dem Testen einer Implementierung, lassen sich Spezifikationsfehler aufdecken [BJK+05, Pre03]. Mit Hilfe von Verifikationstechniken kann bewiesen werden, dass das Modell bestimmte Eigenschaften in Hinblick auf die Anforderungen erfüllt (siehe Abschnitt 9.3).

Wird bei dieser Überprüfung festgestellt, dass die Anforderungsspezifikation und das Entwurfsmodell nicht konform zueinander sind, dann kann dies an Fehlern in der Anforderungsspezifikation als auch an Fehlern in dem Entwurfsmodell liegen. Zur Beseitigung dieser Fehler wird in diesem Fall der Prozess mit der Aktivität der Anforderungsanalyse fortgesetzt. Wurde die Konformität zwischen Anforderungen und Entwurf nachgewiesen, dann wird durch Codegenerierung automatisch eine Implementierung des zu entwickelnden Systems erzeugt.

Schließlich kann entschieden werden, ob das System um neue Funktionalitäten erwei-

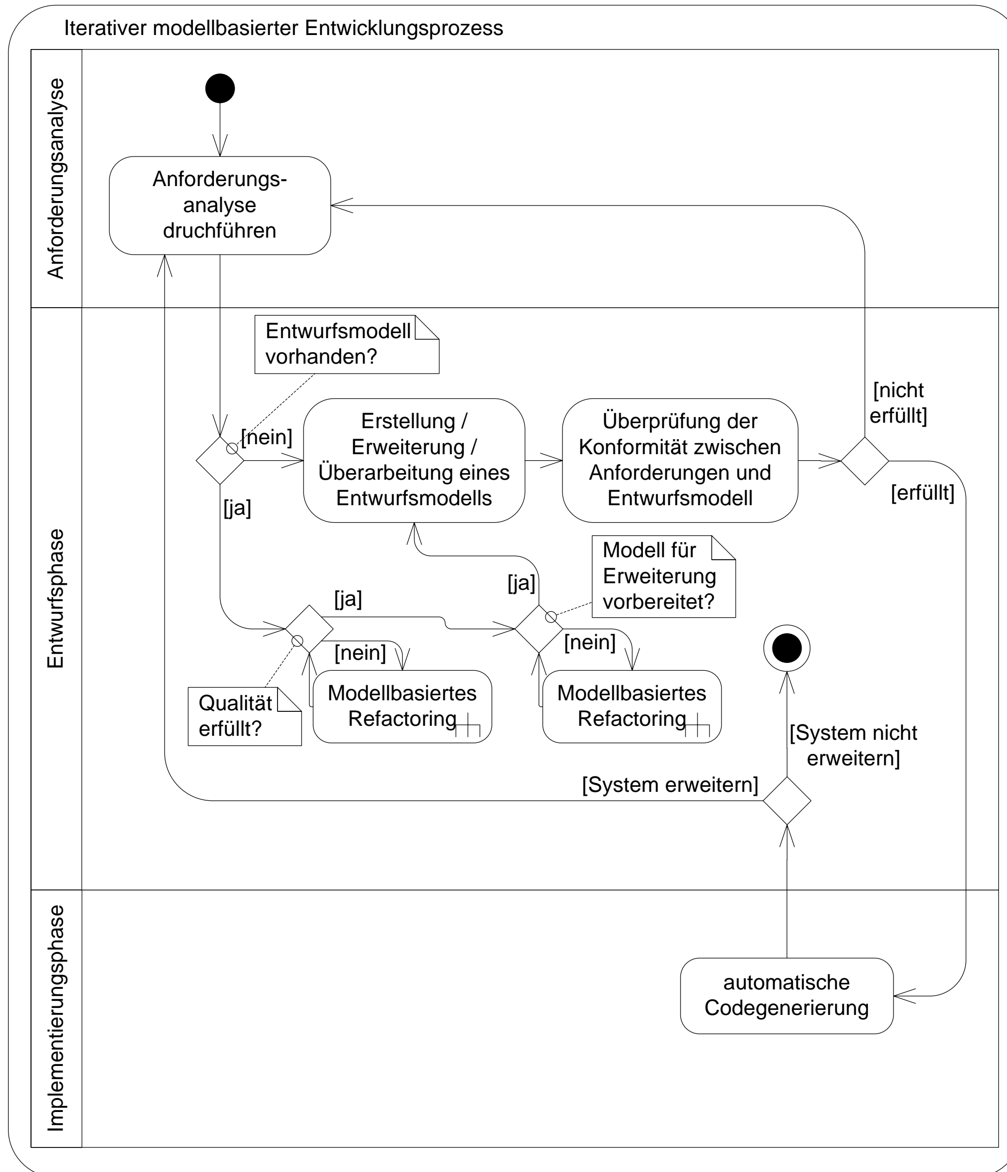


Abbildung 9.1.: Modellbasierter iterativer Entwicklungsprozess.

tert werden soll. Ist dies der Fall, dann wird der Prozess mit einer Anforderungsanalyse der zweiten Version des zu entwickelnden Systems fortgesetzt. Da jetzt bereits ein Entwurfsmodell der ersten Version existiert, kommt nun vor der Erweiterung des Entwurfsmodells um neue Funktionalität die Modell-Refactoring-Technik zum Tragen. Durch wiederholte Anwendung von Modell-Refactorings wird die Qualität der Modellspezifikation in Hinblick auf Verständlichkeit, Lesbarkeit und Wartbarkeit schrittweise verbessert, bis das Modell den festgelegten Qualitätsanforderungen genügt.

Nach Fowler wird das Refactoring ausschließlich zum Zweck der Qualitätssteigerung einsetzt. Wir möchten an dieser Stelle das Anwendungsgebiet von Modell-Refactoring auch auf die gezielte Restrukturierung des Modells zur Vorbereitung einer geplanten Erweiterung des Entwurfsmodells ausdehnen. So ist beispielsweise in dieser Arbeit in Abschnitt 7.2 das *Insert Intermediate State* Refactoring angegeben, das einen nicht benötigten Zwischenzustand in eine Zustandsmaschine einfügt, der später in der Erweiterung zur Realisierung neuer Funktionalität genutzt werden kann. Das gezielte Vorbereiten der Struktur des Modells erscheint auf Grund folgender Argumentation sinnvoll. Die Vorbereitungsschritte, die als Refactorings ausgeführt werden, verändern das Verhalten zunächst nicht. Es können durch diese Schritte also keine neuen Spezifikationsfehler in das Modell integriert werden. Durch die Vorbereitung der Modellstruktur lässt sich der Umfang der für die Erweiterung der Funktionalität erforderlichen verhaltensändernden Modelländerungen im Vergleich zu der nicht speziell vorbereiteten Modellstruktur reduzieren. Durch den reduzierten Umfang von verhaltensändernden Modelländerungen wird das Risiko von Spezifikationsfehlern, die durch die Änderungen hervorgerufen werden, verkleinert.

Nach der Durchführung von Refactoring mit dem Ziel der Qualitätssteigerung wird in dem vorgeschlagenen Prozess das Modell-Refactoring mit dem Ziel der Vorbereitung der Modellstruktur in Hinblick auf die geplante Erweiterung wiederholt eingesetzt. Ist die Modellstruktur für die Erweiterung vorbereitet, dann wird die eigentliche Erweiterung der Funktionalität des Entwurfsmodells entsprechend den in der Anforderungsanalyse festgelegten neuen Anforderungen durchgeführt. Der Entwicklungsprozess wird ab dieser Stelle iterativ fortgesetzt, bis keine Erweiterungen beziehungsweise neuen Versionen des Softwaresystems mehr entwickelt werden.

9.1.2. Prozess der Anwendung eines Modell-Refactorings

Der Prozess der Anwendung eines einzelnen Modell-Refactorings ist in Abbildung 9.2 auf der nächsten Seite als UML Aktivitätsdiagramm gezeigt. Zunächst identifiziert der Entwickler innerhalb der Modellspezifikation eine Teilspezifikation, die verändert werden soll und wählt aus dem Katalog der ihm zur Verfügung stehenden Refactorings ein geeignetes Refactoring aus. Die Auswahl erfolgt je nach dem Ziel, mit dem das Refactoring angewendet wird, unter verschiedenen Gesichtspunkten. Wird das Refactoring mit dem Ziel der Qualitätssteigerung betrieben, dann erfolgt die Auswahl des Refactorings zusammen mit dessen Parametern in Abhängigkeit der durch das Refactoring

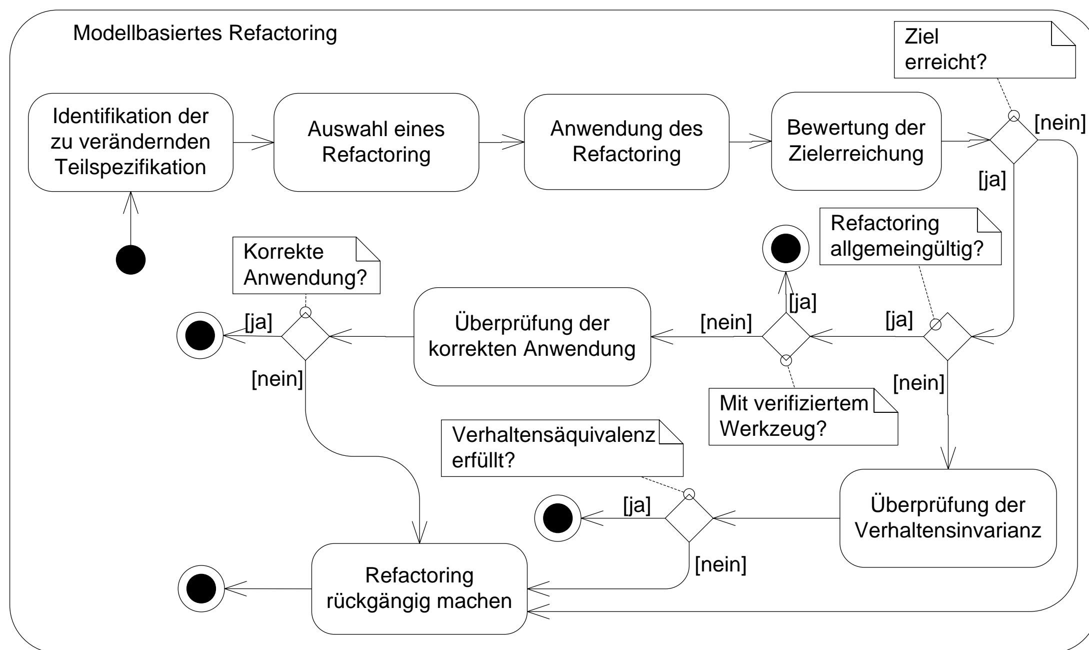


Abbildung 9.2.: Anwendung eines Modell-Refactorings.

zu erwartenden Qualitätssteigerung der Modellspezifikation. An dieser Stelle können Metriken zur Qualitätsbewertung von Entwurfsmodellen eingesetzt werden (siehe Abschnitt 9.2). Wird das Refactoring mit dem Ziel der Vorbereitung der Modellstruktur für nachfolgende Erweiterungen durchgeführt, dann erfolgt die Auswahl in Abhängigkeit der für die Erweiterungen erforderlichen Modelländerungen. Bei der Auswahl von Refactorings zusammen mit deren Parametern ist darauf zu achten, dass die Vorbedingungen der Refactoring-Operationen erfüllt sind.

Im nächsten Schritt wird das Refactoring auf eine Teilspezifikation des Entwurfsmodells mit entsprechenden Parametern angewendet. Die Anwendung kann manuell durch den Entwickler oder automatisiert durch ein Werkzeug erfolgen.

Nach der Anwendung des Modell-Refactorings wird im nächsten Schritt bewertet, ob die Modelländerung zu dem gewünschten Ziel geführt hat beziehungsweise zu diesem beiträgt. Wird das Refactoring mit dem Ziel der Qualitätssteigerung betrieben, dann wird die Qualität des Modells nach dem Refactoring mit der Qualität des Modells vor dem Refactoring verglichen. Wird bei dieser Überprüfung festgestellt, dass das angewendete Refactoring nicht zielführend war, das heißt im Fall der Betrachtung der Qualität, diese nicht verbessert wurde und durch das angewendete Refactoring auch nicht andere qualitätsverbessernde Refactorings vorbereitet wurden, dann wird das Refactoring rückgängig gemacht und das ursprüngliche Entwurfsmodell wiederhergestellt. Wird hingegen festgestellt, dass die für das Refactoring gesetzten Ziele erreicht wur-

den, dann wird betrachtet, ob das angewendete Refactoring die Verhaltensinvarianzeigenschaft erfüllt.

Handelt es sich bei dem angewendeten Refactoring um ein allgemeingültiges Refactoring (Definition 4.4 auf Seite 85) und wurde dieses automatisiert durch ein verifiziertes Refactoring Werkzeug[1] durchgeführt, dann ist das sich aus dem Refactoring ergebende Entwurfsmodell, gegebenenfalls unter einer gewählten Abstraktion, verhaltensäquivalent zu dem Modell vor Anwendung des Refactorings. In diesem Fall ist der Prozess der Anwendung des Refactorings beendet.

Wurde kein verifiziertes Werkzeug zur Durchführung des allgemeingültigen Refactorings eingesetzt, dann muss zur Sicherstellung der Verhaltensinvarianzeigenschaft die korrekte Anwendung des Refactorings auf der veränderten Teilspezifikation überprüft werden. Ist die korrekte Anwendung nachgewiesen, dann ist die Verhaltensinvarianzeigenschaft erfüllt und die Durchführung des Refactorings ist beendet. Wird hingegen festgestellt, dass das Refactoring falsch angewendet wurde, dann wird das gesamte Refactoring rückgängig gemacht und das ursprüngliche Entwurfsmodell wiederhergestellt.

Handelt es sich bei dem angewendeten Refactoring um ein nicht allgemeingültiges Refactoring, dann muss nach der Anwendung dieses Refactorings die Verhaltensäquivalenz des Modells vor dem Refactoring und des Modells nach dem Refactoring gezeigt werden. Der Verhaltensäquivalenznachweis kann je nach Eigenschaften der Kompositionalität der Semantik der verwendeten Modellierungssprache und den zur Beobachtung von Verhalten eingesetzten Abstraktionen entweder auf der Ebene des Schnittstellenverhaltens der veränderten Teilspezifikation erfolgen, oder es muss das Schnittstellenverhalten des Gesamtsystems betrachtet werden. Zum Nachweis der Verhaltensäquivalenz können die in Abschnitt 9.3 aufgeführten Techniken eingesetzt werden.

Wird bei dem Versuch des Nachweises der Verhaltensäquivalenz festgestellt, dass sich das beobachtbare Verhalten der Modellspezifikation durch das Refactoring verändert hat, dann muss das Refactoring rückgängig gemacht und die ursprüngliche Modellspezifikation wiederhergestellt werden. Wird hingegen die Verhaltensäquivalenz nachgewiesen, bleiben die Änderungen durch das Refactoring erhalten und die Durchführung des Refactorings ist abgeschlossen.

9.1.3. Modell-Refactoring in einem inkrementellen Entwicklungsprozess

Inkrementelle Entwicklungsprozesse, wie beispielsweise Cleanroom [PTLP99], propagieren die Entwicklung eines Systems in vielen kleinen Entwicklungsschritten. Diese Entwicklungsschritte werden Inkremente genannt und nach der Durchführung jedes einzelnen Inkrements soll ein ausführbarer Prototyp des Systems zur Verfügung stehen.

[1] Unter verifiziertem Refactoring Werkzeug verstehen wir in diesem Zusammenhang ein Werkzeug, für das die Korrektheit der Anwendung der durch das Programm durchgeführten allgemeingültigen Refactorings nachgewiesen wurde.

