



UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik – Informatik – Mathematik

Institut für Informatik

Fachgebiet Softwaretechnik

Warburger Straße 100

33098 Paderborn

**Ein komponentenbasierter, modellgetriebener
Softwareentwicklungsansatz für vernetzte, mechatronische
Systeme**

Schriftliche Arbeit
zur Erlangung des Grades
„Doktor der Naturwissenschaften“

vorgelegt von

Dipl.-Inform. Stefan Henkler

Obernheideweg 13a

33106 Paderborn

Paderborn, im Juni 2012

Danksagung

An dieser Stelle möchte ich denjenigen Danken, die mich während meiner Zeit als Doktorand unterstützt haben. Beginnen möchte ich mit Prof. Wilhelm Schäfer, der es mir ermöglicht hat an seinem Lehrstuhl zu promovieren. Für die vielen fachlichen Gespräche und der Möglichkeit meine fußballerische Expertise praktisch sowie theoretisch zu vertiefen danke ich dir, Wilhelm!

Besonderem Dank gilt Prof. Holger Giese, der mich davon überzeugen konnte den Weg einer Promotion einzuschlagen. Während meiner gesamten Zeit als Doktorand wurde ich maßgeblich durch die vielen Diskussionen und gemeinsamen Veröffentlichungen von dir inspiriert.

Einen ganz besonderen Dank gilt der Prüfungskommission, dessen Teilnehmer es nicht gescheut haben aus Californien und Potsdam anzureisen. Ich Danke hiermit den Gutachtern Prof. Wilhelm Schäfer, Prof. Ingolf Krüger und Prof. Franz Rammig sowie den Kommissionsmitgliedern Prof. Steffen Becker, Prof. Gregor Engels, Prof. Holger Giese, Prof. Krüger und Prof. Schäfer.

An dieser Stelle möchte ich auch die Gelegenheit nutzen meinem langjährigen Bürokollegen Prof. Martin Hirsch zu danken. Diese Arbeit wäre ohne die zahlreichen Diskussionen, Publikationen und gerade die nicht fachlichen Gespräche mit dir in dieser Form nicht zustande gekommen. Danke Martin!

Meinen zahlreichen Ex-Kollegen am Lehrstuhl Softwaretechnik gilt ebenfalls besonderer Dank. Meine Arbeit wurde durch viele Diskussionen und gemeinsame Publikationen besonders unterstützt durch ((Co-) Autoren der Publikationen): Björn Axenath, Christian Brenner, Christoph Brink, Sven Burmester, Markus von Detten, Tobias Eckardt, Joel Grennyer, Christian Heinemann, Martin Hirsch, Renate Löffler (né Ristov), Jan Meyer, Claudia Priesterjahn, Vladimir Roubin, Andreas Seibel, Florian Stallmann (né Klein), Matthias Tichy und Dietrich Travkin. Für das Korrekturlesen dieser Arbeit möchte ich mich bei Steffen Becker, Nicola Danielzik, Markus von Detten, Martin Hirsch, Jan Meyer, Claudia Priesterjahn, Achim Rettberg und Matthias Tichy bedanken. Was wäre ein Lehrstuhlleben ohne Sekretariat und Administration? Jutta, Sammy herzlichen Dank für eure tolle Unterstützung während der gesamten Zeit! Ahmet, dir möchte ich für die vielen nicht inhaltlichen Diskussionen und Ratschläge Danken!

All meinen Ex-Kollegen aus dem SFB möchte für die exzellente interdisziplinäre Zusammenarbeit danken. Besonders möchte denen danken, die mich in den Arbeitskreisen und weiteren Gremien begleitet haben: Philipp Adelt, Jörg Donoth, Kathrin Flaßkamp, Jens Geisler, Sascha Kahl, Benjamin Klöpffer, Eckehard Münch, Simon Oberthür, Semir Osmic, Christoph Romaus, Alexander Schmidt, Bernd Schulz, Henner Vöcking und Katrin Witting.

Diese Arbeit wäre für mich nicht möglich gewesen ohne die Unterstützung meiner Familie. Meine Eltern haben mir die Ausdauer mitgegeben, einen solchen Schritt anzugehen. Meine Geschwister hatten stets ein offenes Ohr für mich und konnten mich bestens zurück auf das Wesentliche im Leben führen.

Meinem größten Glück widme ich diese Arbeit: Sandra, Gavin & Jarne.

Zusammenfassung

Komplexe mechatronische Systeme, die autonom und flexibel auf Änderungen in ihrer Umwelt reagieren, sind aus unserer Zukunft nicht mehr wegzudenken. Fahrerassistenzsysteme aus dem Transportwesen (z.B. Automobil oder Luftfahrt) oder auch „das Haus der Zukunft“ sind Beispiele hierfür. Diese Systeme werden typischerweise durch eine Vernetzung von (mechatronischen) Komponenten realisiert. Software wird dabei unter anderem eingesetzt, um durch Kommunikation das Wissen von anderen Komponenten zu nutzen, um so benötigte Funktionalität zur Verfügung zu stellen. Im Gegensatz zu reinen Softwareanwendungen bekommt der Sicherheitsaspekt in solchen Systemen einen deutlich höheren Stellenwert, da Fehler zu einer Gefahr für ihre Umwelt und damit auch zu einer Gefahr für Menschenleben führen können. Zudem muss die Wiederverwendung bereits existierender Lösungen (Komponenten) in der Entwicklung von mechatronischen Systemen unterstützt werden, um den Marktanforderungen wie Qualität und Schnelligkeit gerecht zu werden.

Kompositionelle Vorgehensweisen sind weitverbreitete Engineering Ansätze, um solche komplexen Probleme durch kleinere Teilprobleme einfacher zu betrachten und einzelne Komponenten wiederzuverwenden. Wiederverwendung ist dabei das Mittel, um komplexe Probleme durch bekannte (Teil-) Lösungen unterstützend zu entwickeln. Dies führt allerdings zu dem Problem, dass Abhängigkeiten zwischen den verschiedenen Kompositionen, die auch auf unterschiedlichen Hierarchieebenen stattfinden können, berücksichtigt werden müssen, ohne die Eigenschaften der einzelnen Komponenten zu verletzen. Hierbei müssen sowohl Altkomponenten integriert werden, deren Verhalten typischerweise nicht mehr formal durch Modelle beschrieben ist sowie auch Komponenten, die ihre Struktur aufgrund von Veränderungen in ihrer Umwelt zur Laufzeit anpassen.

In dieser Arbeit wird eine Unterstützung für die Komposition und Wiederverwendung von Komponenten in dem modellgetriebenen Entwicklungsansatz MECHATRONIC UML vorgestellt. Die Abhängigkeiten, die bei der Komposition berücksichtigt werden müssen, werden dabei konstruktiv durch einen Syntheseansatz für das Verhalten von Komponenten und analytisch durch eine Verfeinerungsüberprüfung zwischen unterschiedlichen Hierarchieebenen von Verhalten, bzw. von Komponenten unterstützt. Die Verfeinerungsüberprüfung berücksichtigt Altsysteme sowie Strukturanpassungen, deren Ressourcenbeschränkungen in einer Codegenerierung adressiert werden. Der Gesamtansatz wurde an dem RailCab-Projekt der Universität Paderborn validiert.

Inhaltsverzeichnis

1	Einleitung	1
1.1	Ziele und Konzeptüberblick	3
1.2	Anwendungsbeispiel	6
1.3	Übersicht	8
2	Mechatronic UML	9
2.1	Entwicklung hierarchischer Komponentensysteme	10
2.2	Selbstoptimierende, mechatronische Systeme	13
2.3	Komponenten	15
2.4	Echtzeitverhalten	16
2.4.1	Real-Time Coordination Pattern	17
2.4.2	Real-Time Statecharts	19
2.4.3	Parameterized Real-Time Coordination Pattern	27
2.4.4	Parameterized Real-Time Statecharts	28
2.4.5	Rekonfigurationsverhalten	29
2.4.6	Verifikation	35
2.4.7	Verfeinerungen	38
2.5	Hybrides Verhalten	42
2.5.1	Hybrid Reconfiguration Charts	43
2.5.2	Verifikation und Verfeinerung	45
2.6	Timed Story Driven Modeling	45
2.6.1	Metamodell	48
2.6.2	Timed Story Pattern	53
2.6.3	Timed Story Diagrams	57
2.6.4	Timed Story Charts	57
3	Verfeinerung in hierarchischen Komponentensystemen	65
3.1	Verfeinerungsdefinition	72
3.1.1	Real-Time Statecharts	73
3.1.2	Timed Story Charts	81
3.1.3	Diskussion	85
3.2	Verfeinerungsüberprüfung	86
3.2.1	Erreichbarkeitsanalyse	86
3.2.2	Verifikation der Verfeinerung	88
3.2.3	Diskussion	94

4	Integration von Altkomponenten	97
4.1	Gray Box Checking	100
4.1.1	Formalisierungen	101
4.1.2	Initiale Verhaltenssynthese	108
4.1.3	Iterative Verhaltenssynthese	110
4.2	Black Box Checking	116
4.2.1	L* Lernalgorithmus	119
4.2.2	L* für mechatronische Systeme	125
4.3	White Box Checking	128
4.4	Identifikation von Reglerverhalten	136
4.5	Diskussion	138
5	Synthese von Komponentenverhalten	139
5.1	Kompositionsregeln	144
5.1.1	Zustands-Kompositionsregeln	145
5.1.2	Nachrichten-Kompositionsregeln	147
5.2	Synthese	150
5.2.1	Parallele Komposition	150
5.2.2	Anwendung von Zustands-Kompositionsregeln	150
5.2.3	Anwendung von Nachrichten-Kompositionsregeln	155
5.3	Erhalt von Rollenverhalten	158
5.3.1	Rollenkonformität	159
5.3.2	Erhalt von Deadlock Freiheit	168
5.4	Weitere Anwendungsfälle	172
5.5	Diskussion	174
6	Werkzeugunterstützung	175
6.1	Ausführung	175
6.1.1	Laufzeitumgebung	176
6.1.2	Codegenerierung und Laufzeitanalyse	183
6.2	Umsetzung	197
6.3	Validierung	200
6.3.1	Konvoi-Anwendungsszenario	201
6.3.2	Weitere Anwendungsszenarien und Fazit	231
7	Verwandte Arbeiten	233
7.1	Modellgetriebene Entwicklungsansätze	233
7.2	Modellierung und Verfeinerung kompositioneller Strukturanpassungen	235
7.2.1	Modellierung	235
7.2.2	Verfeinerung	237
7.2.3	Verifikation	238
7.3	Analyse von Altkomponenten	239
7.3.1	Reguläre Inferenz	239

7.3.2	Abstraktionstechniken	240
7.4	Synthese von Komponentenverhalten	241
7.4.1	Controller-Synthese	241
7.4.2	Synthese von nicht-zeitbehafteten Komponentenverhalten	242
7.4.3	Synthese von zeitbehafteten Komponentenverhalten	242
8	Zusammenfassung und Ausblick	245
A	Timed Story Charts	249
A.1	Elemente	249
A.1.1	Statechart	249
A.1.2	Zustände	250
A.1.3	Transitionen	251
A.1.4	Clocks	252
A.1.5	Guards	253
A.1.6	Synchronisationen	253
A.1.7	Invariante	258
A.1.8	Time Guards	259
A.1.9	Clock Resets	259
A.1.10	Deadlines	260
A.1.11	Actions und Seiteneffekte	261
A.1.12	WCET und Prioritäten	264
A.2	Zusammengesetzte Ausführung	265
A.2.1	Zustände	265
A.2.2	Transitionen	267
	Abbildungsverzeichnis	271
	Tabellenverzeichnis	275
	Literaturverzeichnis	277

Kapitel 1

Einleitung

Software ist zunehmend für einen schnell wachsenden Bereich von technischen Systemen wie in dem Transportwesen oder der Medizintechnik ein Schlüsselfaktor, um Sicherheit, Effizienz oder Komfort zu steigern [Wir04, BGH05, GH06b, GHH⁺08c]. Die Entwicklung dieser Systeme ist nicht mehr nur Gegenstand der klassischen Ingenieursdisziplinen Maschinenbau, Elektrotechnik und Regelungstechnik, sondern auch der Informatik. Mechatronische Systeme bezeichnen Systeme, die aus der Summe dieser Disziplinen entstehen.

Komplexe mechatronische Systeme, die autonom und flexibel auf Änderungen in ihrer Umwelt reagieren, sind aus unserer Zukunft nicht mehr wegzudenken. Fahrerassistenzsysteme aus dem Transportwesen (z.B. Automobil oder Luftfahrt) oder auch „das Haus der Zukunft“ sind Beispiele hierfür. Diese Systeme werden typischerweise durch eine Vernetzung von (mechatronischen) Komponenten realisiert. Im Fall der Fahrerassistenzsysteme wird z.B. die Motorsteuerung mit der Brems- und Lenksteuerung vernetzt, um bessere Bremswege zu ermöglichen [Rie09]. Software wird dabei unter anderem eingesetzt, um durch Kommunikation das Wissen von anderen Komponenten zu nutzen, um so benötigte Funktionalität zur Verfügung zu stellen. Dabei kann das durch Software gesteuerte Verhalten einer Komponente gegebenenfalls auch an geänderte Bedingungen angepasst werden. Es entstehen komplexe Funktionsnetze aus (Software-) Komponenten, welche sowohl steuerungs- als auch regelungstechnische Aufgaben realisieren. Reagieren diese Systeme optimal, autonom und flexibel auf Änderungen in ihrer Umwelt sprechen wir von selbstoptimierenden, mechatronischen Systemen.

Im Gegensatz zu reinen Softwareanwendungen bekommt der Sicherheitsaspekt in solchen Systemen einen deutlich höheren Stellenwert, da Fehler zu einer Gefahr für ihre Umwelt und damit auch zu einer Gefahr für Menschenleben führen können [LAK92, Sto96]. Zudem muss die Wiederverwendung bereits existierender Lösungen (Komponenten) in der Entwicklung von mechatronischen Systemen unterstützt werden, um den Marktanforderungen wie Qualität und Schnelligkeit gerecht zu werden. Diesen Herausforderungen wird heutzutage mit modellgetriebenen Entwicklungsverfahren begegnet, die Sicherheitsanalysen auf der Modellebene durch Simulation sowie formale mathematisch fundierte Verfahren erlauben. Zudem ermöglicht die komponentenbasierte modellgetriebene Entwicklung durch wohldefinierte Schnittstellen und formale Verhaltensmodelle ein hohes Maß an Wiederverwendungspotential von entwickelten Lösungen [GJM91, Crn02, HKK04].

Auf Grund dieser Anforderungen mechatronischer Systeme ist es notwendig Methoden zu entwickeln, die auf der einen Seite eine geeignete Modellierung und Analyse erlauben und auf der anderen Seite in dem Entwicklungsprozess die Komposition und Wiederverwendung von Komponenten unterstützen, um komplexe Systeme umsetzen zu können [GAO95, IWY00, Gar03].

Modellgetriebene Softwareentwicklung Die hochgradige Vernetzung selbstoptimierender, mechatronischer Systeme ermöglicht auf der einen Seite, wesentlich erweiterte Funktionalität zu realisieren, bedeutet auf der anderen Seite aber auch entsprechend zusätzliche Software zur nachrichtenbasierten Kommunikation zwischen Systemkomponenten. Diese Kommunikation beinhaltet den Austausch von (komplexen) Zustandsinformationen über entsprechende Protokolle und zugrunde liegende Kommunikationskanäle. Das Verhalten der einzelnen Komponenten wird dabei massiv durch diese Kommunikationen beeinflusst.

Um diese Systeme zu beherrschen, wird ein systematischer Entwicklungsansatz gefordert, der Modellierung als eine wesentliche Entwurfsaktivität beinhaltet. Um sicherheitskritische Anforderungen zu adressieren, werden modellbasierte Analyseverfahren sowie eine Quellcodegenerierung aus diesen Modellen benötigt [GH06b]. Die drei zusammenhängenden Aktivitäten Modellierung, Analyse und Quellcodegenerierung werden mit dem Begriff *modellgetrieben* bezeichnet [Ken02].

In [GH06b] haben wir Verfahren zur modellgetriebenen Softwareentwicklung von mechatronischen Systemen verglichen (siehe auch Abschnitt 7.1). Die meisten betrachteten Ansätze unterstützen nur eingeschränkt Konzepte für die Modellierung. Kompositionelle Strukturanpassungen, die neue Elemente der bisherigen Struktur hinzufügen oder Elemente aus der Struktur entfernen, um autonom und flexibel auf Änderungen in der Umwelt reagieren zu können, werden von keinem der Verfahren unterstützt.

Weiterhin ist zu beobachten, dass keiner der Ansätze plattformspezifische Modelle (vollständig) berücksichtigt, um Altsysteme zu integrieren oder eine Wiederverwendung von entwickelten Lösungen zu ermöglichen.

An dem Vergleich nimmt auch der an diesem Lehrstuhl entwickelte Ansatz MECHATRONIC UML teil. Die betrachtete Version aus dem Jahr 2006 berücksichtigt im Wesentlichen die Ergebnisse der Dissertation von Sven Burmester [Bur06]. Der dort vorgestellte Ansatz unterstützt eine hybride Modellierung der Struktur auf Basis von diskreten Softwarekomponenten und kontinuierlichen Reglerkomponenten sowie die Rekonfiguration der Reglerstruktur.

Um den hohen Qualitätsanforderungen an die Kommunikation gerecht zu werden, wurden Muster zur Spezifikation der Kommunikation in der MECHATRONIC UML eingeführt. Die Struktur der sogenannten REAL-TIME COORDINATION PATTERNS besteht aus Rollen der beteiligten Kommunikationspartner sowie einer Verbindung, dem Konnektor, zwischen den Rollen. Das Rollenverhalten wird mittels REAL-TIME STATECHARTS beschrieben, die die bekannten Zustandsmaschinen der UML [Obj05b] im Wesentlichen wohldefiniert um Zeit erweitern.

Auf Basis dieser Muster wird eine Dekomposition des Systems in Komponenten und der Kommunikation zwischen den Komponenten, den REAL-TIME COORDINATION PATTERNS, ermög-

licht. Hierfür wurden Analysetechniken vorgestellt (z.B. [GTB⁺03]), die den modularen Aufbau des Systems ausnutzen.

Das Komponentenverhalten wird implementiert durch Verfeinerung des Rollenverhaltens zu Komponenten-Portverhalten. Wesentliche Aufgaben sind hierbei das Hinzufügen von Reglern zu einzelnen Zuständen, die Beschreibung von Rekonfigurationen der Regler sowie eine Anpassung des Verhaltens, um Abhängigkeiten zwischen mehreren Rollen aufzulösen.

Für diese hybriden Modelle wird eine Quellcodegenerierung unterstützt, die die Echtzeiteigenschaften auf Quellcodeebene korrekt umsetzt. Die Dissertation von Martin Hirsch [Hir08] erweitert diesen Ansatz, um kompositionelle Strukturanpassungen der Muster zu modellieren und zu analysieren. Matthias Tichy hat diesen Ansatz wiederum um kompositionelle Strukturanpassungen der Komponentenstruktur, ohne das Verhalten zu betrachten, erweitert [Tic09].

Damit unterstützt die MECHATRONIC UML einige grundlegende Anforderungen, um selbstoptimierende, mechatronische Systeme zu entwickeln. Eine skalierbare formale Verifikation wird durch einen musterbasierten Ansatz, der eine Dekomposition des Systems ermöglicht, erreicht. Allerdings werden wesentliche Anforderungen der komponentenbasierten Entwicklung komplexer Systeme nicht adressiert.

Die MECHATRONIC UML stellt, wie auch all die in [GH06b] betrachteten Verfahren, keine Unterstützung für eine Verfeinerung in hierarchischen Komponentensystemen mit kompositionellen Strukturanpassungen zur Verfügung. Dies ist allerdings essentiell, damit Kommunikationsmuster mit kompositionellen Strukturanpassungen durch eine Komponente angewandt werden können. Hierdurch wird eine Wiederverwendung von Lösungen ermöglicht. Eine Unterstützung bei der Komposition von Protokollverhalten zu einem Gesamtverhalten einer Komponente wird ebenfalls nur sehr eingeschränkt durch eine von dem Entwickler manuell hinzugefügte Synchronisation adressiert. Hierbei weiß der Entwickler zu keinem Zeitpunkt der Entwicklung, ob eine Komposition der Protokollverhalten überhaupt möglich ist. Zudem können nicht explizit Anforderungen an eine solche Komposition gestellt werden. Darüber hinaus betrachten all diese Ansätze keine Möglichkeit Altkomponenten, von denen kein Modell zur Verfügung steht, die aber einen hohen Wert darstellen, zu integrieren.

Wie in [TOHS99, Crn02, HKK04] beschrieben stellen gerade diese, verallgemeinert dargestellt, Kompositionen und Wiederverwendungen eine wesentliche Herausforderungen dar, um komplexe Systeme ganzheitlich von der Dekomposition des Systems hin zum komponierten Gesamtsystem zu entwickeln. Die in dieser Arbeit vorgestellte Unterstützung für die Komposition und Wiederverwendung von Komponenten in dem modellgetriebenen Entwicklungsansatz MECHATRONIC UML soll genau diese Anforderungen adressieren.

1.1 Ziele und Konzeptüberblick

Ziel dieser Arbeit ist es eine Unterstützung für die Komposition und Wiederverwendung von Komponenten in dem modellgetriebenen Entwicklungsansatz MECHATRONIC UML vorzustel-

len, um eine systematische, modellgetriebene Softwareentwicklung für selbstoptimierende, mechatronische Systeme zu ermöglichen.

Dieser Ansatz baut auf den bisherigen Ergebnissen der MECHATRONIC UML auf, womit insgesamt durch die Verifikationstechniken für vernetzte selbstoptimierende, mechatronische Systeme aus [Hir08] durch Dekomposition des Systems und dem hier vorgestellten Kompositionsansatz ein modellgetriebener Entwicklungsansatz entsteht, der hybrides-, Echtzeitverhalten und Ressourceneinschränkungen für mechatronische Systeme mit kompositionellen Strukturanpassungen unterstützt.

Im Folgenden werden die einzelnen Beiträge, die im Rahmen dieser Arbeit entstanden sind, näher erläutert. Zu jedem Beitrag werden die umfassendsten Veröffentlichungen referenziert. Weitere Veröffentlichungen oder betreute Arbeiten (Master- und Bachelorarbeiten sowie Projektgruppen) werden in den entsprechenden Hauptkapiteln referenziert. Eine vollständige Liste der im Rahmen dieser Arbeit entstandenen Veröffentlichungen ist im Literaturverzeichnis unter „Eigene Veröffentlichungen“ sowie „Betreute Arbeiten“ zu finden.

Die einzelnen Beiträge sind aufgeteilt in die Bereiche *Verfeinerung in hierarchischen Komponentensystemen*, *Integration von Altkomponenten*, *Synthese von Komponentenverhalten* und *Werkzeugunterstützung*. Die ersten drei Beiträge adressieren unmittelbar die Unterstützung der Komposition und Wiederverwendung. Der Beitrag zur *Werkzeugunterstützung* stellt die notwendige Basis zur Verfügung, um Altkomponenten zu integrieren. Dabei werden die Herausforderungen adressiert, um die (erweiterten) Modelle der MECHATRONIC UML auf Code abzubilden.

Verfeinerung in hierarchischen Komponentensystemen Abstraktion und Hierarchisierung sind wesentliche Hilfsmittel bei der Entwicklung von Softwarekomponenten, um komplexe Sachverhalte zu beherrschen. Der musterbasierte Ansatz zur Beschreibung der Kommunikation sowie die Möglichkeit der Hierarchisierung der Komponentenstruktur sind daher feste Bestandteile der MECHATRONIC UML. Die hierdurch entstehenden hierarchischen Kompositionen fordern eine formale Definition der Verfeinerung zwischen den verschiedenen Abstraktionen, um einen Erhalt des abstrakteren Verhaltens durch ein konkreteres Verhalten, bzw. ein Verhalten auf einer niedrigeren Hierarchiestufe zu gewährleisten.

Eine Form der Komposition ist das Einbetten von Komponenten in hierarchische Komponenten. Hierdurch wird Rollenverhalten bzw. Protokollverhalten an vorhandene Komponenten weitergeleitet, die eine Verfeinerung des Protokollverhaltens implementieren. Hierbei können unterschiedliche Rollenstrukturen auf Komponentenstrukturen abgebildet werden.

Der in dieser Arbeit vorgestellte Ansatz unterstützt eine Verifikation der Verfeinerung solcher hierarchischer Komponentenstrukturen mit kompositioneller Strukturanpassung, die sowohl Sicherheits- und Lebendigkeitseigenschaften berücksichtigt, als auch das nach außen sichtbare Echtzeitverhalten [HHH10, HH11].

Integration von Altkomponenten Es kann gerade in der industriellen Praxis häufig vorkommen, dass Altkomponenten wiederverwendet werden, um zum einen den Entwicklungsprozess zu beschleunigen und zum anderen auf bewährte Qualität zurückzugreifen. Unser Ansatz unterstützt eine Integration von Altkomponenten. Zentrale Idee hierbei ist, das relevante Verhaltensmodell für die Integration iterativ zu erlernen und auf dessen Basis dann formal die Integration zu überprüfen [HH08a, GHH08a, HMS⁺10].

Synthese von Komponentenverhalten Wie bereits weiter oben beschrieben, propagieren wir einen musterbasierten Ansatz zur Beschreibung der Kommunikation. Diese Vorgehensweise erlaubt es, das Kommunikationsverhalten getrennt von dem Komponentenverhalten zu modellieren und zu analysieren. Der Ansatz von Hirsch [Hir08] nutzt den musterbasierten Ansatz aus, um kompositionelle Strukturanpassungen der Kommunikationsstruktur formal zu verifizieren.

Diese formal verifizierten Rollenverhalten werden durch eine Komponente angewandt und kombiniert. Da die Rollenverhalten, egal ob sie zu gleichen oder unterschiedlichen Kommunikationsmustern gehören, untereinander häufig Abhängigkeiten aufweisen und dadurch nicht nur einfach parallel von einer Komponente angewandt werden können, unterstützt unser Ansatz eine formale Sprache zur Beschreibung der Abhängigkeiten. Neben der Beschreibung von Abhängigkeiten zwischen neuentwickelten Lösungen, kann in unserem Fall auch eine integrierte Altkomponente berücksichtigt werden.

Die Abhängigkeiten sowie die Rollen sind Eingaben in eine Synthese für das Komponentenverhalten. Diese stellt sicher, dass das synthetisierte Komponentenverhalten eine korrekte Verfeinerung der einzelnen Rollenverhalten ist [HGH⁺09, EH10a].

Werkzeugunterstützung Die entwickelten Konzepte zur Wiederverwendung werden durch ein Werkzeug umgesetzt. Mit der Werkzeugunterstützung wird eine Laufzeitumgebung zur Verfügung gestellt. Diese unterstützt neben dem Ausführen des Systems auf einer Zielplattform eine Simulations- und Verifikationsumgebung für die Integration von Altkomponenten [GH06a, HMS⁺10]. Eine automatische Überprüfung der Integration wird durch eine Codegenerierung basierend auf dem in [BGH⁺07] vorgestellten Ansatz, der hybride und Echtzeitsysteme unterstützt, ermöglicht. Die Codegenerierung wird dahingehend erweitert, dass eine Betrachtung von kompositionellen Strukturanpassungen unter Erhalt von Echtzeitanforderungen erreicht wird [GHH11]. Der in dieser Arbeit vorgestellte Ansatz adressiert den Erhalt der Echtzeitanforderungen durch eine Laufzeitanalyse (Worst Case Execution Time Analyse - WCET Analyse) für diese Modelle [HOGS12].

1.2 Anwendungsbeispiel

Das RailCab Forschungsprojekt¹ der Universität Paderborn dient als ein konkretes Anwendungsbeispiel für ein selbstoptimierendes, mechatronisches System. Das RailCab System erweitert das herkömmliche Schienensystem um einen Linearantrieb sowie passive Weichen. Der eigentliche Antrieb wird über einen Stator im Schienennetz und einem Läufer im RailCab ermöglicht. Durch ein Magnetfeld, welches sich entlang der Schiene fortbewegt, wird das Fahrzeug beschleunigt und gebremst. Eine passive Weiche in Verbindung mit einer aktiven Lenkung ermöglicht das Ausscheren von dicht hintereinander fahrenden Fahrzeugen bei voller Geschwindigkeit.

Eine wesentliche Eigenschaft dieses Systems ist, dass die RailCabs individuell agieren und unabhängig und dezentral Entscheidungen treffen. Das Feder-/Neigemodul des RailCabs tauscht z. B. Informationen mit anderen RailCabs aus, um eine Störung auf den Schienen zu kompensieren und um den Schienenlauf zu optimieren.

Als durchgängiges Anwendungsbeispiel wird in dieser Arbeit das Konvoiszenario betrachtet. RailCabs bilden zur Laufzeit Konvois, um den Energieverbrauch durch Fahren im Windschatten zu reduzieren und um den Streckendurchsatz zu erhöhen (siehe Abbildung 1.1).



(a) RailCab-Konvoi in der Simulation



(b) RailCab-Konvoi auf der Teststrecke

Abbildung 1.1: RailCab Konvoi

Ein Konvoi muss durch ein RailCab koordiniert werden, um die Sicherheit und Stabilität des Konvois nicht zu gefährden [GHH⁺06c, HHG08]. Das System ist z. B. unsicher, wenn die RailCabs aufeinander auffahren können. Ein stabiler Konvoi unterstützt die Sicherheit, in dem ein Übersteuern der unterliegenden Regler beim Anpassen z. B. der Geschwindigkeit kontrolliert wird. Übersteuern beim Anpassen der Geschwindigkeit führt zu dem Effekt, dass die vorgegebene Geschwindigkeit (Sollgeschwindigkeit) kurzfristig überstiegen wird. Aufgrund von Störungen zur Laufzeit, wie Wind, kann dies durch die Regelung nicht vollständig verhindert werden. In einem Konvoi ohne Koordinator kann durch Anpassen von Parametern des Konvois, wie der Geschwindigkeit, ein Ziehharmonikaeffekt auftreten, der den Effekt des Übersteuerns über den

¹<http://www-nbp.uni-paderborn.de/>

aus einer Komposition der Coordinator- und PosCalc-Komponente (Komponentenkomposition). Die Wiederverwendung (Integration) von Altkomponenten wird durch die LegacyRailCab-Komponente verdeutlicht. Diese Anforderung ist gerade relevant, wenn RailCab-Systeme flächendeckend integriert sind und RailCabs von unterschiedlichen Herstellern interagieren.

Dieses Beispiel lässt sich einfach auf andere Transportsysteme, wie z.B. Automobile transferieren. Szenarien, in denen Fahrzeuge koordiniert eine Baustelle oder Kreuzung passieren, werden seit Längerem diskutiert. Das Problem der Wiederverwendung, gerade von Altkomponenten unterschiedlicher Zulieferer, ist für diese Domäne ebenfalls typisch.

1.3 Übersicht

Im nächsten Kapitel werden die Grundlagen für die modellgetriebene Softwareentwicklung anhand der MECHATRONIC UML vorgestellt. Dabei betrachten wir die Grundlagen von Echtzeit- und hybriden Systemen. In Abschnitt 2.1 stellen wir unseren Ansatz zur *Entwicklung von hierarchischen Komponentensystemen* vor, in dem wir die bisherigen Modellierungs- und Analysetechniken der MECHATRONIC UML zusammen mit der in dieser Arbeit bereitgestellten Unterstützung für Komposition und Wiederverwendung darstellen. Wir stellen darüber hinaus mit dem Timed Story Driven Modeling Ansatz (Abschnitt 2.6) die notwendigen Erweiterungen der Modellierungstechniken der MECHATRONIC UML vor, um die Anforderungen dieser Arbeit zu adressieren.

Unseren Ansatz zur *Verfeinerung in hierarchischen Komponentensystemen* stellen wir in Kapitel 3 vor. Wir werden dabei eine Verfeinerungsdefinition und -überprüfung erläutern, die die geforderten kompositionellen Strukturanpassungen unterstützt.

In Kapitel 4 stellen wir unsere *Integration von Altkomponenten* vor. Der in dieser Arbeit entwickelte Ansatz unterstützt drei unterschiedliche Verfahren, um Altkomponenten mit unterschiedlichen zur Verfügung stehenden Informationen (*Black Box*, *White Box* und *Gray Box*) zu integrieren.

Die *Synthese von Komponentenverhalten* wird in Kapitel 5 erläutert. Im Mittelpunkt steht hierbei die Konkretisierung von Protokollverhalten durch Komposition innerhalb einer Komponente.

In Kapitel 6 stellen wir die *Werkzeugunterstützung* vor. Unser Ansatz zur Quellcodegenerierung, Laufzeitanalyse sowie Laufzeitumgebung wird in Abschnitt 6.1 präsentiert. Anschließend werden in Abschnitt 6.2 und 6.3 die Umsetzung der Werkzeugunterstützung sowie eine Validierung des Gesamtansatzes vorgestellt.