Die in dieser Arbeit vorgestellten Modell-Refactorings lassen sich innerhalb eines inkrementellen Entwicklungsprozesses neben herkömmlichen verhaltenserweiternden Inkrementen als eigenständige, wenn auch sehr kleine, Inkremente auffassen, die gezielt zur Veränderung der Struktur in Hinblick auf die spätere inkrementelle Erweiterung eingesetzt werden. Durch die Verhaltensinvarianzeigenschaft der Refactorings ist automatisch sichergestellt, dass ein ausführbares Modell, auf das das Refactoring angewendet wird, weiterhin ausführbar ist. Somit ist sichergestellt, dass sich nach Anwendung eines Refactorings als Entwicklungsinkrement ein ausführbarer Prototyp des Systems, gegebenenfalls durch automatische Codegenerierung, erzeugen lässt.

9.2. Metriken zur Bewertung der Qualität von Entwurfsmodellen

Wird das Modell-Refactoring mit dem Ziel der Qualitätssteigerung des Entwurfsmodells eingesetzt, müssen zunächst Teile des Modells identifiziert werden, die schlechte Qualität aufweisen. Darüber hinaus müssen Refactorings ausgewählt werden, die diese schlechte Qualität potentiell verbessern. Nach der Durchführung des Refactorings sollte kontrolliert werden, ob sich die Qualität tatsächlich verbessert hat. Diese sehr komplexen Aufgaben müssen von dem Entwickler bewältigt werden.

Auf Basis von Heuristiken können wir jedoch Metriken zur Bewertung der Qualität von Entwurfsmodellen definieren, die dem Entwickler Anhaltspunkte zur Einschätzung der Qualität bieten. Diese Metriken lassen sich voll automatisiert auf Entwurfsmodelle anwenden. In [Ben03] sind Metriken zur Bewertung der Qualität von AutoFocus Entwurfsmodellen in Hinblick auf Lesbarkeit, Verständlichkeit und Wartbarkeit angegeben. Die dort festgelegten Metriken übertragen die Idee der seit längerem in der Praxis eingesetzten Codemetriken zur Bewertung der Qualität von Softwareprogrammen, wie beispielsweise die Chidamber und Kemerer Metriken [CK94], auf die Ebene der Entwurfsmodelle. Die definierten AutoFocus Qualitätsmetriken umfassen sowohl die Bewertung einzelner Diagramme als auch die Bewertung gesamter Modelle, die aus diesen einzelnen Diagrammen zusammengesetzt sind. Neben der rein quantitativen Analyse der in den Diagrammen vorkommenden Modellelemente wird insbesondere auch der Kopplungsgrad zwischen den Modellelementen zur Bewertung der Qualität herangezogen.

Durch eine metrikbasierte Qualitätsanalyse können automatisiert Teile der Modellspezifikation identifiziert werden, die potentielle Qualitätsdefizite aufweisen. Diese Ergebnisse liefern dem Entwickler Hinweise, die in seine Qualitätseinschätzung des Entwurfsmodells einfließen. Auf Basis der Ergebnisse einer metrikbasierten Qualitätsanalyse lassen sich auch automatisiert geeignete Refactorings, die die potentiellen Qualitätsdefizite beseitigen sollen, auswählen. Der Entwickler kann auf Basis seiner Qualitätseinschätzung des Entwurfsmodells entscheiden, ob durch die vorgeschlagenen Refactorings tatsächliche Qualitätssteigerungen zu erwarten und diese Refactorings an-

zuwenden sind.

9.3. Nachweis der Verhaltensinvarianzeigenschaft

Die Verhaltensinvarianz ist die zentrale Eigenschaft des Refactoring. Nach Fowler wird die Verhaltensinvarianz der Programme vor und nach dem Refactoring durch ausgedehntes Testen sichergestellt [FBB+99]. Für das Modell-Refactoring sind neben dem Testen von Modellen auch weitere Techniken zum Nachweis der Verhaltensinvarianz möglich.

Grundsätzlich müssen wir zwei Arten des Nachweises unterscheiden. Handelt es sich bei einem betrachteten Refactoring um ein allgemeingültiges Refactoring (Definition 4.4 auf Seite 85), dann kann der Nachweis der Verhaltensinvarianz allgemein auf Basis der Semantik der Modellierungssprache und der Refactoring-Operation durchgeführt werden. Durch einen solchen Nachweis wird allgemein für alle in der Modellierungssprache ausdrückbaren Spezifikationen gezeigt, dass sich, falls die Vorbedingungen des Refactorings erfüllt sind, das Verhalten der Modellspezifikation gegebenenfalls unter Abstraktion durch die Anwendung des Refactorings nicht ändert. Die in dieser Arbeit vorgestellten Refactorings sind mit Ausnahme des *Remove Inactive States* Refactoring alle allgemeingültig. Der Nachweis der Erfüllung der Verhaltensinvarianz wurde durch eine Argumentation auf Basis der Eigenschaften der Semantik der Modellierungssprache geführt. Liegt, wie in dieser Arbeit, eine mathematische Formalisierung des Begriffs des beobachtbaren Verhaltens vor, dann kann die Verhaltensinvarianzeigenschaft durch mathematische Beweise belegt werden. Für die Durchführung solcher Beweise lassen sich Theorembeweissysteme einsetzen. Ein allgemeingültiges Refactoring garantiert, eine korrekte Anwendung des Refactorings vorausgesetzt, die Verhaltensinvarianzeigenschaft. Wird ein allgemeingültiges Refactoring auf eine Modellspezifikation angewendet, dann muss kein Verhaltensäquivalenznachweis auf Basis der Modellspezifikation durchgeführt werden.

Im Gegensatz dazu ist ein Verhaltensäquivalenznachweis auf Basis der veränderten Modellspezifikation nach jeder Anwendung eines nicht allgemeingültigen Refactorings erforderlich. Abhängig von der Kompositionalität der Semantik der eingesetzten Modellierungssprache und der zur Beobachtung von Verhalten verwendeten Abstraktionen muss für den Verhaltensäquivalenznachweis das Verhalten der durch das Refactoring veränderten Teilspezifikation oder das Verhalten der Gesamtspezifikation betrachtet werden. Für den Verhaltensäquivalenznachweis stehen uns eine Reihe von Techniken zur Verfügung. Wie von Fowler vorgeschlagen, kann auch das Testen auf Modellebene [BJK+05, Pre03] vor und nach Durchführung des Refactorings als Methode zum Nachweis der Verhaltensäquivalenz eingesetzt werden. Da durch das Testen in der Regel nicht das vollständige Verhalten überprüft werden kann, wird nur für den durch die Tests abgedeckten Teil des Verhaltens die Invarianz nachgewiesen.

Die Model Checking Technik [CGP00] ermöglicht den automatisierten Nachweis von

Eigenschaften von Systemen mit endlichem Zustandsraum. Es existieren heute zwei Arten von Model Checking Algorithmen. Bei *Explicit State Model Checking* wird der Zustandsraum des betrachteten Systems explizit während der Überprüfung der Eigenschaften aufgebaut. Das bekannteste Model Checking Werkzeug dieser Kategorie ist Spin [Hol97, Hol03]. MIC [WHP00, Wiß99] ist ein Framework zur Realisierung von expliziten Model Checking Werkzeugen unter Verwendung verschiedener Such- und Speicherstrategien. Bei symbolischem Model Checking wird der Zustandsraum des Systems in einem *Binary Decision Diagram* (BDD) dargestellt. Das bekannteste Werkzeug dieser Kategorie ist SMV [McM92]. Die Model Checking Technik ermöglicht es, eine feste Menge an Eigenschaften des Modells vor und nach dem Refactoring zu überprüfen. Somit kann sichergestellt werden, dass die Eigenschaften des Modells, die vor dem Refactoring erfüllt waren, auch nach dem Refactoring gelten. Da die festgelegten Eigenschaften in der Regel nicht das gesamte Verhalten charakterisieren, erfolgt durch dieses Vorgehen nicht der Nachweis der vollständigen Verhaltensinvarianz, sondern nur die Invarianz bezüglich der für das Model Checking festgelegten Eigenschaften. Durch eine Modifikation der Model Checking Algorithmen ist es jedoch denkbar, einen speziellen Model Checker zum vollständigen Verhaltensinvarianznachweis zu konstruieren. Statt der Anwendung des Model Checkers auf eine Spezifikation und eine Menge von Eigenschaften kann auch ein Model Checker entwickelt werden, der die Verhalten zweier Spezifikationen ähnlich dem Prinzip der Bisimulation in den Formalen Sprachen miteinander vergleicht.

Wie beim Nachweis der Verhaltensinvarianzeigenschaft eines allgemeingültigen Refactorings können auch beim Verhaltensäquivalenznachweis auf Modellspezifikationsebene mathematische Beweise unter Zuhilfenahme von Theorembeweissystemen geführt werden.

9.4. Werkzeugunterstützung

Um Fehler bei der Anwendung von Refactorings zu vermeiden und die Effizienz der Anwendung von Refactorings durch deren schnellere Durchführbarkeit zu steigern, empfiehlt sich eine werkzeugbasierte Unterstützung für das Refactoring. Der *Smalltalk Refactoring Browser*[2] [RBJ97] von Roberts und Brant war das erste existierende Refactoring Werkzeug. Es unterstützt das Refactoring von Smalltalk Programmcode. Im Bereich der Code Refactorings existieren heute eine Vielzahl von Refactoring Werkzeugen, die automatisiert Refactorings anwenden. So unterstützt beispielsweise RefactorIt [3] das Refactoring von Java Code. Heutige moderne Entwicklungsumgebungen für die Softwareprogrammierung, wie beispielsweise Eclipse[4] von IBM und JBuilder[5] von Borland, haben bereits elementare Refactorings fest integriert. Die heutige Werkzeugun-

[2] Web-Seite `http://st-www.cs.uiuc.edu/users/brant/Refactory/`.
[3] Web-Seite `http://www.refactorit.com`.
[4] Web-Seite `http://www.eclipse.org`.
[5] Web-Seite `http://www.borland.com/de/products/jbuilder/index.html`.

terstützung beschränkt sich jedoch rein auf die Anwendung von durch den Entwickler ausgewählten in den Programmen fest vordefinierten Refactorings. Die Unterstützung bei der Auswahl von Refactorings, die Unterstützung beim Nachweis der Verhaltensinvarianz der Refactorings sowie die Möglichkeit der Definition domänenspezifischer Refactorings durch den Entwickler wird in den heute zur Verfügung stehenden Code-Refactoring-Programmen nicht unterstützt.

In den heute in der Praxis eingesetzten CASE Werkzeugen, die zur Spezifikation von Entwurfsmodellen benutzt werden, wird ein Modell-Refactoring, wie es in dieser Arbeit vorgestellt wird, nicht unterstützt. In [SHH$^+$03] wird aufgezeigt, dass die in der Studie betrachteten CASE Werkzeuge für Reaktive Systeme mangelhafte Unterstützung für das Verändern bestehender Modelle bieten. In den heute in der Produktion eingesetzten CASE Werkzeugen wird lediglich eine zum Teil sehr rudimentäre *Cut & Paste* Funktionalität unterstützt.

Prototypische CASE Werkzeuge im Forschungsumfeld bieten heute bereits mehr Unterstützung für das Verändern von Modellen an. So ist in AutoFocus das Modelltransformationsframework Aqua und die Transformationsdefinitionssprache *Operation Description Language* (ODL) [Sch01] integriert, die die Definition und die automatisierte Durchführung von Modelltransformationen ermöglichen. Der Einsatz von Aqua und ODL zur Transformation von AutoFocus Modellen ist in [SBHW05] beschrieben. Modell-Refactorings können somit in AutoFocus definiert und automatisiert ausgeführt werden. In dem Werkzeugprototyp *Refactoring Browser for UML* der Gentleware AG [BSF02] wurde ein rudimentäres Refactoring von *StateCharts* und Aktivitätsdiagrammen realisiert.

Nachfolgend wird skizziert, welche Funktionalität eine umfassende Werkzeugunterstützung für das Modell-Refactoring besitzen sollte. Das Werkzeug soll eine umfassende Bibliothek von grundlegenden allgemein verwendbaren Refactorings für die im Werkzeug unterstützte Modellierungssprache bereitstellen. Diese Refactorings sollen möglichst die Eigenschaft der Allgemeingültigkeit (Definition 4.4 auf Seite 85) erfüllen, um automatisch die Verhaltensinvarianzeigenschaft zu gewährleisten. Je nach der Anwendungsdomäne der zu entwickelnden Software können sehr unterschiedliche Refactoring-Operationen sinnvoll sein. Ein Werkzeug soll benutzerdefinierte Refactorings unterstützten, um dem Entwickler die Möglichkeit zu bieten, das Refactoring optimal an seine Anwendungsdomäne anzupassen. Hierzu muss das Werkzeug eine Sprache zur Definition von Refactorings bereitstellen. Definiert der Entwickler selbst allgemeingültige Refactorings, dann muss die Verhaltensinvarianzeigenschaft dieser Refactorings auf Basis der Semantik der Modellierungssprache nachgewiesen werden (vergleiche Abschnitt 9.3). Es ist denkbar, diesen Nachweis der Verhaltensinvarianz automatisiert durch das Werkzeug durchzuführen. Grundsätzlich sollte das CASE Werkzeug in Hinblick auf die Korrektheit der Anwendung von Refactorings verifiziert werden, um die Verhaltensinvarianzeigenschaft in der Anwendung von Refactorings sicherstellen zu können.

Bevor ein Refactoring angewendet werden kann, muss zunächst der Teil der Spezifika-

tion identifiziert werden, der verändert werden soll und ein passendes Refactoring für die Veränderung ausgewählt werden. Wie in Abschnitt 9.2 gezeigt, können Entwurfsmetriken den Entwickler bei der Identifikation von Qualitätsdefiziten und der Auswahl geeigneter Refactorings zur Verbesserung der Qualität unterstützen. Es sollte eine allgemeine Qualitätsmetrik für die Modellierungssprache des CASE Werkzeugs existieren, die von dem Werkzeug automatisiert auf die Modellspezifikationen angewendet werden kann. Da in unterschiedlichen Anwendungsdomänen sehr unterschiedliche Qualitätsmaßstäbe existieren, sollte das CASE Werkzeug eine Sprache zur Definition von Qualitätsmetriken bereitstellen, um dem Entwickler die Möglichkeit der Festlegung eigener Qualitätsmetriken zu ermöglichen. Der Entwickler soll von dem Werkzeug über die Ergebnisse der Qualitätsbewertung umfassend informiert werden. Das Werkzeug soll auf Basis einer Qualitätsverbesserungsvorhersage der einzelnen Refactorings die zur Verbesserung der Qualität potentiell geeigneten Refactorings identifizieren und diese dem Entwickler vorschlagen. Letztendlich sollte der Entwickler selbst entscheiden können, wie er die Qualität der Modellspezifikation einschätzt und welche Refactorings er anwenden möchte. Da die Qualitätsbewertung und die Qualitätsverbesserung von Softwaremodellen sehr komplexe Aufgaben darstellen, erscheint es heute und auch in der nächsten Zukunft nicht sinnvoll, diese automatisiert durch Computerprogramme durchführen zu lassen.

Der Entwickler wählt aus den Vorschlägen des Werkzeuges ein passendes Refactoring zusammen mit einem bestimmten identifizierten Teil der Modellspezifikation aus beziehungsweise gibt selbst ein anzuwendendes Refactoring mit entsprechenden Parametern vor. Die Durchführung des eigentlichen Refactorings sollte voll automatisiert erfolgen. Dem Entwickler sollten hierbei die automatisiert durchgeführten Veränderungen des Modells visualisiert werden, um sicherzustellen, dass dieser das veränderte Modell versteht und dieses weiter entwickeln kann.