Kapitel 7 diskutiert die verwandten Arbeiten und abschließend wird in Kapitel 8 eine Zusammenfassung und Ausblick der Arbeit gegeben.

Kapitel 2

Mechatronic UML

Dieses Kapitel führt in die modellbasierte Softwareentwicklung mittels der MECHATRONIC UML ein, da die in dieser Arbeit vorgestellten Konzepte auf der MECHATRONIC UML basieren, bzw. diese erweitern. In diesem Zusammenhang werden die theoretischen Grundlagen von Timed Automata und Graphtransformationssystemen behandelt.

Die MECHATRONIC UML ist eine Anpassung der UML [Obj05b] für die modellbasierte Entwicklung mechatronischer Systeme. Eine Werkzeugunterstützung wird durch die Fujaba Real-Time Tool Suite¹ angeboten.

Im nächsten Abschnitt beschreiben wir unseren Ansatz zur *Entwicklung von hierarchischen Komponentensystemen*. Wir werden dabei die bisherigen Modellierungs- und Analysetechniken der MECHATRONIC UML zusammen mit der in dieser Arbeit bereitgestellten Unterstützung für Komposition und Wiederverwendung darstellen.

Anschließend erläutern wir die Modellierungselemente und Analysetechniken der MECHATRONIC UML. Zuerst stellen wir in Abschnitt 2.2 genauer die hier betrachteten selbstoptimierenden, mechatronischen Systeme vor. Wir beginnen dann mit der Strukturmodellierung der MECHATRONIC UML in Abschnitt 2.3, die grundlegend in dieser Arbeit genutzt wird. Die Echtzeit-Verhaltensbeschreibung und -Analyse stellen wir in Abschnitt 2.4 vor. Der dort beschriebene musterbasierte Ansatz wird in dieser Arbeit ebenfalls ausgenutzt. Im Zusammenspiel mit dem hierarchischen komponentenbasierten Aufbau der MECHATRONIC UML dient der musterbasierte Ansatz als Grundlage für die Unterstützung der Wiederverwendung von Komponenten und Protokollverhalten. Anschließend stellen wir die hybride Modellierung der MECHATRONIC UML vor (siehe Abschnitt 2.5), die wir für die Integration von Altkomponenten mit regelungstechnischen Anteilen (siehe Abschnitt 4.4) sowie für die Codegenerierung ausnutzen (siehe Abschnitt 6.1.2.4). In Abschnitt 2.6 beschreiben wir den Timed Story Driven Modeling Ansatz, um über eine Datenstruktur gemeinsam Verhalten und Strukturanpassungen zu spezifizieren. Den Timed Story Driven Modeling Ansatz werden wir in Abschnitt 3.1.2 ausnutzen, um eine Verfeinerung für hierarchische Komponentensysteme mit kompositionellen Strukturanpassungen zu definieren.

¹<http://www.fujaba.de/projects/real-time.html>

2.1 Entwicklung hierarchischer Komponentensysteme

In diesem Kapitel wird die MECHATRONIC UML vorgestellt, die für Strukturanpassungen auf der Kommunikationsebene einen Verifikationsansatz auf Basis einer Dekomposition des Systems in Komponenten und Kommunikationen zwischen Komponenten ermöglicht. Um ein Gesamtsystem zu entwickeln wird zudem wie durch [SGW94, TOHS99, Crn02, HKK04], [Obj05b, S. 515ff] und [Obj09, S. 534ff] beschrieben, eine Komposition der getrennt entwickelten und analysierten Komponenten und Kommunikationen zwischen Komponenten zu hierarchischen Komponenten benötigt. In den Kapiteln 3 bis 5 stellen wir Ansätze vor, die genau dieser Forderung nachgehen, indem eine Wiederverwendung von Komponenten und Kommunikationen zwischen Komponenten zu hierarchischen Komponenten unterstützt wird.

Wir werden in diesem Abschnitt einen systematischen Entwicklungsansatz skizzieren, der die bisherigen Entwicklungsschritte der MECHATRONIC UML wie in den folgenden Abschnitten vorgestellt mit denen, die in dieser Arbeit vorgestellt werden, integriert darstellt. Die grundlegende Arbeit zu dem Entwicklungsansatz der MECHATRONIC UML wurde von Giese in [Gie03] vorgestellt.

Der Entwicklungsansatz besteht aus den Schritten *Szenarien modellieren*, *Rollenverhalten synthetisieren*, *Koordinationsmuster analysieren*, *Rollen anwenden* (diese Schritte werden durch die bisherige MECHATRONIC UML unterstützt) und dem Schwerpunkt dieser Arbeit *Verfeinerung in hierarchischen Komponentensystemen*, *Altkomponenten integrieren* und *Komponentenverhalten synthetisieren*. Im Folgenden erläutern wir die einzelnen Schritte anhand von Abbildung 2.1. Wir werden dabei nicht explizit auf mögliche Iterationen zwischen den einzelnen Schritten eingehen.

Szenarien modellieren und Rollenverhalten synthetisieren

Die MECHATRONIC UML unterstützt mit der Aktivität *Szenarien modellieren* die Möglichkeit in den frühen Phasen der Softwareentwicklung formal Kommunikationen zwischen den Rollen eines Musters zu spezifizieren. Ermöglicht wird dies durch eine Anpassung von UML-Sequenzdiagrammen. Hierbei wird die Anforderung unterstützt, Zeit in den frühen Phasen durch eine Parametrisierung für nicht genau bekannte Zeitbedingungen zu beschreiben [BGK05, ACE⁺08]. Das Gesamtverhalten M_i^{Sc} (für alle Muster $i = 1 \dots, i = n$) ergibt sich aus den parallel geschalteten Szenarien ($1 \dots, k$): $M_i^{Sc} = M_{i,1}^{Sc} \parallel \dots \parallel M_{i,k}^{Sc}$ (siehe Abbildung 2.1 unter Parameterized Real-Time Sequence Diagram). Durch den Fokus auf die Kommunikationsbeschreibung zwischen Rollen propagieren wir bereits in den frühen Phasen einen kompositionellen Entwicklungsansatz, im Vergleich zu den klassischen Verfahren zur Synthese von Zustandsverhalten basierend auf Harel [HKP05], die ein Gesamtverhalten synthetisieren.

Um aus den Szenarien Rollenverhalten für ein Koordinationsmuster zu synthetisieren, müssen die Szenarien Synthesebedingungen genügen ($M_i^{Sc} \models c_i$). Es darf z.B. nicht vorkommen, dass eine ausgewiesene Konfiguration (Zustand) unterschiedliche Vorbedingungen in verschiedenen Szenarien besitzt oder dass spezifizierte Konfigurationen nicht erreichbar sind (siehe [GHHK06]).


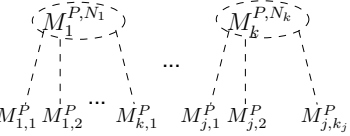
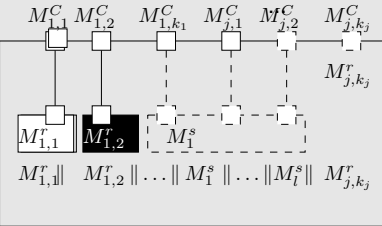
Aktivität	Modelle	Analyse
<p>Szenarien modellieren</p> <p>Rollenverhalten synthetisieren</p>	<p>Parameterized Real-Time Sequence Diagram</p> 	<p>Gesamtverhalten Szenario M_i^{Sc}: $M_i^{Sc} = M_{i,1}^{Sc} \parallel \dots \parallel M_{i,k}^{Sc}$ mit $1, \dots, k$ Teilszenarien Synthesebedingung: $M_i^{Sc} \models c_i$ c_i: Beschreibt Eigenschaften für konfliktfreie Szenarien</p>
<p>Koordinationsmuster analysieren</p> <p>Rollen anwenden</p>	<p>Parameterized Real-Time Coordination Pattern</p> 	<p>Gesamtverhalten Muster M_i^P: $M_i^P = M_{i,1}^P \parallel \dots \parallel M_{i,k}^P \parallel N_k$ mit $1, \dots, k$ Rollenverhalten, N_k Kanalverhalten Korrektheit: $M_i^P \models \phi_i \wedge \neg \delta$ ϕ_i: Sicherheits- und begrenzte Lebendigkeitseigenschaften δ: Deadlock</p>
<p>Komponenten konkretisieren: Komponenten wiederverwenden, Altcomponenten integrieren, Komponentenverhalten synthetisieren</p>	<p>Hierarchische Komponenten</p> 	<p>Gesamtverhalten Komponente M_i^C: $M_i^C = M_{i,1}^r \parallel \dots \parallel M_{j,kj}^r \parallel M_1^s \parallel \dots \parallel M_l^s$ mit $1, \dots, j$ angewandten Mustern, $1, \dots, k$ instanziierten Rollen von Muster j, $1, \dots, l$ komponierten Protokollverhalten Verfeinerungsbedingung: $\forall 1 \leq m \leq j$, $\forall 1 \leq n \leq k_j : M_{m,n}^r \leq M_{m,n}^c$ mit $M_{m,n}^r$: Protokollverhalten eingebettete Komponente, $M_{m,n}^c$ Protokollverhalten hierarchische Komponente Synthesebedingung: $M_i^s \leq M_{i,k}^C \parallel M_{i+1,k+1}^C \parallel \dots$ mit M_i^s komponiertes Protokollverhalten $M_i^s \models \psi_i$ mit ψ_i: Kompositionseigenschaften</p>

Abbildung 2.1: Übersicht Entwicklungsansatz

Dieser Ansatz ermöglicht es damit das Rollenverhalten von Koordinationsmustern zu synthetisieren. Das Koordinationsmuster muss zusätzlich manuell definiert werden. Dies beinhaltet den Namen des Musters festzulegen sowie die Spezifikation zu beschreiben.

Der bisherige Ansatz unterstützt grundsätzlich nur die Synthese von REAL-TIME STATECHARTS (siehe Abschnitt 2.4.2) für eine bilaterale Kommunikation, die über ein REAL-TIME COORDINATION PATTERN beschrieben wird (siehe Abschnitt 2.4.1). Handelt es sich um eine multilaterale Kommunikation, wie dies für unser Konvoi-Beispiel benötigt wird (siehe Abschnitt 1.2), so muss zusätzlich manuell das für die multilaterale Kommunikation benötigte Verhalten beschrieben werden. Dies beinhaltet unter anderem die Beschreibung der Strukturanpassung des Musters durch einen Seiteneffekt. Hiermit werden dann PARAMETERIZED REAL-TIME COORDINATION PATTERNS als Kommunikationsmuster sowie PARAMETERIZED REAL-TIME STATECHARTS, die das Rollenverhalten beschreiben, beschrieben (siehe Abschnitt 2.4.3 und 2.4.4).

Koordinationsmuster analysieren und Rollen anwenden

Die Koordinationsmuster stellen eine wesentliche zentrale Einheit des kompositionellen Vorgehens der MECHATRONIC UML dar (siehe Abschnitt 2.4.1). Hierdurch wird eine Dekomposition des Systems in Komponenten und der Kommunikation zwischen den Komponenten erreicht sowie eine Wiederverwendung bereits auf der Ebene der Protokollverhalten. Diese Vorgehensweise ermöglicht eine kompositionelle formale Verifikation. Das Gesamtverhalten M_i^P (für alle Muster $i = 1 \dots, i = n$) ergibt sich dabei aus den parallel geschalteten Rollenverhalten ($M_{i,1}^P \parallel \dots \parallel M_{i,k}^P$) und einem zusätzlichen Kanalverhalten N_k , welches abstrakt das Netzwerkverhalten spezifiziert (z.B. via Nachrichtenpuffer und Berücksichtigung von Nachrichtenverlust): $M_i^P = M_{i,1}^P \parallel \dots \parallel M_{i,k}^P \parallel N_k$ (siehe Abbildung 2.1 unter PARAMETERIZED REAL-TIME COORDINATION PATTERN).

Dieser Aufbau des Systems wird ausgenutzt, um eine kompositionelle Analyse des Systems durchzuführen. Im Gegensatz zur Überprüfung einer temporallogischen Formel auf dem globalen Zustandsraum nutzt ein kompositioneller Ansatz die Architektur aus, um nur für einzelne Elemente (Komponenten und Kommunikationen) lokale temporallogische Formeln zu überprüfen. Für jedes Muster M_i^P wird entsprechend überprüft ob die lokalen Sicherheits- und begrenzten Lebendigkeitseigenschaften ϕ_i sowie die Deadlockfreiheit $\neg\delta$ erfüllt sind: $M_i^P \models \phi_i \wedge \neg\delta$.

Erfüllen die Muster die Korrektheitsbedingungen, kann auf Basis der Rollen Komponententypen spezifiziert werden. Eine Komponente instanziiert die für den Typ relevanten Rollen. Wir sprechen hier von sogenannten Basiskomponenten (siehe auch [Tic09]), die lediglich Rollen anwenden. Im Folgenden stellen wir vor, wie diese Basiskomponenten konkretisiert werden.

Komponenten konkretisieren: Verfeinerung in hierarchischen Komponentensystemen, Alt-komponenten integrieren und Komponentenverhalten synthetisieren

Dieser Schritt beschäftigt sich mit der Konkretisierung der im vorherigen Schritt *Rollen anwenden* definierten Basiskomponenten. Die Basiskomponenten setzen sich aus den parallel geschalteten Protokollverhalten zusammen. Da die Protokollverhalten in Form von Rollen erst unabhängig von der Komponentenimplementierung entwickelt werden, um eine Wiederverwendung komponentenübergreifend zu ermöglichen, ist eine komponentenspezifische Verfeinerung notwendig.

Wir können im Wesentlichen drei unterschiedliche Ursachen für eine Konkretisierung des Rollenverhaltens unterscheiden. Die offensichtlichste ist die 1) Konkretisierung der unterliegenden Funktionen in Form von Reglern und Seiteneffekten. Dabei kann es sich um eine konkretere Form einer vorliegenden abstrakten Funktion handeln sowie um eine Einbettung von noch nicht spezifizierten Funktionen, wie z.B. das Hinzufügen des Abstandsreglers, der für den Konvoi benötigt wird, oder auch die Funktion zur Beschreibung der Strukturanpassung des Konvois, um z.B. einen weiteren Teilnehmer aufnehmen zu können.

Die Konkretisierung der Funktionen hat häufig zur Folge, dass 2) das zeitliche Verhalten oder auch das Rollenverhalten konkretisiert werden muss. Es kann hierbei zum Beispiel zu einer Anpassung von Guards oder einer Anpassung des Zustandsverhalten durch Hinzufügen weiterer Zustände kommen.

Durch die Komposition von mehreren Rollen führen 3) Abhängigkeiten zwischen diesen Rollen zu einer Konkretisierung des Rollenverhaltens. Die Rollenverhalten können entsprechend nicht mehr nur parallel ausgeführt werden. Hierbei kann ebenfalls wie unter 2) das Echtzeitverhalten konkretisiert werden. Ein Beispiel hierfür ist, dass zwei Zustände unterschiedlicher Rollen nicht gleichzeitig betreten werden dürfen. Das Verhalten der Rollen muss also untereinander synchronisiert werden.

Alle drei Ursachen für eine Konkretisierung können in Kombination miteinander auftreten, wie dies auch einfach an dem Konvoi Beispiel zu sehen ist. Wie in Abschnitt 2.3 zu Abbildung 2.2 erläutert, werden für die Situationen, ob ein RailCab im Konvoi ist oder nicht zwei unterschiedliche Regler (VelocityControl und DistanceControl) eingebettet. Die hierdurch bedingte Rekonfiguration führt zu einer Konkretisierung des Guards. Zudem darf ein RailCab nur an einem Konvoi teilnehmen, wenn es auch registriert ist.

Das Gesamtverhalten einer Komponente M_i^C ergibt sich damit aus den Konkretisierungen der Rollenverhalten $M_{1,1}^r \parallel \dots \parallel M_{j,k_j}^r$ sowie den Synchronisationen zwischen den Rollenverhalten $M_1^s \dots \parallel M_i^s$: $M_i^C = M_{1,1}^r \parallel \dots \parallel M_{j,k_j}^r \parallel M_1^s \dots \parallel M_i^s$. Eine Konkretisierung kann damit erfolgen durch:

- i) manuelles anpassen des Protokollverhaltens (in Abbildung 2.1 durch $M_{1,1}^r$ unter hierarchische Komponenten dargestellt),
- ii) durch vorhandene modellierte Komponenten (wird ebenfalls in Abbildung 2.1 durch $M_{1,1}^r$ unter hierarchische Komponenten dargestellt),
- iii) Altkomponenten (siehe $M_{1,1}^r$ in Abbildung 2.1 unter hierarchische Komponenten),
- iv) hinzufügen von zusätzlichen Abhängigkeiten in Form von Synchronisationsverhalten (siehe M_1^s in Abbildung 2.1 unter hierarchische Komponenten).

Für die Analyse der sich hiermit ergebenden hierarchischen Komponenten muss eine Verfeinerung definiert und überprüft werden, um sicherzustellen, dass die durchgeführten Konkretisierungen nicht zu einer Verletzung des bereits verifizierten Rollenverhaltens führen.

Aus Sicht der Analyse sind die Anwendungsfälle i) und ii) identisch, da zwei bekannte Modelle hinsichtlich einer Verfeinerung überprüft werden (siehe Kapitel 3). Für den Anwendungsfall iii) muss zusätzlich das relevante Verhalten erlernt werden (siehe Kapitel 4) und für iv) können wir konstruktiv durch eine formale Abhängigkeitsbeschreibung das Synchronisationsverhalten (gesamte Komponentenverhalten) synthetisieren (siehe Kapitel 5).

2.2 Selbstoptimierende, mechatronische Systeme

Werden Systeme für mehrere Anwendungssituationen entwickelt, treten häufig Konflikte zwischen den Anforderungen auf. Diese Konflikte müssen im Entwicklungsprozess gefunden und eine mögliche Lösung ausgewählt werden. Da nur eine Lösung ausgewählt werden kann, wird nicht zwangsläufig eine optimale Lösung für alle Anwendungssituationen umgesetzt. Einen Lö-

sungsansatz zur Aufhebung der Anforderungskonflikte stellen selbstoptimierende Systeme dar. Hierdurch ist das entwickelte System in der Lage, mehrere Anwendungssituationen umzusetzen, zur Laufzeit die aktuelle Situation zu erkennen, eine Gewichtung oder Priorisierung der Anforderungen auf wechselnde Umweltbedingungen zu bestimmen und daraus erforderliche Verhaltensanpassungen abzuleiten, die optimal für eine bestimmte Anwendungssituation sind. Definitionen sowie eine Reihe von Anwendungsbeispielen für selbstoptimierende, mechatronische Systeme werden in [ADG⁺09] vorgestellt.

Zur Erfassung der aktuellen Situation nutzt das entwickelte System lokale und globale Netzwerkressourcen, um die Qualität der eigenen Funktionalitäten auf Grundlage einer möglichst umfangreichen Wissensbasis zu verbessern. Teil dieser Systeme ist daher eine Koordination zwischen den einzelnen Teil-Systemen, bzw. Komponenten, um eine umfangreiche Wissensbasis in dem vernetzten System zu erstellen. Die Koordination mit der Umgebung bewirkt damit eine lokale Anpassung des Verhaltens, um den neuen Anforderungen gerecht zu werden.

Wie in [FGK⁺04] beschrieben, kann die Anpassung unterschiedlich erfolgen. Die einfachste Form ist die Parameteranpassung, z.B. das Ändern eines Parameters einer Motorregelung. Darüber hinaus kann die Struktur des Systems angepasst werden, z.B. wird für die Motorregelung im Betrieb „sportlich fahren“ eine andere Reglerstruktur benötigt als im Betrieb „ökonomisch fahren“. Eine Strukturanpassung verändert die Ordnung oder Beziehungen zwischen den Elementen des Systems. Es wird zwischen einer Rekonfiguration und einer kompositionellen Strukturanpassung unterschieden. Eine Rekonfiguration verändert die Beziehungen einer festen Menge von verfügbaren Elementen. Eine kompositionelle Strukturanpassung fügt neue Elemente der bisherigen Struktur hinzu oder entfernt Elemente aus der Struktur.

Darüber hinaus sind wesentliche Merkmale mechatronischer Systeme, dass sie eingebettete, Echtzeit-, hybride und sicherheitskritische Systeme sind [GH06b].

Ein Mikrocontroller, der in einer technischen Umgebung integriert ist, wird *eingebettetes System* genannt. Ein Mikrocontroller steuert, regelt, oder überwacht dabei Teile der technischen Umgebung, in der er eingebettet ist, indem die Software des Mikrocontrollers mit der Hardware (elektrische oder mechanische Module) interagiert. Um einen möglichst günstigen Preis in der Massenproduktion von Mikrocontrollern zu erzielen, sind die Ressourcen (Speicher und CPU) stark eingeschränkt.

Systeme, deren Verhalten von Zeitbedingungen/-restriktionen abhängig sind, werden *Echtzeitsysteme* genannt. Die Korrektheit der Funktionen eines Echtzeitsystems hängen nicht nur von dem logischen Ergebnis einer Berechnung ab, sondern auch von dem Zeitpunkt, wann dieses Ergebnis vorliegt. Ein Airbag-System ist offensichtlich ein Echtzeitsystem. Das Auslösen muss zuverlässig innerhalb eines bestimmten Zeitintervalls passieren.

Eine Integration von kontinuierlichen und diskreten Systemen wird *hybrides System* genannt. Ein Beispiel für ein kontinuierliches System ist eine Motorregelung, die kontinuierlich Eingaben in Form von Sensorsignalen verarbeitet und kontinuierlich Ausgaben berechnet. Diskrete Modi, wie „der Motor ist im Zustand ökonomisches Fahren“ oder „sportliches Fahren“, zwischen den

umgeschaltet werden kann, führen zu einem hybriden System, da hierdurch die Motorregelung beeinflusst wird.

Kann eine Fehlfunktion eine Gefahr für die Umgebung darstellen, handelt es sich um ein *sicherheitskritisches System*. Hierunter fallen sowohl die Gefahr ein Menschenleben zu verlieren, wie auch hohe ökonomische Verluste.

Selbstoptimierende, mechatronische Systeme beschreiben damit komplexe mechatronische Systeme, die optimal, autonom und flexibel auf Änderungen in ihrer Umwelt reagieren können.

2.3 Komponenten

Die Struktur des (Software-) Systems wird in der MECHATRONIC UML mit Komponentendiagrammen spezifiziert [Bur06, HH06]. Es wird dabei zwischen diskreten, kontinuierlichen und hybriden Komponenten² unterschieden. Das Verhalten diskreter Komponenten wird durch Zustandsverhalten spezifiziert (siehe Abschnitt 2.4.2) und kontinuierliche Komponenten durch Reglerverhalten. Eine hybride Komponente besteht aus diskreten und kontinuierlichen Anteilen. Eine Komponente ist in sich abgeschlossen und verbirgt ihre innere Struktur und ihr inneres Verhalten.

Ein Zugriff ist nur über bestimmte Zugangspunkte, die sogenannten *Ports*, möglich. Hierbei wird auch zwischen diskreten, kontinuierlichen und hybriden Ports unterschieden. Ein diskreter Port kann dabei in der MECHATRONIC UML ein *Required Interface* und ein *Provided Interface* mit jeweils einer Menge von Nachrichten spezifizieren. Im Fall von kontinuierlichen Ports sind dies kontinuierliche Ein- und Ausgangsgrößen (Parameter oder Variablen) des Reglerverhaltens. Ein hybrider Port beinhaltet beide Informationen.

Über ein *Required Interface* werden Nachrichten verschickt und über ein *Provided Interface* empfangen. Im Fall eines diskreten Ports wird ein Protokollverhalten (mit REAL-TIME STATECHARTS- siehe Abschnitt 2.4.2) auf Basis der Nachrichtenschnittstelle definiert. Die Spezifikation des Protokollverhaltens ist ein wesentlicher Bestandteil des MECHATRONIC UML-Ansatzes und wird in Abschnitt 2.4 betrachtet.

Komponenten erlauben die Modellierung eines hierarchischen Systems, d.h. die interne Struktur einer Komponente kann sich aus mehreren eingebetteten Komponenten zusammensetzen. Es wird dabei zwischen Basiskomponenten und hierarchischen Komponenten unterscheiden. Im Gegensatz zu einer hierarchischen Komponente enthält eine Basiskomponente keine weiteren Komponenten. Ein Beispiel einer hierarchischen Komponente wurde in Abschnitt 1.2 vorgestellt.

Abbildung 2.2 zeigt eine RailCab Komponente, die im Vergleich zu Abbildung 1.2 zwei regelungstechnische Komponenten (*VelocityController* und *DistanceController*) einbettet. Die Schnittstellen der eingebetteten regelungstechnischen Komponenten stellen dabei kontinuierliche Ein-/Ausgangsgrößen dar. Zur Verringerung der visuellen Komplexität wurden in dem Bei-

²In dieser Arbeit wird aus Gründen der besseren Lesbarkeit allgemein von Komponenten gesprochen, wenn durch den Kontext offensichtlich ist, ob Komponententypen oder -instanzen gemeint sind.

spiel die Schnittstellen nicht zur übergeordneten RailCab Komponente weitergeleitet. Der VelocityController regelt die Geschwindigkeit und der DistanceController den Abstand zum vorausfahrenden RailCab. V und d geben die jeweiligen Sollgeschwindigkeiten für die Geschwindigkeit und Distanz vor. Die Eingänge versehen mit einem * geben die Istwerte zur Geschwindigkeit und Distanz an. Der VelocityController benötigt zudem noch die Position des RailCab, die über X angegeben wird. Der VelocityController gibt an dem Ausgang die Beschleunigung (F^*) an und der DistanceController die Geschwindigkeit V^* .

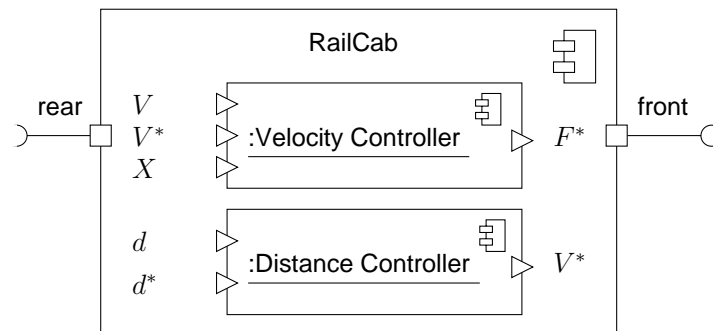


Abbildung 2.2: RailCab Komponente

2.4 Echtzeitverhalten

Systeme, deren Verhalten von Zeitbedingungen/-restriktionen abhängig sind, werden Echtzeitsysteme genannt. Echtzeitsysteme verändern ihren Zustand als eine Funktion über die Zeit. Die Korrektheit eines Ergebnisses einer Funktion hängt damit nicht nur von dem logischen Ergebnis einer Berechnung ab, sondern auch von dem Zeitpunkt, wann dieses Ergebnis vorliegt [Kop97].

Je nach gegebener Anforderung, können wir zwischen harter Echtzeit und weicher Echtzeit unterscheiden. Unter *harter Echtzeit* wird verstanden, dass ein Ergebnis einer Berechnung innerhalb eines bestimmten Zeitfensters (Deadline) vorliegt. Wird diese Zeitfenster nicht eingehalten, können negative oder fatale Konsequenzen entstehen. Bei *weicher Echtzeit* tritt eine positive Wirkung ein, wenn das Zeitfenster eingehalten wird. Wird das Zeitfenster verfehlt, führt das zu einer Verschlechterung des Ergebnisses, allerdings nicht zu fatalen Konsequenzen [Kop97].

Die MECHATRONIC UML ist darauf fokussiert, dass nachrichtenbasierte Echtzeit-Koordinationsverhalten zu beschreiben, das zwischen verschiedenen mechatronischen Komponenten unter harten Echtzeitanforderungen auftritt. Um das Koordinationsverhalten wiederverwenden zu können, werden REAL-TIME COORDINATION PATTERNS eingeführt (siehe Abschnitt 2.4.1). Diese erlauben die Spezifikation von Rollen und Rollenverhalten, die dann durch eine Komponente angewandt werden können.

Das Rollenverhalten wird mit REAL-TIME STATECHARTS beschrieben. REAL-TIME STATECHARTS sind eine Erweiterung von UML State Machines [Obj05b] um spezielle Echtzeiteigen-

schaften für die periodische Ausführung, Echtzeitverhalten, Worst Case Ausführungszeiten und Deadlines zu modellieren. Die Semantik der REAL-TIME STATECHARTS ist über die Semantik der Timed Automata definiert (siehe Abschnitt 2.4.2). Im Folgenden beschreiben wir zuerst REAL-TIME COORDINATION PATTERNS und anschließend REAL-TIME STATECHARTS.

Da mechatronische Systeme zur Laufzeit ihre Struktur anpassen können (z.B. Konvoi, siehe Abschnitt 1.2), kann sich die Kommunikationsstruktur ebenfalls dynamisch zur Laufzeit anpassen. Um diesen Fall betrachten zu können, wurden die PARAMETERIZED REAL-TIME STATECHARTS und PARAMETERIZED REAL-TIME COORDINATION PATTERNS entwickelt (siehe Abschnitt 2.4.4 und 2.4.3).

2.4.1 Real-Time Coordination Pattern

Um das Kommunikationsverhalten von Echtzeitsystemen zu spezifizieren, müssen Nachrichtenverzögerungen berücksichtigt werden und Antwortzeiten garantiert werden. REAL-TIME COORDINATION PATTERNS [GTB⁺03] unterstützen diese Anforderungen. Ein Muster besteht aus den Mitgliedern, die an dem Muster teilnehmen (*Rollen* genannt), das Verhalten der Rollen, den *Konnektor* zwischen den Rollen, das Verhalten der Konnektoren und Invarianten für jede Rolle sowie Musterbedingungen.

Das Verhalten einer Rolle definiert die *externe Kommunikation* (das *Kommunikationsprotokoll*) eines Teilnehmers. Eine Rolle beschreibt nicht das konkrete Verhalten einer Komponente, sondern abstrahiert hiervon. Eine konkrete Komponente muss dieses Verhalten anwenden und darf dabei kein weiteres externes Verhalten hinzufügen. Um dieses Verhalten zu erfüllen, kann eine Komponente zusätzlich internes Verhalten hinzufügen (z.B. das unterlagerte Regelungsverhalten oder konkrete Implementierungen von Seiteneffekten). In Abhängigkeit von den Kommunikationen, an denen eine Komponente teilnimmt, wendet eine Komponente Rollen aus verschiedenen Mustern an. In der Folge einer Anwendung einer Rolle durch eine Komponente wird das Rollenverhalten durch einen Port der Komponente realisiert (konkretisiert). Um die Eigenschaften einer Rolle bzw. Musters nicht zu verletzen muss das Portverhalten eine *Verfeinerung* des Rollenverhaltens sein (siehe Abschnitt 2.4.2).

Die Konnektoren spezifizieren die *Kommunikationsverbindung* (*Link*) zwischen den Kommunikationsteilnehmern (Rollen). Das Verhalten eines Konnektors beschreibt abstrakt das unterliegende Netzwerkverhalten (z.B. UDP), in dem die Qualität des Netzwerkprotokolls, wie Verzögerung oder Nachrichtenverlust, berücksichtigt werden.

Um das Verhalten einer Rolle oder Konnektors zu beschreiben, verwenden wir REAL-TIME STATECHARTS (siehe Abschnitt 2.4.2). Das Verhalten kann, wie wir z.B. in [GHHK06] und [ACE⁺08] gezeigt haben, auch aus Szenarien synthetisiert werden.

Eine *Rolleninvariante* beschreibt Eigenschaften, die durch den Teilnehmer garantiert werden und *Mustereigenschaften* beschreiben Eigenschaften, die durch das Muster erfüllt werden sollen. Rolleninvarianten und Mustereigenschaften beschreiben Sicherheits- und Lebendigkeitseigenschaften. Sicherheitseigenschaften beschreiben, dass etwas Schlechtes niemals passieren wird. Wäh-

rend Lebendigkeitseigenschaften beschreiben, dass etwas Gutes eventuell passieren wird. Im Rahmen von Echtzeitsystemen findet typischerweise eine eingeschränkte Form von Lebendigkeitseigenschaften, die begrenzten Lebendigkeitseigenschaften, Anwendung. Hiermit wird eine zeitliche Obergrenze festgelegt, in der etwas Gutes passieren soll.

Diese Eigenschaften können durch Model Checking basierend auf dem Verhalten der Rollen und Konnektoren verifiziert werden. Im Fall von REAL-TIME STATECHARTS, die über die Semantik von Timed Automata definiert sind (siehe Abschnitt 2.4.2), ist dies mittels des UPPAAL Model Checkers³ möglich.