Wurde ein nicht allgemeingültiges Refactoring angewendet, dann muss auf Basis der Modellspezifikation ein Verhaltensäquivalenznachweis der Modelle vor und nach dem Refactoring geführt werden (siehe Abschnitt 9.1.2). Eine umfassende Werkzeugunterstützung sollte den Entwickler bei der Durchführung dieses Nachweises unter Verwendung der in Abschnitt 9.3 aufgeführten Techniken unterstützen.

10. Schlussfolgerungen

Zu Beginn der Arbeit wurde in der Einleitung ein Fall beschrieben, bei dem ein Kind auf Grund eines Softwarefehlers in dem Steuergerät eines Airbags ums Leben gekommen ist. Die in dieser Arbeit vorgestellte Modell-Refactoring-Technik hätte vor der Erweiterung der Software des Steuergeräts eingesetzt werden können, um die Qualität des Softwareentwurfs zu verbessern. Vielleicht wäre auf Grund der besseren Verständlichkeit des Entwurfs der Fehler nicht gemacht beziehungsweise rechtzeitig erkannt worden.

In dem Anschnitt 10.1 wird das Fazit dieser Arbeit in Hinblick auf die Softwareentwicklung Reaktiver Systeme gezogen. Die Übertragbarkeit der AutoFocus Refactorings auf andere Modellierungssprachen Reaktiver Systeme wird in Abschnitt 10.2 betrachtet. Mögliche zukünftige Arbeiten im Umfeld des Modell-Refactorings werden in Abschnitt 10.3 skizziert. Abschließend wird in dem Abschnitt 10.4 eine Vision der Softwareentwicklung in der Zukunft aufgezeigt, zu deren Verwirklichung das Modell-Refactoring einen Beitrag leisten kann.

10.1. Bedeutung des Modell-Refactorings für die Entwicklung Reaktiver Systeme

In der Arbeit wurde gezeigt, dass sich für Modellierungssprachen Modell-Refactorings definieren lassen, die es ermöglichen weitgehende Veränderungen von Entwurfsmodellen, bis hin zu komplexen Architekturveränderungen, vorzunehmen. Diese Veränderungen können unter verschiedenen Zielsetzungen erfolgen. So lässt sich das Modell-Refactoring entsprechend dem klassischen Code-Refactoring mit dem Ziel der Qualitätssteigerung in Hinblick auf Verständlichkeit, Lesbarkeit und Wartbarkeit der Modelle durchführen. Darüber hinaus sind auch andere Zielsetzungen wie beispielsweise das gezielte Vorbereiten einer Architektur in Hinblick auf eine geplante Erweiterung oder ein Modell-Refactoring zur Performanz- und Resourcenoptimierung des modellierten Systems denkbar. Durch das Modell-Refactoring existiert somit eine sehr weitreichende Restrukturierungstechnik, die sich durch eine umfassende Werkzeugunterstützung sehr effizient einsetzen lässt.

Auf Grund der zunehmenden Etablierung von Modellen in der Softwareentwicklung und einer Tendenz zur automatischen Generierung von Programmcode aus Modellen ist in dem Anwendungsbereich der Reaktiven Systeme davon auszugehen, dass das Modell-Refactoring das Code-Refactoring in diesem Bereich in Zukunft ersetzen wird.

Aber auch in den anderen Bereichen der Softwareentwicklung wird in absehbarer Zeit auf Grund des vermehrten Einsatzes von Modellen eine Nachfrage nach Refactorings für diese Modelle entstehen.

Die zentrale Eigenschaft der Refactoring-Technik ist die Verhaltensinvarianz. Das Modell vor dem Refactoring soll das gleiche beobachtbare Verhalten aufweisen, wie das Modell nach dem Refactoring. Ist diese Eigenschaft erfüllt, dann ist durch die Anwendung des Refactorings das vorhandene Verhalten nicht verändert worden und somit sind durch das Refactoring auch keine neuen Softwarefehler zur Spezifikation hinzugekommen. Tatsächlich hängt die Effizienz der Refactoring-Technik sehr stark von der Frage ab, in wie weit die Verhaltensinvarianz der Refactorings garantiert werden kann. Heute wird hauptsächlich ein schwacher Verhaltensäquivalenzbegriff, der durch strukturiertes Testen überprüft wird, eingesetzt. Bei einem solchen Äquivalenzbegriff besteht weiterhin das Risiko, dass durch das Refactoring Fehler in das Programm oder das Modell integriert werden, ohne dass dies bemerkt wird, da keiner der verwendeten Testfälle diesen Fehler aufdecken kann. Um Fehler beim Refactoring ausschließen zu können, wird ein wesentlich stärkerer Verhaltensäquivalenzbegriff benötigt.

Vor allem beim Einsatz von Refactoring im Anwendungsbereich der sicherheitskritischen Systeme ist es notwendig, die Verhaltensinvarianz von Refactorings zu garantieren, um Fehler durch das Refactoring ausschließen zu können. In der Arbeit wurde am Beispiel der Modellierungssprache AutoFocus gezeigt, dass sich ein formaler Verhaltensäquivalenzbegriff für das Refactoring definieren lässt, auf dessen Basis die Eigenschaft der Verhaltensinvarianz von Modell-Refactorings über die Semantik der Modellierungssprache allgemein für alle in der Sprache ausdrückbaren Modelle nachgewiesen werden kann. Die so verifizierten Refactorings garantieren somit, eine korrekte Anwendung vorausgesetzt, die Erhaltung des Systemverhaltens. Durch die Erfüllung dieser Eigenschaft wird die Effizienz der Durchführung von Refactorings stark gesteigert. Zum einen muss keine Überprüfung der Verhaltensäquivalenz durch den Entwickler erfolgen. Da die Refactorings immer korrekt sind, müssen diese im Gegensatz zu Refactorings, die potentiell doch das Verhalten verändern, nie rückgängig gemacht werden. Zum anderen können durch die Anwendung formal verifizierter Refactorings keine Fehler in die Spezifikation einschleichen. Somit entfällt der Aufwand für die Suche und Beseitigung von durch das Refactoring bedingten Fehlern.

Die Arbeit hat ergeben, dass die hier genannten Vorteile von Modell-Refactorings unter formaler Betrachtung von Verhaltensäquivalenz den erhöhten Aufwand für die Formalisierung bei weitem rechtfertigen. Falls die Komplexität der Semantik einer Sprache es zulässt, ist eine formal semantische Betrachtung von Refactoring empfehlenswert.

Die Auswirkungen von Zeit in verschiedenen semantischen Modellen wurde in der Arbeit untersucht. Es hat sich gezeigt, dass zeitasynchrone Semantiken grundsätzlich wesentlich flexibler gegenüber Modelländerungen als zeitsynchrone Semantiken sind und sich somit besser für das Modell-Refactoring eignen. Eine Notwendigkeit der Berücksichtigung von Refactoring bei der Konstruktion von Modellierungssprachen wurde identifiziert. In Bezug auf das zeitsynchrone AutoFocus hat sich herausgestellt, dass

das implizite Vorhandensein von festen Zeitbedingungen, dahingehend, dass pro Zeittakt in einer Zustandsmaschine genau eine Transition gefeuert wird, die Möglichkeiten der Durchführung von Refactorings sehr stark einschränken. Das in dieser Arbeit definierte zeitasynchrone AutoFocus hat sich dagegen als für das Modell-Refactoring sehr gut geeignet erwiesen. Diese Eignung wurde durch ein sehr komplexes Refactoring, das die Zerteilung atomarer Komponenten und somit sehr weitgehende Architekturänderungen ermöglicht, nachgewiesen.

Der Einsatz von Modell-Refactoring setzt eine Integration dieser Technik in den Entwicklungsprozess voraus. In dieser Arbeit wurde die Modell-Refactoring-Technik in einen iterativen modellbasierten Entwicklungsprozess eingebettet. Die Arbeit hat die Möglichkeit der Verwendung von Entwurfsmetriken zur Steuerung des Modell-Refactorings aufgezeigt. Der effiziente Einsatz von Modell-Refactoring erfordert eine umfassende Werkzeugunterstützung, wie sie in der vorliegenden Arbeit skizziert wurde.

10.2. Bezug zu anderen Modellierungssprachen für Reaktive Systeme

In der vorliegenden Arbeit wurde am Beispiel der Modellierungssprache AutoFocus die Modell-Refactoring-Technik für Reaktive Systeme vorgestellt. Diese Technik lässt sich auf andere Modellierungssprachen für Reaktive Systeme auf Grund der Ähnlichkeit der verwendeten Diagrammarten übertragen. Hierbei müssen die Besonderheiten der jeweiligen Sprachen berücksichtigt werden.

Die Arbeit hat gezeigt, dass eine zeitasynchrone Semantik der Modellierungssprache eine wesentlich höhere Flexibilität in Bezug auf Modelländerungen bietet als eine zeitsynchrone Semantik. Das Modell-Refactoring kann somit wesentlich einfacher für Modellierungssprachen mit zeitasynchroner Semantik, wie beispielsweise die *System Description Language* (SDL) [ITU02], adaptiert werden.

Grundsätzlich hat die Komplexität der Notation und der Semantik der Modellierungssprache starke Auswirkungen auf die Möglichkeit und Einfachheit der Durchführung von Modell-Refactorings. Die Modellierungssprache AutoFocus ist eine sehr schlanke Sprache, die sich auf die Darstellung der wesentlichen Konzepte zur Modellierung von Systemen beschränkt. Andere Modellierungssprachen für Reaktive Systeme sind in dieser Hinsicht wesentlich komplexer.

So legt beispielsweise die Semantik der *StateCharts* von Harel [Har87] eine Priorisierung für das Feuern von Transitionen in Zustandsmaschinen bezüglich ihrer Position innerhalb der Zustandsmaschinenhierarchie fest, die dazu führt, dass rein strukturelle Verschiebungen von Transitionen innerhalb dieser Hierarchie, wie sie in dieser Arbeit für AutoFocus in Abschnitt 5.2 definiert sind, in *StateCharts* zu Verhaltensänderungen führen können. Das Refactoring von *StateCharts* wird somit stark erschwert.

Bei der Adaption der Modell-Refactoring-Technik auf die Modellierungssprache *Real-Time Object-Oriented Modeling* (ROOM) [SGW94] und dessen Nachfolger *UML Real-Time* (UML-RT), die in die UML in der Version 2.0 [OMG04] integriert wurden, ist beim Refactoring der Strukturdiagramme, die als *Capsule Diagrams* bezeichnet werden, darauf zu achten, dass Restrukturierungen dieser Diagramme auch zu Änderungen der Klassendiagramme führen. Ist eine Klasse in den *Capsule Diagrams* einer Modellspezifikation mehrfach instanziiert, dann führen Refactorings hier häufig dazu, dass Varianten der betroffenen Klasse gebildet werden müssen. Die Möglichkeit der verschachtelten Verhaltenspezifikation auf unterschiedlichen Hierarchieebenen der Systemstruktur und die damit verbundene Unterscheidung zwischen weitervermittelnden Schnittstellen (*Relay-Ports*) und verarbeitenden Schnittstellen (*End-Ports*), die jeweils nach außen sichtbar oder unsichtbar sein können, erschwert ebenfalls das Refactoring von *UML-RT* Modellen.

Keine der heute in der Praxis eingesetzten Modellierungssprachen für Reaktive Systeme erweist sich als flexibel gegenüber Veränderungen der Modellstruktur unter Beibehaltung des Modellverhaltens. Somit eignen sich diese Sprachen nur bedingt für das Refactoring. Der Grund hierfür liegt vermutlich in der Tatsache, dass zum Zeitpunkt deren Entwicklung Modellrestrukturierung und Modellerweiterung nicht als Anforderungen an eine Modellierungssprache gesehen wurden. Wir möchten darauf hinweisen, dass heute fast jedes Softwaresystem, auch Software im Bereich der Reaktiven Systeme und Eingebetteten Systeme, großen Erweiterungen unterliegt. Die Unterstützung von Modellrestrukturierung muss als zentrale Anforderung bei der Definition von Modellierungssprachen berücksichtigt werden.

10.3. Zukünftige Arbeiten im Umfeld von Modell-Refactoring

Das Themengebiet des Modell-Refactorings wurde in dieser Arbeit sowohl aus theoretischer als auch aus praktischer Sicht umfassend betrachtet. Es konnten jedoch auf Grund der Vielfältigkeit und Komplexität dieses Themas nicht alle Facetten behandelt werden.

In der Arbeit wurde das Thema des Modell-Refactorings unter Verwendung eines formalen Verhaltensäquivalenzbegriffs betrachtet. Die Verhaltensäquivalenzeigenschaft wurde mathematisch in Prädikatenlogik unter Verwendung der Spezifikationssprache Focus und mit formal definierten Zeitabstraktionen in Focus stromverarbeitenden Funktionen dargestellt. Die Verhaltensinvarianzeigenschaft von Refactorings wurde durch eine Argumentation auf Basis der Semantik der Modellierungssprache AutoFocus nachgewiesen. Mathematische Beweise, die die Erfüllung der Verhaltensinvarianzeigenschaft der Refactoring-Operationen belegen, wurden in dieser Arbeit nicht durchgeführt. Bereits einfache in dieser Arbeit vorgestellte Modell-Refactorings, wie beispielsweise das *Push Down Component* Refactoring, das eine Komponente um eine Hierarchieebene nach unten verschiebt, bewirken eine Vielzahl von Diagrammänderungen, die alle in dem mathematischen Beweis der Verhaltensinvarianz berücksich-

tigt werden müssen und somit diesen sehr komplex werden lassen. Es bleibt einer zukünftigen Arbeit zu zeigen, in wie weit sich die Verhaltensinvarianz von komplexen Modell-Refactorings mathematisch beweisen lässt.

In dem AutoFocus CASE Werkzeug existiert mit dem *Aqua Framework* und der *Operation Description Language* (ODL) eine Unterstützung für die Spezifikation und Durchführung von Modelltransformationen. Im Rahmen dieser Arbeit wurden die AutoFocus Refactorings nicht in der Sprache ODL definiert. Bei dem Versuch der Definition von AutoFocus Refactorings in ODL hat sich herausgestellt, dass die deklarative Beschreibung von Operationen, wie sie in der ODL vorgenommen werden, für komplexere Operationen sehr schnell für den Entwickler unverständlich wird. Operative Beschreibungen, wie sie in dieser Arbeit verwendet werden, sind in den meisten Fällen besser verständlich. Für die Verifikation von Eigenschaften scheinen jedoch deklarative Definitionen von Modelltransformationen besser geeignet zu sein, als operative. In Bezug auf den vorangehend erwähnten mathematischen Nachweis der Verhaltensinvarianzeigenschaft von Refactorings ist die Formalisierung von AutoFocus Refactorings in ODL interessant. Schließlich bleibt es zu zeigen, wie gut sich ODL für die Durchführung mathematischer Beweise zum Nachweis von Eigenschaften von Modelltransformationen, insbesondere der Verhaltensinvarianzeigenschaft, auf Basis der Semantik von AutoFocus eignet.