Um zu zeigen, dass ein Gesamtsystem bzgl. seiner Spezifikation korrekt umgesetzt wurde, wird in einem ersten Schritt überprüft, ob jedes Muster seine Spezifikation erfüllt. Hierzu gehört, dass die Eigenschaften des Musters erfüllt sind und Deadlock-Freiheit gezeigt wurde. In einem zweiten Schritt wird überprüft, ob jede Komponente korrekt ist. Eine Komponente ist korrekt, wenn ihre Eigenschaften erfüllt sind, die Komponente keine Deadlocks enthält, die Invarianten der Rollen eingehalten werden und die angewandten Rollen korrekt verfeinert werden. Wenn diese beiden Schritte erfolgreich sind, dann ist ein syntaktisch korrekt komponiertes System, welches aus Mustern und Komponenten besteht, ebenfalls korrekt [GTB⁺03].

Abbildung 2.3 zeigt das DistanceCoordination-Muster mit den Rollen rear und front, die über einen Kanal miteinander verbunden sind. Die Intention dieses Musters ist die Koordination zwischen zwei hintereinanderfahrenden RailCabs in einem Konvoi zu beschreiben. Dabei befindet sich das vorherfahrende RailCab in der Rolle front und das hinterherfahrende RailCab in der Rolle rear. Das Verhalten der Rollen, welches genauer in dem nächsten Abschnitt 2.4.2 beschrieben wird, muss dabei bestimmte Eigenschaften erfüllen, die dem REAL-TIME COORDINATION PATTERN zugeordnet werden. Zum einen, dies gilt für jedes Muster, muss die Deadlock Freiheit gelten. Zum anderen gibt es musterspezifische Eigenschaften, wie, wenn das RailCab in der Rolle rear im Konvoi (Konvoizustand) ist, dann muss auch das vorherfahrende RailCab im Konvoi (Konvoizustand) sein (rear.convoy implies front.convoy). Diese Eigenschaft wird benötigt, um sicherzustellen, dass ein rechtzeitiges Bremsen des RailCabs in der Rolle rear möglich ist. Hiermit wird impliziert, dass eine entsprechende regelungstechnische Komponente in diesen Zuständen aktiv ist, die z. B. für eine Abstandskontrolle sorgt (siehe Abschnitt 2.5).

Ein weiteres Beispiel ist das REAL-TIME COORDINATION PATTERN Registration. Hiermit wird die Koordination zwischen einem RailCab und der Streckenabschnittskontrolle beschrieben. Die Streckenabschnittskontrolle ist die Entität im RailCab-System, die für die Bestromung des Systems sowie der Bekanntmachung der RailCabs untereinander innerhalb eines Streckenabschnitts zuständig ist. Eine Eigenschaft, die dabei sichergestellt sein muss ist, dass wenn ein RailCab (welches hier in der Rolle registree ist) registriert ist, die zugehörige Streckenabschnittskontrolle (die in der Rolle registrar ist) den gleichen Status hält (registree.registered implies registrar.registered). Ansonsten kann z. B. nicht garantiert werden, dass ein RailCab die relevanten Daten der RailCabs in dem gleichen Streckenabschnitt erhält. Dies ist wiederum eine Voraussetzung, damit die RailCabs sich untereinander koordinieren können.

³www.uppaal.com

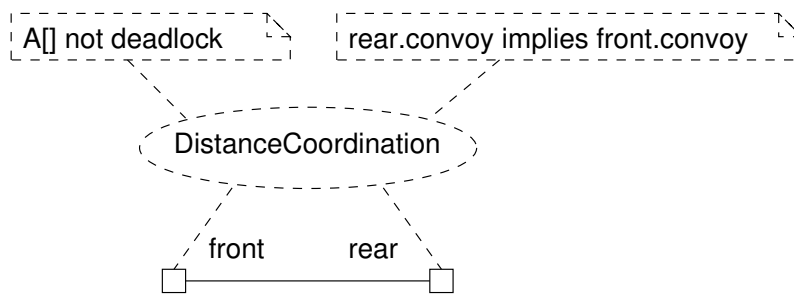


Abbildung 2.3: Convoy-Koordinationsmuster

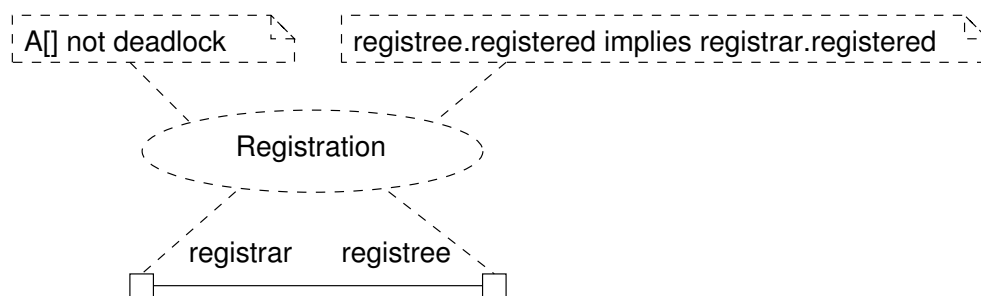


Abbildung 2.4: Registration-Koordinationsmuster

2.4.2 Real-Time Statecharts

REAL-TIME STATECHARTS [GB03] sind eine Erweiterung von UML State Machines [Obj05b], um den Einsatz in eingebetteten Systemen zu ermöglichen. Bis auf das after- und when- Konstrukt besitzt ein Realtime Statechart alle Eigenschaften von UML State Machines. Um Echtzeitverhalten modellieren zu können, werden die Transitionen und Zustände um Clocks erweitert. Zustände werden um Zeitinvarianten, Clock Resets, die mit den entry()- und exit()- Methoden assoziiert sind, WCETs (Worst Case Execution Time) zu den entry()-, do()- und exit()- Methoden und ein Periodenintervall für die do()- Methode erweitert. Transitionen werden um Time-Guards, Clock Resets, Prioritäten, Deadlines, WCETs und Synchronisationskanäle erweitert. Die Semantik der REAL-TIME STATECHARTS ist über Hierarchical Timed Automata [DMY02] definiert, so dass eine formale Verifikation der Modelle mit dem Model Checker UPPAAL ermöglicht wird.

Ein Beispiel für ein REAL-TIME STATECHART zeigt Abbildung 2.5 und Abbildung 2.6. Die Abbildungen zeigen eine Implementierung der Rolle front und rear aus Abbildung 1.2. Das REAL-TIME STATECHART der Rolle front hat zwei verschachtelte Zustände noConvoy und convoy mit Unterzuständen default und wait im Fall des Zustands noConvoy und Unterzustand default im Fall des Zustands convoy. Die Struktur des REAL-TIME STATECHARTS der Rolle rear ist gleich aufgebaut. Die Kommunikation zwischen den beiden Rollen wird durch die Rolle rear initiiert. Initial schickt diese eine convoyProposal-Nachricht, die die front-Rolle empfangen kann und in-

nerhalb des Intervalls von $0 \leq 1000$ den Konvoi über eine startConvoy-Nachricht starten kann und zu einem beliebigen Zeitpunkt durch Verschicken der Nachricht breakConvoy wieder beenden kann.

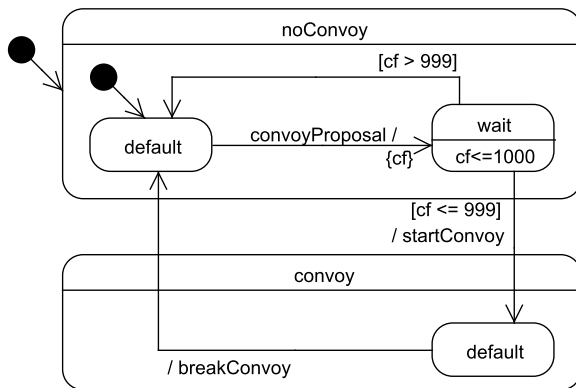


Abbildung 2.5: REAL-TIME STATECHARTS der Rolle front

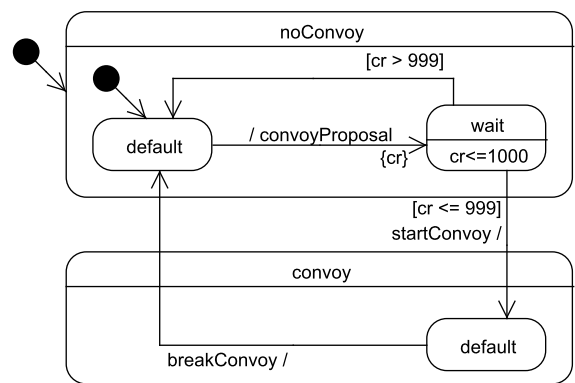


Abbildung 2.6: REAL-TIME STATECHARTS der Rolle rear

Abbildung 2.7 und 2.8 zeigt eine Erweiterung des bisherigen Beispiels um Echtzeitkommunikationsverhalten mit einer Streckenabschnittskontrolle (siehe Abschnitt 2.4.1). Für die Rolle registrar und registree wird ein entsprechendes REAL-TIME STATECHART beschrieben. Beide weisen die gleiche Struktur auf: zwei verschachtelte Zustände unregistered und registered. default ist der Unterzustand des Zustands unregistered. default und waiting sind die Unterzustände von registered. Initiiert wird die Koordination durch eine register-Nachricht der Rolle registree. Innerhalb des Zeitintervalls $ca \leq 2000$ verschickt die registree eine requestUpdate-Nachricht. Vor Betreten des Zustands waiting wird die Clock ta bzw. ce zurückgesetzt (auf null gesetzt). Innerhalb von 500 Zeiteinheiten verschickt die Rolle registrar dann eine performUpdate-Nachricht. Vor Betreten des Zustands default wird dann jeweils wieder die entsprechende Clock (ca bzw. ce) zurückgesetzt.

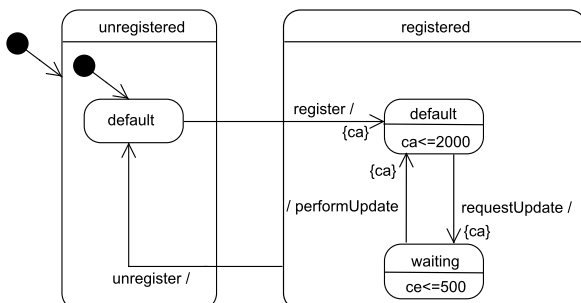


Abbildung 2.7: REAL-TIME STATECHART der Rolle registrar

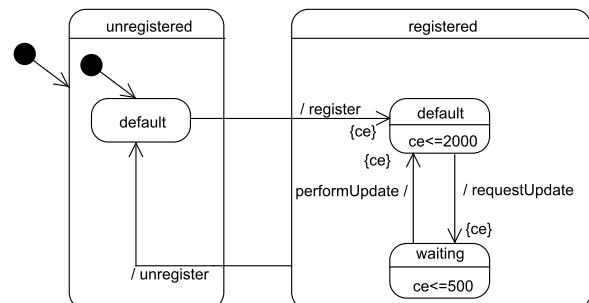


Abbildung 2.8: REAL-TIME STATECHART der Rolle registree

Aus Sicht einer Komponente werden die Rollen im Idealfall unabhängig voneinander ausgeführt. Es existiert also ein übergeordneter Zustand (Railcab in Abbildung 2.9), der die einzelnen Rollen parallel ausführt. Dieses Idealbild kann und wird auch häufig verletzt, indem Abhängigkeiten zwischen den verschiedenen Rollen existieren. Wendet eine Komponente z.B. gleichzeitig die Rolle rear und registree an, so muss gelten, dass das RailCab registriert sein muss, um an einem Konvoi teilzunehmen. Diese Anforderung wird in der MECHATRONIC UML durch ein zusätzliches Synchronisationsverhalten (Beobachter-Automat) realisiert, der durch aktive Synchronisation mit den Portverhalten solche übergreifenden Anforderungen realisiert.

Initial ist das Synchronisationsstatechart im Zustand unregistered. Wenn über den Trigger when(bsAvailable) signalisiert wird, dass eine Streckenabschnittskontrolle in der Nähe ist, wird die Registrierung über die Synchronisation doRegister gestartet. Im Vergleich zu Nachrichten wird eine Synchronisation über ein ! bzw. ? kodiert, wie dies in UPPAAL üblich ist. Im Zustand noConvoy verweilt das Synchronisationsstatechart für wenigstens 2500 Zeiteinheiten, damit das RailCab die aktuellen Streckendaten empfangen kann. Wenn ein Konvoi nützlich für ein RailCab ist (when(convoyUseful)), dann wird über die Synchronisation buildConvoy ein Konvoi initiiert. So lange das RailCab im Zustand convoy ist, kann es nicht in den Zustand register wechseln. Entsprechend kann das RailCab nicht gleichzeitig im Zustand convoy und unregistered sein.

Eine Verfeinerung muss zudem sicherstellen, dass die verifizierten Eigenschaften des Musters (der Rolle) nicht durch das Synchronisationsverhalten verletzt werden (siehe Abschnitt 2.4.1).

2.4.2.1 Formalisierungen

Die bisher informal eingeführten REAL-TIME STATECHARTS werden im Folgenden über Timed Automata definiert. Für eine Abbildung von REAL-TIME STATECHARTS auf Timed Automata sei auf [GB03] verwiesen.

Bei der Verifikation von auf Timed Automata basierenden Systemen ist deren unendlicher Zustandsraum problematisch, an dessen Stelle daher eine geeignete Abstraktion analysiert werden muss. Für diesen Zweck werden oft *Zone Graphen* eingesetzt, welche auch die Grundlage für die in Abschnitt 3 vorgestellten Erreichbarkeitsanalyse und die in Abschnitt 5 eingeführten Synthese darstellen.

Timed Automata [AD90] basieren auf endlichen Automaten und sind, wie diese, ein zustandsbasiertes Verhaltensmodell. Sie definieren ebenfalls Transitionen und Entsprechungen zu den Zuständen endlicher Automaten, die jedoch als *Locations* bezeichnet werden. Der Grund für die unterschiedliche Benennung ist, dass eine Location, für sich genommen, nicht den Gesamtzustand des Modells definiert, da dieser im Unterschied zu (gewöhnlichen) endlichen Automaten auch zeitabhängig ist. Wenn allerdings keine explizite Unterscheidung notwendig ist, werden wir im Rahmen dieser Arbeit den Begriff Zustand auch für Timed Automata verwenden.

Zeitbehaftetes Verhalten wird auf dieselbe Weise modelliert, wie dies bereits für REAL-TIME STATECHARTS (siehe Abschnitt 2.4.2) eingeführt wurde, also mit Clocks, Clock Resets und Time Guards sowie Invarianten, die auf die Clocks Bezug nehmen.

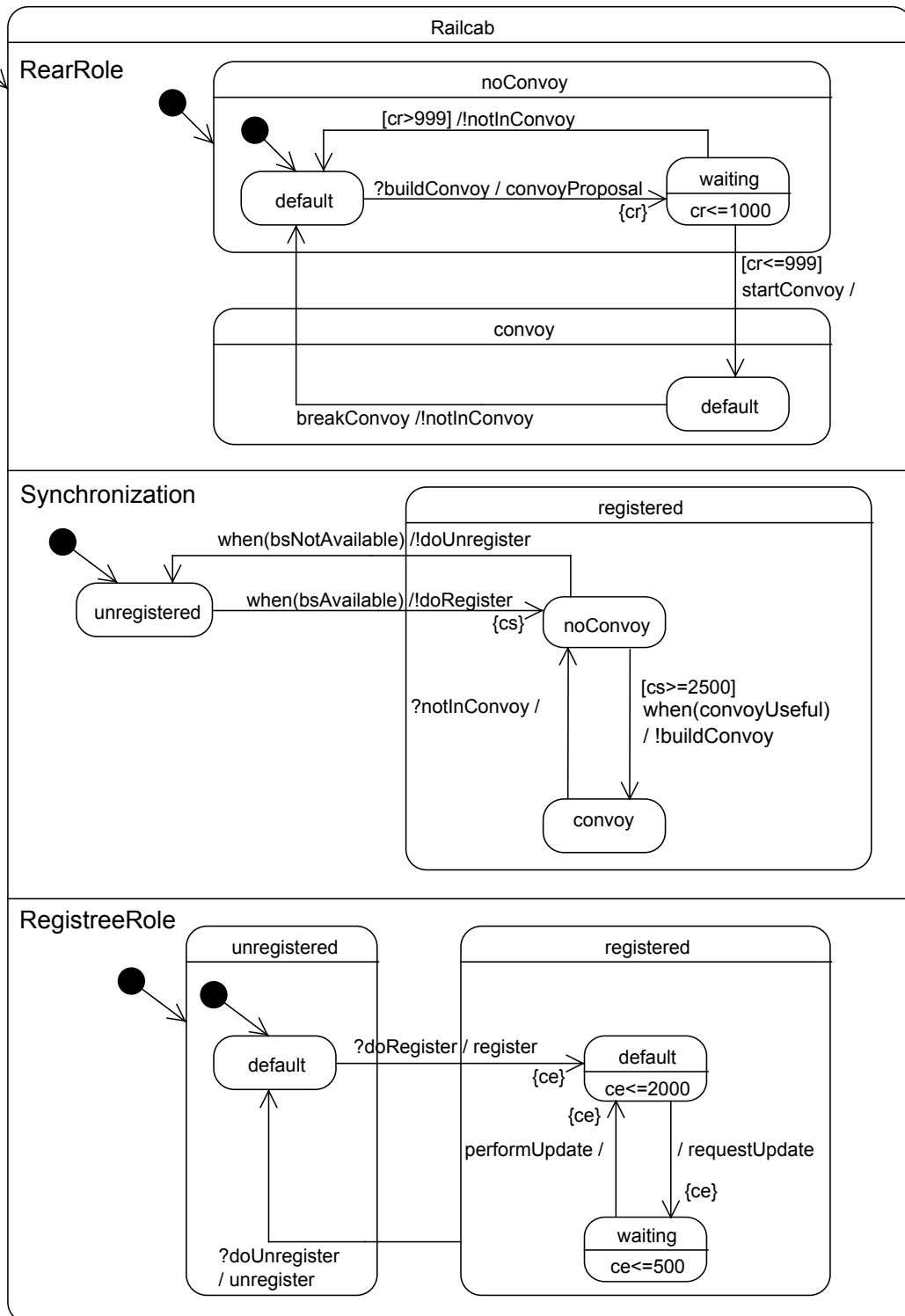


Abbildung 2.9: Verhalten RailCab Komponente

Ein Timed Automaton sei hier basierend auf [GB03] und [JLS00] wie folgt definiert:

Definition 1 (Timed Automaton)

Ein Timed Automaton A ist ein Tupel $(L, l_0, \Sigma, C, I, T)$, wobei

- L endliche, nichtleere Menge von Locations
- $l_0 \subseteq L$ Teilmenge von Startlocations
- Σ eine endliche Menge von Events, mit den internen Events ϵ
- C eine endliche Menge von Clocks
- $I : L \rightarrow \Phi(C)$ ordnet jedem Zustand einen Clock Constraint (Invariante) zu
- $T \subseteq L \times \Sigma \times \Phi(C) \times 2^C \times L$ eine endliche Menge von Transitionen $t = (l, a, g, r, l') \in T$ mit
 - $l \in L$ Quell-Location
 - $a \in \Sigma$ ein Event
 - $g \in \Phi(C)$ ein Clock Constraint (Time Guard)
 - $r \in C$ eine Menge von Clock Resets
 - $l' \in L$ die Ziel-Location

Die parallele Komposition mehrerer Automaten basiert auf der Komposition in Prozessalgebren [Mil89]. Für Timed Automata wurde dies bereits durch die *vernetzten Timed Automata* definiert [YPD94, Pet99, BDL04].

Definition 2 (Parallele Komposition Timed Automata)

Seien $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ zwei Timed Automata mit $C_1 \cap C_2 = \emptyset$ und $\Sigma_1 \cap \Sigma_2 = \emptyset$. Wir definieren die parallele Komposition $A_1 \parallel A_2$ als einen Produktautomat $A_P = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, mit

- $L_P = L_1 \times L_2$,
- $l_P^0 = (l_1^0, l_2^0)$,
- $\Sigma_P = \Sigma_1 \cup \Sigma_2$,
- $I_P : L_P \rightarrow \Phi(C_1) \cup \Phi(C_2)$ mit $I_P((l_1, l_2)) = I_1(l_1) \wedge I_2(l_2)$,
- $C_P = C_1 \cup C_2$,
- $T_P \subseteq L_P \times \Sigma_P \times \Phi(C_P) \times 2^{C_P} \times L_P$, mit
 - $((l_1, l_2), e_1, g_1, r_1, (l_1', l_2)) \in T_P \Leftrightarrow (l_1, e_1, g_1, r_1, l_1') \in T_1$, und
 - $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_P \Leftrightarrow (l_2, e_2, g_2, r_2, l_2') \in T_2$.

Die Menge der Zustände ergibt sich aus dem Kreuzprodukt der Zustände der einzelnen Automaten. Die Nachrichten sind entsprechend eine Vereinigung der separaten Automaten, wie dies auch

für die Uhren C_P gilt. Für Invarianten von komponierten Zuständen $I((l_1, l_2))$ werden die Invarianten der einzelnen Zustände $I(l_1)$ und $I(l_2)$ miteinander verbunden, da beide Invarianten in der parallelen Ausführung betrachtet werden müssen. Die Menge der Transitionen T_P reflektiert genau die verschachtelte, nebenläufige Ausführung der Nachrichten der parallelen Ausführung der separaten Automaten. Die Transition korrespondiert entweder zu Automaten A_1 oder A_2 . Im Vergleich zu [Mil89] betrachtet diese Definition keine Synchronisationen von Nachrichten, da die separaten Rollenautomaten der MECHATRONIC UML unabhängig voneinander sind.

Asynchrone Kommunikation kann durch Modellierung eines Puffers als zusätzlicher Automat auf synchrone Kommunikation abgebildet werden. Im Rahmen dieser Arbeit wird diese Vorgehensweise vorausgesetzt und daher generell von der Verwendung von Synchronisationskanälen ausgegangen.

Die Priorität p modelliert wie bei den RTSCs, dass Transitionen mit höherer Priorität (also höherer Zahl p) bevorzugt zu alternativen Transitionen geschaltet werden müssen. Analog zu [GB03] gelten hier alle Transitionen mit $p > 0$ als urgent, alle mit $p = 0$ als nicht-urgent.

Die verschiedenen existierenden Varianten von Timed Automata unterscheiden sich in einigen Details, beispielsweise darin, ob urgent-Transitionen unterstützt werden oder ob grundsätzlich von einer synchronen oder asynchronen Kommunikation ausgegangen wird. Die hier vorgestellte Variante orientiert sich an derjenigen, die den Extended Hierarchical Timed Automata (ExHTA), einer Erweiterung der Timed Automata, zugrunde liegt, da die Semantik der REAL-TIME STATECHARTS über diese definiert ist.

Die in Definition 1 verwendeten Clock Constraints sind wie folgt definiert.

Definition 3 (Clock Constraint)

Für eine Menge C von Clocks, ist die Menge $\Phi(C)$ von Clock Constraints φ definiert über die Grammatik

$$\varphi := x \leq c, \quad c \leq x, \quad x < c, \quad c < x, \quad x - y \leq c, \quad x - y < c, \quad \varphi_1 \wedge \varphi_2$$

mit $x, y \in C$ Clocks und c ist eine Konstante aus \mathbb{Q} . (vgl. [Alu99, BY03])

Ein Clock Constraint vergleicht den Wert einer Clock mit einer Konstanten oder mit einer anderen Clock.

Timed Transition System Ein Timed Transition System (TTS) [Alu99] ist, ebenso wie ein Timed Automaton, ein zustandsbasiertes, zeitbehaftetes Verhaltensmodell in Form eines Graphen. Im Unterschied zum Timed Automaton repräsentieren die Knoten in diesem Graphen jedoch die tatsächlichen Zustände des Systems, die zusätzlich zur aktuellen Location des dazugehörigen Timed Automaton auch durch die aktuellen Werte aller Clocks bestimmt wird. Selbst bei einem diskreten Zeitmodell ist der Zustandsraum und damit die Größe des TTS unendlich. Wird, wie in dieser Arbeit, ein kontinuierliches Zeitmodell (Clock-Werte aus \mathbb{R}_0^+) vorausgesetzt, dann gibt es sogar überabzählbar viele Zustände.

Der Zustand eines Timed Transition System wird durch ein Paar $(l, v) \in S \times C_v$ charakterisiert, wobei l die aktuelle Location und v die aktuelle *Clock Bewertung* ist. Bei dieser handelt es sich um eine Bindung aller Clocks des Automaten, die als $v \in C_v = [C \rightarrow \mathbb{R}_0^+]$ definiert ist; sie legt also die aktuellen Werte aller Clocks fest. Der Startzustand ist (l_0, v^0) , wobei v^0 eine Bindung ist, die allen Clocks den Wert 0 zuordnet.

Ein Timed Transition System sei hier wie folgt definiert:

Definition 4 (Timed Transition System)

Ein Timed Transition System ist ein Tupel $(S, s_0, \Sigma, \longrightarrow)$ mit Zustandsmenge S , Startzustand $s_0 \in S$, Alphabet $\Sigma = \Sigma_D \cup \Sigma_E$ und Transitionsrelation $T \subseteq S \times \Sigma \times S$. Dabei sei D die Menge der möglichen Delays $\{\delta \mid \delta \in \Sigma_D = \mathbb{R}_0^+\}$ und Σ_E die Menge der Nachrichten (bzw. Synchronisationskanäle), einschließlich des speziellen internen Ereignis τ . Dabei muss T folgende Bedingungen erfüllen (mit $s, s', s'' \in S$ und $\delta, \delta_1, \delta_2 \in \Sigma_D$):

1. $s \xrightarrow{\delta} s' \wedge s \xrightarrow{\delta} s'' \implies s' = s''$ („Zeitdeterminismus“)
2. $s \xrightarrow{\delta_1 + \delta_2} s'' \Leftrightarrow s \xrightarrow{\delta_1} s' \xrightarrow{\delta_2} s''$ mit s' beliebig („Zeitadditivität“)
3. $s \xrightarrow{0} s' \Leftrightarrow s = s'$ („Zero-Delay“)

Auf Basis von Timed Transition Systems kann die Semantik der Timed Automata nach [BY03] wie folgt definiert werden:

Definition 5 (Semantik der Timed Automata)

Die Semantik eines Timed Automaton wird definiert durch ein Timed Transition System, dessen Zustände Paare (l, v) sind und dessen Transitionsrelation T definiert ist durch:

- $(l, v) \xrightarrow{\delta} (l, v + \delta)$, wenn $v \in I(l) \wedge (v + \delta) \in I(l)$ für ein $\delta \in \mathbb{R}_0^+$
- $(l, v) \xrightarrow{\mu} (l', v')$, wenn $(l, g, \mu, r, l') \in T, v \in g, v' = [r \mapsto 0]v$ und $v' \in I(l')$

Dabei sind $v, v' \in \mathbb{R}_0^+$ Clock-Bewertungen. $v \in g$ bedeutet hier, dass die Clock-Bewertung v den Time Guard g erfüllt. Entsprechend gibt $v \in I(l)$ an, dass v die Invariante von Location l erfüllt.

Weiterhin ist $v + \delta$ für $\delta \in \mathbb{R}_0^+$ diejenige Clock-Bewertung, die alle Clocks $c \in C$ auf $v(c) + \delta$ setzt. Für Clock Resets $r \subseteq C$ ist $[r \mapsto 0]v$ die Clock-Bewertung, die alle Clocks in r auf 0 setzt und alle anderen Clocks $c \in C \setminus r$ unverändert lässt.

Zone Graph Ein Timed Transition System eines Timed Automaton kann potentiell unendlich groß sein. Um ein solches System analysieren zu können, wurden *Zone Graphen* [Alu99, BY03] eingeführt, deren Zustände jeweils mehrere Zustände des durch sie repräsentierten Timed Transition Systems mit identischem Zeitverhalten zusammenfassen. Die Zustände des Zone Graph beinhalten neben der jeweils aktuellen Location auch eine sogenannte *Clock Zone*. Letztere repräsentiert die Menge aller Clock-Bewertungen, die für die gegebene Location eine bestimmte Clock Constraint (siehe Definition 3) erfüllen.

Eine Clock Zone fasst TTS-Zustände zusammen, die in derselben Location dasselbe Verhalten beschreiben. Konkret heißt das, dass innerhalb einer Clock Zone für eine gegebene Location immer dieselben Transitionen schalten können. Eine Clock Zone kann für jede Clock eine obere und eine untere Schranke definieren, sie legt also ein Intervall fest, in dem der Wert der Clock liegen muss, um in dieser Clock Zone enthalten zu sein. Eine Clock Zone für n Clocks ist demnach ein konvexer Körper im n -dimensionalen Raum.

Ein Zone Graph wird, ausgehend von einer gegebenen Clock Zone, die dem Startzustand des dazugehörigen Timed Automaton (bzw. dessen TTS) entspricht, konstruiert, indem solange die Nachfolge Zones aller bereits erzeugten Zones konstruiert werden, bis auf diese Weise keine neuen Zones mehr erzeugt werden können. Dafür wird eine Operation verwendet, welche zu einer gegebenen Zone eine mögliche Nachfolge Zone für diese zurück liefert, also eine Zone, die durch Schalten einer Synchronisation oder aber dem Vergehen von Zeit erreicht werden kann. Die Konstruktion des gesamten Zone Graphen entspricht also der Konstruktion der Abgeschlossenheit zu dieser Operation und dem gegebenen Startzustand.

Definition 6 (Clock Zone, Zone, Zone Graph)

Sei $A = (L, l_0, \Sigma, C, I, T)$ ein Timed Automaton. Eine Clock Zone z ist eine \wedge -Verknüpfung mehrerer Clock Constraints über Clocks in C . Eine Zone z_0 ist ein Tupel $\langle s, z \rangle$ mit einer Location $s \in L$ und einer Clock Zone z . Eine Zone beschreibt für die Location s die Menge der zulässigen Clock-Bewertungen, die äquivalent zueinander sind. Ein Zone Graph wird ausgehend von der Clock Zone des Startzustands l_0 konstruiert. Es wird dabei solange eine Nachfolge Zone der bereits erzeugten Zones konstruiert, bis keine neuen Zones erzeugt werden können. Ein Zone Graph ist entsprechend definiert durch Zustände die Zones sind und der Transitionen zwischen zwei Zones, falls A einen Übergang zwischen diesen Zones erlaubt.

Difference Bound Matrice Difference Bound Matrices [Dil89, CGP00] sind eine effiziente Form der Repräsentation von Clock Zones über eine Matrix. Für eine Clock Zone mit n Clocks erhält man eine $n \times n$ -Matrix, deren i -te Zeile und Spalte zu Clock i der Clock Zone gehören. Zusätzlich zu den Clocks der Clock Zone wird eine Clock x_0 eingeführt, deren Wert immer 0 ist. Dies erlaubt es, Vergleiche einer Clock mit einer Konstanten als Differenz über x_0 darzustellen, d.h. $x < c$ wird zu $x - x_0 < c$. Der Eintrag $d_{i,j}$ der Matrix hat die Form (c, \prec) , wobei c für eine Konstante oder ∞ steht und \prec für $<$ oder \leq . Er kodiert damit die Ungleichung $x_i - x_j \prec c$. Ein Beispiel für eine Difference Bound Matrix zu der Clock Zone $0 \leq t \wedge t \leq 10$ zeigt Abbildung 2.10.

	0	1
0	(0, \leq)	(0, \leq)
1	(10, \leq)	(0, \leq)

Abbildung 2.10: Difference Bound Matrice für eine Clock Zone mit einer Clock.

Die Einträge der 0-ten Spalte entsprechen den oberen Schranken der Clocks, die Einträge der 0-ten Zeile entsprechen den negierten unteren Schranken der Clocks. Die Einträge auf der Diagonalen sind immer $(0, \leq)$, da hier eine Clock mit sich selbst verglichen wird.

Difference Bound Matrices erlauben durch eine Normalisierung eine kanonische Darstellung von Clock Zones und somit einen einfachen Vergleich, ob zwei Clock Zones identisch sind. Diese Eigenschaft wird für die Verifikation der Verfeinerung benötigt.

Der *Zustand* eines Timed Automaton ist der Zustand des diesem entsprechenden Timed Transition Systems. Die Semantik von Zustandsübergängen und damit die gesamte Semantik des Timed Automaton wird daher über Timed Transition Systems definiert.

2.4.3 Parameterized Real-Time Coordination Pattern

Bei REAL-TIME COORDINATION PATTERNS muss die Anzahl der beteiligten System-Instanzen bei der Definition des Musters statisch festgelegt werden. Dieser Mangel an Flexibilität ist allerdings beim Modellieren von Situationen problematisch, in denen neue Kommunikationsteilnehmer hinzukommen oder vorhandene die Kommunikationsbeziehung verlassen. Derartige dynamische Änderungen der Kommunikationsstruktur kommen bei verteilten eingebetteten Systemen relativ häufig vor. Um solche Fälle modellieren zu können, wurde daher das Konzept der REAL-TIME COORDINATION PATTERNS zu dem der PARAMETERIZED REAL-TIME COORDINATION PATTERNS [GHH⁺06c, Hir08, HHG08, HHH10, HHPS10] erweitert.