In der Arbeit wurden gezeitete Modelle für den Entwurf Reaktiver Systeme betrachtet. Es hat sich gezeigt, dass zeitasynchrone Modelle wesentlich flexibler gegenüber Änderungen sind als zeitsynchrone Modelle und sich somit besser für das Modell-Refactoring eignen. Aus den Anforderungen der Anwendungsdomäne kann die Verwendung von zeitsynchronen Modellen jedoch erforderlich sein. In einem solchen Fall ist eine Modellierungssprache mit zeitsynchroner Semantik wünschenswert, die trotz der engen Zeitbindung die flexible Veränderung der Modellstruktur und somit das Modell-Refactoring unterstützt. In dem zeitsynchronen AutoFocus ist dies auf Grund der impliziten Verknüpfung zwischen Zeitschritten und dem Feuern von Transitionen nicht gegeben. Es ist eine zeitsynchrone Modellierungssprache wünschenswert, in der die Zeitbedingungen explizit durch den Entwickler modelliert werden. Diese Zeitbedingungen legen die Grenzen für die verhaltensinvarianten Veränderungen fest. Für das Refactoring hat eine explizite Modellierung der Zeitbedingungen gegenüber einer impliziten den Vorteil, dass der Entwickler dazu gezwungen wird, die Zeitbedingungen festzulegen und es hierdurch vermieden wird, dass unbeabsichtigt Zeitbedingungen spezifiziert werden. Hierdurch werden nicht erforderliche Zeitbedingungen vermieden und somit mehr zeitliche Freiheitsgrade in der Spezifikation zugelassen. Des Weiteren erscheint die Einführung von Konzepten in zeitsynchrone Sprachen, die eine Zeitflexibilisierung innerhalb genau definierter Zeitschranken bewirken, in Hinblick auf das Refactoring sinnvoll.

In der vorliegenden Arbeit wurde die Notwendigkeit der Berücksichtigung der Veränderbarkeit von Modellspezifikationen als zentrale Anforderung bei der Definition von Modellierungssprachen identifiziert. Diese ist notwendig, um das Modell-Refactoring

optimal zu unterstützen und die Erweiterung von Modellen zu vereinfachen. Der von Stefănescu in [Ste00] vorgestellte theoretische Ansatz der Definition einer Algebra für Netzwerkstrukturen, die Axiome zur invarianten Umformung festlegt und sich auf konkrete Strukturen, wie beispielsweise Datenflussdiagramme und Endliche Automaten, anwenden lässt, erscheint in diesem Zusammenhang als sehr vielversprechend. Es ist denkbar solche Algebren bei der Definition von in der Praxis einsetzbaren Modellierungssprachen zu verwenden, um eine Flexibilisierung der Sprache gegenüber Veränderungen von Modellspezifikationen zu erreichen.

Eine Fragestellung wurde bisher bei dem Modell-Refactoring vollständig ausgeklammert. Die heute in der Praxis eingesetzten Modellierungssprachen stellen neben verschiedenen graphischen Diagrammarten immer auch eine meist zu Programmiersprachen sehr ähnliche textuelle Sprache bereit, die zur Datentypdefinition, Funktionsdefinition und Verhaltensbeschreibung dient. So enthält beispielsweise AutoFocus die funktionale Sprache Quest/F. Neben dem reinen Refactoring von graphischen Modellen wird folglich auch eine Refactoring Unterstützung für den innerhalb von graphischen Modellen verwendeten Programmcode benötigt, um dessen Qualität verbessern zu können. In vielen Fällen kann das Verhalten entweder graphisch in Zustandsmaschinen oder in der textuellen Sprache dargestellt werden. Diese unterschiedlichen Darstellungsformen haben starke Auswirkungen auf die Lesbarkeit, Verständlichkeit und Wartbarkeit der Modellspezifikationen. Es werden Refactorings zwischen diesen beiden Darstellungsformen benötigt, um beispielsweise ein Verhalten, das rein textuell durch eine Funktion beschrieben ist, in einen Teil einer Zustandsmaschine umzuwandeln.

Ein Thema das ebenfalls bisher nicht behandelt wurde, ist die Frage, wie sich das Modell-Refactoring von Strukturdiagrammen und Zustandsmaschinen auf bestehende Ablaufbeschreibungen des Modells in Form von Sequenzdiagrammen auswirkt. Dies ist insbesondere in dem Kontext der Nutzung dieser Abläufe als Testfälle für Regressionstests, sei es auf der Ebene von Modellen oder auch auf der Ebene von Programmcode, sehr interessant.

Neben den in dieser Arbeit vorgestellten Modell-Refactorings, die allgemein auf Modellspezifikationen Reaktiver Systeme anwendbar sind, werden in den verschiedenen speziellen Anwendungsdomänen auch spezialisierte Modell-Refactorings benötigt. So sind beispielsweise im Automobilbereich Refactorings zur Integration von in diesem Bereich verwendeten Kommunikationsinfrastrukturen, wie beispielsweise das *Controller Area Network* (CAN) und die *Time-Triggered Architecture* (TTA), denkbar. Für sicherheitskritische Systeme im Sinne der Datensicherheit lassen sich Refactorings zur Integration von Sicherheitsmechanismen, wie beispielsweise der Verschlüsselung von Kommunikation, realisieren. Für die verschiedenen speziellen Anwendungsdomänen können in Zukunft eine Vielzahl von interessanten Refactorings identifiziert und definiert werden.

Das Thema der domänenspezifischen Refactorings ist im direkten Zusammenhang mit dem Thema der domänenspezifischen Entwurfsmuster zu sehen. Es werden Entwurfs-

muster benötigt, die die Besonderheiten der Anwendungsdomäne in qualitativ hochwertige Architekturkonzepte umsetzen. Modell-Refactorings lassen sich einsetzen, um diese domänenspezifischen Entwurfsmuster in bestehende Softwarearchitekturen zu integrieren.

In der Arbeit wurde angedeutet, dass neben der Qualitätssteigerung das Refactoring auch mit anderen Zielsetzungen durchgeführt werden kann. Hier bleibt es zu untersuchen, für welche Zielsetzungen sich das Modell-Refactoring eignen kann. Wie bei der Durchführung von Refactoring zur Qualitätssteigerung können auch bei anderen Zielsetzungen Metriken zu deren Bewertung hilfreich sein.

Die Arbeit hat die Idee des Einsatzes von Qualitätsmetriken für Entwurfsmodelle zur Steuerung des Modell-Refactorings skizziert. Je besser diese Metriken den Qualitätsbegriff fassen können, um so mehr lässt sich das Modell-Refactoring automatisieren. Diese Qualitätsmetriken müssen entwickelt werden, um die Automatisierung der Softwareentwicklung voranzutreiben.

Es werden empirische Studien benötigt, die den Nutzen der Modell-Refactoring-Technik an sich, als auch den Nutzen von bestimmten vorgeschlagenen Modell-Refactorings bewerten.

Schließlich wird für die effiziente Nutzung von Modell-Refactoring eine umfassende Werkzeugunterstützung benötigt. Hier wird sich in Zukunft der Trend zu immer mehr Automatisierung abzeichnen. Es müssen eine Vielzahl von neuen Techniken entwickelt werden, um diese Automatisierung für das Modell-Refactoring realisieren zu können.

10.4. Vision der Softwareentwicklung in der Zukunft

Abschließend soll eine Vision der Softwareentwicklung in der Zukunft angedacht werden, zu der das Modell-Refactoring einen Beitrag leisten kann. Das Erstellen beziehungsweise das Verändern einer Softwarearchitektur lässt sich als ein klassisches Suchproblem in der Informatik auffassen. So lässt sich die Suche nach einer guten Softwarearchitektur mit einem Computerschachspiel vergleichen. Der Entwickler erweitert laufend das bestehende Entwurfsmodell um neue Funktionalitäten und der Computer spielt in dem Sinn dagegen, dass dieser mit Hilfe von Entwurfsmetriken laufend die Qualität des Entwurfsmodells automatisch bewertet und mit Hilfe von Modell-Refactorings automatisch die Qualität verbessert. Die Modell-Refactorings legen hier die möglichen erlaubten Spielzüge fest und spannen somit einen Suchraum auf. Die Entwurfsmetriken dienen hierbei als Bewertungsfunktion für die Suche. Der Entwickler muss sich in diesem Szenario nicht um die Qualität eines Modells kümmern. Ihm werden, wie bei den gegnerischen Zügen eines Schachspiels, lediglich die durch den Computer getätigten Veränderungen des Modells gezeigt.

Eine solche automatische Suche von qualitativ guten verhaltensinvarianten Architekturen setzt eine bis heute nicht annähernd erreichte Formalisierung des Begriffs der Qualität eines Modells voraus, der zudem in unterschiedlichen Anwendungsbereichen

sehr unterschiedliche Ausprägungen aufweisen kann. Des Weiteren muss, um ein solches Szenario realisieren zu können, die Größe des Suchraums der verhaltensinvarianten Architekturen technisch beherrschbar sein, was nach dem heutigen Stand der Technik nicht gegeben ist. Werden neben der Qualität auch weitere Kennzeichen einer guten Softwarearchitektur formalisiert und bei der automatisierten Suche von geeigneten Softwarearchitekturen berücksichtigt, können hierdurch zunehmend Aufgaben des Softwareentwurfs automatisiert werden.

Schließlich kann in Zukunft die Frage gestellt werden, ob es möglich und praktikabel ist, die Erstellung von Softwarearchitekturen beziehungsweise von Softwareentwürfen voll zu automatisieren und eine direkte automatische Übersetzung von formalen Anforderungen in fertige Softwareprogramme zu realisieren.

A. Bezeichner und Symbole

Nachfolgend sind die in dieser Arbeit speziell verwendeten Bezeichner und Symbole von Operationen, Abbildungen, Mengen, Tupel und Datentypen aufgelistet. Hierbei sind die Bezeichner und Symbole, die in herkömmlichen Focus Termen verwendet werden, nicht aufgeführt. Diese sind in [BS01, Anhang B] angegeben.

$[\![S]\!]_{AF}$
: Denotation der AutoFocus Modellspezifikation S als prädikatenlogische Formel (Definition 3.5 auf Seite 53 und Definition 3.9 auf Seite 56).

$\otimes_{\alpha_x} \in (\underline{\mathcal{N}} \times \mathcal{L}_{Syn}) \times (\underline{\mathcal{N}} \times \mathcal{L}_{Syn}) \rightarrow \mathcal{P}(\underline{\mathcal{N}})$
: Kompositionsoperator für abstraktes Schnittstellenverhalten (Definition 3.23 auf Seite 80).

$\otimes_K \in (\underline{\mathcal{M}} \times \mathcal{L}_{Syn}) \times (\underline{\mathcal{M}} \times \mathcal{L}_{Syn}) \rightarrow \underline{\mathcal{M}}$
: Kompositionsoperator für konkretes Schnittstellenverhalten (Definition 3.15 auf Seite 62).

$\otimes_{Syn} \in \mathcal{L}_{Syn} \times \mathcal{L}_{Syn} \rightarrow \mathcal{L}_{Syn}$
: Komposition zweier syntaktischer Schnittstellen.

$\oplus_{Tup}$
: Konkatenation zweier Tupel zu einem Tupel:

$$(x_1, x_2, ..., x_n) \oplus_{Tup} (y_1, y_2, ..., y_m) \overset{\text{def}}{=} (x_1, x_2, ..., x_n, y_1, y_2, ..., y_m) \tag{A.1}$$

$\odot_{i=1}^{n} x_i$
: Konstruktion eines Tupels aus den Elementen $x_1, x_2, \ldots, x_n$:

$$\odot_{i=1}^{n} x_i \overset{\text{def}}{=} (x_1, x_2, \ldots, x_n) \tag{A.2}$$

%
: Symbol zur Kennzeichnung von Kommentaren innerhalb von Focus Spezifikationen.

$\times_{i=1}^{n} X_i$
: Kreuzprodukt der Mengen $X_1, X_2, \ldots, X_n$:

$$\times_{i=1}^{n} X_i \overset{\text{def}}{=} X_1 \times X_2 \times \ldots \times X_n \tag{A.3}$$

$\bigwedge_{i=1}^{n} x_i$
Konjunktion der Terme $x_1, x_2, \ldots, x_n$:

$$\bigwedge_{i=1}^{n} x_i \stackrel{\mathsf{def}}{=} x_1 \wedge x_2 \wedge \ldots \wedge x_n \tag{A.4}$$

$\bigwedge_{x \in X} e(x)$
Konjunktion der Terme $e(x)$ aller Elemente x, die in X enthalten sind.

$\bigvee_{x \in X} e(x)$
Disjunktion der Terme $e(x)$ aller Elemente x, die in X enthalten sind.

$\bigcup_{i=1}^{n} X_i$
Vereinigung der Mengen $X_1, X_2, \ldots, X_n$:

$$\bigcup_{i=1}^{n} X_i \stackrel{\mathsf{def}}{=} X_1 \cup X_2 \cup \ldots \cup X_n \tag{A.5}$$

$\bigcup_{x \in X} E(x)$
Vereinigung der Mengen $E(x)$ aller Elemente $x \in X$.

$\{x \in X \text{ so that } e(x)\}$
Menge aller Elemente x aus der Menge X für die die Eigenschaft $e(x)$ gilt.

$x \in X \text{ so that } e(x)$
x wird so belegt, dass die Eigenschaft $e(x)$ erfüllt ist.

$\alpha_x \in M^{\underline{\omega}, \mathsf{ts}} \rightarrow N^{\underline{\omega}, \mathsf{ts}}$ oder $\alpha_x \in M^{\underline{\omega}, \mathsf{ts}} \rightarrow N^{\underline{\omega}}_{noVoid}$
Abstraktion von AutoFocus Schnittstellenabläufen (α_{NI} Definition 3.16 auf Seite 69, α_{ZA1} Definition 3.17 auf Seite 71, α_{ZA2} Definition 3.18 auf Seite 76 und α_{ZA3} Definition 3.20 auf Seite 77).

$\widetilde{\alpha}_x \in \mathcal{M} \rightarrow \mathcal{N}$
Abstraktion von AutoFocus Schnittstellenverhalten.

$\underline{\widetilde{\alpha}}_x \in \underline{\mathcal{M}} \rightarrow \underline{\mathcal{N}}$
Verallgemeinerung der Abstraktion von Schnittstellenverhalten auf beliebige Schnittstellensignaturen.

$\widetilde{\gamma}_x \in \mathcal{N} \rightarrow \mathcal{M}$
Konkretisierung von abstraktem AutoFocus Schnittstellenverhalten (Definition 3.21 auf Seite 78).

$\underline{\widetilde{\gamma}}_x \in \underline{\mathcal{N}} \rightarrow \underline{\mathcal{M}}$
Verallgemeinerung der Konkretisierung von Schnittstellenverhalten auf beliebige Schnittstellensignaturen.

$\underline{A}$
: Verallgemeinerung einer Verhaltensabbildung auf allgemeines Verhalten unter Verwendung einer beliebigen Schnittstellensignatur (Definition 3.12 auf Seite 59).

$\widetilde{A}$
: Erweiterung einer Abbildung von Abläufen auf eine Abbildung von Schnittstellenverhalten (Definition 3.13 auf Seite 60).

$AbsSyn \in \mathcal{L}_{Syn} \to \mathcal{L}_{Syn}$
: Abbildung, die zu einer konkreten syntaktischen Schnittstelle deren abstrakte syntaktische Schnittstelle durch Weglassung der speziellen Verarbeitungskanäle liefert.

$Bez \in \mathcal{L}_{Syn} \to String \times ... \times String$
: Abbildung, die zu einer gegebenen syntaktischen Schnittstelle das Tupel der darin verwendeten Variablenbezeichner liefert.

Body
: Unterer Teil einer Focus Spezifikation, in dem die Beziehungen zwischen Ein- und Ausgaben ausgedrückt sind.

$C_{Focus} \in \mathcal{L}_A \to \mathcal{L}_F$
: Übersetzung von AutoFocus in Focus.

$I = \{i_1, \ldots, i_n\}$
: Menge der Bezeichner der Eingabekanäle einer von AutoFocus in Focus übersetzten Spezifikation.

$I_1, \ldots, I_n$
: Datentypen der Eingabekanäle einer von AutoFocus in Focus übersetzten Spezifikation.