PARAMETERIZED REAL-TIME COORDINATION PATTERNS können neben gewöhnlichen Rollen an deren Stelle auch *Multi-Rollen* enthalten. Diese stehen jeweils für mehrere Instanzen einer Rolle. Das Rollenverhalten wird daher im Allgemeinen nicht durch ein einfaches, sondern durch ein *parametrisiertes Realtime Statechart* beschrieben (siehe Abschnitt 2.4.4). Zusätzlich ist die Angabe einer *Multiplizität* (auch: *Kardinalität*) der Rolle, der oberen Grenze für die Anzahl von Instanzen (n für unbeschränkt), möglich. Optional kann zudem durch ein spezielles Attribut $\{ordered\}$ spezifiziert werden, dass die einzelnen Rollen-Instanzen geordnet sein müssen.

Um das Hinzufügen und Entfernen von Rolleninstanzen zu beschreiben, definiert ein parametrisiertes Koordinationsmuster zusätzlich eine Menge von *Erweiterungsregeln* sowie eine Menge von *Reduzierungsregeln*. Bei diesen handelt es sich jeweils um zeitbehaftete Graphtransformationssysteme (*Timed Graph Transformation Systems (TGTS)*), einer speziellen Form von *Graphtransformationsregeln*. Diese Strukturanpassungen werden durch einen Seiteneffekt der Rollenverhalten implementiert (siehe Abschnitt 2.4.5). TGTS werden in Abschnitt 2.4.5.1 behandelt.

Weiterhin definiert ein PARAMETERIZED REAL-TIME COORDINATION PATTERN eine Menge von *Profilen*, sowie zusätzliche Eigenschaften für diese. Sie beschreiben das kontinuierliche Systemverhalten in einer Konfiguration. Für diese Arbeit sind diese allerdings nicht von weiterer Relevanz und es sei daher auf [Hir08] für Details verwiesen.

Zusätzlich zu den bereits bei REAL-TIME COORDINATION PATTERNS möglichen Einschränkungen des Verhaltens durch Eigenschaften können bei PARAMETERIZED REAL-TIME COOR-

DINATION PATTERNS zusätzlich *verbotene Strukturregeln* spezifiziert werden, die unsicheren Konfigurationen des Systems entsprechen (siehe Abschnitt 2.4.5.1).

Abbildung 1.2 zeigt das PARAMETERIZED REAL-TIME COORDINATION PATTERN Convoy Coordination, mit den Rollen coordinator und member. An Stelle des einfachen Quadrats für eine Rolle, wird eine Multi-Rolle durch zwei überlappende Quadrate dargestellt (siehe coordinator Rolle). Multi-Rollen mit der Multiplizität 1 entsprechen dabei einfachen Rollen.

2.4.4 Parameterized Real-Time Statecharts

PARAMETERIZED REAL-TIME STATECHARTS wurden als Erweiterung der REAL-TIME STATECHARTS eingeführt, um mehrere Instanzen desselben Statecharts mit leicht verändertem Verhalten modellieren zu können. Die Einführung erfolgte in [Hir08] zusammen mit den PARAMETERIZED REAL-TIME COORDINATION PATTERNS, in deren Kontext sie für mehrfach instanziierte Komponenten (welche Multi-Rollen implementieren) verwendet werden.

PARAMETERIZED REAL-TIME STATECHARTS umfassen grundsätzlich die Syntax der gewöhnlichen RTSCs, wobei auch die Semantik dieselbe bleibt. Die wesentliche Erweiterung ist, dass Instanzen von PARAMETERIZED REAL-TIME STATECHARTS ein Parameter k zugeordnet wird, welcher eine eindeutige ID der Instanz darstellt.

Auf den Parameter des PARAMETERIZED REAL-TIME STATECHARTS kann direkt in Guards an Transitionen zugegriffen werden, in denen diese mit numerischen Konstanten verglichen werden können. Die Transition darf dann also nur für bestimmte Instanzen schalten, während das betreffende Verhalten für die übrigen ausgeschlossen wird. Prinzipiell ist es also möglich, für bestimmte Parameter k völlig eigene Unter-Statecharts (beispielsweise in Form von OR-Zuständen) zu definieren.

Eine weitere Anwendung der Parameter ermöglichen die in PARAMETERIZED REAL-TIME STATECHARTS ebenfalls neu eingeführten *parametrisierten Synchronisationskanäle*: Sie werden zusätzlich zu ihrem Namen anhand eines eigenen Parameters identifiziert, der als Index angegeben werden kann (also beispielsweise $x_3!$ für ein Senden über Kanal x mit Parameter 3). Dieser muss für eine Synchronisation ebenfalls identisch sein. Der Kanal-Parameter kann in einem PARAMETERIZED REAL-TIME STATECHART durch einen numerischen Ausdruck, beispielsweise eine Addition oder eine einzelne Variable bestimmt werden. Insbesondere ist aber auch ein Bezug auf den Parameter des PARAMETERIZED REAL-TIME STATECHARTS möglich.

PARAMETERIZED REAL-TIME STATECHARTS werden wie REAL-TIME STATECHARTS über Timed Automaton definiert. Hierbei handelt es sich allerdings ebenfalls um eine parametrisierte Version, also parametrisierte Timed Automaton, die die oben beschriebenen Erweiterungen gegenüber Timed Automaton beinhalten.

Definition 7 (Parametrisierter Timed Automaton [Hir08])

Ein parametrisierter Timed Automaton A ist ein 7-Tupel $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, \text{Sig}(l, P), T)$, wobei Σ ein endliches Eingabealphabet, \mathcal{S} eine endliche Menge an Locations, $\mathcal{S}^0 \subseteq \mathcal{S}$ eine endliche Menge von Start-Locations, $X := (x_1, \dots, x_n)$ eine endliche Menge an Clock Variablen

mit $x_i \in \mathbb{R}^+$, I eine Zuordnungsfunktion $I \rightarrow \mathcal{C}(X)$, welche den einzelnen Locations eine Menge an Ungleichungen zuordnet, die so genannten Invarianten, $Sig(l, P)$ eine Menge von Signalen die mit l parametrisiert und die Eigenschaft P ist hierbei eine spezielle Eigenschaft/Profil des Automaten. T ist die Menge der Transitionen. $\mathcal{C}(X)$ ist eine Menge von Bedingungen über Clock-Variablen aus X . Dabei besteht $\mathcal{C}(X)$ aus einer Menge an Ungleichungen der Form $x_i \prec c \vee c \prec x_i$, wobei \prec entweder $<$ oder \leq ist und $c \in \mathbb{N}^+$. Für T , die Menge der Transitionen, gilt $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{C}(X) \times 2^X \times Sig(l, p) \times \mathcal{S}$. Eine Transition von Location s nach s' läßt sich durch ein 6-Tupel $(s, a, \varphi, \lambda, sig, s')$ beschreiben. Dabei ist $a \in \Sigma$ die Beschriftung der zugehörigen Kante, φ eine Bedingung, die erfüllt sein muss damit die Transition schalten kann und $\lambda \subseteq X$ eine Anzahl an Clockvariablen, die beim Schalten auf 0 zurückgesetzt werden. $sig \subseteq Sig(l, P)$ ist ein durch einen Parameter l gekennzeichnetes Signal, das den Wert $p \in P$ übermittelt.

Die parallele Komposition (II) zweier parametrisierter Automaten A^i und A^j ist wie folgt definiert:

Definition 8

Gegeben sei ein parametrisierter Timed Automaton $A^i := (\Sigma^i, \mathcal{S}^i, \mathcal{S}^{0i}, X^i, I^i, Sig(l^i, P^i), T^i)$ und ein parametrisierter Timed Automaton $A^j := (\Sigma^j, \mathcal{S}^j, \mathcal{S}^{0j}, X^j, I^j, Sig(l^j, P^j), T^j)$ wie in Definition 7 definiert. Jeder Automat verhält sich lokal wie ein Timed Automaton. Nur über die parametrisierten Signale $Sig(l^i, P^i)$ und $Sig(l^j, P^j)$ findet eine Synchronisation statt, wenn $i = j$ ist. Dabei wird $P^j := P^i$, falls $i \leq j$

2.4.5 Rekonfigurationsverhalten

Rekonfigurationsverhalten wird in der MECHATRONIC UML nicht zustandsbasiert, sondern durch Graphtransformationen [Roz97] beschrieben, die sich auf den Objektgraphen der aktuellen Konfiguration (auch: Instanzsituation) des Systems beziehen. Graphtransformationen beschreiben dabei im Allgemeinen Änderungen eines Graphen, in diesem Fall eine Rekonfiguration des Systems, also eine Strukturänderung zur Laufzeit. Der Prozess der graphbasierten Verhaltensmodellierung, der in MECHATRONIC UML eingesetzt wird, wird als *Story Driven Modeling* bezeichnet.

Rekonfigurationsverhalten wird als Seiteneffekte an Transitionen von (PARAMETERIZED) REAL-TIME STATECHARTS aufgerufen. Das Rekonfigurationsverhalten wird mittels *Story-Diagrammen* spezifiziert, die in Abschnitt 2.4.5.4 behandelt werden.

Bevor in Abschnitt 2.4.5.2 die Graphtransformationen selbst behandelt werden, werden zunächst Objektgraphen im folgenden Abschnitt 2.4.5.1 betrachtet.

2.4.5.1 Objektgraphen

Wir führen im Folgenden Objektgraphen nach Zündorf ein [Zün01]. Es handelt sich hierbei um gerichtete, typisierte und attributierte Graphen mit Beschriftungen an Kanten und Knoten. Typ-

informationen werden durch eine *Schema Information* festgelegt, die zudem die möglichen Attribute und Assoziationen der einzelnen Typen definiert (und einschränkt). Die Schema Information wird durch Klassendiagramme definiert. Im Fall der MECHATRONIC UML Komponenten werden diese aus Komponentendiagrammen generiert. Der Graph eines Objektdiagramms wird als *Extension* der jeweiligen Schema Information bezeichnet. Ein Objektgraph ist damit wie folgt definiert:

Definition 9 (Objektgraph)

Ein Objektgraph G ist ein Tupel (SI, Ext) mit SI Schema Information und Ext Extension der Schema Information.

$SI := (NL, EL, A, IsAs, Assoc, Attrs)$ mit

- NL , Endliche Menge von Knotenbeschriftungen
- EL , Endliche Menge von Kantenbeschriftungen
- A , Endliche Menge von Attributnamen
- $IsAs \subseteq Relation(desc \in NL, anch \in NL)$, Vererbungsbeziehung
- $Assoc \subseteq Function((el \in EL) \rightarrow (src \in NL, srcCard \in \{one, many\}, assocType \in P(AssocTypes := \{ordered, qualified, aggregation\}), tgt \in NL, tgtCard \in \{one, many\}))$, Assoziationen
- $Attrs \subseteq Function((A) \rightarrow NL \times BaseTypes)$, Attribute von Knoten

$BaseTypes$ sind alle Grunddatentypen wie z.B. Integer, Float, Boolean, String, usw.

$Ext := (V, E, nl, av)$

- V , Endliche Menge von (eindeutig identifizierbaren) Objekten
- $E \subseteq Relation(src \in V, el \in EL, tgt \in V)$, Menge von beschrifteten Kanten
- $nl : V \rightarrow NL$, Funktion, die jedem Knoten einen Namen zuordnet
- $av : (N, A) \rightarrow$ Attributwerte, Attributwertfunktion, die jedem Attribut eines Knotens einen Wert zuweist.

Attributwerte sind alle Instanzen der $BaseTypes$ oder Referenzen auf andere Objekte.

2.4.5.2 Graphtransformationssysteme

Graphtransformationen [Roz97] beschreiben Änderungen auf Graphen (häufig *Wirtsgraph* genannt). Ein Graph kann dabei z.B. als ein Objektgraph gegeben sein (siehe Definition 9). Die Änderungen auf den Graphen werden durch Regeln beschrieben. Eine Regel wird über jeweils zwei Teilgraphen definiert. Diese werden nach der Seite bezeichnet, auf der sie in der Regel dargestellt sind. *LHS* (Left Hand Side) für den Graphen auf der linken Seite der Regel und *RHS* (Right Hand Side) für den auf der rechten.

Die Anwendungsbedingung der Regel wird durch die LHS beschrieben. Als Voraussetzung für die Anwendung der Regel, muss die hierdurch definierte Struktur eine Entsprechung im Wirts-

graphen haben. Es handelt sich hierbei um einen *Homomorphismus* der LHS zu einem Teilgraphen des Wirtsgraphen. Dies bezeichnen wir mit *Matching*.

Eine Zuordnung sämtlicher Elemente eines Graphen zu Elementen (Knoten, Kanten und Label, einschließlich Attributen und Typen) eines anderen Graphen unter Einhaltung von deren Struktur ist ein Homomorphismus. Soll ausgeschlossen werden, dass mehrere Elemente der LHS auf dasselbe Element des Wirtsgraphen abgebildet werden können, wird ein bijektiven Homomorphismus (genannt *Isomorphismus*) gefordert.

Soll die Anwendbarkeit einer Regel *ausgeschlossen* werden, kommen sogenannte *Negative Anwendungsbedingungen* zum Einsatz. Die anderen Elemente des Wirtsgraphen, die in der LHS nicht enthalten sind, schließen eine Anwendung *nicht* aus, auch wenn sie in einem strukturellen Zusammenhang mit denen in der LHS enthaltenen stehen.

Die durch die Regel vorgenommene Änderung wird durch die RHS der Regel zusammen mit der LHS definiert. Für die Änderung ist der Unterschied zwischen diesen beiden relevant. Wir können dabei zwischen den Fällen unterscheiden, 1) in denen das Element auf beiden Seiten vorkommt. In diesem Fall verbleibt es bei Anwendung der Regel im Wirtsgraphen. 2) Das Element kommt nur in der RHS vor. Dann wird bei Regelanwendung ein entsprechendes neues Element im Wirtsgraphen erzeugt. 3) Das Element kommt nur in der LHS vor. In diesem Fall muss das Element im Matching enthalten sein. Bei Anwendung der Regel im Wirtsgraphen wird das Element gelöscht.

Hiermit kommen wir nun schließlich zu der Definition eines *Graphtransformationssystems* (GTS). Ein GTS wird durch eine Menge von Graphtransformationen und einem Typgraphen definiert. Ein Zustand eines solchen Systems entspricht dabei einem Graphen.

Definition 10 (Graphtransformationssystem)

Ein Graphtransformationssystem $\mathcal{G} = \langle TG, TR \rangle$ ist ein 2-Tupel aus einem Typgraphen TG gemäß Definition 9 und einer Menge von Transformationsregeln TR . Die Menge $GRAPH_{TG}$ bezeichnet die Menge aller Objektgraphen über TG .

Der Typgraph TG wird im Rahmen dieser Arbeit durch ein Klassendiagramm beschrieben. Die Elemente der Menge $GRAPH_{TG}$ sind Objektdiagramme des Klassendiagramms. Damit entsprechen das Klassendiagramm der Schemainformation und die Objektdiagramme den Extensions aus Definition 9. Die Menge TR besteht aus Graphtransformationen.

2.4.5.3 Zeitbehaftete Graphtransformationssysteme

Eine Anforderung der hier betrachteten Systeme ist, auch zeitliche Bedingungen beschreiben zu können. In [Hir08] wurden daher Graphtransformationssysteme zu zeitbehafteten Graphtransformationssystemen erweitert. Hiermit kann eine Spezifikation erfolgen, so dass die Ausführung der Graphtransformation nur unter bestimmten zeitlichen Bedingungen erfolgen darf oder dass eine bestimmte Instanzsituation nur für eine bestimmte Zeit vorliegen darf. Im Folgenden werden die durch [Hir08] zusätzlich eingeführten Elemente beschrieben.

Clocks: Clocks wurden basierend auf der Theorie der Timed Automata nach [AD94, Alu99, CGP00] eingeführt. Durch eine Clock wird das Vergehen von Zeit über die Menge \mathbb{R}_0^+ beschrieben. Wie bei einem Timed Automaton kann ein zeitbehaftetes GTS eine Menge von Clocks haben. In [Hir08] wird eine Abbildung der zeitlichen Elemente eines Timed Automaton auf Graphtransformationenregeln beschrieben. Im Unterschied zu einem Graphen, der über Graphtransformationenregeln aufgebaut und verändert wird, sind die Elemente eines Timed Automaton bereits zu Beginn der Ausführung vollständig vorhanden und verändern sich auch während der Ausführung nicht. Daher werden die zeitlichen Elemente mit den Graphtransformationenregeln assoziiert. Im Folgenden betrachten wir dies genauer.

Clock-Instanzen Zeitliche Bedingungen werden über die in Abschnitt 2.4.2.1 beschriebenen Clocks definiert. Die Werte einer Clock sind abhängig vom Erzeugungszeitpunkt der Elemente. Da grundsätzlich eine Clock von mehreren Elementen genutzt werden kann, müsste die Clock unterschiedliche Werte für verschiedene Elemente annehmen können, um unterschiedliche Erzeugungszeitpunkte von Elementen gerecht zu werden. Hirsch führt daher Clock-Instanzen ein. Clock-Instanzen gelten für eine bestimmte Menge von Elementen aus dem Graphen. Dabei hat die Clock-Instanz Referenzen auf alle Objekte des aktuellen Graphen. Clock-Instanzregeln können automatisch aus den LHS der Anwendungsregeln erzeugt werden, die diese Clock-Instanzen benutzen.

Time Guards Ein Graphtransformationssystem mit einem Time Guard zeigt Abbildung 2.11. Das Beispiel zeigt das Erweitern des Konvois um ein RailCab. Der Time Guard wird als zeitliche Bedingung über die Clock c1 angegeben. Die Transformation der LHS zur RHS ist im Beispiel nur möglich, wenn der Wert der Clock c1 zwischen 5 und 10 liegt. Im Allgemeinen wird hier ebenfalls durch einen Time Guard spezifiziert, dass eine Bedingung über die Werte einer oder mehrerer Clocks erfüllt sein muss, damit eine Transition schalten kann.

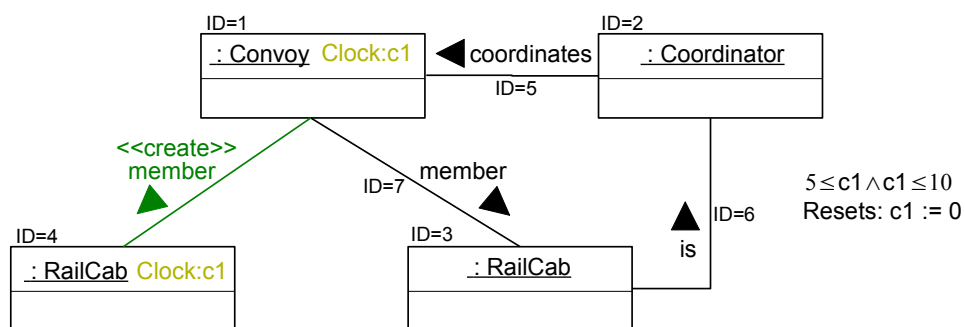


Abbildung 2.11: Erweiterung des Konvois um ein RailCab mit einem Time Guard und einem Clock Reset

Resets Durch einen Clock Reset wird der Wert einer Clock auf 0 zurück gesetzt. Zeitbehaftete GTS führen Clock Resets aus, wenn die Transformation, an der sie gebunden ist, abgeschlossen wurde. Abbildung 2.11 zeigt einen Clock Reset für Clock-Instanz c1.

Invarianten Invarianten beschreiben einen Teilgraphen, der nur für eine bestimmte Zeit im Graphen vorkommen darf. Invariantenregeln haben nur eine LHS, da die Invariante nur im Graphen gefunden werden muss. Abbildung 2.12 zeigt ein Beispiel für eine Invariantenregel. Hiermit wird ausgedrückt, dass ein RailCab, das nicht im Konvoi ist, nur für 10 Zeiteinheiten im System existieren darf.

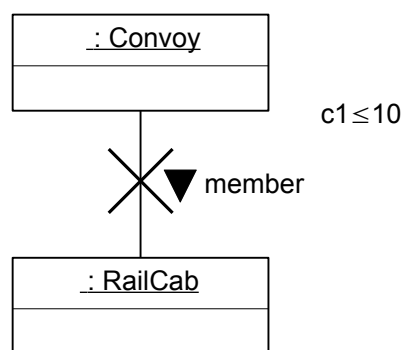


Abbildung 2.12: Eine Invariantenregel über einen Teilgraphen

2.4.5.4 Story Patterns und Story Diagramme

In der MECHATRONIC UML werden Graphtransformationsregeln durch *Story Patterns*, einer speziellen Notation für Graphtransformationen, definiert. Sie können in *Story-Diagrammen*, einer Erweiterung der UML-Aktivitätsdiagramme, in einen sequentiellen Kontrollfluss eingebunden werden. An Transitionen von REAL-TIME STATECHARTS können diese wiederum als Seiteneffekte aufgerufen werden.

Story Patterns Story Patterns beschreiben die Spezifikation von LHS und RHS einer Graphtransformationsregel in einem einzigen Graphen abgekürzt [Zün01]. Elemente, die Voraussetzung der Anwendung sind (aber nicht gelöscht werden), werden in Story Patterns in schwarz dargestellt. Zu löschende Elemente (nur in LHS) werden in rot notiert und zudem mit dem Stereotyp `<< -- >>` (alternativ: `<< destroy >>`) versehen. Erzeugte Elemente (nur in RHS) werden in grün notiert und mit `<< ++ >>` (oder `<< create >>`) gekennzeichnet. Für die Anwendung eines Story Patterns muss ein Isomorphismus (siehe Abschnitt 2.4.5.2) des durch die rot und schwarz dargestellten Elemente definierten Subgraphen des Patterns zu einem Subgraphen des Wirtsgraphen bestehen. Hierdurch wird vermieden, dass eine Regel für einzelne Elemente gleichzeitig den Erhalt und Löschung fordert. Für die Kanten gelöschter Knoten gilt, dass diese

ebenfalls entfernt werden, auch wenn dies nicht explizit durch die Regel gefordert wird. So werden „dangling edges“ vermieden. Das sind Kanten, die nicht zwei Knoten miteinander verbinden (ein offenes Ende besitzen).

In Abbildung 2.13 wird ein Story Pattern gezeigt, welches die Aufnahme eines RailCabs in den Konvoi spezifiziert. Die Assoziation vom Objekt Convoy zu dem Objekt RailCab wird in dem Pattern neu erstellt.

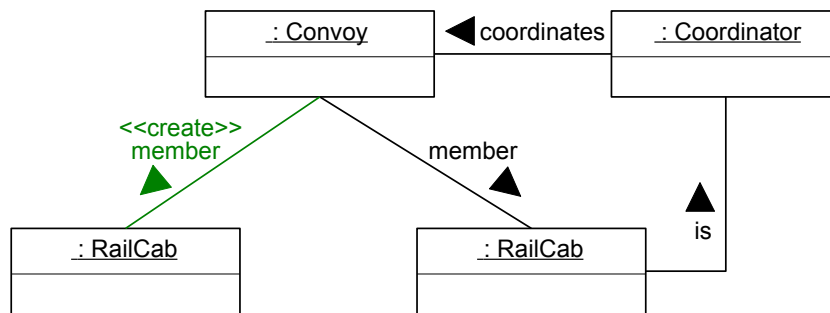


Abbildung 2.13: Ein Story Pattern zur Erweiterung des Konvois um ein RailCab

Das Story Pattern in Abbildung 2.14 beschreibt das Austreten eines RailCabs aus dem Konvoi. Dazu wird die entsprechende Kante member zwischen dem Objekt Convoy und dem betreffenden Objekt RailCab entfernt.

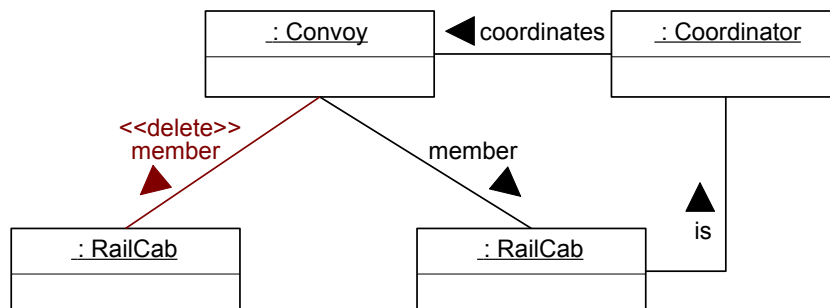


Abbildung 2.14: Ein Story Pattern zur Reduzierung des Konvois um ein RailCab

Story-Diagramme In [Zün01] wurden Story-Diagramme eingeführt, um zusätzlich zu den regelbasierten Änderungen auf Graphen auch einen Kontrollfluss durch zustandsbasiertes Verhalten verbindlich vorzugeben. Story Diagramme sollen damit eine Kombination der UML-Aktivitätsdiagramme [Obj05b] mit Story Patterns beschreiben. Hierdurch werden Folgen von Aktivitäten festgelegt sowie Fallunterscheidungen in der Syntax der Aktivitätsdiagramme. Innerhalb der einzelnen Aktivitäten, die in Story-Diagrammen als *Stories* bezeichnet werden, können Story Patterns definiert werden. Für diese wird im Unterschied zu Graphtransformationssystemen nur dann Matching gesucht, wenn sie vom Kontrollfluss der Story-Diagramme erreicht werden.

Zu einer Story kann zudem eine Bedingung spezifiziert werden. Diese Bedingung ist dabei im Allgemeinen eine aussagenlogische Formel, die mehrere Teilbedingungen über die Elemente, wie Vergleiche von Attributwerten, miteinander verknüpfen kann. Ein Story Pattern wird nur dann ausgeführt, wenn die Bedingung mit *true* ausgewertet wird. Zusätzlich kann bei den ausgehenden Transitionen der einzelnen Stories danach unterschieden werden, ob ein Matching gefunden werden konnte, oder nicht. Kann ein Matching gefunden werden, wird die Transition mit dem Label [*success*] geschaltet, andernfalls die mit dem Label [*failure*]. Im Erfolgsfall bleiben die Objektbindungen der verlassenen Story in der nächsten gültig.

Um alle möglichen Matchings eines Story Patterns zu betrachten, werden *iterierte Stories* eingeführt, die durch doppelte Rahmen dargestellt werden. So lange ein Matching gefunden werden kann wird die Story durch eine Transition mit dem Label [*eachtime*] verlassen. Andernfalls wird eine zweite ausgehende Transition mit [*end*]-Label geschaltet.

Durch Story-Diagramme werden Methoden(aufrufe) modelliert. Daher werden sie auch ausgenutzt, um Seiteneffekte an REAL-TIME STATECHART-Transitionen zu beschreiben. Eine Besonderheit ist dabei, dass innerhalb der einzelnen Stories das *this*-Objekt immer an dasjenige Objekt gebunden ist, auf dem die Methode aufgerufen wird. Innerhalb von Story-Diagrammen können ebenfalls Methoden aufgerufen werden. Dies ist mittels *Collaboration Messages* für jedes innerhalb einer Story gebundene Objekt möglich. Der Methodenaufruf wird dazu in der Syntax der Zielsprache der für Story-Diagramme verwendeten Codegenerierung (beispielsweise Java) textuell in der Story aufgeführt. Ein von diesem Text ausgehender Pfeil zeigt auf das Objekt, auf das sich der Aufruf bezieht. Ist kein Pfeil angegeben, so ist das aktuelle Objekt *this* gemeint.

Ein Beispiel für ein Story Diagramm zeigt Abbildung 2.15. Die Methode `addMember()` beschreibt, dass ein RailCab in den Konvoi aufgenommen wird und anschließend die Anzahl der RailCabs im Konvoi um 1 erhöht wird. Das Objekt `Convoy`, auf dem die Methode aufgerufen wird, ist in jeder Story über das *this*-Objekt gebunden und dient als Referenzpunkt für das Matching des Story Patterns.

2.4.6 Verifikation

In Abschnitt 2.4.1 und 2.4.3 haben wir (parametrisierte) REAL-TIME COORDINATION PATTERNS eingeführt. Komponenten können mehrere Rollen unterschiedlicher Koordinationsmuster anwenden. Das Verhalten der Komponente muss das Verhalten der angewandten Rollen verfeinern. Die Auftrennung von Kommunikationsverhalten und Komponentenverhalten führt zu einem kompositionellen Modell, welches in der Verifikation ausgenutzt wird [GTB⁺03].

Wie bereits in Abschnitt 2.4.1 beschrieben, müssen wir zum einen zeigen, dass das Muster korrekt ist und zum anderen, dass die Komponente korrekt ist. Um die Korrektheit zu zeigen werden dabei *Model Checker* (im speziellen UPPAAL) eingesetzt, die die Gültigkeit einer Eigenschaft über das (Teil-)System überprüfen bzw. eine Verfeinerung überprüft, um zu zeigen, dass eine Komponente korrekt eine Rolle verfeinert.

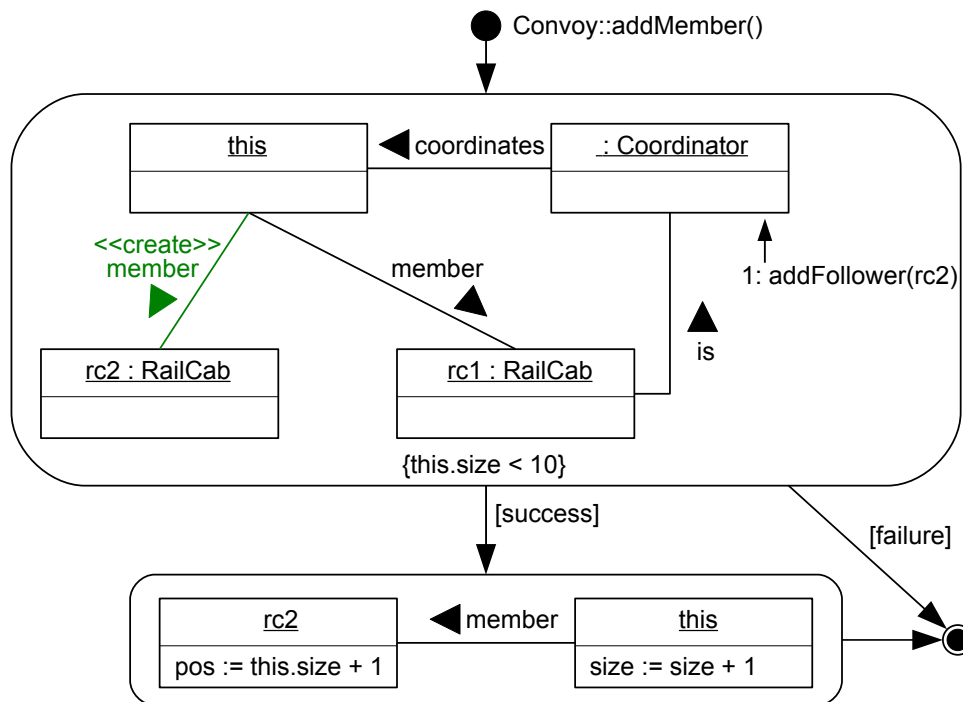


Abbildung 2.15: Ein Story Diagramm zur Erweiterung des Konvois um ein RailCab

Üblicherweise werden die Eigenschaften in Form von *temporallogischen Formeln* beschrieben. Diese werden wir im Folgenden Abschnitt 2.4.6.1 näher betrachten. In Abschnitt 2.4.6.2 diskutieren wir den Ansatz der kompositionellen Verifikation und anschließend in Abschnitt 2.4.7 erläutern wir einige relevante Verfeinerungsbeziehungen.

2.4.6.1 Eigenschaften

Die Spezifikation von Anforderungen an ein System können mit *Temporallogiken* beschrieben werden. Temporallogiken beziehen sich auf die zeitliche Abfolge von Zuständen oder Ereignissen. Eine temporallogische Formel, ist für ein System genau dann erfüllt, wenn dessen Verhalten die durch die Formel ausgedrückten Einschränkungen einhält. Eine (formale) Verifikation beschreibt den Vorgang der Überprüfung eines Systems S auf Gültigkeit einer solchen Formel ϕ (geschrieben: $S \models \phi$) [BK08, CGP00]. Ein vollautomatisches Verifikationsverfahren ist das Model Checking.

Eine verbreitete Temporallogik, ist die *Computation Tree Logic* (CTL) [CGP00]. Ihre Formeln beziehen sich auf den *Berechnungsbaum* des Systems. Dieser Baum enthält sämtliche möglichen Pfade durch das System, ausgehend von dessen Startzustand. Er ist daher für Systeme, die Zyklen enthalten, unendlich groß. CTL-Formeln können aussagenlogische Formeln sein, die für einen Zustand des Systems bzw. einen entsprechenden Knoten im Pfadbaum erfüllt sind, wenn die Formel für die atomaren Aussagen (auch: atomare Propositionen), die für diesen Zustand

gelten, erfüllt sind. Für jeden Zustand wird dabei eine Menge von gültigen atomaren Aussagen als gegeben vorausgesetzt.