$I_x^{\infty}, I_x^{\omega}$
: Menge aller entsprechend dem Datentyp des Eingabekanals möglichen unendlichen beziehungsweise endlichen und unendlichen gezeiteten Ströme.

$I_x^{\infty,\mathsf{ts}}, I_x^{\omega,\mathsf{ts}}$
: Menge aller entsprechend dem Datentyp des Eingabekanals möglichen unendlichen beziehungsweise endlichen und unendlichen zeitsynchronen Ströme (siehe Definition 3.10 auf Seite 57).

$KonSyn \in \mathcal{L}_{Syn} \to \mathcal{L}_{Syn}$
: Abbildung, die zu einer abstrakten syntaktischen Schnittstelle deren konkrete syntaktische Schnittstelle durch Hinzufügen der speziellen Verarbeitungskanäle liefert.

$\mathcal{L}_A$
: Menge aller syntaktisch korrekten Spezifikationen in der Sprache AutoFocus.

$\mathcal{L}_F$
: Menge aller syntaktisch korrekten Spezifikationen in der Sprache Focus.

$\mathcal{L}_{Syn}$
Menge aller syntaktischen Schnittstellen in der Sprache Focus.

$\mathcal{L}_{V}$
Menge aller syntaktisch korrekten zeitrobusten atomaren Komponentenspezifikationen in der Sprache AutoFocus, die Gegenstand von Veränderungen durch den Entwickler sein können.

$\mathcal{L}_{ZR}$
Menge aller syntaktisch korrekten zeitrobusten Spezifikationen in der Sprache AutoFocus.

$M = I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m \times \left(\times_{j=1}^{n} (Signal \cup VoidDT) \right)$
Konkrete Schnittstellensignatur einer AutoFocus Spezifikation einschließlich der Verarbeitungskanäle unter Verwendung fester Bezeichner.

$\underline{M}$
Menge aller AutoFocus Schnittstellensignaturen.

$M^{\underline{\infty},\mathsf{ts}} = I_1^{\underline{\infty},\mathsf{ts}} \times \ldots \times I_n^{\underline{\infty},\mathsf{ts}} \times O_1^{\underline{\infty},\mathsf{ts}} \times \ldots \times O_m^{\underline{\infty},\mathsf{ts}} \times \left(\times_{j=1}^{n} (Signal \cup VoidDT)^{\underline{\infty},\mathsf{ts}} \right)$
Menge der entsprechend einer fest vorgegebenen Schnittstellensignatur M möglichen konkreten unendlichen AutoFocus Abläufe von Ein- und Ausgaben einschließlich der speziellen Verarbeitungskanäle. Diese werden in Focus als Menge von Tupel von zeitsynchronen Strömen dargestellt (siehe Definition 3.11 auf Seite 57).

$M^{\underline{\omega},\mathsf{ts}} = I_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times I_n^{\underline{\omega},\mathsf{ts}} \times O_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times O_m^{\underline{\omega},\mathsf{ts}} \times \left(\times_{j=1}^{n} (Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}} \right)$
Menge der entsprechend einer fest vorgegebenen Schnittstellensignatur M möglichen konkreten endlichen und unendlichen AutoFocus Abläufe von Ein- und Ausgaben einschließlich der speziellen Verarbeitungskanäle. Diese werden in Focus als Menge von Tupel von zeitsynchronen Strömen dargestellt.

$\mathcal{M} = \mathcal{P}(M^{\underline{\infty},\mathsf{ts}})$
Menge aller entsprechend der vorgegebenen Signatur möglichen konkreten Schnittstellenverhalten.

$\underline{\mathcal{M}} \supset \mathcal{M}$
Menge aller in AutoFocus spezifizierbaren konkreten Schnittstellenverhalten aller in AutoFocus ausdrückbaren Schnittstellensignaturen.

$N = I_1 \times \ldots \times I_n \times O_1 \times \ldots \times O_m$
Abstrakte Schnittstellensignatur einer AutoFocus Spezifikation ohne Verarbeitungskanäle unter Verwendung fester Bezeichner.

$N_{noVoid} = I_1 \setminus \{void\} \times \ldots \times I_n \setminus \{void\} \times O_1 \setminus \{void\} \times \ldots \times O_m \setminus \{void\}$
Abstrakte Schnittstellensignatur einer AutoFocus Spezifikation ohne Verarbeitungskanäle und ohne *void* Nachrichten unter Verwendung fester Bezeichner.

$N^{\underline{\omega}} = I_1^{\underline{\omega}} \times \ldots \times I_n^{\underline{\omega}} \times O_1^{\underline{\omega}} \times \ldots \times O_m^{\underline{\omega}}$

Menge der entsprechend einer fest vorgegebenen Schnittstellensignatur M möglichen abstrakten nicht zwingend zeitsynchronen endlichen und unendlichen AutoFocus Abläufe von Ein- und Ausgaben ohne die speziellen Verarbeitungskanäle. Diese werden in Focus als Menge von Tupel von gezeiteten Strömen dargestellt.

$N^{\underline{\omega},\mathsf{ts}} = I_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times I_n^{\underline{\omega},\mathsf{ts}} \times O_1^{\underline{\omega},\mathsf{ts}} \times \ldots \times O_m^{\underline{\omega},\mathsf{ts}}$

Menge der entsprechend einer fest vorgegebenen Schnittstellensignatur M möglichen abstrakten endlichen und unendlichen zeitsynchronen AutoFocus Abläufe von Ein- und Ausgaben ohne die speziellen Verarbeitungskanäle. Diese werden in Focus als Menge von Tupel von zeitsynchronen Strömen dargestellt (siehe Definition 3.5 auf Seite 59).

$\mathcal{N} = \mathcal{P}(N^{\underline{\omega}})$

Menge aller entsprechend der vorgegebenen Signatur möglichen abstrakten Schnittstellenverhalten.

$\underline{\mathcal{N}} \supset \mathcal{N}$

Menge aller abstrakten AutoFocus Schnittstellenverhalten mit beliebiger Schnittstellensignatur.

$O = \{o_1, \ldots, o_m\}$

Menge der Bezeichner der Ausgabekanäle einer von AutoFocus in Focus übersetzten Spezifikation.

$O_1, \ldots, O_m$

Datentypen der Ausgabekanäle einer von AutoFocus in Focus übersetzten Spezifikation.

$O_x^{\underline{\infty}}, O_x^{\underline{\omega}}$

Menge aller entsprechend dem Datentyp des Ausgabekanals möglichen unendlichen beziehungsweise endlichen und unendlichen gezeiteten Ströme.

$O_x^{\underline{\infty},\mathsf{ts}}, O_x^{\underline{\omega},\mathsf{ts}}$

Menge aller entsprechend dem Datentyp des Ausgabekanals möglichen unendlichen beziehungsweise endlichen und unendlichen zeitsynchronen Ströme (siehe Definition 3.10 auf Seite 57).

$\mathcal{P}(X)$

Potenzmenge der Menge X.

Par

Menge der möglichen Tupel von Parametern einer Modelltransformation beziehungsweise eines Refactorings.

$Pre_T \in \mathcal{L} \times Par \rightarrow \{wahr, falsch\}$

Vorbedingung eines Refactorings.

$Q \in \mathcal{L} \to \mathbb{N}_0$
: Qualitätsmetrik.

$\mathcal{R}_S$
: Ein-/Ausgabereation einer von AutoFocus in Focus übersetzten Spezifikation.

$Sig \in \mathcal{L}_{Syn} \to \underline{M}$
: Abbildung, die zu einer gegebenen syntaktischen Schnittstelle deren Schnittstellensignatur liefert.

Signal
: AutoFocus Datentyp *data Signal=Present*, in Focus als *type Signal={Present}* dargestellt.

$SV_{AF} \in \mathcal{L}_A \to \mathcal{M}$
: Abbildung, die zu einer AutoFocus Spezifikation mit vorgegebener Schnittstellensignatur deren Schnittstellenverhalten in Form von Mengen von Tupel von Strömen (Mengen von Abläufen) liefert (Definition 3.11 auf Seite 57).

$\underline{SV}_{AF} \in \mathcal{L}_A \to \underline{\mathcal{M}}$
: Verallgemeinerung der SV Abbildung auf beliebige Schnittstellensignaturen.

Syn_{AF}
: Abbildung, die zu einer atomaren oder komponierten AutoFocus Komponentenspezifikation deren syntaktische Schnittstelle entsprechend der Focus Übersetzung liefert (Definition 3.2 auf Seite 52 und Definition 3.6 auf Seite 54).

$T \in \mathcal{L} \times Par \to \mathcal{L}$
: Modelltransformation beziehungsweise Refactoring.

$V = \{v_1, \ldots, v_n\}$
: Menge der Bezeichner der speziellen Ausgabekanäle einer von AutoFocus in Focus übersetzten Spezifikation, die zur Kommunikation der Verarbeitung von Eingaben dienen.

VoidDT
: Focus Datentyp zur Darstellung des Nichtvorhandenseins von AutoFocus Nachrichten *type VoidDT={void}*.

B. Definition von Refactorings auf Basis der AutoFocus Modell API

Nachfolgend sind AutoFocus Refactorings in der Programmiersprache Java unter Verwendung der AutoFocus Modell API definiert. Der hier gezeigte Programmcode dient der operationellen Definition der aufgeführten Refactorings. Zur Verwendung in dem CASE Werkzeug AutoFocus sind Anpassungen dieses Codes erforderlich.

Folgende Hilfsfunktionen werden in dem Programmcode verwendet:

`public Component Component::topLevelComponent()` liefert die oberste Komponente des Modells, in dem sich die betrachtete Komponente befindet.

`public boolean Component::containsComponent(Component x)` liefert den Wert *true*, falls die Komponente x in der betrachteten Komponente über mehrere Hierarchieebenen hinweg enthalten ist.

`public Component Component::getContainingSubComponent (Component x)` liefert die Unterkomponente einer Komponente, die in ihrem Unterbaum die Komponente x enthält.

`public Port Component::correspondingPort(Port x)` liefert den Port der betrachteten Komponente, der äquivalent zu dem Port x einer anderen Komponente ist, das heißt den gleichen Namen besitzt.

In Abschnitt B.1 ist das *Push Down Component* Refactoring festgelegt. Das *Wrap Components* Refactoring ist in Abschnitt B.2, das *Move Components* Refactoring in Abschnitt B.3 und das *Remove Single Component Hierarchy* Refactoring ist in Abschnitt B.4 definiert.

B.1. Push Down Component Refactoring

```
public static boolean pushDownComponent(Component A,
    Component B){
  // Vorbedingung des Push Down Component Refactorings
  // 1. Es wurden verschiedene Komponenten gewählt
  if ( ( A != B ) &&
    // 2. Beide Komponenten befinden sich im gleichen SSD
    ( A.getSuperComponent() == B.getSuperComponent() ) &&
    // 3. B ist nicht atomar
    ( B.hasSubComponents() == true )
```

```
){
  // Schritt 1: Kopiere Komponente A
  Component A2 = A.deepCopyComponent();
  B.addSubComponents(A2);
  // Alle Ports der Komponente A betrachten
  Enumeration e = A.getPorts().elements()
  while ( e.hasMoreElements() ) {
    Port m = (Port) e.nextElement();
    // Gegenüberliegenden Port ermitteln
    Channel o;
    Port n;
    if ( m.getDirection().isExit() == true ){
      o = (Channel) m.getOutChannels().firstElement();
      n = o.getDestinationPort();
    } else {
      o = m.getInChannel();
      n = o.getSourcePort();
    }
    // Falls der Port m über einen Kanal mit einem Port der
    // Komponente B verbunden ist
    if (n.getComponent() == B) {
      // Weiterführenden Kanal von Port n in SSD C ermitteln
      Channel p;
      if ( n.getDirection().isEntry() == true ){
        p = (Channel) n.getOutChannels().firstElement();
      } else {
        p = n.getInChannel();
      }
      // Schritt 2a: Lösche Kanäle zwischen Ports der
      // Komponenten A und B in SSD C
      C.removeChannels(o);
      // Schritt 2b: Umleiten der Kanäle in B
      if ( n.getDirection().isEntry() == true ) {
        p.setSourcePort(A2.correspondingPort(m));
      } else {
        p.setDestinationPort(A2.correspondingPort(m));
      }
      // Nicht mehr benötigten Port löschen
      B.deletePorts(n);
    } else {
      // Kanal gefunden, der mit der Komponente A nicht
      // aber mit B verbunden ist
      // Schritt 3a: Schnittstelle von B erweitern
      Port q=new Port(m.getName());
```

```
        q.setType(m.getType());
        q.setDirection(m.getDirection());
        B.addPorts(q);
        // Schritt 3b: Umleiten der Kanäle über neue Ports von
        // B in dem SSD C
        if ( m.getDirection().isExit() == true ) {
          o.setSourcePort(q);
        } else {
          o.setDestinationPort(q);
        }
        // Einfügen des neuen Kanals in dem SSD B
        Channel p=new Channel();
        p.setName(o.getName());
        p.setType(o.getType());
        if (q.getDirection().isEntry()==true){
          p.setSourcePort(q);
          p.setDestinationPort(A2.correspondingPort(m));
        } else {
          p.setSourcePort(A2.correspondingPort(m));
          p.setDestinationPort(q);
        }
        B.addChannels(p);
      }
    }
    // Schritt 4: Löschen der Komponente A
    C.removeSubComponents(A);
    // Refactoring erfolgreich durchgeführt
    return true;
  } else {
    // Vorbedingungen des Refactorings sind nicht erfüllt
    return false;
  }
}
```

B.2. Wrap Components Refactoring

```
public static boolean wrapComponents(Vector components, String
componentName) {
  // Vorbedingungen
  boolean precondition=true;
  Enumeration e=components.elements();
  // Mindestens eine Komponente gewählt
```

```
  if (components.size()>=1){
    // Alle Komponenten im gleichen SSD
    Component parentComponent=e.nextElement().
        getSuperComponent();
    while (e.hasElements()){
      if (e.nextElement().getSuperComponent()!=
          parentComponent) {
        precondition=false;
      }
    }
  } else {
   precondition=false;
  }
  if (precondition == true) {
    // Vorbedingung erfüllt
    // Neue nichtatomare Komponente erzeugen
    Component newComponent=new Component();
    newComponent.setName(componentName);
    parentComponent.addComponents(newComponent);
    e=components.elements();
    while (e.hasElements()){
      // Komponenten durch das Push Down Component Refactoring
      // in die neue Komponente verschieben
      pushDownComponent(e.nextElement(),newComponent);
    }
    // Refactoring erfolgreich durchgeführt
    return true;
  } else {
    // Vorbedingung nicht erfüllt
    return false;
  }
}
```

B.3. Move Components Refactoring

```
public static boolean moveComponent(Component A, Component B){
  // Vorbedingung
  if ( ( B.hasSubComponents()== true ) &&
      ( A.topLevelComponent() == B.topLevelComponent() ) ) {
    // Vorbedingung erfüllt
    Component x=A.getSuperComponent()
    // Komponente so lange unter Verwendung des Pull Up
```

```
    // Component Refactorings in der Hierarchie nach oben
    // verschieben, bis die Zielkomponente in dem Unterbaum
    // des SSDs enthalten ist
    while (!x.containsComponent(B)){
      pullUpComponent(A);
      x=A.getSuperComponent();
    }
    // Komponente so lange unter Verwendung des Push Down
    // Component Refactorings in der Hierarchie nach unten
    // in Richtung der Zielkomponente verschieben, bis diese
    // in der Zielkomponente enthalten ist
    while (A.getSuperComponent()!=B) {
      pushDownComponent(A,
        A.getSuperComponent.getContainingSubComponent(B));
    }
    // Refactoring erfolgreich durchgeführt
    return true;
  } else {
    // Vorbedingung nicht erfüllt
    return false;
  }
}
```

B.4. Remove Single Component Hierarchy Refactoring

```
public static boolean removeSingleComponentHierarchy
    (Component A){
  // Vorbedingung
  if (A.getSubComponents().size()==1 &&
      A.hasSuperComponent()==true){
    // Unterkomponente durch Anwendung des Pull Up Component
    // Refactorings eine Hierarchieebene nach oben verschieben
    pullUpComponent(A.getSubComponents().firstElement());
    // Nicht mehr benötigte Komponente löschen
    A.getSuperComponent().removeComponents(A);
    return true;
  } else {
    // Vorbedingung nicht erfüllt
    return false;
  }
}
```

C. Puffer in der zeitasynchronen AutoFocus Semantik

C.1. Datentyp- und Funktionsdefinitionen der Pufferkomponente

Nachfolgend ist die Datentypdefinition der von der Pufferkomponente zur Definition der zeitasynchronen AutoFocus Semantik verwendeten Listentypen für die Pufferung von *Integer* Werten zusammen mit den benötigten Funktionen in der Sprache Quest/F angegeben.

```
data IntList = Cons(Int, IntList) | EmptyIntList;

fun lastIntElement(Cons(x:Int, y:IntList)) =
  if y == EmptyIntList then
     x
  else
     lastIntElement(y)
  fi;

fun deleteLastInt(Cons(x:Int, y:IntList)) =
  if y == EmptyIntList then
    EmptyIntList
  else
    Cons(x,deleteLastInt(y))
  fi;

fun intListLength(EmptyIntList) = 0 |
    intListLength(Cons(x:Int, y:IntList)) =
    1 + intListLength(y);
```

C.2. Transitionen des Pufferkomponente

Nachfolgend sind die Transitionen des Zustandsübergangsdiagramms (STD) *Buffer* aus der Definition der zeitasynchronen AutoFocus Semantik aus Abschnitt 6.6.4 aufgeführt.