Die temporallogischen Aussagen können in CTL durch einen *Pfadquantor* (E für \exists oder A für \forall), gefolgt von einem *temporalen Operator* formuliert werden. Ein temporaler Operator bezieht sich auf eine oder zwei temporallogische Formeln. Der Pfadquantor gibt an, ob die durch den Rest der Formel definierte Bedingung für mindestens einen (E) oder für alle (A) Pfade gelten muss, die vom aktuellen Zustand ausgehen. Der temporale Operator kann einer der folgenden sein:

- $X\phi$, „Next“: ϕ muss für den Nachfolge-Zustand des aktuellen Zustands auf dem Pfad gelten.
- $G\phi$, „Globally“: ϕ muss ab dem aktuellen Zustand auf dem gesamten Pfad gelten.
- $F\phi$, „Finally“: ϕ muss ab dem aktuellen Zustand irgendwann auf dem Pfad gelten.
- $\phi U \psi$, „Until“: ϕ muss ab dem aktuellen Zustand irgendwann auf dem Pfad gelten; bis ψ (für mindestens einen Zustand) gilt, muss immer ϕ gelten.
- in manchen Dialekten: $\phi W \psi$, „Weak Until“: ab dem aktuellen Zustand muss auf dem gesamten Pfad ϕ gelten, bis (für mindestens einen Zustand) ψ gilt. Es ist jedoch nicht erforderlich, dass ψ jemals gilt.

Eine Beispiel CTL-Formel für folgende Aussage: „Auf allen Pfaden muss für immer gelten, dass es mindestens einen Pfad gibt, auf dem irgendwann $p \wedge q$ gilt“, ist $AG(EFp \wedge q)$.

Um zusätzlich Zeitbedingungen an den temporalen Operatoren zu erlauben, wurde die *Timed Computation Tree Logic* (TCTL) [ACD93] eingeführt. Hiermit können für einen temporalen Operator O Bedingungen in der Form $O_{\sim c}$ mit $c \in \mathbb{N}_0$, $\sim \in \{<, \leq, =, \geq, >\}$, $O \in \{G, F, U\}$ angegeben werden. Eingeschränkte Varianten der CTL bzw. TCTL sind die *ACTL* bzw. *ATCTL*, die die Untermengen der (T)CTL-Formeln erlauben, die mit A beginnen und deren temporaler Operator sich nur auf einfache aussagenlogische Formeln bezieht.

2.4.6.2 Kompositionelle Verifikation

Standard Model-Checking-Verfahren sind in ihrer Laufzeit exponentiell abhängig von der Größe des Zustandsraums der untersuchten Systeme. Wird ein System in kleinere Subsysteme zerlegt und die Verifikation jeweils einzeln auf diese angewendet, dann ist die Gesamtlaufzeit der Verifikation insgesamt deutlich geringer, da die exponentielle Laufzeit für kleinere Systeme gilt. Kann die Größe der Subsysteme sogar als konstant angenommen werden, dann ist die Verifikation insgesamt nur linear abhängig von der Anzahl der Systeme. Die Voraussetzung für die Korrektheit einer solchen kompositionellen Verifikation (siehe z.B. [GTB⁺03, JLS00]) ist, dass aus der Gültigkeit der überprüften Eigenschaft ϕ für alle Teilsysteme S_1, S_2, \dots, S_n des Systems S auch die Gültigkeit für das Gesamtsystem $S = S_1 \parallel S_2 \parallel \dots \parallel S_n$ folgt (also: $(S_1 \models \phi \wedge S_2 \models \phi \wedge \dots \wedge S_n \models \phi) \implies (S \models \phi)$).

Der grundlegende kompositionelle Verifikationsansatz der MECHATRONIC UML wurde in [GTB⁺03] vorgestellt. Eine Erweiterung zur Betrachtung von auseinander driftenden Uhren wurde in [GHH06a] betrachtet. Eine Verifikation, die auch eine kompositionelle Anpassung der Kommunikationsstruktur berücksichtigt, wird durch den Ansatz vorgestellt in [HHG08, Suc08, HHPS10] ermöglicht (siehe auch Abschnitte 2.4.1 und 2.4.3).

Abstraktion ist eine weitere wirksame Technik zur Verbesserung der Skalierbarkeit. Anstatt ein Model Checking auf einem konkreten System K durchzuführen wird ein abstraktes und meist kleineres System A an dessen Stelle analysiert. Ein weiterer Vorteil ergibt sich, wenn es mehr als eine konkrete Implementierung von A gibt, da auch die übrigen dann nicht mehr gesondert verifiziert werden müssen. Daraus, dass A die Überprüfung besteht, wird geschlossen, dass auch das konkrete System bezüglich der überprüften Formel korrekt ist. Damit dieser Schluss $A \models \phi \implies K \models \phi$ erlaubt ist, muss eine geeignete Verfeinerungsbeziehung von K zu A bestehen, die eine Übertragbarkeit von Verifikationsergebnissen für Formeln der Art von ϕ (beispielsweise TCTL oder lediglich ACTL, je nach Verfeinerung) garantiert. Im folgenden Abschnitt betrachten wir, wie Eigenschaften einer Verifikation auf einem abstrakten Modell auch in einer Konkretisierung dieses Modells erhalten bleiben, ohne erneut die Konkretisierung bzgl. der gestellten Eigenschaften zu überprüfen.

2.4.7 Verfeinerungen

Im Rahmen der MECHATRONIC UML können Rolleninvarianten und Mustereigenschaften für Koordinationsmuster spezifiziert werden, die sich jeweils nur auf eine beschränkte Anzahl von Teilsystemen beziehen und damit eine kompositionelle Verifikation ermöglichen [GTB⁺03] (siehe Abschnitt 2.4.6.2). Das Model Checking wird dabei auf dem abstrakten Kommunikationsverhalten der Rollen durchgeführt. Eine Übertragbarkeit der Ergebnisse erfordert somit eine Verfeinerungsbeziehung von den Ports der konkreten Komponenten zu den Rollen.

Eine weitere Form der Abstraktion stellt die Delegation von Verhalten an Subkomponenten dar, deren Portverhalten daher in einer Verfeinerungsbeziehung zum Verhalten des implementierten Ports stehen muss. Es kann dabei sein, dass eine Verfeinerung ausgeschlossen ist, weil die strukturellen Voraussetzungen nicht erfüllt sind: Beispielsweise könnte der Fall auftreten, dass einer der Ports der analysierten Komponente in der aktuellen Konfiguration nicht implementiert ist, da eine Subkomponente durch eine Strukturanpassung nicht mehr erreichbar ist. In Kapitel 3 stellen wir einen Ansatz vor, der auch diese Abhängigkeiten berücksichtigt.

Im Folgenden werden einige für die MECHATRONIC UML relevante Definitionen für Verfeinerungen aufgeführt. Dies sind die Timed Simulation, die Timed Ready Simulation und die Timed Bisimulation.

Die Definitionen in diesem Abschnitt beziehen sich auf Timed Transition Systems (TTS) nach Definition 4 (Abschnitt 2.4.2.1). Ein TTS definiert die Semantik eines Timed Automaton nach Definition 1 (Abschnitt 2.4.2.1). Für ein Timed Automata A und K mit TTS T_A und T_K und

einer Verfeinerung \sqsubseteq gilt $K \sqsubseteq A$ genau dann, wenn $T_K \sqsubseteq T_A$ gilt. Eine Verfeinerung zweier Timed Automat kann also über die dazugehörigen TTS gezeigt werden.

Für die Verfeinerungen ist im Folgenden das externe sichtbare Protokollverhalten relevant. Dies ist bestimmt durch (externe) Nachrichten und den Zeitabständen dazwischen. Die im Folgenden betrachtete Timed Simulation und Timed Bisimulation werden nach [WL97] als „schwache“ Simulationen bezeichnet. Diese unterscheiden sich von den „starken“ darin, dass sie nicht das Vorhandensein einzelner Transitionen (Delay-Transitionen oder Transitionen mit internen Ereignissen) fordern, sondern sich auf Transitionsfolgen beziehen, die im Bezug auf das beobachtbare Verhalten äquivalent sind. Diese werden nach [WL97] als *Weak Transition Relation* bezeichnet und sind für TTS nach Definition 4 wie folgt definiert:

Definition 11 (Weak Transition Relation)

Sei ein Timed Transition System gegeben mit $s, s', s'', s''' \in S$, $\mu \in \Sigma_E$, $\delta, \delta_0, \delta_1, \delta_2 \in \Sigma_D$, $\delta_0 = 0$, sowie dem internen Ereignis $\tau \in \Sigma_E$. Bezeichne weiterhin $\xrightarrow{\tau^*}$ einen beliebig langen Pfad über ausschließlich τ -Transitionen ($\xrightarrow{\tau}$). Eine Weak Transition Relation ist dann die kleinste Relation \Longrightarrow , für die gilt:

1. $s \xrightarrow{\mu} s'$ gdw. $s \xrightarrow{\tau^*} s'' \xrightarrow{\mu} s''' \xrightarrow{\tau^*} s'$ für bel. s'', s''' , und
2. $s \xrightarrow{\delta_0} s'$ gdw. $s \xrightarrow{\tau^*} s'$ und
3. $s \xrightarrow{\delta} s'$ gdw. $s \xrightarrow{\delta_1} s'' \wedge s'' \xrightarrow{\delta_2} s' \wedge \delta = \delta_1 + \delta_2$ für bel. s''

Es dürfen nach \Longrightarrow vor und nach einem extern sichtbaren Ereignis (Nachricht) μ beliebig viele Transitionen mit dem internen Ereignis τ geschaltet werden (Bedingung 1). Bedingung 2 stellt sicher, dass beliebig viele τ -Transitionen einer Delay-Transition mit Dauer 0 entsprechen. Durch Bedingung 3 wird das Aufteilen einer Verzögerung in eine beliebig lange Folge von Delay-Transitionen erlaubt, wobei sich diese zu einer Gesamt-Verzögerung aufaddieren.

2.4.7.1 Timed Simulation

Im Folgenden betrachten wir die Timed Simulation unter Anwendung der Weak Transition Relation \Longrightarrow nach Definition 11 (vgl. [WL97] und [JLS00]). Diese Form der Timed Simulation ist die „schwächste“ der hier behandelten Verfeinerungsbeziehungen. Hiermit wird für ein Paar eines abstrakten und eines konkreten Systems gefordert, dass das konkretere System kein sichtbares Verhalten definiert, das nicht bereits im abstrakten System definiert wurde.

Definition 12 (Timed Simulation)

Seien T_A und T_K Timed Transition Systems mit Zustandsmengen S_A bzw. S_K und Startzuständen s_{0A} bzw. s_{0K} . Sei weiterhin Ω eine Relation $\Omega \subseteq S_K \times S_A$. Dann ist Ω eine Timed Simulation $T_K \leq_{TS} T_A$, wenn gilt:

1. $(s_{0K}, s_{0A}) \in \Omega$ und
2. $\forall (s_K, s_A) \in \Omega : s_K \xrightarrow{\mu} s'_K \xRightarrow{Impl.} \exists s'_A : s_A \xrightarrow{\mu} s'_A \wedge (s'_K, s'_A) \in \Omega$ und

$$3. \forall (s_K, s_A) \in \Omega : s_K \xrightarrow{\delta_1} s'_K \xRightarrow{\text{Impl.}} \exists s'_A : s_A \xrightarrow{\delta_2} s'_A \wedge (s'_K, s'_A) \in \Omega \\ \wedge \delta_1 = \delta_2$$

Erfüllt Ω die Bedingungen 1 und 2 und ist 3. bis auf $\delta_1 = \delta_2$ erfüllt, dann ist das hinreichend dafür, dass Ω eine (nicht zeitbehaftete) Simulation $T_K \leq_S T_A$ ist.

Die Simulationsbeziehung wird durch die Ω Relation umgesetzt, die die korrespondierenden Zustände des abstrakten und konkreten Systems zuordnet. Die korrespondierenden Zustände werden dabei induktiv beginnend mit dem Startzustand in der Relation aufgenommen (siehe Bedingung 2 und 3). Es muss dabei für alle Zustände eines konkreten Systems sichergestellt werden, dass ausgehende Weak Transitions eine Entsprechung im abstrakten System haben. Bedingung zwei bezieht sich dabei auf Transitions und Bedingung drei auf die Einhaltung der Zeitintervalle. Die Verwendung einer Weak Transition Relation $\xRightarrow{\quad}$ nach Definition 11 führt dazu, dass das konkrete System zusätzlich zu den Transitions mit sichtbarem Verhalten beliebig viele interne Transitions $\xrightarrow{\tau}$ und Delay-Transitions $\xrightarrow{\delta}$ schalten kann, sofern sich diese zur Gesamtverzögerung aufaddieren.

Erhalt temporallogischer Eigenschaften Nach [CGP00] werden durch nicht zeitbehaftete Simulationsbeziehungen der Erhalt von ACTL-Formeln zugesichert (siehe Abschnitt 2.4.6.1). Da durch Zeitbedingungen zusätzlich Time Stopping Deadlocks auftreten können, muss zusätzlich für eine Timed Simulation nachgewiesen werden, dass genau diese nicht auftreten.

Bedingung 3 sichert zudem zu, dass ATCTL durch die Verfeinerung erhalten bleiben. ATCTL Formeln beziehen sich auf Zeitintervalle, die eben durch Bedingung 3 zugesichert werden (hierdurch bleiben die zeitlichen Abstände dieselben).

Da eine Simulationsbeziehung nicht fordert, dass Verhalten des abstrakten Systems erhalten bleiben muss, werden in der Praxis häufig Bismulationen für eine Definition der Verfeinerungsbeziehung verwendet. Im Folgenden werden wir daher auf diese näher eingehen.

2.4.7.2 Timed Bisimulation

Zusätzlich zu einer Timed Simulation fordert eine Timed Bisimulation, dass sämtliches im abstrakten System mögliche Verhalten vom konkreten System ebenfalls unterstützt wird. Das sichtbare Protokollverhalten (beobachtbare Verhalten) muss im konkreten System identisch zum abstrakten System sein. Hiermit wird eine Timed Simulation (siehe Definition 12) in beide Richtungen gefordert. Eine Timed Bisimulation ist damit wie folgt definiert:

Definition 13 (Timed Bisimulation)

Seien T_A und T_K Timed Transition Systems. Dann ist Ω eine Timed Bisimulation $T_K \approx_{TBS} T_A$, wenn gilt:

1. Ω ist eine Timed Simulation $T_K \leq_{TS} T_A$ und
2. Ω ist eine Timed Simulation $T_A \leq_{TS} T_K$

Ist Ω eine (nicht zeitbehaftete) Simulation \leq_S in beide Richtungen, dann ist Ω zumindest eine (nicht zeitbehaftete) Bisimulation $T_K \approx_{BS} T_A$.

Erhalt temporallogischer Eigenschaften Nach [CGP00] erhält eine Bisimulation CTL-Formeln. Da die Timed Bisimulation die Bisimulation umfasst, bleiben ebenfalls durch eine Timed Bisimulation CTL-Formeln erhalten. Wie im Fall von Timed Simulationen muss allerdings zusätzlich sichergestellt werden, dass Time-Stopping Deadlocks ausgeschlossen werden. TCTL-Formeln bleiben ebenfalls erhalten, da die Zeitintervalle, auf die sich die TCTL-Formeln beziehen, durch die Timed Bisimulation erhalten bleiben. Da eine Timed Bisimulation sehr restriktiv bzgl. der möglichen Verfeinerungen ist, betrachten wir im Folgenden eine Verfeinerung, die auf Kompromisse eingeht, um eine größere Anzahl an Verfeinerungen zu erlauben.

2.4.7.3 Timed Ready Simulation

Eingeführt wurde die Timed Ready Simulation in [JLS00]. Motivation für diese Verfeinerung ist, dass auch bei Gültigkeit einer Timed Simulation $T_K \leq T_A$, in der urgent-Transitionen oder globale Variablen betrachtet werden, nicht garantiert ist, dass bei Einbettung der Systeme in einen Kontext T_C auch $T_K \parallel T_C \leq T_A \parallel T_C$ gilt. Problematisch hieran ist, dass aus $T_A \parallel T_C \models \varphi$ nicht auch $T_K \parallel T_C \models \varphi$ gefolgert werden kann. Die Verfeinerungsbeziehung verliert damit ihr Gültigkeit bei paralleler Komposition der Systeme. Um aus $T_{K1} \leq T_{A1} \wedge T_{K2} \leq T_{A2}$ auf $T_{K1} \parallel T_{K2} \leq T_{A1} \parallel T_{A2}$ schließen zu können, ist dies aber erforderlich.

Begründet ist dies damit, dass urgent-Transitionen und globale Variablen in einem Automaten zur Nicht-Erreichbarkeit von Verhalten in einem anderen Automaten führen können, welcher dieselben Urgent-Synchronisationen oder Variablenzugriffe anbietet. Um dieses Problem auszuschließen werden durch eine Timed Ready Simulation zusätzliche Bedingungen für globale Variablen und urgent-Transitionen eingeführt. Eine Timed Ready Simulation sei damit wie folgt definiert:

Definition 14 (Timed Ready Simulation)

Seien T_A und T_K Timed Transition Systems mit Zustandsmengen S_A bzw. S_K und Startzuständen s_{0A} bzw. s_{0K} . Sei weiterhin Ω eine Relation $\Omega \subseteq S_K \times S_A$. Dann ist Ω eine Timed Ready Simulation $T_K \leq_{TRS} T_A$, wenn gilt:

1. Ω ist eine Timed Simulation $T_K \leq_{TS} T_A$ und
2. $\forall (s_K, s_A) \in \Omega : s_A \xrightarrow{\mu} s'_A \wedge \xrightarrow{\mu} \in (T_u)_A$
 $\xRightarrow{Impl.} s_K \xrightarrow{\mu} s'_K \wedge \xrightarrow{\mu} \in (T_u)_K$

Dabei bezeichne $(T_u)_A, (T_u)_K$ jeweils die Menge der urgent-Transitionen in T_A bzw. T_K .

Aus Vereinfachungsgründen haben wir in der Definition auf die Berücksichtigung von globalen Variablen verzichtet, da dies für die MECHATRONIC UML nicht relevant ist. Begründet

ist dies damit, dass die REAL-TIME STATECHARTS nicht über globale Variablen Informationen austauschen, sondern über Nachrichten. Damit ergibt zusätzlich die Forderung, dass urgent-Transitionen des abstrakten Systems im konkreten System erhalten bleiben müssen.

Erhalt temporallogischer Eigenschaften Da eine Timed Ready Simulation eine Timed Simulation zusätzlich einschränkt, bleiben für eine Timed Ready Simulation die gleichen Formeln erhalten wie dies für eine Timed Simulation gilt. Zusätzlich bleiben (T)CTL-Formeln für die Teile erhalten, die vollständig durch urgent-Transitionen definiert sind. Sobald ein Pfad eine nicht urgent Transition beinhaltet, gilt dies allerdings nicht mehr.

2.5 Hybrides Verhalten

Wie in Kapitel 1 beschrieben, sind die hier betrachteten mechatronischen Systeme hybride Systeme. Hybride Systeme sind dadurch gekennzeichnet, dass sie sowohl aus einem diskreten wie auch einem kontinuierlichen Anteil bestehen. Ein *hybrider Automat* [Hen96] stellt eine Erweiterung zum Timed Automaton dar (siehe Abschnitt 2.4.2.1), da er zusätzlich zu einem diskreten zustandsbasierten Verhalten auch ein kontinuierliches Verhalten beschreibt. Neben der Einbettung von kontinuierlichen Verhalten ermöglicht der hybride Automat wie auch der Timed Automaton die Spezifikation von Zeitangaben.

In einem mechatronischen System wird das kontinuierliche Verhalten typischerweise durch Ansätze der Regelungstechnik beschrieben. Das Verhalten wird durch Differentialgleichungen beschrieben, die dafür sorgen, dass sich das System wie gewünscht verhält [Föl05].

Allgemein wird dabei zwischen einer Steuerung und Regelung unterschieden. Das Problem einer Steuerung kann wie folgt beschrieben werden: Gegeben sei das Ziel eines Systems, die Stellgröße (control) y und die Zustandsgröße/Ausgangsgröße (controlled) x . Die Aufgabe der Steuerung ist die Beeinflussung von x durch y in der Art und Weise, dass ein gewünschtes Verhalten trotz Einwirkung von Störgrößen z (disturbance), die nicht immer bekannt sind, erreicht wird.

Steuerungen reagieren schneller auf a priori bekannte Störungen, allerdings nicht auf unbekannte Störungen. Regler reagieren durch einen Regelkreis (Rückkopplung des Ausgangs auf den Eingang) auf jede Art von Störungen, allerdings nur, wenn die Zustandsgrößen und die Abweichungen messbar sind. Das Ziel einer Regelung ist es, die Differenz zwischen einem Vorgabewert und der Realität gegen 0 zu regeln. Eine gängige Technik für die Modellierung von Reglerstrukturen sind hierarchische Block Diagramme [Föl05].

Im Folgenden beschreiben wir den Modellierungsansatz der MECHATRONIC UML zur Spezifikation von hybriden Systemen. Hierbei wird besonderer Fokus auf die Beschreibung von Reglerkonfigurationen (ein Zustand indem eine Menge von Regler(-Instanzen) aktiv sind) und deren Rekonfiguration gelegt. Anschließend werden wir die Verifikation und die Verfeinerung von hybriden Verhalten in Abschnitt 2.5.2 nach dem MECHATRONIC UML-Ansatz diskutieren.

2.5.1 Hybrid Reconfiguration Charts

Aufgrund der steigenden Komplexität regelungstechnischer Komponenten werden diese vermehrt modular entworfen. Dies hat auch den Vorteil, dass Ressourcen eingespart werden können, da nicht ein Regler, der alle Funktionen umsetzt aktiv sein muss, sondern nur die regelungstechnische Komponente, die aktuell tatsächlich benötigt wird. Die in Abbildung 2.2 dargestellte RailCab-Komponente bettet z.B. zwei kontinuierliche Unterkomponenten ein. Die Aktivierung und Deaktivierung, wird dabei von Software übernommen.

Das Modell der REAL-TIME STATECHART (siehe Abschnitt 2.4.2) wurde dementsprechend erweitert zu so genannten HYBRID RECONFIGURATION CHARTS, die die Aktivierung und Deaktivierung von (kontinuierlichen) Komponenteninstanzen beschreiben können. Die Komponenteninstanzen werden Zuständen zugeordnet, die wir Zustandskonfigurationen nennen.

Im Vergleich zum klassischen hybriden Automaten [Hen96] ermöglichen HYBRID RECONFIGURATION CHARTS eine modulare Rekonfiguration (siehe Abschnitt 1) zur Laufzeit [BGH05a]. Wie auch bei hybriden Automaten bettet ein HYBRID RECONFIGURATION CHART Komponenteninstanzen in Zustände ein und tauscht diese durch einen Zustandswechsel aus. Jedoch bieten HYBRID RECONFIGURATION CHARTS zusätzlich die Möglichkeiten, die Struktur und den internen Zustand der Komponenten durch einen Zustandswechsel zu modifizieren. Hierdurch wird eine Rekonfiguration des Systems ermöglicht.

Um die beschriebenen Vorteile umzusetzen, verwenden HYBRID RECONFIGURATION CHARTS im Gegensatz zum hybriden Automaten ein verändertes kontinuierliches Modell. In diesem Modell werden die Zustands-, Eingabe- und Ausgabevariablen in Abhängigkeit des jeweiligen Zustands angegeben. Zudem wird die Umschaltung zwischen den Reglern oder kontinuierlichen Verhalten explizit betrachtet und analysiert, so dass hier keine Störungen auftreten, die zu einer Beeinträchtigung der Sicherheit bzw. Stabilität führen können [OMT⁺08].

Für die Spezifikation der RailCab-Komponente aus Abbildung 2.2 muss definiert werden, in welchen Zuständen des RailCab-REAL-TIME STATECHARTS, welche kontinuierliche Komponente aktiv ist. Das HYBRID RECONFIGURATION CHART für die rear-Rolle zeigt Abbildung 2.16. Dies ist eine Erweiterung des korrespondierenden Rollenverhaltens der rear-Rolle (siehe Abbildung 2.6) um die Reglereinbettungen in Form von kontinuierlichen Komponenten. In Zustand noConvoy ist nur der VelocityController aktiv und in Zustand convoy zusätzlich der DistanceController. Der DistanceController wird benötigt, um im Konvoibetrieb zusätzlich den Abstand zum vorherfahrenden RailCab für die Berechnung der Beschleunigung zu berücksichtigen.

Für die Verfeinerung des Systems müssen die Reglerkonfigurationen betrachtet werden, wenn eine Komponente mehrere Rollen anwendet. Die Komponente befindet sich z.B. in einen nicht sicheren Zustand, wenn der DistanceController gleichzeitig von zwei Statecharts aktiviert (genutzt) wird. Dies liegt daran, dass der DistanceController eine einzelne Ressource ist und eine mehrfach Aktivierung in unterschiedlichen Statecharts zu widersprüchlichen Eingaben führen kann. Dies muss entsprechend bei der Entwicklung des Komponentenverhaltens aufgelöst werden. Der in Kapitel 5 vorgestellte Syntheseansatz löst diese Konflikte automatisch auf.

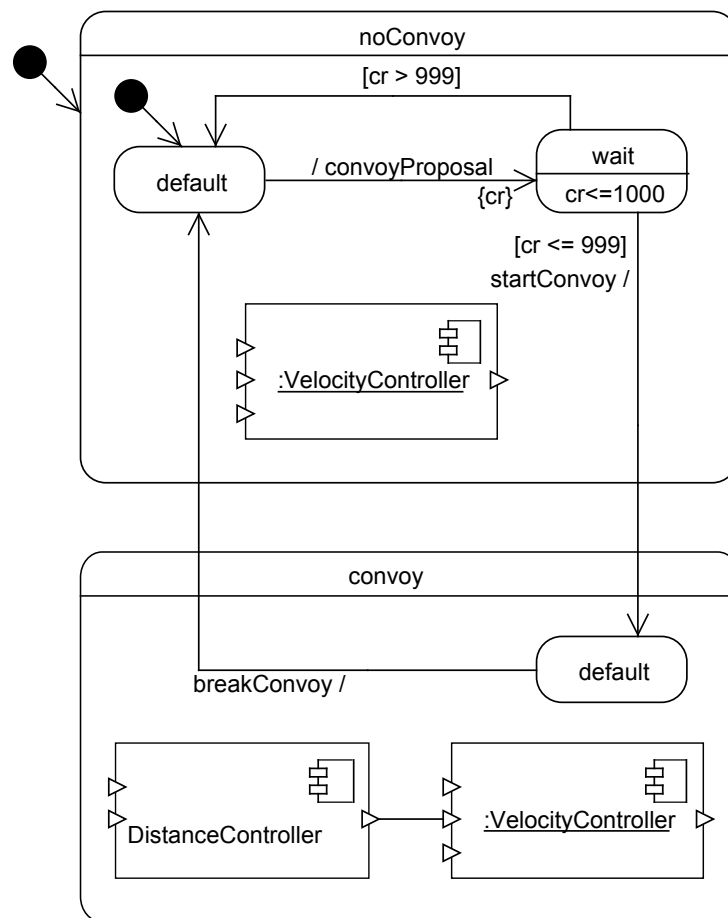


Abbildung 2.16: HYBRID RECONFIGURATION CHART für die rear Rolle

2.5.2 Verifikation und Verfeinerung

Für allgemeine hybride Systeme ist die Erreichbarkeit nicht entscheidbar, sondern nur die eingeschränkte Klasse der Rectangula Automata⁴ [HKPV95]. Selbst für diese eingeschränkte Klasse ist die Verifikation durch Model Checking nur für kleine Beispiele anwendbar [Hir08, HHP08, Dor08].

Daher beschränkt sich die Analyse der MECHATRONIC UML auf das reine Echtzeitverhalten sowie der Erreichbarkeit von konsistenten Konfigurationen mit wohl-definierten kontinuierlichen Komponenten [Hir08]. Erreicht wird dies durch Abstraktion von dem kontinuierlichen Verhalten eines HYBRID RECONFIGURATION CHART, indem nur die Clocks betrachtet werden [Bur06]. Auf diesem Modell kann dann wie unter Abschnitt 2.4 beschrieben eine Verifikation und Verfeinerung durchgeführt werden. Eine Wesentliche Herausforderung besteht daher darin eine für die Analysen gültige Abstraktion zu beschreiben. Im Folgenden werden wir dies nicht näher betrachten.

2.6 Timed Story Driven Modeling

Selbstoptimierende, mechatronische Systeme passen ihr Verhalten zur Laufzeit den Umweltbedingungen an. Um diese Anpassungen zu modellieren und zu analysieren, wird unter anderem eine enge Verzahnung zwischen der Verhaltensbeschreibung dieser Systeme sowie den (kompositionellen) Strukturanpassungen benötigt, wie zu Abbildung 1.2 und im Folgenden erläutert.

Abbildung 2.17 illustriert die Notwendigkeit einer Anpassung des Konvois. Damit sich das RailCab RC3 zwischen den RailCabs RC1 und RC2 im Konvoi einordnen kann, muss der Konvoi restrukturiert werden. Die Strukturanpassungen des Multi-Ports zur Koordination des RailCab Konvois wird durch ein Adaptionsverhalten angestoßen (siehe Abschnitt 2.4.3 und 2.6.1). Das Adaptionsverhalten ist wiederum eng verzahnt mit den Portinstanzen, die hierdurch indirekt eine Strukturanpassung steuern können. Eine Portinstanz darf z. B. nicht gelöscht werden, wenn diese noch mit einem RailCab kommuniziert.

Die Beeinflussung des Verhaltens durch eine Strukturanpassung ist inhärent, da eine Strukturanpassung das Verhalten verändert (siehe Abschnitt 2.2). Durch z. B. das Hinzufügen eines weiteren Ports, um ein neues Mitglied im Konvoi aufzunehmen, wird nicht nur einfach ein weiteres Verhalten parallel zu den anderen ausgeführt. Je nach Position des neuen RailCabs im Konvoi müssen die Abhängigkeiten zu den direkt benachbarten Ports angepasst werden. Es muss dabei sichergestellt sein, dass eine Restrukturierung nur dann stattfindet, wenn die beteiligten Ports in einem dafür geeigneten Zustand sind (Quiescent State [KM98, ZC06]). Es gilt wieder, dass keine Restrukturierung durchgeführt werden darf, wenn z. B. gerade Konvoiparameter (wie das Bremsverhalten) ausgetauscht werden.

⁴Rectangula Automata beschreiben analoge Trajektorien mit teilweise-linearer Entwicklung und Sprüngen

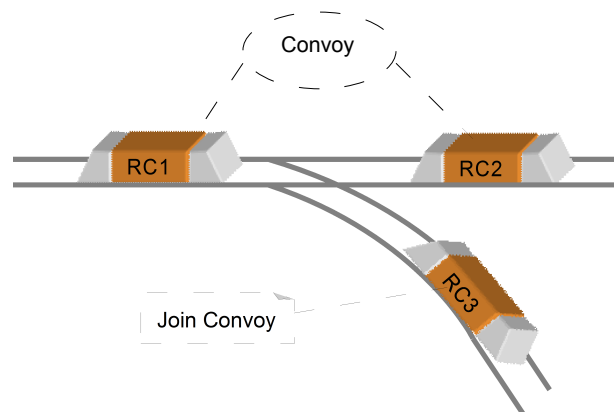


Abbildung 2.17: Beispiel Konvoirestrukturierung

Die in Abschnitt 2.4.3 vorgestellten PARAMETERIZED REAL-TIME COORDINATION PATTERNS wurden für diesen Anwendungsfall entwickelt. Hierüber ist es allerdings nicht möglich mit den unterliegenden Formalismen für das Verhalten (PARAMETERIZED REAL-TIME STATECHARTS, siehe Abschnitt 2.4.4) und den Strukturanpassungen (TGTS, siehe Abschnitt 2.4.5.3) über die gleichen Objekte, wie Nachrichten oder Ports, das Systemverhalten (Komponenten und Muster) zu spezifizieren. Wird also ein Seiteneffekt, mit dem eine Strukturanpassung über TGTS umgesetzt werden kann, implementiert, so kann nicht auf die Objekte des Statecharts zugegriffen werden. Dies wird aber genau in dem skizzierten Szenario benötigt, um sicherzustellen, dass die beiden Statecharts der Portinstanzen für RC1 und RC2 in einem aktuellen Zustand sind, der eine Strukturanpassung erlaubt.

Der unterlagerte Ansatz der Fujaba Tool Suite, der Story Driven Modeling Ansatz von Zündorf [Zün01], ermöglicht dies allerdings, in dem Statecharts und Graphtransformationssysteme objektorientiert definiert werden. Der Formalismus zur Verhaltensbeschreibung wird dabei Story Charts genannt und der zur Beschreibung von Strukturanpassungen Story Patterns, bzw. Story Diagramme (siehe Abschnitt 2.4.5).

Diese Voraussetzungen nutzen wir aus, um in diesem Abschnitt den Timed Story Driven Modeling Ansatz als eine Erweiterung des Story Driven Modeling Ansatzes um Zeit, der die unterliegenden Formalismen der PARAMETERIZED REAL-TIME COORDINATION PATTERNS integriert, vorzustellen. Unser Ansatz erweitert, bzw. passt daher die Formalismen des Story Driven Modeling Ansatzes (Story Diagramme, Story Pattern und Story Charts) um Zeit an zu TIMED STORY DIAGRAMS, TIMED STORY PATTERNS und TIMED STORY CHARTS.

Diese Formalismen werden eingeführt, indem wir zuerst die einzelnen Elemente des Formalismus durch Abbildung von dem Ursprungsformalismus (z. B. PARAMETERIZED REAL-TIME STATECHARTS) in den neuen Formalismus (z. B. TIMED STORY CHARTS) beschreiben. Da gerade die Ausführung eines TIMED STORY CHARTS durch eine Kombination einzelner Elemente geprägt ist, beschreiben wir zudem eine Abbildung der Ausführungssemantik von PARAMETERIZED REAL-TIME STATECHARTS auf TIMED STORY CHARTS, die entsprechend eine Kombi-

nation aller Elemente berücksichtigt. Ein formaler Beweis über die semantische Äquivalenz der Formalismen ist ein Ausblick für weiterführende Arbeiten.

Die Modellierung des Verhaltens sowie der Struktur kann durch diese Vorgehensweise weiterhin über die in Abbildung 2.1 dargestellten Formalismen durchgeführt werden, da die Strukturmodellierung unverändert bleibt und das Verhalten über die Semantik von `PARAMETERIZED REAL-TIME STATECHARTS` definiert ist. Dies ist wichtig, da z. B. eine Verhaltensspezifikation mit `TIMED STORY CHARTS` für die meisten Entwickler ungewohnt ist und zudem einen höheren Aufwand erfordert, da hier die einzelnen Statechart Objekte und Transitionen explizit spezifiziert werden müssen (siehe Abschnitt 2.6.4). Die Spezifikation der Strukturanpassungen mit `TIMED STORY PATTERN` und `TIMED STORY DIAGRAMS` wird nicht durch eine andere Syntax ersetzt, da diese bereits die Syntax von Story Diagrammen aufweisen, die in der `MECHATRONIC UML` zur Beschreibung von Strukturanpassungen verwendet werden.

Wir werden zuerst `TIMED STORY PATTERNS` (siehe Abschnitt 2.6.2) und `TIMED STORY DIAGRAMS` (siehe Abschnitt 2.6.3) vorstellen. Anschließend werden wir `TIMED STORY CHARTS` (siehe Abschnitt 2.6.4) auf Basis der `TIMED STORY PATTERN` und `TIMED STORY DIAGRAMS` beschreiben. Grundlegende Arbeiten hierzu wurden in [HHG08, Hei09, HHZ09, HHH10] vorgestellt. Bevor wir auf die Modellierungssprachen eingehen, werden wir in Abschnitt 2.6.1 ein domänenspezifisches Metamodell für `MECHATRONIC UML` Komponenten beschreiben, auf Basis dessen die Modellierungssprachen angewandt werden. Als Einführung in die Formalismen, werden wir im folgenden Paragraphen einen Ausschnitt des Eingangs beschriebenen Einfädelszenarios skizzieren.

Beispielanwendung Abbildung 2.18 und 2.19 zeigt am Beispiel der Coordinator Komponente die Restrukturierung eines Konvois, ausgelöst durch das Hinzufügen eines weiteren Konvoiteilnehmers. Abbildung 2.17 illustriert die Situation, die die Restrukturierung auslöst. RC3 möchte an dem Konvoi von RC1 und RC2 teilnehmen und sich zwischen diesen beiden RailCabs einordnen.

Das Komponentenmodell zeigt die Struktur der Coordinator Komponente (siehe Abbildung 2.18). Wie bereits in Abschnitt 1.2 erläutert, ist die Coordinator Komponente verantwortlich für das Berechnen der Konvoiparameter. Das entfaltete Komponentenmodell zeigt, welche Ports für die Koordination von zwei RailCabs miteinander verbunden sind. Mit gestrichelten Linien wird dabei angedeutet, wie sich das neue RailCab einordnen soll.

`updatePort` zeigt das `TIMED STORY PATTERN`, welches den Port für RailCab RC3 in den bisherigen Multi-Port der Coordinator Komponente integriert. Dies wird nur erlaubt, wenn das zu dem Port gehörige Statechart in dem Zustand `NoUpdate` ist⁵ und die Clock (siehe Abschnitt 2.4) `c` im Intervall zwischen fünf und zehn ist. Damit wird ausgedrückt, dass das Hinzufügen im Intervall zwischen fünf und zehn Zeiteinheiten stattfinden muss.

⁵Aus Vereinfachungsgründen wird in dem Timed Story Pattern nur auf die Verbindung zwischen den Hauptklassen eingegangen. Z. B. wurden die Port Objekte zwischen den `next` Assoziationen nicht extra aufgeführt.

updatePart() implementiert im Wesentlichen die gleiche Strukturanpassung wie updatePort. Der Unterschied ist lediglich, dass Part statt Port Objekte verwendet werden.

updateDel() zeigt die Strukturanpassung der Delegation. In diesem Fall ist das einfach, da lediglich eine Delegation erzeugt werden muss, die die beiden neuen Ports (p3 und pc3) miteinander verbindet.

Abbildung 2.19 zeigt, wie die einzelnen Story Pattern zur Strukturanpassung der Port-, Part- und Delegations-Elemente durch ein TIMED STORY DIAGRAM verknüpft werden. Neben dem einfachen Ausführen der TIMED STORY PATTERN durch Aufrufen der entsprechenden Methode, wird Initial eine ClockInstance angelegt, über die nach jedem Aufruf der TIMED STORY PATTERN, eine Invariante überprüft wird und die Clock zurückgesetzt wird.

In den folgenden Abschnitten 2.6.2 bis 2.6.4 werden wir die einzelnen Formalismen beschreiben, die wir bereits in dem Beispiel zum Teil verwendet haben. Vorher werden wir in Abschnitt 2.6.1 ein Metamodell für MECHATRONIC UML Komponenten vorstellen, welches die Formalismen nutzen, um das Konvoi Beispiel illustrativ umzusetzen.

2.6.1 Metamodell

Hierarchische Komponenten sind ein mächtiges Mittel, um das interne Verhalten einer Komponente von ihrem externen Verhalten zu trennen [SGW94]. Wie bereits in Abschnitt 2.3 vorgestellt, unterstützt die MECHATRONIC UML hybride hierarchische Komponenten. Um den Anforderungen von Strukturanpassungen gerecht zu werden, wird allerdings eine Erweiterung des zugrunde liegenden Metamodells benötigt. Die RailCab-Komponente (siehe Abschnitt 1.2) ist z. B. mit dem bisherigen Komponentenmodell nicht realisierbar, da das Verhalten welches die Strukturanpassung steuert, nicht berücksichtigt ist.

Wie in unseren eigenen Arbeiten [HHG08, HHH10] und auch in [ZC06] beschrieben hat sich für die Modellierung von Strukturanpassungen eine hierarchische Modellierung durchgesetzt, in der auf der obersten Ebene das Verhalten für die Anpassung beschrieben wird. Diese explizite Betrachtung ermöglicht es, Strukturanpassungen und das dafür benötigte Verhalten unabhängig von dem Zustandsverhalten der Komponente oder Ports zu betrachten. Wir bezeichnen dies mit Adaptionverhalten. Das Adaptionverhalten muss nun für hierarchische Komponenten explizit für Multi-Ports (siehe Abschnitt 2.4.3), Multi-Parts (siehe Abschnitt 2.3) und Delegationen, die Multi-Ports und Multi-Parts verbinden, spezifiziert werden können.

Wir haben daher in [BBB⁺09, HHH10] einen Ansatz vorgestellt, der für die Elemente Multi-Port, Multi-Part und Delegation eine extra Adaptionklasse berücksichtigt. Dieses ermöglicht es uns speziell für diese strukturellen Elemente Adaptionverhalten zu beschreiben.

Für eine spezifizierte Komponente unterstützen wir eine automatische Synthese des komponentenspezifischen Klassendiagramms, auf Basis dessen Strukturanpassungen beschrieben werden können. Das Klassendiagramm beinhaltet Klassen für jede Komponente, Ports, Parts und für alle Delegationen. Die Struktur des Klassendiagramms basiert auf dem Metamodell für Komponen-

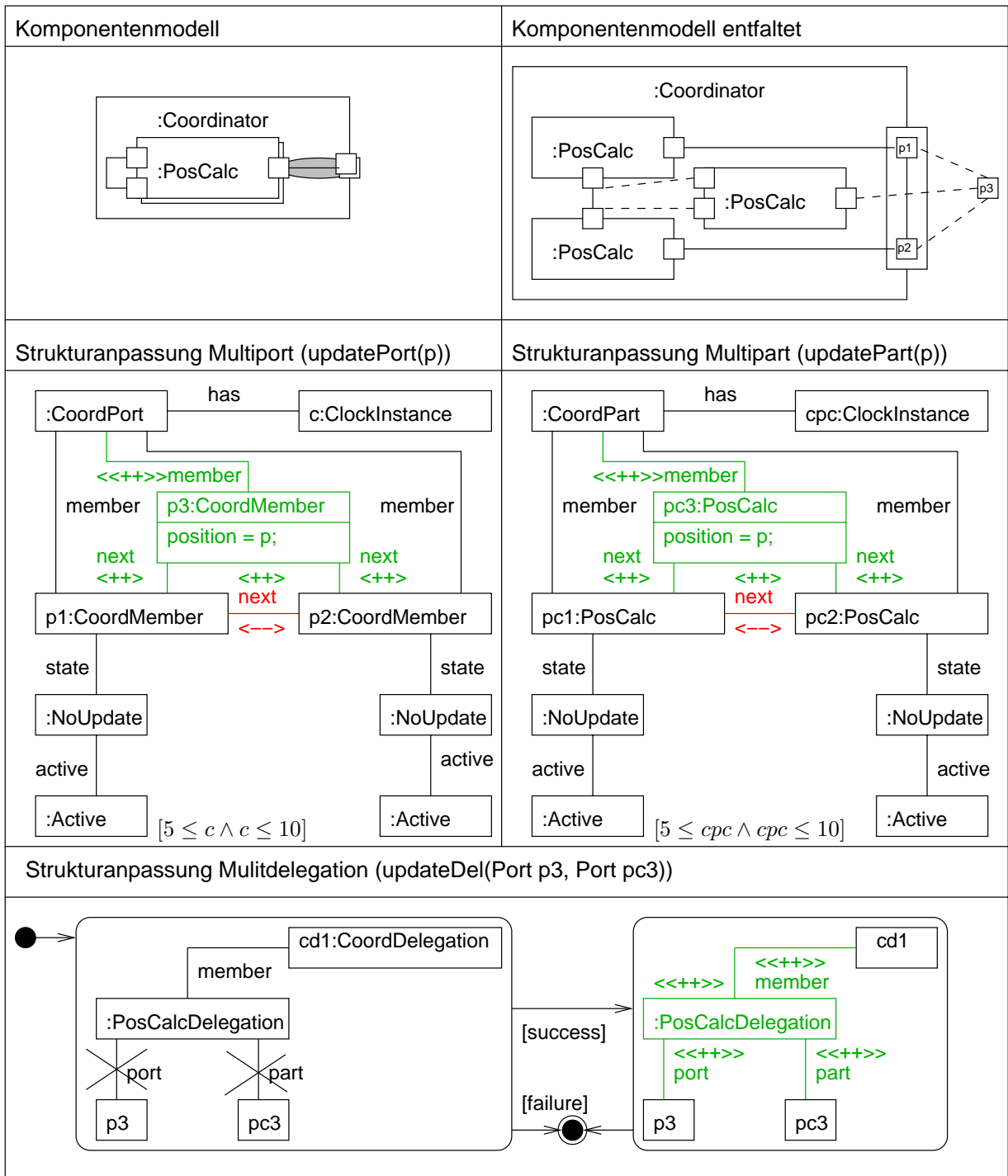


Abbildung 2.18: Konvoirestrukturierung: Überblick

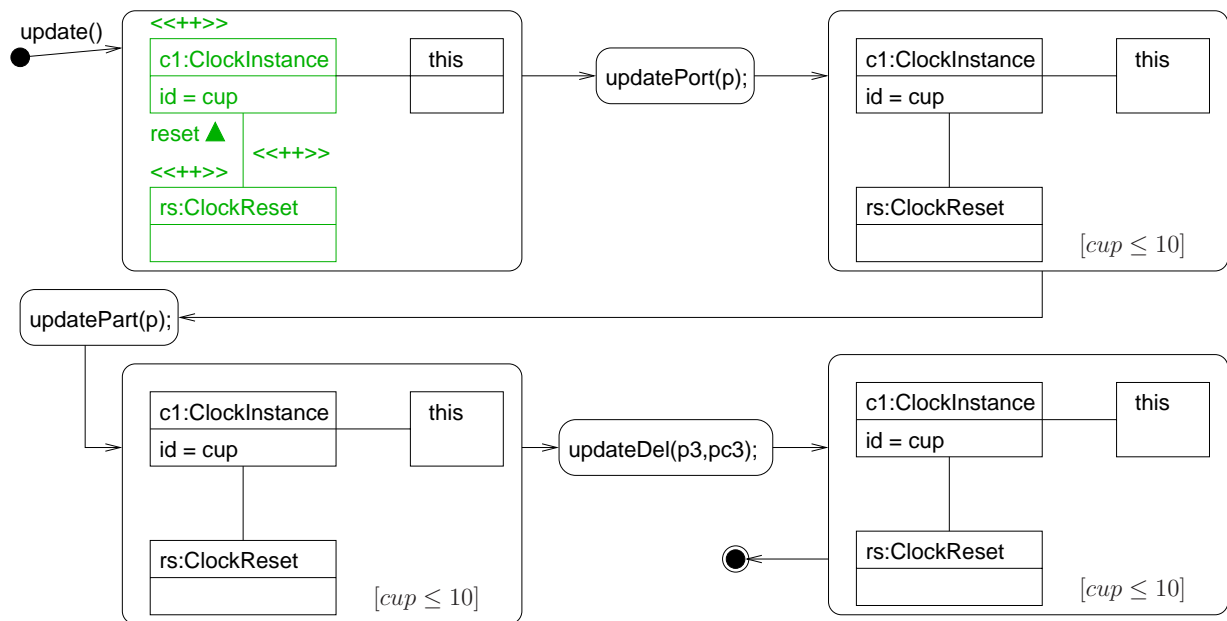


Abbildung 2.19: Konvoirestrukturierung: Story

ten (siehe Abbildung 2.20). Der dort gezeigte Ausschnitt orientiert sich an dem Metamodell von Komponentenstorydiagrammen (siehe Abschnitt 2.3 [Tic09]). Die Coordinator-Klasse realisiert dabei die Stellvertreterklasse, um Adaptionverhalten beschreiben zu können. Die Assoziation zur Port-, ComponentPart-, PortPart- und DelegationTyp-Klasse realisieren die geforderte Verknüpfung zu den Multielementen, um dessen Struktur anpassen zu können. Die Selbstassoziation erlaubt eine Verknüpfung der Adaptionen untereinander.

Abbildung 2.21 verdeutlicht die diskutierte Adaptionsschicht an der Coordinator-Komponente. Für jedes Multielement sowie Delegationen zwischen Multielementen wird eine Adaptionsschicht angelegt (Multi-Part-Adaption, Multi-Port-Adaption, Delegation-Adaptation), die jede Instanz des Multielements (z. B. `coordPortPart1`, ..., `coordPortPartk`) koordinieren kann (siehe auch 2.4.3).

Ein Beispiel-Klassendiagramm für die Coordinator-Komponente ist in Abbildung 2.22 gezeigt. `CoordPort`, `CoordPart` und `CoordDelegation` repräsentieren die Adaptionsschichten.

Eine durch das Adaptionverhalten gesteuerte Strukturanpassung kann mit Story Diagrammen, bzw. Story Pattern beschrieben werden (siehe Abschnitt 2.4). Um Zeit bei der Spezifikation der Strukturanpassung zu berücksichtigen führen wir in Abschnitt 2.6.2 TIMED STORY PATTERN und in Abschnitt 2.6.3 TIMED STORY DIAGRAM ein.

Strukturanpassungen werden als Seiteneffekt von REAL-TIME STATECHART, die das Adaptionverhalten implementieren, ausgeführt. Ein Seiteneffekt wird durch eine Methode in der unterliegenden Verhaltensklasse, hier also der Adaptionsschicht, definiert. Hierdurch haben die Story Diagramme eine Assoziation zu den Multielementen und können diese entsprechend strukturell

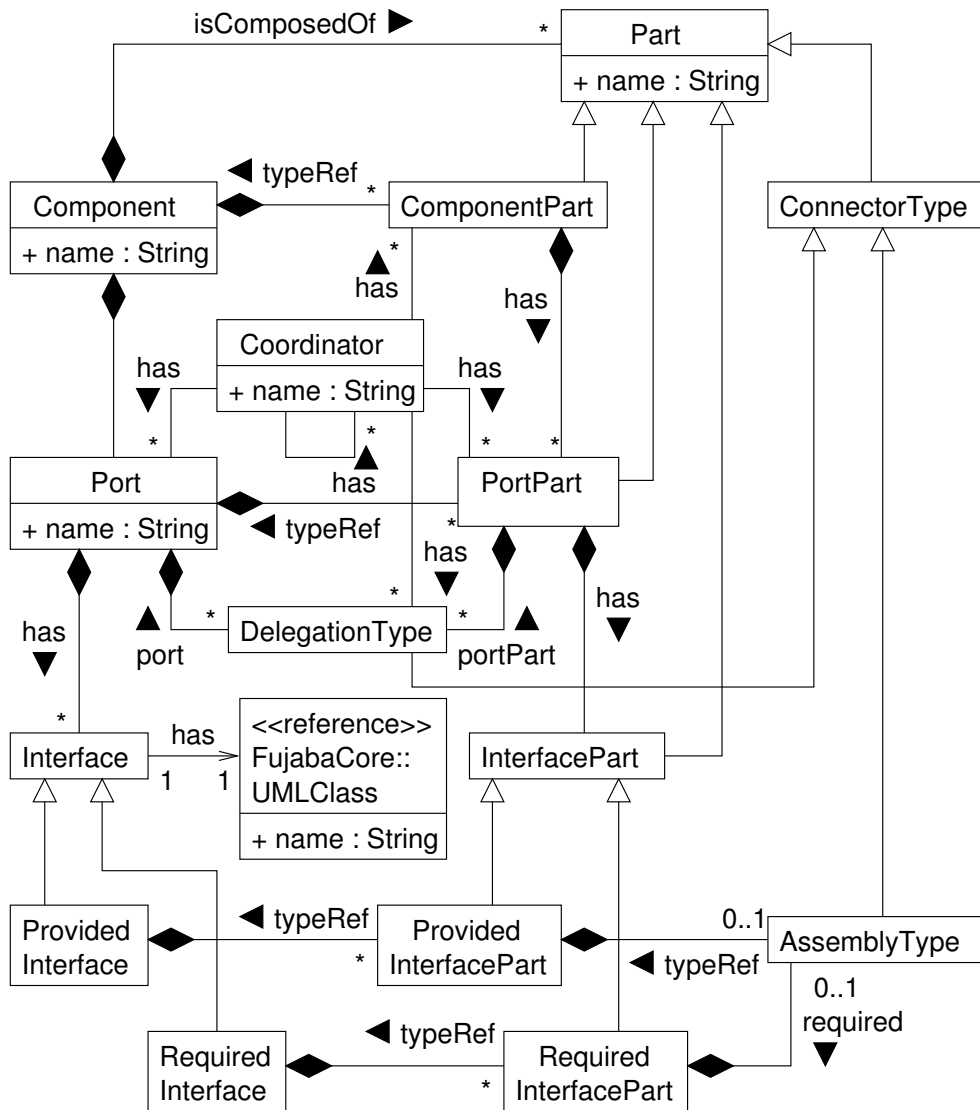


Abbildung 2.20: Komponenten und -parts Metamodell

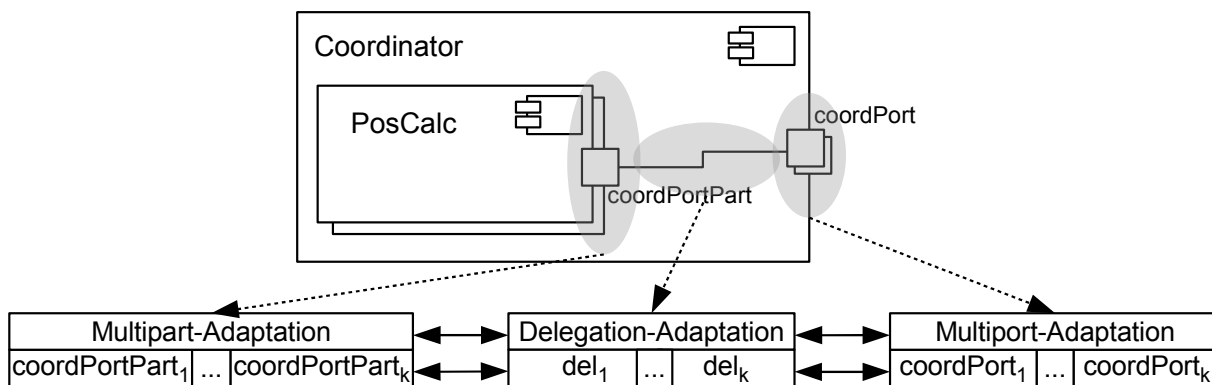


Abbildung 2.21: Multi-Part, -Port, und -Delegation

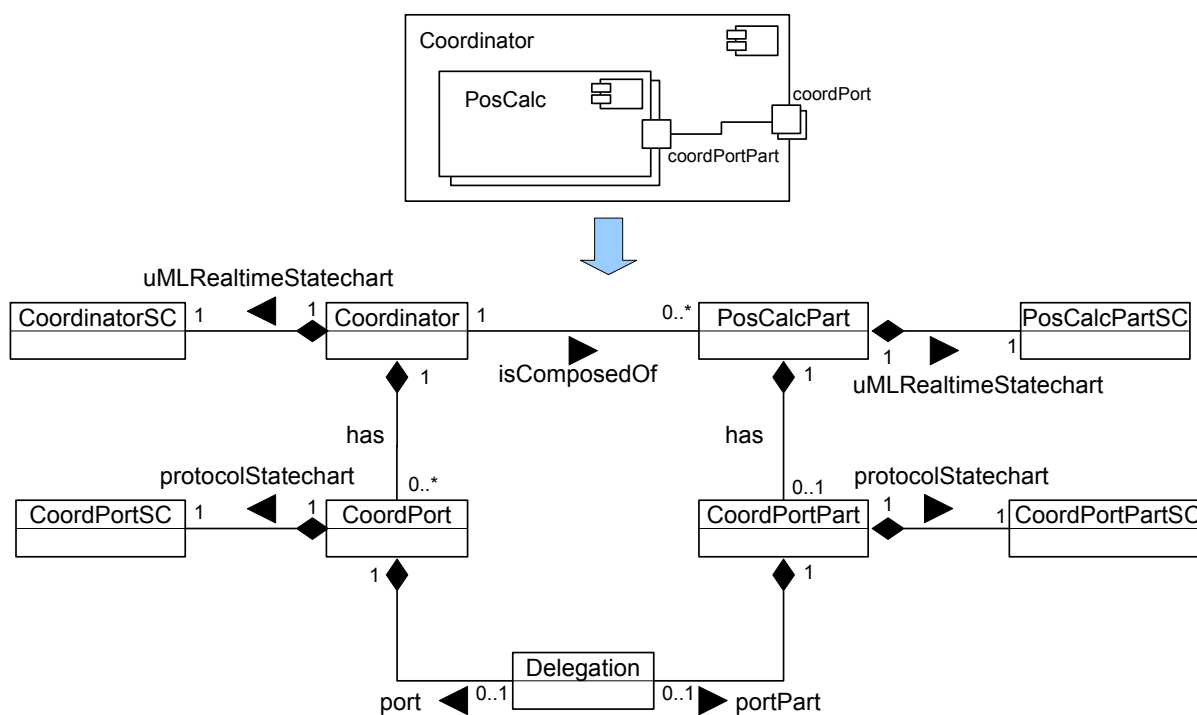


Abbildung 2.22: Beispiel-Klassendiagramm

verändern (siehe Abschnitt 2.21). Eine Strukturanpassung kann die innere Struktur der Komponente verändern, aber nicht direkt die Struktur außerhalb der Komponente. Es ist jedoch möglich, dass eine Strukturanpassung über Nachrichten oder eine Synchronisation andere Komponenten ebenfalls zur Strukturanpassung veranlasst, wie dies z.B. über die Delegations-Adaption durchgeführt wird. Es kann so ein kausaler Zusammenhang zwischen verschiedenen Strukturanpassungen bestehen, die aber einzeln nur innerhalb einer Komponente Veränderungen vornehmen können.

Ein Adaptionsverhalten kann über Strukturanpassungen Parts und Ports erzeugen und entfernen, sowie Assemblies zwischen den Partinstanzen anlegen und entfernen. Die Instanzen eines Ports oder Parts können durch die entsprechenden Adaptionsverhalten erzeugt und entfernt werden. Über eine Delegations-Adaption können Delegationen zwischen Multi-Ports und -Parts angelegt und entfernt werden.

Die Statecharts zu einem Port, bzw. einer Komponente, müssen beim Erstellen des Ports, bzw. der Komponente, mit erzeugt werden. Das Löschen geschieht automatisch über die verwendeten Kompositionsbeziehungen, so dass es ausreichend ist, in einem Story Diagramm nur den Port, bzw. die Komponente, zu löschen.

2.6.2 Timed Story Pattern

TIMED STORY PATTERN erweitern Story Pattern [Zün01] um zeitliche Bedingungen. Ein weit verbreiteter Formalismus um zeitliche Bedingungen zu spezifizieren ist der Timed Automata-Formalismus [AD90, AD94]. Daher wird der Zeitformalismus nach den Timed Automata als Grundlage dienen. Dies ermöglicht die Spezifikation von Zeitbedingungen für das Verändern von Strukturen.

In [Hir08] wurden zeitbehaftete Graphtransformationssysteme (Timed Graph Transformation Systems - TGTS) eingeführt, um die Instanzierungsdauer von Elementen und eine kontinuierliche Bewegung spezifizieren zu können (siehe auch Abschnitt 2.4.5.3). Um einen konsistenten, gemeinsamen Formalismus zu spezifizieren benötigen wir allerdings einen Ansatz der basierend auf einem gemeinsamen Metamodell Strukturveränderungen beschreiben kann.

Da TGTS grundsätzlich den gestellten Anforderungen gerecht werden, definieren wir TIMED STORY PATTERN als eine Erweiterung von Story Pattern über der Semantik von TGTS. Damit ermöglichen TIMED STORY PATTERN zum einen objektorientierte Strukturen zu verändern und des Weiteren zeitliche Bedingungen für Strukturanpassungen zu spezifizieren. Um die Syntax von Story Pattern nicht zu verändern, werden die benötigten Elemente in der Syntax von Story Pattern definiert.

Die benötigten Elemente, um zeitliche Bedingungen zu spezifizieren sind: Uhren (Clock-Instanz), Uhren Resets (Clock Resets) und Zeitbedingungen (Time Guards und Invarianten). Wie im Timed Automata-Formalismus können mehrere Uhren definiert werden, da sich dies für die Modellierung von Zeitbedingungen bewährt hat [Alu99].

Da Story Pattern über Objekte, bzw. Objektinstanzen, Strukturanpassungen spezifizieren, wird ein *Clock-Objekt* eingeführt. Wird eine Uhr spezifiziert, dann wird also eine Instanz des Clock-Objektes angelegt. Daher verwenden wir hier den Begriff der *Clock-Instanz*.

Um die hier betrachteten Systeme spezifizieren zu können, müssen Eigenschaften von reaktiven Systemen berücksichtigt werden. Elementar ist daher die Spezifikation von relativen Zeitbedingungen, da häufig Bedingungen relativ zu einem Ereignis definiert werden müssen.

Das Konzept der Clock Resets wird benötigt, um eine Uhr auf null zurückzusetzen. Clock Resets werden ebenfalls durch Objekte definiert.

Um zeitliche Bedingungen zu beschreiben werden wie bei dem Timed Automata-Formalismus die folgenden Bedingungen ϕ erlaubt: $\phi ::= x \sim n \mid x - y \sim n \mid \phi \wedge \phi \mid true \mid false$, mit $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$. Wie in [AD94, Alu99] beschrieben haben sich diese Bedingungen als nützlich herausgestellt und sind im Allgemeinen auch nicht erweiterbar, um z. B. Addition von Uhren, um die Analysefähigkeit nicht zu verlieren.

Um die Zeitbedingungen anwenden zu können, muss das zugrunde liegende Metamodell um spezielle Clock-Instanz- und Clock-Reset-Objekte erweitert werden. Das kann entweder ganz allgemein für alle Objekte definiert werden oder einschränkend für eine bestimmte Menge an Objekten.

Für unser Komponentenmetamodell (siehe Abbildung 2.20) ist z. B. eine Einschränkung nur auf die Objekte notwendig, deren Struktur angepasst werden kann. Daher reicht es aus, der Klasse Coordinator eine Assoziation dem Clock-Objekt hinzuzufügen. Dadurch kann ein Story Pattern, welches durch eine Methode einer konkreten Coordinator Klasse definiert wird, auf die Multi-Elemente Port, Part und Delegation zugreifen und entsprechend Zeitbedingungen für eine Strukturanpassung definieren (siehe Abbildung 2.23).

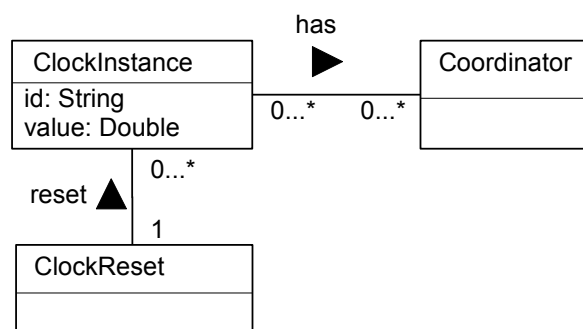


Abbildung 2.23: Erweiterung Komponentenmetamodell um Zeit

Im Folgenden beschreiben wir die Uhren-Elemente in der Syntax von Story Pattern. Weiterhin erläutern wir die Semantik über TGTS.

Clock-Instanz Eine Clock-Instanz wird definiert durch ein ClockInstance Objekt mit einem id und value Attribut (siehe Abbildung 2.23). Das value Attribut vom Typ Double beschreibt

den aktuellen Wert der Clock-Instanz. Die *id* gibt den Namen der Clock-Instanz an. Durch eine Assoziation zu einer Clock-Instanz zu den Objekten des Pattern oder allgemein des zugrunde liegenden Graphen wird die Zugehörigkeit einer Clock-Instanz zu einzelnen Objekten spezifiziert. Das Erstellen einer Clock-Instanz wird über die Standard-Modifizierer «++» von Story Pattern erreicht. Clock-Instanzen werden über spezielle Clock-Instanzregeln dem Objektgraphen zugewiesen. Da die Regeln aus Hirsch [Hir08] auf Story Pattern anwendbar sind, sei für Details auf diese Arbeit verwiesen. Über diese Clock-Instanzregeln wird es zudem ermöglicht, dass sich eine Clock-Instanz auf eine Kante des Objektgraphen bezieht, obwohl diese nicht als extra Objekt definiert wird.

Abbildung 2.24 zeigt die Spezifikation einer Clock-Instanz. Die ClockInstance *c* wird angelegt, wenn ein RailCab an einem Convoy teilnimmt.

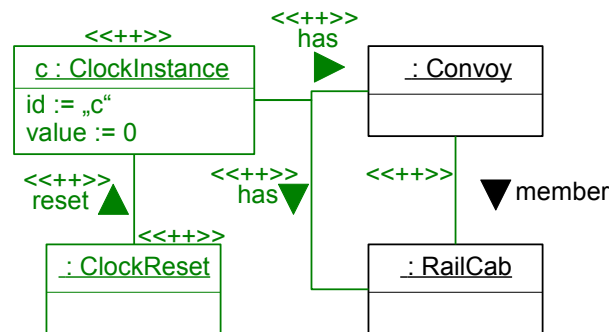


Abbildung 2.24: Definition einer Clock-Instanz und eines Clock Resets

Clock Resets Ein Clock Reset ist beschrieben über ein ClockReset Objekt und der Zugehörigkeit zu einer Uhr über eine reset Assoziation (siehe Abbildung 2.23). Eine ClockReset Instanz wird automatisch mit einer Clock-Instanz erzeugt. Ein Clock Reset wird spezifiziert, indem das mit der Clock-Instanz angelegte ClockReset Objekt gebunden wird.

In Abbildung 2.24 ist das Anlegen eines Clock Resets zu sehen. Abbildung 2.25 zeigt ein zurücksetzen der Clock *c*.