Die Bezeichnerplatzhalter *name* und *type* sind durch den Namen und den Datentyp des Eingabe-Ports des Puffers zu ersetzen.

```
Bezeichnung:   Next Empty
Vorbedingung:  typeListLength(bufferVar) = 1
Eingabe:       nameCont?Present; nameP?
Ausgabe:
Zuweisung:     bufferVar = deleteLastType(bufferVar)

Bezeichnung:   Next Not Empty
Vorbedingung:  typeListLength(bufferVar) > 1
Eingabe:       nameCont?Present; nameP?
Ausgabe:
Zuweisung:     bufferVar = deleteLastType(bufferVar)

Bezeichnung:   Next Output
Vorbedingung:  typeListLength(bufferVar) > 1
Eingabe:       nameCont?Present; nameP?
Ausgabe:       name!lastTypeElement(deleteLastType(bufferVar))
Zuweisung:     bufferVar = deleteLastType(bufferVar)

Bezeichnung:   Next Save
Vorbedingung:
Eingabe:       nameP?x; nameCont?Present
Ausgabe:
Zuweisung:     bufferVar = Cons(x,deleteLastType(bufferVar))

Bezeichnung:   Next Save Output
Vorbedingung:  typeListLength(bufferVar) > 0
Eingabe:       nameP?x; nameCont?Present
Ausgabe:       name!lastTypeElement(Cons(x,
                 deleteLastType(bufferVar)))
Zuweisung:     bufferVar = Cons(x,deleteLastType(bufferVar))

Bezeichnung:   Output
Vorbedingung:  typeListLength(bufferVar) > 0
Eingabe:       nameP?; nameCont?
Ausgabe:       name!lastTypeElement(bufferVar)
Zuweisung:

Bezeichnung:   Output Delay
Vorbedingung:
Eingabe:       nameP?
Ausgabe:       name!lastTypeElement(bufferVar)
```

```
Zuweisung:

Bezeichnung:   Save
Vorbedingung:
Eingabe:       nameP?x
Ausgabe:
Zuweisung:     bufferVar = Cons(x,bufferVar)

Bezeichnung:   Save Output
Vorbedingung:
Eingabe:       nameP?x; nameCont?
Ausgabe:       name!lastTypeElement(Cons(x,bufferVar))
Zuweisung:     bufferVar = Cons(x,bufferVar)

Bezeichnung:   Save Output Delay
Vorbedingung:
Eingabe:       nameP?x;
Ausgabe:       name!lastTypeElement(Cons(x,bufferVar))
Zuweisung:     bufferVar = Cons(x,bufferVar)

Bezeichnung:   Save Output Empty
Vorbedingung:
Eingabe:       nameP?x
Ausgabe:       name!lastTypeElement(Cons(x,bufferVar))
Zuweisung:     bufferVar = Cons(x,bufferVar)

Bezeichnung:   Wait
Vorbedingung:
Eingabe:       nameP?
Ausgabe:
Zuweisung:     bufferVar = Cons(x,bufferVar)
```

C.3. Transitionen der Zustandsmaschine der Multiplexing Komponente

Nachfolgend sind die Transitionen der Zustandsmaschine der Multiplexing Komponente, die zur Realisierung von zwei zu eins Kommunikationsbeziehungen in der zeitasynchronen AutoFocus Semantik verwendet wird (siehe Abschnitt 6.6.6), aufgelistet.

```
Bezeichnung:   Input o1
Vorbedingung:
Eingabe:       o1?x; o2?
```

```
Ausgabe:        o!lastTypeElement(Cons(x,bufferVar))
Zuweisung:      bufferVar = deleteLastType(Cons(x,bufferVar))

Bezeichnung:    Input o2
Vorbedingung:
Eingabe:        o1?; o2?x
Ausgabe:        o!lastTypeElement(Cons(x,bufferVar))
Zuweisung:      bufferVar = deleteLastType(Cons(x,bufferVar))

Bezeichnung:    Simultaneous Input 1
Vorbedingung:
Eingabe:        o1?x; o2?y
Ausgabe:        o!lastTypeElement(Cons(y,Cons(x,bufferVar)))
Zuweisung:      bufferVar =
                deleteLastType(Cons(y,Cons(x,bufferVar)))

Bezeichnung:    Simultaneous Input 2
Vorbedingung:
Eingabe:        o1?x; o2?y
Ausgabe:        o!lastTypeElement(Cons(x,Cons(y,bufferVar)))
Zuweisung:      bufferVar =
                deleteLastType(Cons(x,Cons(y,bufferVar)))

Bezeichnung:    No Input Buffer Not Empty
Vorbedingung:   typeListLength(bufferVar) > 0
Eingabe:        o1?; o2?
Ausgabe:        o!lastTypeElement(Cons(x,bufferVar))
Zuweisung:      bufferVar = deleteLastType(Cons(x,bufferVar))
```

D. Kombinierte Betrachtung von eins zu zwei Kommunikation

Existiert in einem zeitasynchronen AutoFocus Modell eine eins zu zwei Kommunikationsbeziehung zwischen zwei Eingabe-Ports, dann können die auf diesen Eingabe-Ports auftretenden Eingabeströme auf Grund der Eigenschaften dieser speziellen Kommunikationsform zu einem Eingabestrom zusammengefasst werden. Wir definieren eine stromverarbeitende Funktion *combine12Comm*, die zwei in eins zu zwei Kommunikationsbeziehung stehende Eingabeströme zu einem Strom zusammenfasst. Diese Funktion wird in dieser Arbeit zum Verhaltensinvarianznachweis des *Split Component* Refactorings in dem Abschnitt 7.5.3 eingesetzt. Dort wird der Einsatz der *combine12Comm* Funktion ebenfalls motiviert.

In diesem Abschnitt wird die *combine12Comm* Funktion formal als Focus stromverarbeitende Funktion definiert. Die Funktion fasst zwei Eingabeströme zusammen. Zunächst werden in den beiden Eingabeströmen entsprechend der Abstraktion α_{NI} (siehe Definition 3.16 auf Seite 69) nur verarbeitete Eingaben berücksichtigt. Zur Durchführung dieser Abstraktion werden neben den eigentlichen beiden Eingabeströmen auch die speziellen, die Verarbeitung dieser Eingaben anzeigenden, Ausgabeströme als Parameter der *combine12Comm* Funktion übergeben. Auf Basis der nur die verarbeiteten Eingaben berücksichtigenden Eingabeströme wird jeweils die zeitlich früher auftretende Nutznachricht zweier in beiden Strömen vorkommenden äquivalenten Nutznachrichten berücksichtigt. Zur Realisierung der *combine12Comm* Funktion wird eine Hilfsfunktion *combineRec* verwendet, die rekursiv aufgerufen wird und jeweils zusätzlich zu den zu betrachtenden Restströmen der beiden Eingabeströme zwei Zähler übergibt, die die Anzahlen der in den jeweils beiden Strömen bisher beobachteten Nutznachrichten enthalten. Die Funktion betrachtet hierbei jeweils Nachrichten in beiden Eingabeströmen, die zueinander gleichzeitig auftreten.

Die *combine12Comm* Funktion wird auf zwei konkrete Eingabeströme sowie deren die Verarbeitung von Eingaben anzeigenden Strömen aufgerufen und ist wie folgt definiert:

$$combine12Comm \in I^{\underline{\omega},\mathsf{ts}} \times I^{\underline{\omega},\mathsf{ts}} \times (Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}} \times$$

$$(Signal \cup VoidDT)^{\underline{\omega},\mathsf{ts}} \rightarrow I^{\underline{\omega},\mathsf{ts}}$$

where *combine12Comm* so that

$$\forall s_1, s_2 \in I^{\underline{\omega},\text{ts}}, \forall v_1, v_2 \in (Signal \cup VoidDT)^{\underline{\omega},\text{ts}} :$$

$$combine12Comm(s_1, s_2, v_1, v_2) = CombineRec(\alpha_{NI}(s_1, v_1), 0, \alpha_{NI}(s_2, v_2), 0) \quad \text{(D.1)}$$

Die von der *combine12Comm* Funktion verwendete *combineRec* Funktion sind wie folgt definiert:

$$X = I \setminus VoidDT$$

$$Y = \{(a, b) \in \mathbb{N}_0 \times \mathbb{N}_0 \text{ so that } a \geq b\}$$

$$Z = \{(c, d) \in \mathbb{N}_0 \times \mathbb{N}_0 \text{ so that } c > d\}$$

$$combineRec \in I^{\underline{\omega},\text{ts}} \times \mathbb{N}_0 \times I^{\underline{\omega},\text{ts}} \times \mathbb{N}_0 \rightarrow I^{\underline{\omega},\text{ts}}$$

$$\text{where } combineRec \text{ so that } \forall x, y \in \mathbb{N}_0, (a, b) \in Y, (c, d) \in Z, w, z \in X, s_1, s_2 \in I^{\underline{\omega},\text{ts}} :$$

Fall 1: In beiden Eingabeströmen ist zum betrachteten Zeitpunkt keine Nutznachricht vorhanden.

$$combineRec(void\&s_1, x, void\&s_2, y) = void\&combineRec(s_1, x, s_2, y) \quad \text{(D.2)}$$

Fall 2: In beiden Eingabeströmen ist an der betrachteten Stelle ein Focus Tick Symbol.

$$combineRec(\checkmark\&s_1, x, \checkmark\&s_2, y) = \checkmark\&combineRec(s_1, x, s_2, y) \quad \text{(D.3)}$$

Fall 3: Im ersten Eingabestrom ist an der betrachteten Stelle eine Nutznachricht vorhanden und an der selben Stelle in dem zweiten Strom ist keine Nutznachricht vorhanden. In dem zweiten Strom ist die zu der Nutznachricht im ersten Strom äquivalente Nutznachricht noch nicht verarbeitet worden. Folglich ist die im ersten Strom gefundene Nutznachricht zeitlich vor der äquivalenten Nutznachricht im zweiten Strom. Die Nutznachricht wird zu dem betrachteten Zeitpunkt in den Ergebnisstrom aufgenommen.

$$combineRec(w\&s_1, a, void\&s_2, b) = w\&combineRec(s_1, a + 1, s_2, b) \quad \text{(D.4)}$$

Fall 4: Im ersten Eingabestrom ist an der betrachteten Stelle eine Nutznachricht vorhanden und an der selben Stelle in dem zweiten Strom ebenfalls eine Nutznachricht vorhanden. Die Nutznachricht in Strom eins ist zeitlich vor oder gleichzeitig mit der äquivalenten Nutznachricht in Strom zwei. Die Nutznachricht des ersten Stroms wird zu dem betrachteten Zeitpunkt in den Ergebnisstrom aufgenommen und die Nutznachricht des zweiten Stroms wird ignoriert.

$$combineRec(w\&s_1, a, z\&s_2, b) = w\&combineRec(s_1, a+1, s_2, b+1) \quad \text{(D.5)}$$

Fall 5: Im zweiten Eingabestrom wird eine Nutznachricht identifiziert, die zeitlich vor der äquivalenten Nutznachricht in Strom eins auftritt. Zum betrachteten Zeitpunkt ist keine Nutznachricht in Strom eins vorhanden.

$$combineRec(void\&s_1, b, z\&s_2, a) = z\&combineRec(s_1, b, s_2, a+1) \quad \text{(D.6)}$$

Fall 6: Im ersten Eingabestrom ist an der betrachteten Stelle eine Nutznachricht vorhanden und an der selben Stelle in dem zweiten Strom ebenfalls eine Nutznachricht vorhanden. Die Nutznachricht in Strom zwei ist zeitlich vor der äquivalenten Nutznachricht in Strom eins. Die Nutznachricht des zweiten Stroms wird zu dem betrachteten Zeitpunkt in den Ergebnisstrom aufgenommen und die Nutznachricht des ersten Stroms wird ignoriert.

$$combineRec(w\&s_1, d, z\&s_2, c) = z\&combineRec(s_1, d+1, s_2, c+1) \quad \text{(D.7)}$$

Fall 7: An der betrachteten Stelle in Strom eins befindet sich keine Nutznachricht und in Strom zwei eine Nutznachricht. Zu dieser Nutznachricht existiert jedoch eine äquivalente Nachricht im ursprünglichen Strom eins, die zeitlich zuvor auftritt. Die Nutznachricht in Strom zwei wird folglich ignoriert.

$$combineRec(void\&s_1, c, z\&s_2, d) = void\&combineRec(s_1, c, s_2, d+1) \quad \text{(D.8)}$$

Fall 8: An der betrachteten Stelle in Strom eins befindet sich eine Nutznachricht und in Strom zwei keine Nutznachricht. Zu der gefundenen Nutznachricht existiert jedoch eine äquivalente Nachricht im ursprünglichen Strom zwei, die zeitlich zuvor auftritt. Die Nutznachricht in Strom eins wird folglich ignoriert.

$$combineRec(w\&s_1, d, void\&s_2, c) = void\&combineRec(s_1, d+1, s_2, c) \quad \text{(D.9)}$$

Fall 9: Die Rekursion terminiert, falls die Funktion auf leeren Strömen aufgerufen wird.

$$combineRec(\langle\rangle, x, \langle\rangle, y) = \langle\rangle \quad \text{(D.10)}$$

Literaturverzeichnis

[Ast02] Astels, Dave: Refactoring With UML. In: Marchesi, Michele (Hrsg.) ; Succi, Giancarlo (Hrsg.): *Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*. Alghero, Sardinia, Italy, Mai 2002, S. 67–70

[Aut06a] AutoFocus Team: *AutoFocus 2 Code Documentation*. `http://www4.in.tum.de/~af2/api`. Version: Januar 2006

[Aut06b] AutoFocus Team: *AutoFocus Model API*. `http://autofocus.in.tum.de`. Version: Januar 2006. – Bestandteil der AutoFocus Distribution