Time Guards Ein Time Guard wird spezifiziert über das Bedingungs-Element eines Story Pattern. Ein Time Guard nimmt Bezug zu einer Clock-Instanz und spezifiziert eine Bedingung über den Wert *value* dieser Instanz. Die Bedingung wird zu einem Booleschen Wert evaluiert. Um einen Time Guard über eine Clock-Instanz zu spezifizieren, muss die Clock-Instanz in dem Story Pattern gebunden sein. Es werden folgende Bedingungen berücksichtigt: $\phi ::= x \sim n \mid x - y \sim n \mid \phi \wedge \phi \mid true \mid false$, mit $x, y \in C, \sim \in \{\leq, <, =, >, \geq\}, n \in \mathbb{N}$ (vgl. [AD94, Alu99]).

Abbildung 2.25 zeigt einen Time Guard $5 \leq c \wedge c \leq 10$. Damit kann eine Übereinstimmung mit dem Objektgraphen nur im Intervall fünf bis zehn erfolgen. Ein neues Konvoimitglied muss also innerhalb dieses Intervalls hinzugefügt werden.

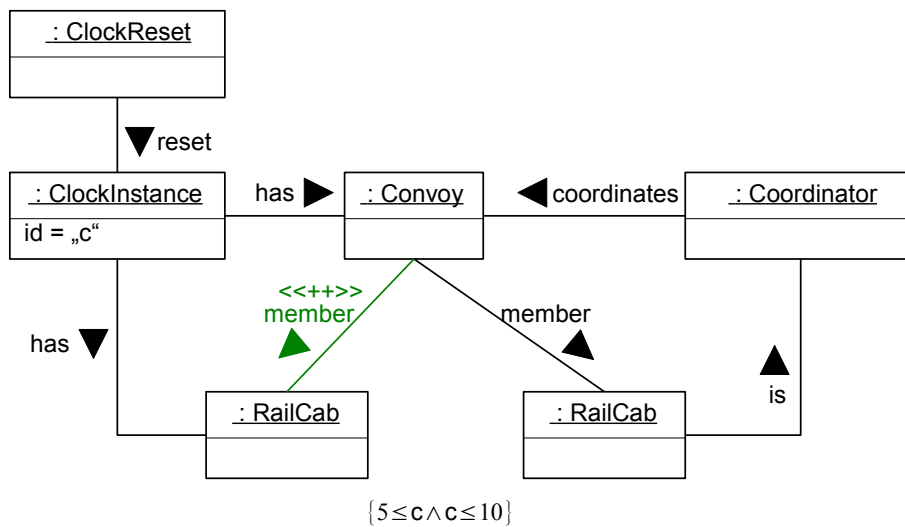


Abbildung 2.25: Clock Reset und Time Guard

Invarianten Eine Invariante wird durch ein Story Pattern ohne rechte Seite durch ein Bedingungs-Element von Story Pattern spezifiziert. Die Clock-Instanz, über die die Invariante spezifiziert wird, muss in dem Story Pattern gebunden sein.

Abbildung 2.26 zeigt ein Beispiel einer Invariante über die Clock *c*. Die Invariante spezifiziert, dass ein RailCab nicht länger als zehn Zeiteinheiten ohne Convoy sein soll.

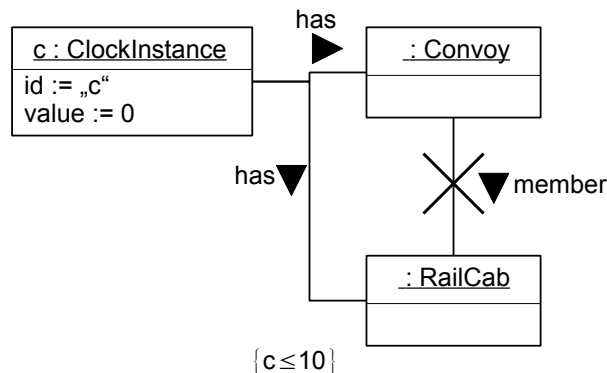


Abbildung 2.26: Invariante

Die Semantik von TIMED STORY PATTERN wird über die von TGTS definiert. Da TIMED STORY PATTERN lediglich eine andere Syntax verwenden und auf Objektgraphen statt allgemein auf Knoten agieren, bleibt die Semantik unverändert (vgl. Definition von Story Pattern über Graphtransformationssysteme [Zün01]). Die Definitionen sowie die Semantik von TGTS, Clock-Instanzregeln und Invariantenregeln [Hir08] sind damit uneingeschränkt auf TIMED STORY PATTERN anwendbar. Dies gilt auch für die Berechnung eines Folgegraphen über eine linke

und rechte Regelseite, da ein Story Pattern, wie in [Zün01] beschrieben, hierauf abbildbar ist, bzw. die linke und rechte Regelseite aus Vereinfachungsgründen in einer Sicht dargestellt werden.

2.6.3 Timed Story Diagrams

TIMED STORY DIAGRAMS erweitern Story Diagramme um Zeit. Zeitbedingungen werden dabei in Aktivitäten des Story Diagramms in Form von TIMED STORY PATTERN berücksichtigt.

Zeitbedingungen in Timed Story Diagrams Eine Zeitbedingung wird in TIMED STORY DIAGRAMS über TIMED STORY PATTERN definiert. TIMED STORY PATTERN können an Stelle von Story Pattern eine Aktivität spezifizieren.

Im Unterschied zu Story Diagrammen können damit TIMED STORY DIAGRAMS TIMED STORY PATTERN einbetten. Die Syntax bleibt daher unverändert. Diese Form der Definition führt zu dem Effekt, dass nur Zeit in einer Aktivität eines TIMED STORY DIAGRAM durch Anwenden eines TIMED STORY PATTERN vergehen kann. Wenn auch Zeit zwischen den Aktivitäten vergehen soll, muss ein extra TIMED STORY PATTERN eingeführt werden, welches entsprechend die zeitlichen Bedingungen realisiert.

Da die Definition der Semantik von Story Diagrammen unabhängig von den eingebetteten Story Pattern definiert ist, verändert sich die Definition der Semantik durch die Berücksichtigung von TIMED STORY PATTERN nicht. Entscheidend für die Ausführung eines Story Diagramms ist nur die Anwendbarkeit der eingebetteten Pattern. Die Anwendbarkeit ist wiederum durch das Pattern selbst definiert (siehe Abschnitt 2.6.2).

2.6.4 Timed Story Charts

TIMED STORY PATTERN und TIMED STORY DIAGRAMS ermöglichen die Beschreibung von Strukturanpassungen. Die hier vorgestellten TIMED STORY CHARTS schließen den Timed Story Driven Modeling Ansatz ab.

TIMED STORY CHARTS beschreiben wie Story Charts [Zün01] Zustandsverhalten. Um Verhalten für Echtzeitsysteme zu beschreiben dessen Struktur zur Laufzeit angepasst wird ist der Statchart-Ansatz nach Harel [Har87, HPSS87], der den Story Charts zu Grunde liegt, allerdings ungenügend. Zum einen fehlt die Möglichkeit Zeitbedingungen zu spezifizieren und zum anderen gibt es keine Möglichkeit die unterschiedlichen Verhaltensvarianten und Instanzen gesondert zu betrachten.

PARAMETERIZED REAL-TIME STATECHARTS stellen die benötigten Konstrukte zur Verfügung [HHG08, Hir08, HHH10] (siehe auch Abschnitt 2.4.4), um Zeitbedingungen sowie Verhalten spezifisch für eine Instanz zu beschreiben. Wir werden daher die Semantik von TIMED STORY CHARTS durch eine Abbildung von PARAMETERIZED REAL-TIME STATECHARTS auf be-

stimmte TIMED STORY DIAGRAM Elemente definieren. Damit wird ein konsistenter Formalismus für Multielemente, die eine dynamische Änderung der Kommunikationsstruktur beschreiben, gewahrt, da TIMED STORY CHARTS durch TIMED STORY DIAGRAMS ebenfalls über Story Pattern und Story Diagramme definiert werden. Dies stellt damit sicher, dass, wie in dem einleitenden Beispiel beschrieben, auch in der Strukturanpassung die Elemente des Kommunikationsprotokolls berücksichtigt werden können. Die Beschreibung von Zeitbedingungen wird über TIMED STORY DIAGRAMS durch TIMED STORY PATTERN ermöglicht, womit die unterlagerte Semantik über zeitbehaftete Graphtransformationssysteme (Timed Graph Transformation Systems) beschrieben ist (siehe Abschnitt 2.6.2).

Um einen gemeinsamen, konsistenten Formalismus zu definieren, müssen TIMED STORY CHARTS ebenfalls über Objekte definiert werden. Eine einfache und weit verbreitete Möglichkeit, um Statecharts objektorientiert darzustellen ist das Zustandsmuster [GHJV95]. Stallmann hat einen ersten Ansatz für die Abbildung von REAL-TIME STATECHARTS auf Objekte in [Sta08] vorgestellt. Ereignisse, Zeit, Parametrisierungen und Synchronisationen wurden allerdings nicht beschrieben. Weiterhin führen wir in dem hier vorgestellten Ansatz Transitionen nicht als extra Objekte ein sondern implizit über Regeln, da hierdurch keine relevanten Informationen verloren gehen und zudem eine Analyse erleichtert wird, da nicht explizit Objekte für die Transition erzeugt werden müssen.

Der durch Zündorf vorgestellte Story Chart Ansatz [Zün01] betrachtet zwar nur Statecharts, jedoch werden Ereignisse berücksichtigt sowie eine Semantik definiert. Folgend werden die wesentlichen Details und Unterschiede zu diesem Ansatz dargestellt.

Der Ansatz von [Zün01] beruht auf der Definition eines Framework, welches die Schaltregeln von Statecharts umsetzen. Das Framework wird mit Story Diagrammen und Story Pattern definiert. Dem zugrunde liegt ein Metamodell für Statecharts. Ein konkretes Statechart wird als Objektgraph des Metamodells spezifiziert. Dieser Objektgraph ist dann Eingabe für das Framework. Das Framework überprüft die möglichen zu schaltenden Transitionen und feuert gegebenenfalls, falls ein Transition schalten kann, Ereignisse nach einer sequentiellen Ausführungssemantik.

Neben dem Problem, dass dieser Ansatz keine Zeit, Parametrisierung sowie Synchronisationen berücksichtigt, ist der Framework Ansatz nicht gut geeignet, um Analysen durchzuführen. Es ist z. B. nicht ohne weiteres zu erkennen, in welcher Reihenfolge Transitionen ausgeführt wurden, da das Schalten einer Transition nur durch eine Framework-Methode umgesetzt ist.

Grundlegende Idee der Umsetzung von TIMED STORY CHARTS ist, dass wir für jedes PARAMETERIZED REAL-TIME STATECHART Element eine Abbildungsvorschrift bestimmen, wie dieses Element mit Story Pattern oder einem Story Diagramm (für kompliziertere Konstrukte) spezifiziert wird. Darauf basierend beschreiben wir eine Ausführungssemantik, in dem wir die einzelnen Elemente durch ein Story Diagramm verknüpfen. Das Story Diagramm spiegelt dabei die Ausführungssemantik von PARAMETERIZED REAL-TIME STATECHART wider. Diese Vorgehensweise ermöglicht eine einfache Anpassung der Ausführungssemantik durch den vorgeschlagenen modularen Aufbau. Die einzelnen Elemente (Stories) bleiben dabei unverändert. Andersherum lassen sich einzelne Elemente anpassen, ohne Auswirkung auf die Ausführungssemantik.

Weiterhin ermöglicht dieser Ansatz eine gute Nachvollziehbarkeit der geschalteten Transitionen, die direkt an den ausgeführten Story Diagrammen abgelesen werden können.

Die Abbildung wird im Rahmen dieser Arbeit auf die Sprachkonstrukte beschränkt, die für das in Abbildung 1.2 eingeführte Anwendungsbeispiel notwendig sind. Damit werden gemäß [Hir08] flache PARAMETERIZED REAL-TIME STATECHARTS mit einem hierarchischen Zustand unterstützt. Dies ermöglicht allerdings alle bisherigen Protokolle umzusetzen (siehe [May09]). Eine vollständige Unterstützung aller Sprachkonstrukte ist entsprechend ein Ausblick.

2.6.4.1 Übersicht Abbildung

Der Abbildung von PARAMETERIZED REAL-TIME STATECHARTS in TIMED STORY CHARTS liegt das in Abbildung 2.27 vorgestellte Metamodell zu Grunde. Zeitbedingungen werden wie bereits in Abschnitt 2.6.2 vorgestellt durch ein `ClockInstance` sowie `ClockReset` Objekt ermöglicht. Die Assoziation des `ClockInstance` Objekts mit dem Statechart und State Objekt ermöglicht die Spezifikation von Zeitbedingungen für Zustände sowie für das gesamte Statechart. Durch den in Abschnitt 2.27 vorgestellten Ansatz können zudem Zeitbedingungen für Transitionen spezifiziert werden. Weitere Merkmale sind, dass wir durch die `parameter` Attribute eine Parametrisierung ermöglichen, in dem für unterschiedliche Statechart Instanzen unterschiedliche Parameter vergeben werden. Im Folgenden werden wir den Ereignismechanismus erläutern (wir werden dabei den englischen Begriff `Event` verwenden) und den Ansatz zusammenfassend diskutieren. Eine detaillierte Beschreibung der einzelnen Elemente sowie die zusammengesetzte Ausführung der Elemente erfolgt in Appendix A.

Events Ein Event ist definiert durch ein Event Objekt. Der Name des Events wird über ein `name` Attribut angegeben. Ein Event Objekt kann über ein `Parameter` Objekt eine geordnete Menge von Parametern zugewiesen werden. Der Wert eines Parameters wird durch eine `value` Assoziation auf ein Object definiert. Events werden über eine Event-Queue verwaltet. Die Queue ist spezifiziert als eine einfache verkettete Liste, dessen Anfang und Ende mit der Event-Queue assoziiert sind. Eine Event-Queue ist genau einem Statechart zugeordnet. Dies gilt auch für Instanzen eines Statecharts.

Abbildung 2.28 zeigt die Abbildung von PARAMETERIZED REAL-TIME STATECHART-Events. Das Story Diagramm bindet zunächst die Zustände wie in Abschnitt A.1.2 dargestellt. Zusätzlich wird das erste Event (Event a) aus der Event-Queue des zugehörigen Statecharts gebunden. Wurde dieser Graph gebunden, so wird das ausgehende Event mit Namen b erzeugt und der Zustand gewechselt. Die Details der `enqueue` und `dequeue` Methode werden im Folgenden Semantik-Abschnitt erläutert.

Wie in [GB03] dargestellt, sind für Echtzeitsysteme ein asynchrones Event-Handling notwendig, um den verteilten Anforderungen der Systeme gerecht zu werden. Die von Harel eingeführte Mikro-Step- ([Har87]) und Super-Step-Semantik ([HN96]) ist für ein verteiltes Echtzeitsystem damit nicht praktikabel, da eine Nullzeit-Ausführung der Transitionen und ein unmittelbares

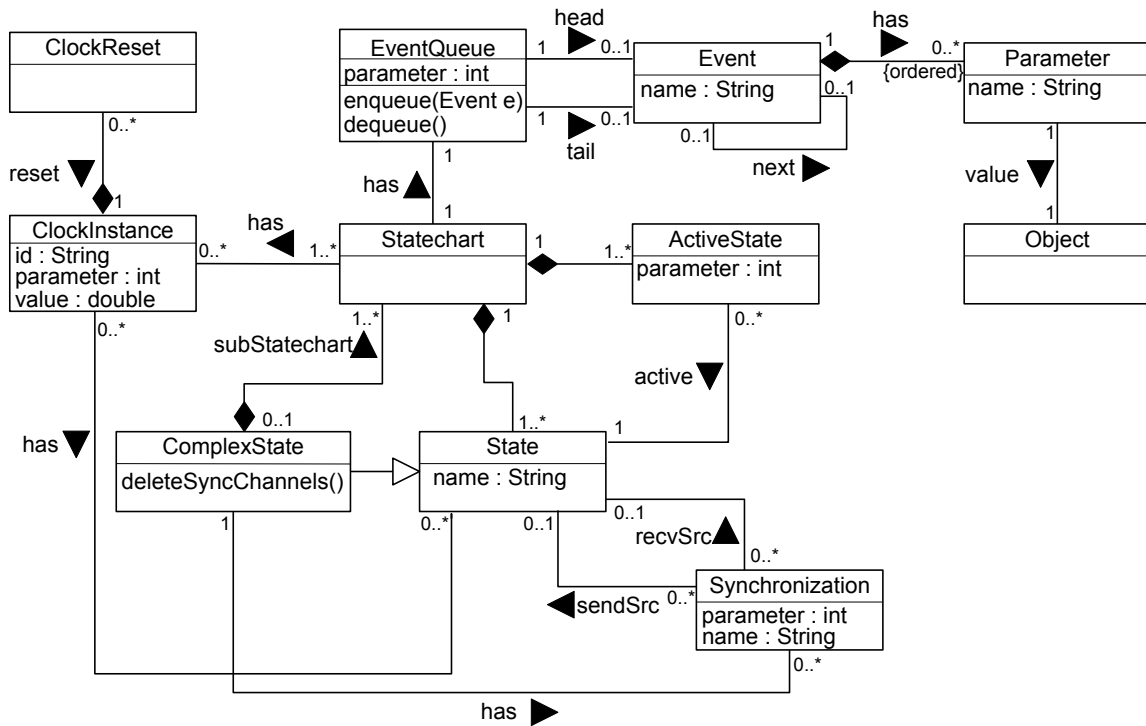


Abbildung 2.27: Metamodell für die Abbildung von Realtime Statecharts auf Story Diagramme

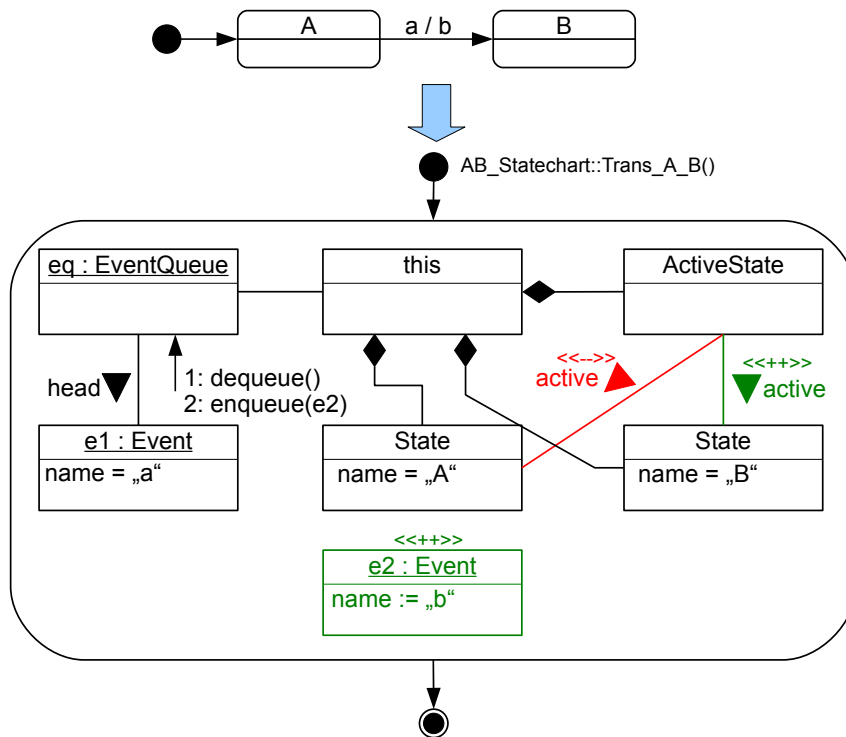


Abbildung 2.28: Schalten einer Transition mit Events

Konsumieren von Events nicht der Realität entspricht. Daher wurde in [GB03] ein asynchrones Event-Handling für REAL-TIME STATECHARTS eingeführt auf denen auch PARAMETERIZED REAL-TIME STATECHARTS basieren. Die konkrete Realisierung, ob z. B. nur das erste Element einer Queue überprüft wird oder alle Elemente der Queue, ist abhängig von dem konkreten plattformspezifischen Netzwerkverhalten. Dies wird typischerweise über Connectoren zwischen den Strukturelementen, wie Ports, definiert, die wiederum durch ein Statechart-Verhalten implementiert werden. Hierüber lassen sich dann z. B. Nachrichtenverluste implementieren. Grundsätzlich wurde in [GB03] festgelegt, dass unabhängig von dem konkreten Netzwerkverhalten Fifo-Queues für jedes Statechart definiert werden, die elementar für ein asynchrones Event-Handling sind. Damit werden die eingegangenen Events nacheinander in der Reihenfolge ihres Eingangs bearbeitet. Die definierten Event und Event-Queue Objekte setzen genau diese Semantik um.

Ein Zustandswechsel ist demnach nur möglich, wenn das Trigger-Event in der Queue enthalten ist. Wird eine Transition geschaltet und ein Event verschickt, so wird dieses Event in eine ausgehende Event-Queue gelegt oder direkt in die eingehende Event-Queue des Statecharts, welches dieses Event konsumieren soll. Durch die `dequeue` und `enqueue` Methode wird das Konsumieren der Events umgesetzt. Eine Transition kann demnach nur schalten, wenn die `dequeue` Methode das geforderte Event binden kann. Ist dies der Fall wird das Event aus der Queue entfernt und alle ausgehenden Events (Raised-Events) über die `enqueue` Methode der entsprechenden Queue hinzugefügt. Durch die vorgegebene eindeutige Struktur, auch im Falle eines Multicast, ist das Einsortieren der ausgehenden Events einfach möglich (siehe Abschnitt 2.6.1). Dieses gehört zum Statechart des Ports, der mit dem Port des sendenden Statecharts über eine Assembly verbunden ist. Die Semantik der PARAMETERIZED REAL-TIME STATECHARTS bleibt so offensichtlich erhalten. Im Folgenden wird noch eine Implementierung für die `dequeue` und `enqueue` Methode vorgeschlagen.

Die `dequeue` Methode versucht das erste Element der Event-Queue zu binden. Ist dies der Fall, wird dieses Event aus der Queue entfernt⁶ (siehe Abbildung 2.29). Die erste Story versucht das erste Element der Event-Queue zu binden. Kann kein Event gebunden werden, wird das Story Diagramm über die `failure` Kante verlassen und das Statechart kann die Transition nicht schalten. Die zweite Story spezifiziert das Entfernen des gebundenen Events aus der Event-Queue. Weiterhin wird der `head` Zeiger auf das `next` Element umstrukturiert. Ist dies nicht möglich, befindet sich nur noch ein Event in der Event-Queue. Damit können beide Links gelöscht werden.

Das Versenden eines Events wird durch die `enqueue` Methode umgesetzt. Diese Methode fügt das übergebene Event in die assoziierte Queue ein (siehe Abbildung 2.30). Beim Einfügen des Events in die Queue wird versucht das Event an die letzte Position einzufügen. Ist dies nicht möglich, ist die Event-Queue leer und das Element wird als erstes und letztes Element eingefügt. Andernfalls wird die `last` Assoziation auf das neue Event `e` umgesetzt und eine `next` Assoziation zwischen dem neuen vorletzten und letzten Element eingefügt.

⁶Die dargestellte Umsetzung überprüft nur das erste Element. Der Vollständigkeit halber müsste über die gesamte Queue iteriert werden.

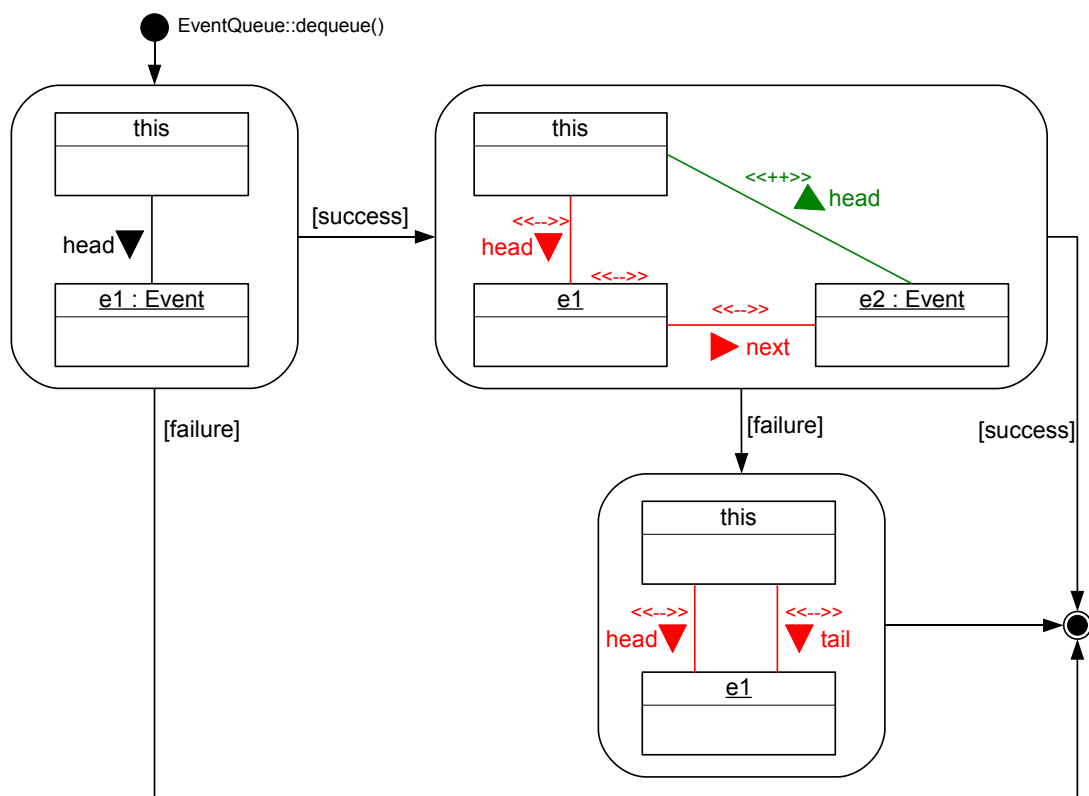


Abbildung 2.29: Dequeue der Event Handling Queue

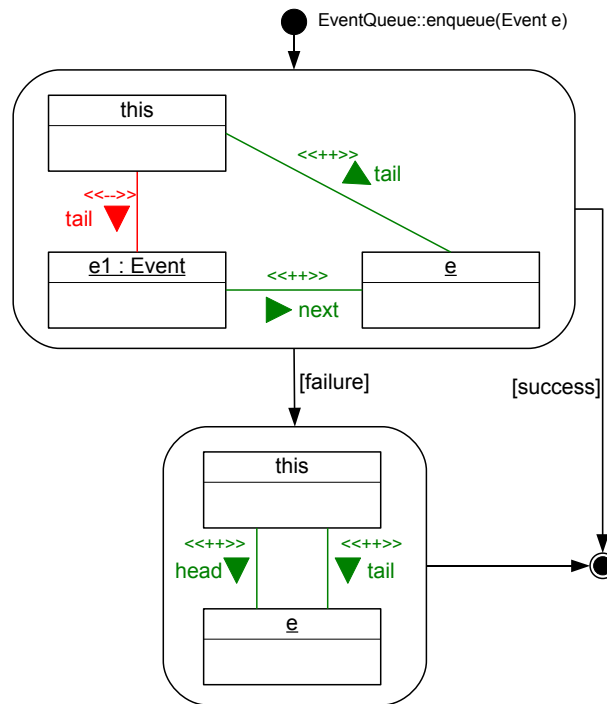


Abbildung 2.30: Enqueue der Event Handling Queue

Diskussion Durch den gemeinsamen Formalismus für das Verhalten und die Strukturänderungen ermöglicht dieser Ansatz im Vergleich zu dem bisherigen MECHATRONIC UML Ansatz prinzipiell eine Verifikation von Sicherheits- und Lebendigkeitseigenschaften (siehe Abschnitt 2.4.1 und 2.4.6.1), die sowohl Struktur als auch Verhalten betrachten. Im Rahmen aktueller Arbeiten wird dies adressiert [HSJZ10, HHPS10, EHH⁺11]. Wir werden eine formale Verifikation von Sicherheits- und Lebendigkeitseigenschaften im Folgenden allerdings nicht näher betrachten, da dies nicht der Fokus dieser Arbeit ist. In Kapitel 3 werden wir eine Wiederverwendung von modellierten Komponenten vorstellen, in dem wir auf Basis der TIMED STORY CHARTS eine Verfeinerung definieren und diese anschließend verifizieren.

Kapitel 3

Verfeinerung in hierarchischen Komponentensystemen

Für die in Abschnitt 2.1 auf Seite 13 vorgestellten Konkretisierungen muss eine Verfeinerung definiert und überprüft werden, um sicherzustellen, dass die durchgeführten Konkretisierungen nicht zu einer Verletzung des bereits verifizierten Protokollverhaltens führen. Für die betrachteten Systeme, ist dies eine Herausforderung, da sowohl die Verifikationsergebnisse, wie auch das nach außen sichtbare Echtzeitverhalten des übergeordneten (abstrakten) Protokollverhalten erhalten bleiben müssen, unter Berücksichtigung von kompositionellen Strukturanpassungen.

Wir stellen in diesem Kapitel einen Ansatz vor, der genau diese Anforderungen adressiert. Darüber hinaus betrachten wir die Anforderung an eine Verfeinerung möglichst viele Konkretisierungen zuzulassen, um die Wiederverwendung existierender Lösungen zu fördern. Im Vergleich zu bisherigen Ansätzen (wie [JLS00, GRPS02, Bey02, GTB⁺03, HT04, Bur06, Gie07, ÖM07]) können wir daher zum einen überhaupt durch die Betrachtung der Strukturanpassungen in Kombination mit Echtzeitverhalten eine Verfeinerungsüberprüfung für selbstoptimierende, mechatronische Systeme durchführen. Zum anderen ermöglicht unsere Verfeinerungsdefinition durch Fokussierung auf die MECHATRONIC UML tatsächlich einen höheren Grad an Wiederverwendung existierender Lösungen.

Handelt es sich bei der Wiederverwendung um Multielemente, muss eine Verfeinerung für TIMED STORY CHARTS definiert werden, da nicht nur das Verhalten, sondern auch die Strukturanpassung eine Auswirkung auf die Verfeinerung hat (siehe Kapitel 2.6). Für Einfachelemente muss nur eine Verfeinerung für das Verhalten, also REAL-TIME STATECHARTS (siehe Abschnitt 2.4.2), definiert werden. Da die Verfeinerung für TIMED STORY CHARTS auf der von REAL-TIME STATECHARTS aufbaut, definieren wir zuerst eine Verfeinerung für REAL-TIME STATECHARTS in Abschnitt 3.1.1 und anschließend für TIMED STORY CHARTS in Abschnitt 3.1.2. In Abschnitt 3.2 beschreiben wir eine Verfeinerungsüberprüfung auf Basis dieser Definitionen.

Im Folgenden werden wir zuerst das in Abschnitt 1.2 eingeführte Beispiel konkretisieren, um hieran die geschilderte Problematik zu verdeutlichen. Die Anforderungen und Voraussetzungen an die Verfeinerung betrachten wir in dem darauf folgenden Paragraphen auf Seite 67.

Beispielanwendung Zur Veranschaulichung betrachten wir wieder die Coordinator-Komponente des RailCab Konvoibeispiels (siehe Abbildung 2.18). Abbildung 3.1 zeigt einen Ausschnitt des REAL-TIME STATECHARTS der Coordinator-Komponente. Es wird sowohl das Kommunikationsverhalten der Coordinator-Komponente als auch der eingebetteten PosCalc-Komponente gezeigt (rechte Spalte der Abbildung). In der linken Spalte wird das Adaptionverhalten des Multi-Ports Coordinator, des Multi-Parts PosCalc und der Multi-Delegation gezeigt. Die drei Adaptionverhalten rufen jeweils die dazugehörige Strukturanpassung auf (siehe Abbildung 2.18).

Das Kommunikationsverhalten der Multi-Rolle Coordinator wird über das Adaptionverhalten der Multi-Rolle durch eine `next[k]`-Synchronisationsnachricht angestoßen. Innerhalb von fünf Zeiteinheiten verschickt das Kommunikationsverhalten eine parametrisierte `update(para)`-Nachricht an den Konvoiteilnehmer k , um diesen die aktuellen Konvoiparameter für dieses RailCab zu schicken. Innerhalb der Zeitinvariante von $c1 \leq 25$ wird eine `ack()`-Nachricht von dem Konvoiteilnehmer erwartet. Wurde bereits allen Konvoiteilnehmern eine `update(para)`-Nachricht zugeschickt, so wird dem Adaptionverhalten mitgeteilt, dass alle Teilnehmer aktualisiert wurden (`done`-Synchronisationsnachricht). Ist dies nicht der Fall, wird die nächste Portinstanz über die `next[k+1]`-Synchronisationsnachricht angestoßen, um den zugeteilten Konvoiteilnehmer ebenfalls aktuelle Konvoiparameter zu zuschicken.