[BBB+85] Bauer, F. L. ; Berghammer, R. ; Broy, M. ; Dosch, W. ; Geiselbrechtinger, F. ; Gnatz, R. ; Hangel, E. ; Hesse, W. ; Krieg-Brückner, B. ; Laut, A. ; Matzner, T. ; Möller, B. ; Nickl, F. ; Partsch, H. ; Pepper, P. ; Samelson, K. ; Wirsing, M. ; Wössner, H.: *The Munich Project CIP, Vol. 1: The Wide Spectrum Language CIP-L*. Springer Verlag, 1985 (Lecture Notes in Computer Science (LNCS) 183)

[BD03] Brügge, Bernd ; Dutoit, Allen H.: *Object-Oriented Software Engineering: Using UML, Patterns and Java*. 2. Prentice Hall, 2003. – ISBN 0–13–047110–0

[BDD+92] Broy, Manfred ; Dederich, Frank ; Dendorfer, Claus ; Fuchs, Max ; Gritzner, Thomas ; Weber, Rainer: The Design of Distributed Systems - An Introduction to FOCUS / Institut für Informatik, Technische Universität München. 1992 (TUM-I9202). – Forschungsbericht. `http://www4.in.tum.de/~broy/papers/TUM-I9202.pdf`

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. – ISBN 0201616416

[Ben03] Bender, Katrin: *Metriken zur Bewertung der Güte von AutoFocus Modellen*, Fakultät für Informatik, Technische Universität München, Diplomarbeit, 2003

[Ber91] Bergstein, Paul L.: Object-Preserving Class Transformations. In: Paepcke, Andreas (Hrsg.): *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91), Sixth Annual Conference, 6-11 October 1991, Phoenix, Arizona, USA, Proceedings*, 1991 (SIGPLAN Notices 26/11)

[Ber97] BERGSTEIN, Paul L.: Maintenance of Object-Oriented Systems During Structural Evolution. In: *Theory and Practice of Object Systems* 3 (1997), Nr. 3, S. 185–212

[Ber98] BERRY, Gérard: The Foundations of Esterel. In: PLOTKIN, G. (Hrsg.) ; STIRLING, C. (Hrsg.) ; TOFTE, M. (Hrsg.): *Proof, Language and Interaction: Essays in Honour of Robin Milner*, MIT Press, 1998

[BHS99] BROY, Manfred ; HUBER, Franz ; SCHÄTZ, Bernhard: AutoFocus – Ein Werkzeugprototyp zur Entwicklung Eingebetteter Systeme. In: *Informatik Forschung und Entwicklung* 14 (1999), Nr. 3, S. 121–134

[BJK+05] BROY, Manfred ; JONSSON, Bengt ; KATOEN, Joost-Pieter ; LEUCKER, Martin ; PRETSCHNER, Alexander: *Model-Based Testing of Reactive Systems*. Springer Verlag, 2005 (Lecture Notes in Computer Science (LNCS) 3472). – ISBN 3–540–26278–4

[BLS00] BRAUN, Peter ; LÖTZBEYER, Heiko ; SLOTOSCH, Oscar: *Quest Users Guide*. Fakultät für Informatik, Technische Universität München, 2000. `http://www4.in.tum.de/proj/quest/papers/UserGuide.pdf`

[BM02] BRAUN, Peter ; MARSCHALL, Frank: Transforming Object Models with BOTL. In: *Electronic Notes on Theoretical Computer Science* Bd. 72, 2002

[BMR+96] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-oriented Software Architecture: A System of Patterns*. Bd. 1. John Wiley & Sons, 1996. – ISBN 0471958697

[BPPT03] BOTTONI, Paolo ; PARISI-PRESICCE, Francesco ; TAENTZER, Gabriele: Coordinated Distributed Diagram Transformation for Software Evolution. In: *Electronic Notes on Theoretical Computer Science (ENTCS)* 72 (2003), Nr. 4

[Bro93] BROY, Manfred: Functional Specification of Time Sensitive Communicating Systems. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2 (1993), Nr. 1, S. 1–46

[Bro95] BROY, Manfred: Characterizing the Behavior of Reactive Systems by Trace Sets / Fakultät für Informatik, Technische Universität München. 1995 (TUM-I9102). – Forschungsbericht. `http://wwwbroy.informatik.tu-muenchen.de/publ/papers/TUM-I9102.pdf`

[Bro97] BROY, Manfred: Refinement of Time. In: BERTRAN, M. (Hrsg.) ; RUS, Th. (Hrsg.): *Transformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Palma, Mallorca, Spain, May 21-23, 1997, Proceedings*, Springer Verlag, 1997 (Lecture Notes in Computer Science (LNCS) 1231), S. 44–63

[Bro01] BROY, Manfred: Refinement of time. In: *Journal of Theroetical Computer Science* 253 (2001), Nr. 1, S. 3–26

[Bro03] BROY, Manfred: Abstractions from time. In: MCIVER, A. (Hrsg.) ; MORGAN, C. (Hrsg.): *Programming Methodology. Monographs in Computer Science*, Springer-Verlag New York, Inc., 2003. – ISBN 0–387–95349–3, S. 95–107

[Bro04] BROY, Manfred: Time, Abstraction, Causality and Modularity in Interactive Systems: Extended Abstract. In: *Electronic Notes in Theoretical Computer Science, Proceedings of the First International Workshop on Formal Foundations of Embedded Software and Component-based Software Architectures FESCA 2004* 108 (2004), S. 3–9

[BS01] BROY, Manfred ; STØLEN, Ketil: *Specification and Development of Interactive Systems – Focus on Streams, Interfaces, and Refinement*. Springer-Verlag New York, Inc., 2001 (Monographs in Computer Science). – ISBN 0–387–95073–7

[BS04] BROY, Manfred ; STEINBRÜGGEN, Ralf: *Modellbildung in der Informatik*. Springer Verlag, 2004. – ISBN 3–540–44292–8

[BSF02] BOGER, Marko ; STURM, Thorsten ; FRAGEMANN, Per: Refactoring Browser for UML. In: MARCHESI, Michele (Hrsg.) ; SUCCI, Giancarlo (Hrsg.): *Proceedings of the 3nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP2002)*. Alghero, Sardinia, Italy, Mai 2002, S. 77–81

[BT95] BATORY, Don ; TOKUDA, Lance: Automated Software Evolution via Design Pattern Transformations / The University of Texas at Austin, Department of Computer Sciences. 1995 (CS-TR-95-06). – Technical Report. `ftp://ftp.cs.utexas.edu/pub/techreports/tr95-06.ps.gz`

[BW82] BAUER, Friedrich L. ; WÖSSNER, Hans: *Algorithmic Language and Program Development.* Springer Verlag, 1982

[Cas94] CASAIS, Eduardo: Automatic Reorganization of Object-Oriented Hierarchies: A Case Study. In: *Object-Oriented Systems* 1 (1994), Dezember, Nr. 2, S. 95–115

[CGP00] CLARKE, Edmund M. ; GRUMBERG, Orna ; PELED, Doron A.: *Model Checking*. MIT Press, 2000. – ISBN 0262032708

[CK94] CHIDAMBER, Shyam R. ; KEMERER, Chris F.: A metrics Suite for Object Oriented Design. In: *IEEE Transactions on Software Engineering* 20 (1994), Nr. 6, S. 476–493

[Com05] Common Criteria Consortium: *Common Criteria for Information Technology Security Evaluation*. Version 2.3. August 2005. `http://www.commoncriteriaportal.org/public/expert/index.php?menu=2`

[Dij71] DIJKSTRA, Edsger W.: Structured Programming. In: HOARE, C.A.R. (Hrsg.) ; DAHL, O. (Hrsg.) ; DIJKSTRA, E.W. (Hrsg.): *Notes on Structured Programming*, Academic Press, 1971

[Dou99] DOUGLASS, Bruce P.: *Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999 (Object Technology Series). – ISBN 0–201–49837–5

[EHKG02] ENGELS, Gregor ; HECKEL, Reiko ; KÜSTER, Jochen M. ; GROENEWEGEN, Luuk: Consistency-Preserving Model Evolution through Transformations. In: JÉZÉQUEL, Jean-Marc (Hrsg.) ; HUSSMANN, Heinrich (Hrsg.) ; COOK, Stephen (Hrsg.): *UML 2002 - The Unified Modeling Language. Model Engineering, Languages, Concepts, and Tools. 5th International Conference, Dresden, Germany, September/October 2002, Proceedings*, Springer Verlag, 2002 (Lecture Notes in Computer Science (LNCS) 2460), S. 212–226

[FBB+99] FOWLER, Martin ; BECK, Kent ; BRANT, John ; OPDYKE, William ; ROBERTS, Don: *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999 (Object Technology Series). – ISBN 0–201–48567–2

[FGSK03] FRANCE, Robert ; GHOSH, Sudipto ; SONG, Eunjee ; KIM, Dae-Kyoo: A Metamodeling Approach to Pattern-Based Model Refactoring. In: *IEEE-SOFTWARE* 20 (2003), September/Oktober, Nr. 5, S. 52–58. – ISSN 0740–7459

[FS03] FOWLER, Martin ; SCOTT, Kendall: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3. Addison-Wesley, 2003. – ISBN 0321193687

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph E. ; VLISSIDES, John: *Design Patterns*. Addison-Wesley, 1995. – ISBN 0201633612

[GJ03] GARRIDO, Alejandra ; JOHNSON, Ralph: Refactoring C with Conditional Compilation. In: *18th IEEE International Conference on Automated Software Engineering (ASE 2003), 6-10 October 2003, Montreal, Canada*, 2003. – ISBN 0–7695–2035–9, S. 323–326

[GSMD03] GORP, Pieter V. ; STENTEN, Hans ; MENS, Tom ; DEMEYER, Serge: Towards automating source-consistent UML Refactorings. In: *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, Springer Verlag, 2003 (Lecture Notes in Computer Science (LNCS) 2863). – ISBN 3–540–20243–9

[Har87] HAREL, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Nr. 3, S. 231–274

[Hec95] HECKEL, Reiko: *Algebraic graph transformation with application conditions*, Technische Universität Berlin, Diplomarbeit, 1995

[HMS+98] HUBER, Franz ; MOLTERER, Sascha ; SCHÄTZ, Bernhard ; SLOTOSCH, Oscar ; VILBIG, Alexander: Traffic Lights - An AutoFocus Case Study. In: *1st International Conference on Application of Concurrency to System Design (ACSD '98), 23-26 March 1998, Fukushima, Japan*, IEEE Computer Society, 1998. – ISBN 0–8186–8350–3, S. 282ff

[HN96] HAREL, David ; NAAMAD, Amnon: The STATEMATE Semantics of Satecharts. In: *ACM Transactions on Software Engineering and Methodology* 5 (1996), Oktober, Nr. 4, S. 293–333

[Hol97] HOLZMANN, Gerard J.: The Model Checker Spin. In: *IEEE Transactions on Software Engineering* 23 (1997), Mai, Nr. 5, S. 279–295

[Hol03] HOLZMANN, Gerard J.: *The Spin Model Checker, Primer and Reference Manual*. Addison-Wesley, 2003. – ISBN 0321228626

[HP85] HAREL, David ; PNUELI, Amir: On the development of reactive systems. In: APT, Krzysztof R. (Hrsg.): *Logics and models of concurrent systems*, Springer Verlag, 1985 (NATO ASI Series 13). – ISBN 0–387–15181–8, S. 477–498

[HP02] HOUDEK, Frank ; PAECH, Barbara: Das Türsteuergerät - eine Beispielspezifikation / Fraunhofer Institut Experimentelles Software Engineering (IESE). 2002 (IESE-Report Nr. 002.02/D). – Technical Report

[HSE97] HUBER, Franz ; SCHÄTZ, Bernhard ; EINERT, Geralf: Consistent Graphical Specification of Distributed Systems. In: FITZGERALD, John (Hrsg.) ; JONES, Cliff B. (Hrsg.) ; LUCAS, Peter (Hrsg.): *FME '97: 4th International Symposium of Formal Methods Europe*, Springer Verlag, 1997 (Lecture Notes in Computer Science (LNCS) 1313), S. 122–141

[HU94] HOPCROFT, John E. ; ULLMAN, Jeffrey D.: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie*. Addison-Wesley, 1994

[ITU02] International Telecommunication Union (ITU): *Specification and Description Language*. 2002. `http://www.itu.int/ITU-T/studygroups/com17/languages/Z100.pdf`

[KS03] KOF, Leonid ; SCHÄTZ, Bernhard: Combining Aspects of Reactive Systems. In: BROY, Manfred (Hrsg.) ; ZAMULIN, Alexandre V. (Hrsg.): *Perspectives of Systems Informatics 5th International Andrei Ershov Memorial Conference, PSI 2003, Akademgorodok, Novosibirsk, Russia, July 9-12, 2003, Revised Papers.*, Springer Verlag, 2003 (Lecture Notes in Computer Science (LNCS) 2890). – ISBN 3–540–20813–5

[KSTW04] KOF, Leonid ; SCHÄTZ, Bernhard ; THALER, Ingomar ; WISSPEINTNER, Alexander: Service-Based Development of Embedded Systems. In: *Proceedings of Net.Object Days 2004, Objektorientierte Software-Enwicklung OOSE Workshop,*

27.-30. September 2004, Erfurt, Germany, 2004. – ISBN 3–9808628–3–6, S. 171–185

[Lam93] LAMPORT, Leslie: Verification and Specification of Concurrent Programs. In: BAKKER, J.W.de (Hrsg.) ; ROEVER, W.-P.de (Hrsg.) ; G.ROZENBERG (Hrsg.): *A Decade of Concurrency - Reflexions and Perspectives*, Springer Verlag, 1993 (Lecture Notes in Computer Science (LNCS) 803), S. 347–374

[LBM+01] LÉDECZI, Ákos ; BAKAY, Arpad ; MAROTI, Miklos ; VÖLGYESI, Péter ; NORDSTROM, Greg ; SPRINKLE, Jonathan ; KARSAI, Gabor: Composing Domain-Specific Design Environments. In: *IEEE Computer* 34 (2001), Nr. 11, S. 44–51

[LPS+02] LÖTZBEYER, Heiko ; PRENNINGER, Wolfgang ; SLOTOSCH, Oscar ; STEINBRÜGGEN, Ralf ; STRÖSE, Thomas ; WINHARD, Ferdinand: *Einführung in die Systemmodellierung mit AutoFocus*. Fakultät für Informatik, Technische Universität München, 2002. http://autofocus.in.tum.de/nelli/html/

[LRT03] LI, Huiqing ; REINKE, Claus ; THOMPSON, Simon: Tool support for refactoring functional programs. In: *Proceedings of the ACM SIGPLAN workshop on Haskell (HASKELL-03)*. New York : ACM Press, August 2003, S. 27–38

[Mar05] MARSCHALL, Frank: *Modelltransformationen als Mittel der modellbasierten Entwicklung von Software-Systemen*, Fakultät für Informatik, Technische Universität München, Dissertation, 2005

[McM92] MCMILLAN, Ken L.: *Symbolic Model Checking: An Approach to the State Explosion Problem*. Pittsburgh, PA, School of Computer Science, Carnegie Mellon University, PhD Thesis, Mai 1992. http://embedded.eecs.berkeley.edu/Alumni/kenmcmil/thesis.ps. – Online–Ressource. – CMU-CS-92-131

[MDJ02] MENS, Tom ; DEMEYER, Serge ; JANSSENS, Dirk: Formalising Behaviour Preserving Program Transformations. In: CARRADINI, A. (Hrsg.) ; EHRIG, H. (Hrsg.) ; KREOWSKI, H.-J. (Hrsg.) ; ROZENBERG, G. (Hrsg.): *International Conference on Graph Transformations (ICGT)*, Springer Verlag, 2002 (Lecture Notes in Computer Science (LNCS) 2505)

[MEDJ05] MENS, Tom ; EETVELDE, Niels V. ; DEMEYER, Serge ; JANSSENS, Dirk: Formalizing refactorings with graph transformations. In: *Journal of Software Maintenance and Evolution: Research and Practice* 17 (2005), Nr. 4, S. 247–276

[Moo96] MOORE, Ivan: Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In: *Proceedings of the 1996 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '96), San Jose, California, October 6-10 1996*, 1996 (SIGPLAN Notices 31/10), S. 235–250

[MT04] MENS, Tom ; TOURWÉ, Tom: A Survey of Software Refactoring. In: *IEEE Transactions on Software Engineering (IEEETSE)* 30 (2004)

[OMG02] Object Modeling Group (OMG): *Meta-Object Facility 1.4 (MOF)*. 2002. http://www.omg.org/cgi-bin/apps/doc?formal/02-04-03.pdf

[OMG03a] Object Management Group (OMG): *MDA Guide Version 1.0.1*. Juni 2003. http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf

[OMG03b] Object Modeling Group (OMG): *UML 2.0 Object Constraint Language (OCL) Specification*. 2003. http://www.omg.org/docs/ptc/03-10-14.pdf

[OMG04] Object Modeling Group (OMG): *UML 2.0 Superstructure Specification*. 2004. http://www.omg.org/cgi-bin/apps/doc?formal/05-07-04.pdf

[Opd92] OPDYKE, William F.: *Refactoring Object-Oriented Frameworks*. Urbana-Champaign, IL, USA, University of Illinois at Urbana-Champaign, PhD Thesis, 1992

[Por03] PORRES, Ivan: Model Refactorings as Rule-Based Update Transformations. In: STEVENS, Perdita (Hrsg.) ; WHITTLE, Jon (Hrsg.) ; BOOCH, Grady (Hrsg.): *UML 2003 - The Unified Modeling Language, Modeling Languages and Applications, 6th International Conference, San Francisco, CA, USA, October 20-24, 2003, Proceedings*, Springer Verlag, 2003 (Lecture Notes in Computer Science (LNCS) 2863). – ISBN 3–540–20243–9, S. 159–174

[PR97] PHILIPPS, Jan ; RUMPE, Bernhard: Refinement of Information Flow Architectures. In: *First IEEE International Conference on Formal Engineering Methods, ICFEM 1997, November 12-14, 1997, Hiroshima, Japan, Proceedings*, IEEE Computer Society, 1997, S. 203–212. – Electronic Publication

[PR99] PHILIPPS, Jan ; RUMPE, Bernhard: Refinement of Pipe-and-Filter Architectures. In: WING, Jeannette M. (Hrsg.) ; WOODCOCK, Jim (Hrsg.) ; DAVIES, Jim (Hrsg.): *FM'99 - Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, France, September 20-24, 1999, Proceedings, Volume I*, Springer Verlag, 1999 (Lecture Notes in Computer Science (LNCS) 1708). – ISBN 3–540–66587–0, S. 96–115

[PR01] PHILIPPS, Jan ; RUMPE, Bernhard: Roots of Refactoring. In: BACLAVSKI, Kenneth (Hrsg.) ; KILOV, Haim (Hrsg.): *Proceedings of 10th OOPSLA Workshop on Behavioral Semantics: Back to Basics*. Tampa Bay, Florida, USA : Northeastern University Press, Oktober 2001, S. 187–199

[PR03] PHILIPPS, Jan ; RUMPE, Bernhard: Refactoring of Programs and Specifications. In: KILOV, H. (Hrsg.) ; BACLAWSKI, K. (Hrsg.): *Practical foundations of business and system specifications*. Kluwer Academic Publishers, 2003, S. 281–297

[Pre03] PRETSCHNER, Alexander: *Zum modellbasierten funktionalen Test Reaktiver Systeme*, Fakultät für Informatik, Technische Universität München, Dissertation, August 2003. `http://tumb1.biblio.tu-muenchen.de/publ/diss/in/2003/pretschner.pdf`. – Online–Ressource

[Pre05] PRENNINGER, Wolfgang Ludwig J.: *Inkrementelle Entwicklung von Verhaltensmodellen zum Test von Reaktiven Systemen*, Fakultät für Informatik, Technische Universität München, Dissertation, Juli 2005

[PS01] PRETSCHNER, Alexander ; SCHÄTZ, Bernhard: Modellbasiertes Testen mit AutoFocus/Quest. In: *Softwaretechnik-Trends* 21 (2001), Februar, Nr. 1, S. 20–23

[PTLP99] PROWELL, Stacy J. ; TRAMMELL, Carmen J. ; LINGER, Richard C. ; POORE, Jesse H.: *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, 1999. – ISBN 0201854805

[RB03] RUI, Kexing ; BUTLER, Gregory: Refactoring Use Case Models: The Metamodel. In: OUDSHOORN, Michael J. (Hrsg.): *Computer Science 2003, Twenty-Sixth Australasian Computer Science Conference (ACSC2003), Adelaide, South Australia, February 2003*, Australian Computer Society, 2003 (CRPIT 16). – ISBN 0–909–92594–1, S. 301–308

[RBJ97] ROBERTS, Don ; BRANT, John ; JOHNSON, Ralph E.: A Refactoring Tool for Smalltalk. In: *Theory and Practice of Object Systems (TAPOS)* 3 (1997), Nr. 4, S. 253–263

[RJB04] RUMBAUGH, James ; JACOBSON, Ivar ; BOOCH, Grady: *The Unified Modeling Language Reference Manual*. 2. Addison-Wesley, 2004. – ISBN 0321245628

[Rob99] ROBERTS, Don: *Practical Analysis for Refactoring*, University of Illinois, PhD Thesis, 1999

[Rum96] RUMPE, Bernhard: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*, Fakultät für Informatik, Technische Universität München, Dissertation, 1996

[Rum98] RUMPE, Bernhard: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. In: *Ausgezeichnete Informatikdissertationen 1997*, B. G. Teubner Stuttgart, 1998

[SBHW03] SCHÄTZ, Bernhard ; BRAUN, Peter ; HUBER, Franz ; WISSPEINTNER, Alexander: Consistency in Model-Based Development. In: *10th IEEE International Conference on Engineering of Computer-Based Systems (ECBS 2003), 7-10 April 2003, Huntsville, AL, USA*, IEEE Computer Society, 2003. – ISBN 0–7695–1917–2

[SBHW05] SCHÄTZ, Bernhard ; BRAUN, Peter ; HUBER, Franz ; WISSPEINTNER, Alexander: Checking and Transforming Models with AutoFOCUS. In: *12th IEEE International Conference on the Engineering of Computer-Based Systems (ECBS 2005), 4-7 April 2005, Greenbelt, MD, USA*, IEEE Computer Society, 2005. – ISBN 0–7695–2308–0, S. 307–314

[Sch98] SCHOLZ, Peter: *Design of Reactive Systems and their Distributed Implementation with Statecharts*, Fakultät für Informatik, Technische Universität München, Dissertation, 1998

[Sch01] SCHÄTZ, Bernhard: The ODL Operation Definition Language and the AutoFocus/Quest Application Framework AQuA / Fakultät für Informatik, Technische Universität München. 2001 (TUM-I0111). – Technischer Bericht. `http://wwwbib.informatik.tu-muenchen.de/infberichte/2001/TUM-I0111.ps`

[SFGP05a] SCHÄTZ, Bernhard ; FLEISCHMANN, Andreas ; GEISBERGER, Eva ; PISTER, Markus: Model-Based Requirements Engineering with AutoRAID. In: CREMERS, Armin B. (Hrsg.) ; MANTHEY, Rainer (Hrsg.) ; MARTINI, Peter (Hrsg.) ; STEINHAGE, Volker (Hrsg.): *INFORMATIK 2005 - Informatik LIVE! Band 2, Beiträge der 35. Jahrestagung der Gesellschaft für Informatik e.V. (GI), Bonn, 19. bis 22. September 2005*, Gesellschaft für Informatik Verlag, 2005 (LNI 68). – ISBN 3–88579–397–0, S. 511–515

[SFGP05b] SCHÄTZ, Bernhard ; FLEISCHMANN, Andreas ; GEISBERGER, Eva ; PISTER, Markus: Modellbasierte Anforderungsentwicklung. In: *Workshop Object-Orientied Software-Engineering (OOSE), NetObjectDays 2005, 19.-22. September 2005, Proceedings*, 2005. – ISBN 3–9808628–4–4

[SGMZ98] SCHULZ, Benedikt ; GENSSLER, Thomas ; MOHR, Berthold ; ZIMMER, Walter: On the Computer Aided Introduction of Design Patterns into Object-Oriented Systems. In: *Proceedings of the TOOLS 27 Conference (Asia '98)*, IEEE Computer Society Press, 1998

[SGW94] SELIC, Bran ; GULLEKSON, Garth ; WARD, Paul T.: *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994. – ISBN 0–471–59917–4

[SHH+03] SCHÄTZ, Bernhard ; HAIN, Tobias ; HOUDEK, Frank ; PRENNINGER, Wolfgang ; RAPPL, Martin ; ROMBERG, Jan ; SLOTOSCH, Oscar ; STRECKER, Martin ; WISSPEINTNER, Alexander: CASE Tools for Embedded Systems / Fakultät für Informatik, Technische Universität München. 2003 (TUM-I0309). – Technischer Bericht. `http://wwwbib.informatik.tu-muenchen.de/infberichte/2003/TUM-I0309.pdf`

[Slo00] SLOTOSCH, Oscar: Modelling and Validation: AutoFocus and Quest. In: *Formal Aspects of Computing* 12 (2000), Nr. 4, S. 225–227

[SPHP02] SCHÄTZ, Bernhard ; PRETSCHNER, Alexander ; HUBER, Franz ; PHILIPPS, Jan: Model-Based Development of Embedded Systems. In: BRUEL, Jean-Michel (Hrsg.) ; BELLAHSENE, Zohra (Hrsg.): *Advances in Object-Oriented Information Systems, OOIS 2002 Workshops, Montpellier, France, September 2, 2002, Proceedings*, Springer Verlag, 2002 (Lecture Notes in Computer Science (LNCS) 2426). – ISBN 3–540–44088–7, S. 298–312

[SPTJ01] SUNYÉ, Gerson ; POLLET, Damien ; TRAON, Yves L. ; JÉZÉQUEL, Jean-Marc: Refactoring UML Models. In: GOGOLLA, Martin (Hrsg.) ; KOBRYN, Cris (Hrsg.): *UML 2001 - The Unified Modeling Language. Modeling Languages, Concepts, and Tools. 4th International Conference, Toronto, Canada*, Springer Verlag, Oktober 2001 (Lecture Notes in Computer Science (LNCS) 2185), S. 134–148

[SPW04] SCHÄTZ, Bernhard ; PISTER, Markus ; WISSPEINTNER, Alexander: Anforderungsanalyse in der modellbasierten Entwicklung am Beispiel von AutoFocus. In: *Softwaretechnik-Trends* 24 (2004), Nr. 1

[SRS+03] SCHÄTZ, Bernhard ; ROMBERG, Jan ; SLOTOSCH, Oscar ; STRECKER, Martin ; WISSPEINTNER, Alexander ; HAIN, Tobias ; PRENNINGER, Wolfgang ; RAPPL, Martin ; SPIES, Katharina: Modeling Embedded Software: State of the Art and Beyond. In: *Proceedings of 16th International Conference on Software and Systems Engineering and their Applications ICSSEA 2003, Paris, France, 2. - 4. December*, 2003

[SSL01] SIMON, Frank ; STEINBRÜCKNER, Frank ; LEWERENTZ, Claus: Metrics Based Refactoring. In: SOUSA, Pedro (Hrsg.) ; EBERT, Jürgen (Hrsg.): *Proceedings of the Fifth Conference on Software Maintenance and Reengineering, CSMR 2001, 14-16 March 2001, Lisbon, Portugal*, IEEE Computer Society, 2001. – ISBN 0–7695–1028–0, S. 30–38

[Ste00] STEFANESCU, Gheorghe: *Network algebra*. Springer Verlag, 2000. – ISBN 185233195X

[SW99] STROBL, Frank ; WISSPEINTNER, Alexander: Specification of an Elevator Control System – An AutoFocus Case Study / Fakultät für Informatik, Technische Universität München. 1999 (TUM-I9906). – Technischer Bericht. `http://wwwbib.informatik.tu-muenchen.de/infberichte/1999/TUM-I9906.ps`

[TDDN00] TICHELAAR, Sander ; DUCASSE, Stéphane ; DEMEYER, Serge ; NIERSTRASZ, Oscar: A Meta-model for Language-Independent Refactoring. In: *International Symposium on Principles of Software Evolution ISPSE 2000, Kanazawa, Japan*, IEEE, 2000, S. 157–167

[VWW02] VETTERLING, Monika ; WIMMEL, Guido ; WISSPEINTNER, Alexander: Secure Systems Development Based on the Common Criteria. In: *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering*

FSE 2002, Charleston, South Carolina, USA, November 18. - 22. 2002, ACM Press, 2002. – ISBN 1–58113–514–9

[WHP00] WISSPEINTNER, Alexander ; HUBER, Franz ; PHILIPPS, Jan: Model Checking and Random Competition A Study Using the Model Checking Framework MIC. In: GRABOWSKI, Jens (Hrsg.) ; HEYMER, Stefan (Hrsg.): *Formale Beschreibungstechniken für verteilte Systeme, 10. GI/ITG Fachgespräch*, Shaker Verlag, Juni 2000. – ISBN 3–8265–7491–5, S. 91–100

[Wiß99] WISSPEINTNER, Alexander: *Model-Checking Strategien mit MIC*, Fakultät für Informatik, Technische Universität München, Diplomarbeit, November 1999. http://www4.in.tum.de/~wisspein/publications/masters_thesis_mic.pdf. – Online–Ressource

[Wiß03] WISSPEINTNER, Alexander: *Refactoring AutoFocus Design Models*. 2003. – F. Tip (Org.), G. Snelting (Org.) und R. Johnson (Org.), Seminar Nr. 03091 Program Analysis for Object-Oriented Evolution, Dagstuhl 23.02.-28.02.2003, 2003. Foliensatz http://www.dagstuhl.de/files/Materials/03/03091/03091.WiszpeintnerAlexander.Slides.pdf – Online-Ressource

[Wir71] WIRTH, Niklaus: Program Development by Stepwise Refinement. In: *Communications of the ACM CACM* 14 (1971)

[Wüs99] WÜST, Christian: Vom Retter erschlagen. In: *Der Spiegel* (1999), Nr. 12

[WW01] WIMMEL, Guido ; WISSPEINTNER, Alexander: Extended Description Techniques for Security Engineering. In: DUPUY, Michel (Hrsg.) ; PARADINAS, Pierre (Hrsg.): *Trusted Information—The New Decade Challenge, IFIP TC11 16th International Conference on Information Security (IFIP/Sec'01), June 11-13, 2001, Paris, France*, Kluwer Academic Publishers, 2001. – ISBN 0–7923–7389–8, S. 470–485

[ZLG05] ZHANG, Jing ; LIN, Yuehua ; GRAY, Jeff: Generic and Domain-Specific Model Refactoring using a Model Transformation Engine. In: BEYDEDA, Sami (Hrsg.) ; BOOK, Matthias (Hrsg.) ; GRUHN, Volker (Hrsg.): *Model-driven Software Development - Research and Practice in Software Engineering*. Springer-Verlag, 2005, S. 199–218. – ISBN 354025613X