Das Adaptionverhalten der Coordinator-Multi-Rolle, welches parallel zu dem Kommunikationsverhalten ausgeführt wird, koordiniert die Portinstanziierung sowie die erzeugten Ports untereinander. Initial befindet sich das Adaptionverhalten im Zustand `NoConvoy`. Wird die `initPort`-Synchronisationsnachricht empfangen, wird der Seiteneffekt `updatePort(1)` ausgeführt (siehe Abbildung 3.3). Die `initPort`-Synchronisationsnachricht kann dabei z. B. durch eine Synchronisation mit dem `Registry-Port` angestoßen werden, wenn ein RailCab über die gleiche Registrierung verwaltet wird und zudem mit diesem RailCab verhandelt wurde, dass ein gemeinsamer Konvoi gebildet werden soll.

Der `updatePort`-Seiteneffekt beschreibt zusätzlich zu dem TIMED STORY DIAGRAM in Abbildung 2.18 die Situation, dass noch kein oder nur ein Port angelegt wurde. Die erste Story überprüft, ob das `CoordPort`-Objekt bereits eine Verbindung zu einem `CoordMember`-Objekt hat. Das `CoordPort`-Objekt ist die Stellvertreter-Klasse für das Adaptionverhalten, um die einzelnen Port-Klassen, die durch das `CoordMember`-Objekt repräsentiert werden, zu verwalten. Wurde noch kein Port angelegt, so wird der (erste) Port angelegt und mit der übergebenen Portposition initialisiert. Für die Initialisierung des Konvois, wird diese Regel über den `updatePort(1)`-Seiteneffekt ausgelöst. Wurden bereits Ports hinzugefügt, so wird überprüft, ob der Vorgänger-, bzw. Nachfolge-Port bereits eine `next`-Assoziation zu einem Port hat. Ist dies nicht der Fall, wird ein neuer Port und eine entsprechende `next`-Assoziation zu dem direkten Vorgänger-, bzw. Nachfolger-Port erzeugt. Soll der Port andernfalls zwischen zwei bisherigen Ports eingebunden werden, wird die letzte Story ausgeführt, wie in Abbildung 2.18 beschrieben.

Ist der erste Port erzeugt, wird gewartet, bis die PosCalc-Rolle eine entsprechende Part-Komponente erzeugt hat und die Delegation eine Verbindung zwischen diesen anlegt. Dies wird über die Synchronisationsnachrichten `createPort`, `addPart`, `portCreated` und `partCreated`

erreicht. Ist dies der Fall befindet sich das Adaptionsverhalten der Coordinator-Multi-Rolle entweder in einer Schleife, in der ein weiterer Port hinzugefügt (initialisiert über eine addPort-Synchronisationsnachricht) oder eine Aktualisierung der Konvoiparameter durch eine next[1]-Synchronisationsnachricht ausgelöst wird.

Das Adaptionsverhalten der Rolle PosCalc verhält sich ähnlich zu dem Adaptionsverhalten der Multi-Rolle Coordinator. Als Seiteneffekt wird die updatePart()-Methode aufgerufen, die, wie zu Abbildung 2.18 beschrieben, PosCalc-Parts erzeugt.

Das Kommunikationsverhalten der PosCalc-Rolle unterscheidet sich allerdings merklich von dem Kommunikationsverhalten der Coordinator-Multi-Rolle (siehe Abbildung 3.1). Es unterscheiden sich die Anzahl der Zustände, die Zustandsnamen sowie die Zeitintervalle. Der Zustand ComputeParam des PosCalc-Kommunikationsverhaltens bettet zudem eine Steuerung zur Berechnung der Konvoiparameter ein, die periodisch mit aktuellen Parametern aufgerufen wird und dessen Ergebnisse mittels der parametrisierten Nachricht update verschickt werden. Um dies zu ermöglichen, bettet die PosCalc-Komponente ebenfalls auf der strukturellen Ebene die Steuerung CPCController ein (siehe Abbildung 3.2). Es ist nicht nur das Verhalten unterschiedlich umgesetzt, sondern auch die verwendeten strukturellen Elemente. Zum einen wird eine Multi-Rolle restrukturiert und zum andern ein Multi-Part.

Das Delegationsverhalten beschreibt eine alternative Umsetzung für das TIMED STORY DIAGRAM in Abbildung 2.19. Vorteil ist hier die konsequente Aufteilung in Adaptions- und Kommunikationsverhalten. Dies führt wiederum zu einer entkoppelten Spezifikation der Restrukturierungen.

Bisherige Ansätze, wie in Abschnitt 2.4.7 und 7.2 beschrieben (dies beinhaltet auch die Verfeinerungsdefinition der MECHATRONIC UML), würden diese Konkretisierung nicht zulassen. Dies liegt an dem allgemeineren Charakter dieser Ansätze, wodurch keine Relaxierung der Zeit (Zeitintervallverschiebungen) erlaubt werden (können). Darüber hinaus wäre grundsätzlich eine Verfeinerungsüberprüfung nicht möglich, da keine Strukturanpassungen berücksichtigt werden. Unser Ansatz wird diese Konkretisierung zu lassen, da wir spezifisch für die MECHATRONIC UML eine Zeitintervallverschiebung in bestimmten Bereichen erlauben können, unter Berücksichtigung von Strukturanpassungen. Im Folgenden werden wir die Anforderungen und Voraussetzungen genauer erläutern.

Anforderungen und Voraussetzungen Um die in der Beispielanwendung diskutierte Konkretisierung zu erlauben, nutzen wir den spezifischen Ansatz der MECHATRONIC UML und die damit verbundene Anwendungsdomäne aus.

Begrenzte Zeitbedingungen. Der unterlagerte Verhaltensformalismus der MECHATRONIC UML, die Timed Automata, ermöglichen die Modellierung eines Systems mit anwachsender, nicht beschränkter Zeit (Uhren). Dies kann potentiell zu einer nicht Analysierbarkeit des modellierten Systems führen, bzw. eine nicht Implementierbarkeit des Modells.

Die in dieser Arbeit betrachteten harten Echtzeitsysteme (siehe Abschnitt 2.4) fordern allerdings, dass die ausgeführten Berechnungen vorhersagbar in einer bestimmten Zeit ein Ergebnis liefern

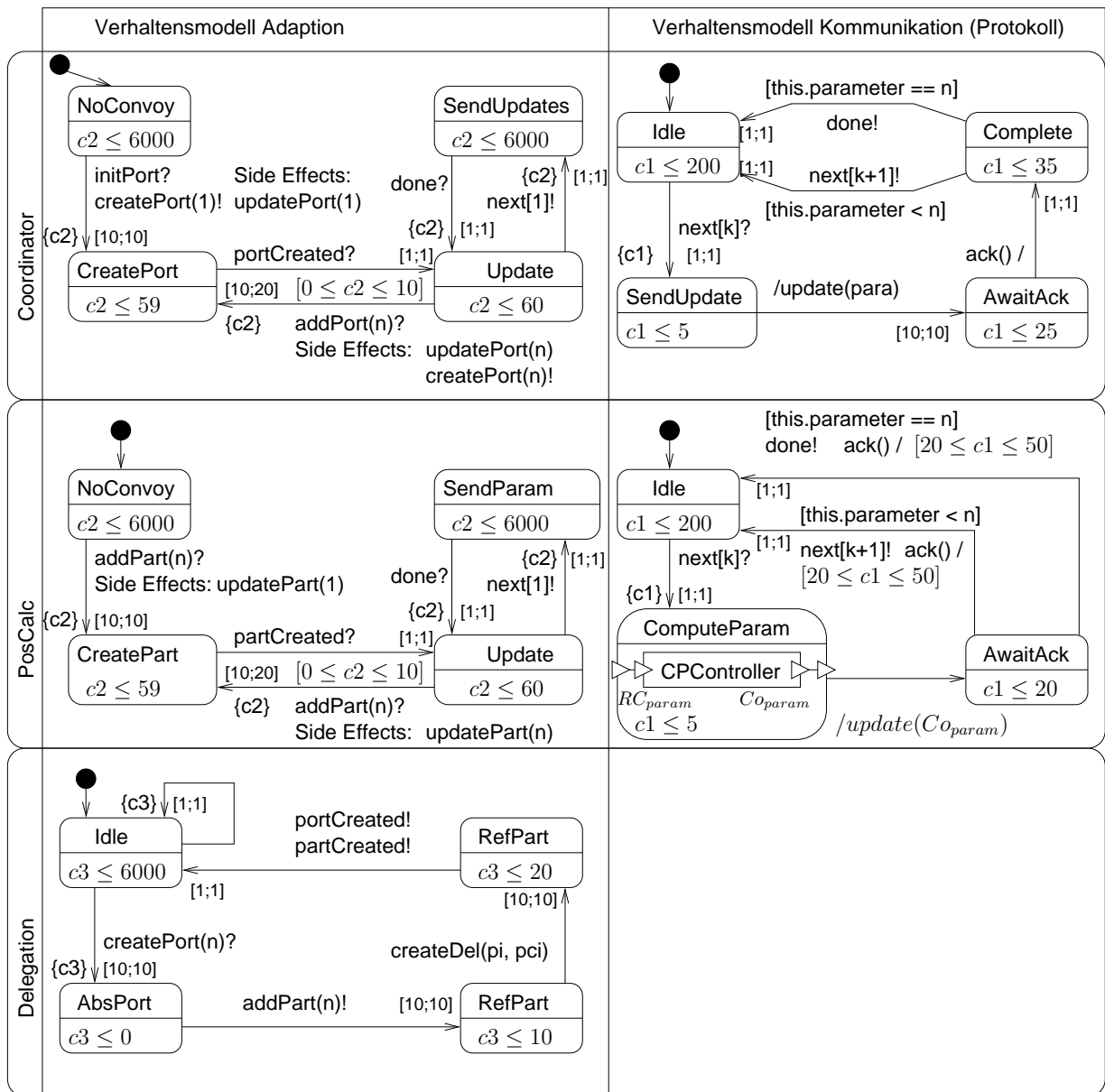


Abbildung 3.1: Verhaltensmodell Coordinator-Komponente

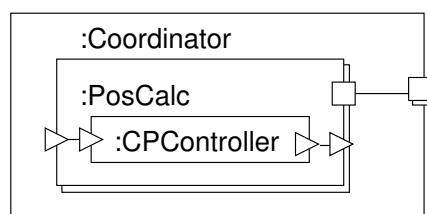


Abbildung 3.2: Coordinator-Komponente mit eingebetteten Regler

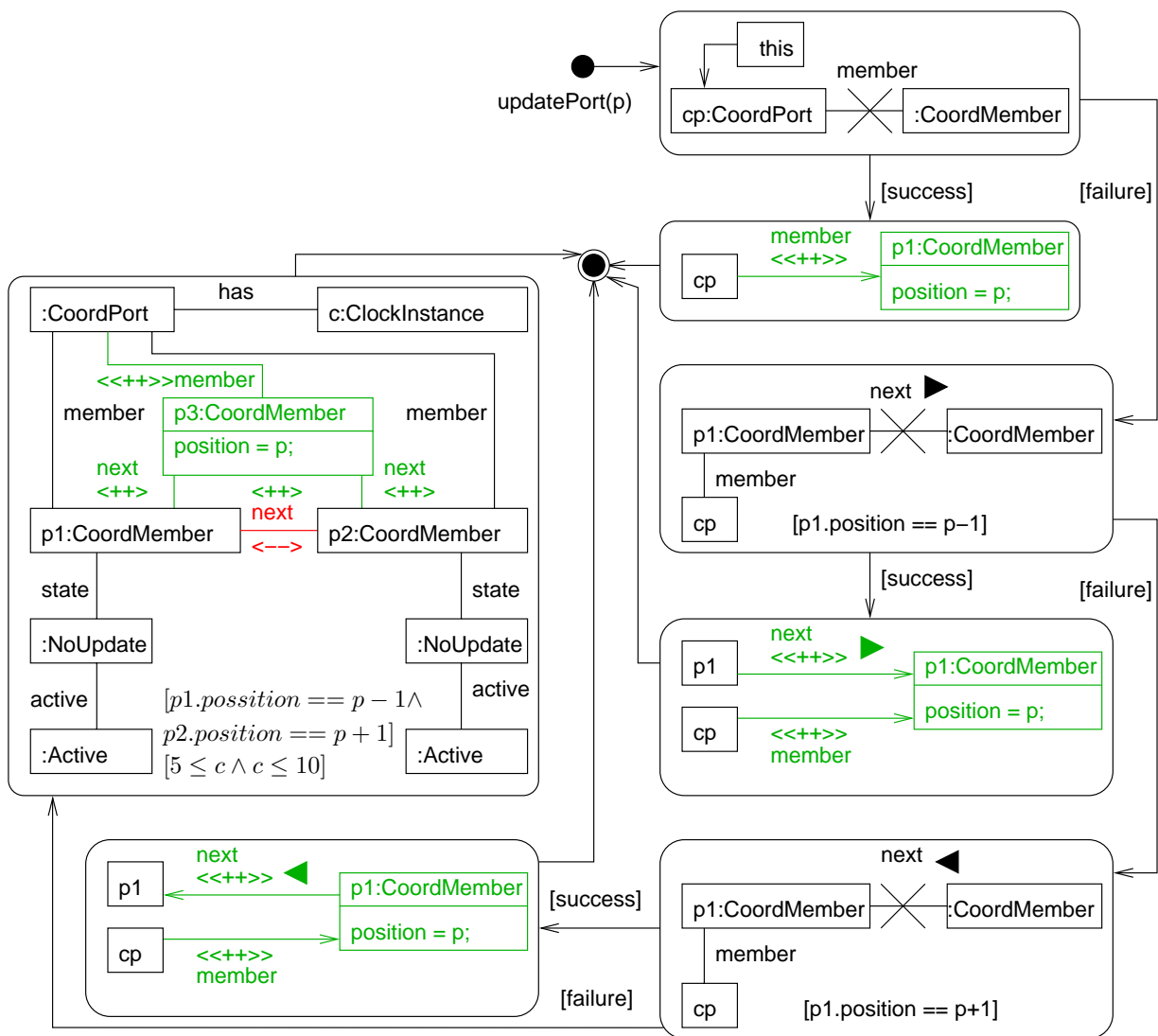


Abbildung 3.3: Verfeinertes UpdatePort Timed Story Diagramm

[But05, Kop97]. In der MECHATRONIC UML, wird dies erreicht, indem Deadlines für die ausgeführten Berechnungen und Invarianten spezifiziert werden. Hierdurch wird Fortschritt im System erzwungen und ein Wachsen der Zeit über alle Grenzen verhindert. Diese Einschränkungen führen zu einem endlichen System, welches durch z.B. eine Erreichbarkeitsanalyse auf bestimmte Eigenschaften überprüft werden kann (siehe z.B. [Bur06]). Das Problem des Zeno Schaltens, womit unendlich viele Schaltvorgänge in endlicher Zeit möglich sind, wird hiermit ebenfalls umgangen, da das Schalten einer Transition einen Fortgang der Zeit erzwingt.

Zeitliche Bedingungen für Strukturanpassungen. Die Strukturanpassungen des Systems werden ausschließlich als Seiteneffekte von Statecharts aufgerufen (siehe Adaptionsverhalten in Abbildung 3.1). Eine Transition, die einen Seiteneffekt aufruft, spezifiziert die zeitlichen Bedingungen der Strukturanpassungen.

Dazu muss zusätzlich separat geprüft werden, ob die WCET der Strukturanpassung die Deadline der Transition einhalten kann ([BBB⁺09]). Dies gilt auch für Transitionen, die keine Strukturanpassung ausführen, da auch deren Ausführung in einem realen System Zeit verbraucht. Diese Überprüfung wird nicht innerhalb der Verfeinerungsüberprüfung durchgeführt, sondern als ein explizierter zusätzlicher Schritt (siehe Kapitel 6.1).

Asynchrone Echtzeitkommunikation. Eine asynchrone Echtzeitkommunikation ist nach [Dou02] implizit (immer) durch ein Watchdog Muster in Kombination mit einem Puffer implementiert. Ein Watchdog Echtzeitkommunikationsmuster erwartet nach jedem Verschicken einer Nachricht nach einer bestimmten Zeit eine Antwort von dem Kommunikationspartner, bevor weitere Aktionen ausgeführt werden. Dieses Muster ist damit Elementar für alle REAL-TIME COORDINATION PATTERNS (siehe auch [May09]). Diese Informationen über den Kommunikationspartner werden durch unseren Ansatz ausgenutzt, um eine Zeitintervallverschiebung durch eine Konkretisierung zu erlauben, ohne die Eigenschaften der bisherigen (abstrakten) Kommunikation zu verletzen.

Kontinuierliche Zeit. Die bisherige Verfeinerung der MECHATRONIC UML für Rollenverhalten basiert auf diskreter Zeit [Gie03, GTB⁺03]. Für asynchrone Systeme ist ein diskretes Zeitmodell allerdings nicht anwendbar, da im allgemeinen Fall eine Erreichbarkeitsanalyse nicht möglich ist ([CGP00]). Für physikalische Systeme ist zudem ein kontinuierliches Zeitverhalten in der plattformunabhängigen Modellierungsphase von Vorteil, da eine Taktung zwangsläufig zu einem komplizierten Modell führt und zudem häufig nicht eindeutig bestimmbar ist, da z.B. regelungstechnische Modelle typischerweise ebenfalls von kontinuierlicher Natur sind. Ein diskretes Zeitsystem führt daher im Allgemeinen zu einem eingeschränkteren plattformunabhängigen Modell. Eine automatische Diskretisierung erst während der Implementierungsphase bzw. Codegenerierungsphase vorzunehmen erleichtert zudem die Entwicklung dieser Systeme und ist, wie in [MPS95, AMPS98] vorgestellt, für Timed Automata möglich. Aus diesem Grund verwenden wir kontinuierliche Zeit (auch *dense-time* genannt). Da sich die Eigenschaften von diskreter Zeit im Vergleich zu kontinuierlicher Zeit stark unterscheiden [AD94], kann die Definition aus [GTB⁺03] nicht (einfach) angewandt werden.

Deterministische Modelle. Ein deterministisches Modell ist Voraussetzung für eine Verfeinerung, da für beliebige nichtdeterministische Automaten die Korrektheit der Verfeinerung nicht über

Pfade gezeigt werden kann [AD94]. Dies liegt daran, dass die Auswahl der Pfade im abstrakten und verfeinerten Verhalten unterschiedlich bestimmt werden können.

Die betrachteten Modelle können grundsätzlich nichtdeterministisches Verhalten aufweisen. Um das Verhalten aber tatsächlich auf ein reales System umsetzen zu können, muss es eine Abbildung in ein deterministisches Modell geben (da Rechnerarchitekturen deterministisch sind, bzw. nur deterministisches Verhalten umsetzen können). Wie bereits in [MPS95, AMPS98, Sto02] gezeigt wurde, lassen sich Timed Automata automatisch auf deterministische Automaten abbilden, wodurch es entsprechend auch eine solche Abbildung für REAL-TIME STATECHARTS gibt.

Die in dieser Arbeit definierte Verfeinerung wird daher direkt für deterministisches Verhalten beschrieben.

Wohldefinierte Architektur. Für zwei beliebige Strukturen und Verhalten ist es prinzipiell schwierig eine Verfeinerung zu zeigen, wenn keine konkrete Verbindung zwischen den abstrakten und konkreten Modellen vorliegt.

Durch die wohldefinierte Komponentenstruktur ist eine Verbindung zwischen den Komponenten auf unterschiedlicher Hierarchieebene durch eine Delegation gegeben. Dies kann wiederum für eine Verfeinerung des Verhaltens und der Strukturanpassung ausgenutzt werden, da die beteiligten Strukturen und die dazu gehörigen Verhaltensbeschreibungen eindeutig in Verbindung stehen.

Anforderungen an die Verfeinerung. Die Merkmale der Anwendungsdomäne haben einen wesentlichen Einfluss auf die Verfeinerungsdefinition. In unserem Fall ist besonders hervorzuheben, dass wir sicherheitskritische Echtzeitsysteme betrachten. Hieraus folgt, dass Zeitbedingungen und der Erhalt von Verifikationsergebnissen eine wichtige Rolle spielen. Zudem gibt es die Forderung, möglichst viele existierende Lösungen wiederverwenden zu können. Damit ergeben sich folgende Anforderungen an eine Verfeinerung:

1. Das extern sichtbare Protokoll (die Echtzeit-Nachrichtenkommunikation) des abstrakten Protokolls muss durch das verfeinerte Protokoll erhalten bleiben.
2. Die auf dem abstrakten Protokoll durchgeführten Verifikationen sollen auch für die Verfeinerung gelten. Dies sind für einen kompositionellen Ansatz alle ATCTL Formeln. Das sind alle TCTL-Formeln (Timed Computation Tree Logic, [ACD93]), die ausschließlich Allquantoren und keine Negationen vor Allquantoren enthalten (siehe Abschnitt 2.4.6 und 2.4.7).
3. Die Verfeinerungsdefinition soll möglichst viele existierende Lösungen zu lassen, bzw. möglichst viele Änderungen an dem abstrakten Protokollverhalten erlauben. Hierdurch soll die Wiederverwendung existierender Lösungen und die Entwicklung von notwendigen neuen verfeinerten Protokollverhalten vereinfacht werden.

Abbildung 3.4 verdeutlicht die Anforderungen an die Verfeinerung. Das PosCalc Protokollverhalten soll zum einen das abstrakte Coordinator Protokollverhalten erfüllen und zum anderen sollen die verifizierten Eigenschaften des Coordinator Protokollverhaltens für das PosCalc Protokollverhalten erhalten bleiben. Diese beiden Anforderungen lassen sich durch eine restriktive

Verfeinerung, die keine Änderung des Protokollverhaltens erlaubt, einfach erfüllen. Hiermit wird allerdings die letzte Anforderung nicht erfüllt. In unserem Beispiel führt dies dazu, wie auch in Abschnitt Beispielanwendung auf Seite 65 beschrieben, dass das PosCalc Protokollverhalten keine gültige Verfeinerung des Coordinator Protokollverhaltens ist. Die im Folgenden Abschnitt beschriebene Verfeinerung zeigt, wie diese kontroversen Anforderungen erfüllt werden können.

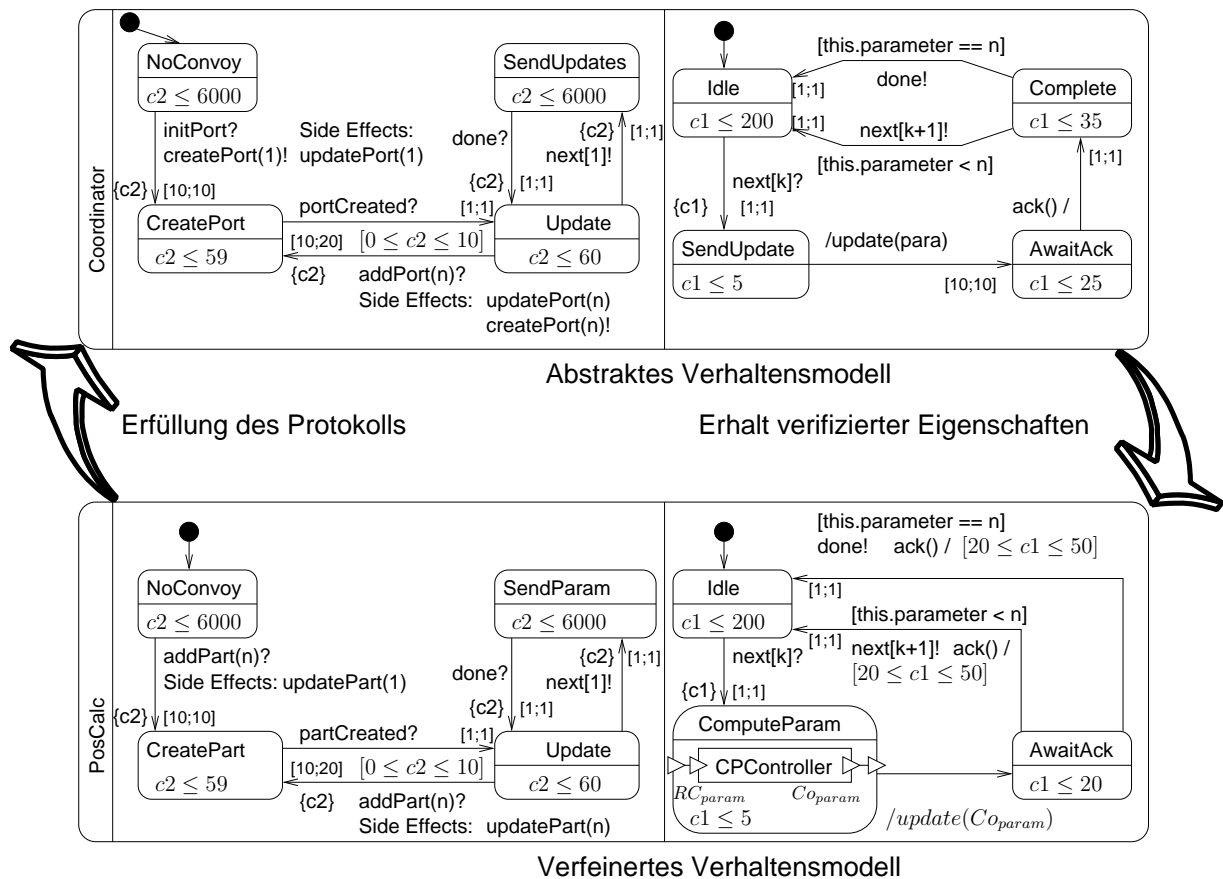


Abbildung 3.4: Anforderungen an die Verfeinerung

3.1 Verfeinerungsdefinition

Zuerst betrachten wir im Folgenden eine Verfeinerung für Einaufbauelemente, die keine Strukturanpassung berücksichtigen. In diesem Fall muss nur eine Verfeinerung für das Verhalten definiert werden. Für den Modellierungsansatz MECHATRONIC UML muss also eine Verfeinerung für REAL-TIME STATECHARTS definiert werden, die wir in Abschnitt 3.1.1 vorstellen. Für Multielemente muss nicht nur das Verhalten, sondern auch die Strukturanpassung berücksichtigt werden. Daher definieren wir eine Verfeinerung für TIMED STORY CHARTS

in Abschnitt 3.1.2, die genau dies berücksichtigt. Grundlegende Arbeiten hierzu wurden in [HHG08, Hei09, HHZ09, HHH10, Bre10, HH11] vorgestellt.

3.1.1 Real-Time Statecharts

Die Anforderungen an eine Verfeinerung für REAL-TIME STATECHARTS ergeben sich aus dem spezifischen Einsatz in der MECHATRONIC UML, wie in Abschnitt Anforderungen und Voraussetzungen auf Seite 67 beschrieben. Damit soll durch eine Verfeinerung das nach außen sichtbare Verhalten und Verifikationsergebnisse erhalten bleiben. Weiterhin soll die Verfeinerung möglichst viele Konkretisierungen zu lassen.

In Abschnitt 2.4.7 haben wir bereits relevante Verfeinerungen für die MECHATRONIC UML vorgestellt (die Timed Simulation, Timed Bisimulation und Timed Ready Simulation). Diese erfüllen allerdings nur zum Teil die gestellten Anforderungen. Die auf Simulationen basierenden Verfeinerungen erhalten zwar relevante Eigenschaften der Verifikation (\leq_S), jedoch wird hierdurch nicht gefordert, dass sämtliches im abstrakten System mögliche Verhalten vom konkreten System ebenfalls unterstützt wird. Eine Timed Simulation erhält zudem die Zeitintervalle T_A (\leq_{TS}) der Abstraktion. Die Timed Ready Simulation bezieht sich zusätzlich zur Timed Simulation auf den Erhalt von urgent-Transitionen (\leq_{TRS}). Die auf Bisimulation aufbauenden Verfeinerungen erfüllen den Erhalt beider Richtungen (\leq_S und \geq_S). Eine Timed Bisimulation erhält zudem die Zeitintervalle (\leq_{TBS}).

All den zeitbehafteten Verfeinerungen ist gemein, dass sie keine Relaxierung der Zeitintervalle erlauben. Wir werden daher eine *relaxierte, zeitbehaftete Bisimulation* einführen (*Relaxed Timed Bisimulation, RTBS*), die die Voraussetzungen durch die MECHATRONIC UML ausnutzt, um Zeitintervallverschiebungen zu erlauben. In Abbildung 3.5 haben wir zusammenfassend die Beziehung zwischen der Relaxed Timed Bisimulation und der hiermit in Bezug stehenden (Timed) Bisimulation dargestellt.

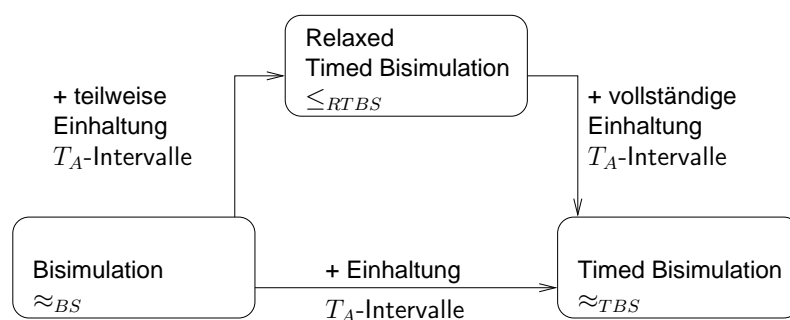


Abbildung 3.5: Beziehung zwischen RTBS und (Timed) Bisimulation

Um eine Überprüfung der Verfeinerung zu ermöglichen, die wir in Abschnitt 3.2 vorstellen, betrachten wir im Folgenden eine Definition der Verfeinerung direkt über Clock Zones (siehe Definition 6). Dies ermöglicht uns eine Implementierung der Verfeinerung über die Difference

Bound Matrice, die eine effiziente Repräsentation der Clock Zones durch eine Matrix zur Verfügung stellen (siehe Abschnitt 2.4.2.1).

Um die Clock Zones direkt in der Verfeinerung zu berücksichtigen, müssen wir die Pfade des externen Echtzeitverhaltens direkt darstellen. Hiermit können wir über die jeweiligen Intervalle und deren Clock Zone argumentieren. Dies wird über sogenannte Timed Traces [YJ94] ermöglicht, die wie folgt definiert sind.

Definition 15 (Timed Trace)

Sei M ein Timed Automaton (siehe Definition 1) mit extern sichtbaren Ereignissen (Nachrichten) $A = A_i \cup A_o$ mit A_i empfangene Nachrichten, A_o gesendete Nachrichten und $A \subseteq \Sigma$. Ein Timed Trace ist ein Ausführungspfad π von M für den gilt:

$$\pi = (s_0, t_0) \Rightarrow_{\delta_0} (s_0, t_0 \oplus \delta_0) \Rightarrow_{a_0} (s_1, t_1) \Rightarrow_{\delta_1} (s_1, t_1 \oplus \delta_1) \dots$$

wobei \Rightarrow_{δ_0} dem Vergehenlassen einer Zeitspanne δ_0 entspricht und \Rightarrow_{a_0} einem Zustandswechsel auf Basis einer Nachricht $a_0 \in A$.

Ein Zustand eines Traces ist über Zones nach Definition 6 wie folgt definiert:

Definition 16 (Zustände eines Timed Trace)

Sei M ein Timed Automaton. Ein Zustand S eines Timed Trace ξ zu M ist eine Zone $\langle s, z \rangle$, wobei s eine Location aus M und z eine Clock Zone ist. Es bezeichnet $S.s$ die Location des Timed Automaton und $S.z$ die Clock Zone von S . Es bezeichnet weiterhin $z.c$ die Menge der Clock Constraints über die Clock c der Clock Zone z .

Die Beschränkung eines Timed Traces auf das extern sichtbare Verhalten führt dazu, dass die Transitionen eines Timed Traces das interne Verhalten verbergen.

Definition 17 (Transitionen eines Timed Trace)

Seien S, T Zustände eines Timed Trace, ε das intern ausgeführte Verhalten der Transitionen und d_i Zeitintervalle, dann gilt:

1. $S \Rightarrow_a T$ falls $S(\Rightarrow_{\varepsilon})^* \Rightarrow_a (\Rightarrow_{\varepsilon})^* T$
2. $S \Rightarrow_{\delta} T$ falls $S(\Rightarrow_{\varepsilon})^* \Rightarrow_{d_1} (\Rightarrow_{\varepsilon})^* \dots (\Rightarrow_{\varepsilon})^* \Rightarrow_{d_n} (\Rightarrow_{\varepsilon})^* T$ mit $\delta = \sum_{i=1}^n d_i$

Mit Hilfe dieser Definition wird es ermöglicht, dass nach jeder extern sichtbaren Transition beliebig viele interne Transitionen geschaltet werden können (1.). Interne Transitionen können z.B. Synchronisationen (wie next[k]) und Seiteneffekte (wie updatePort(n)) ausführen. Dieses Prinzip wird ebenfalls in der Definition von der Stutter Verfeinerungen nach [BK08] angewandt. Im Unterschied zu dieser Definition müssen wir das zeitliche Verhalten berücksichtigen.

Das extern sichtbare Zeitverhalten ist nicht nur von den Transitionen beeinflusst, die extern sichtbar sind (also die Transitionen eines Timed Trance), sondern auch von den anderen (internen) Transitionen, die z.B. ein Clock Reset ausführen können. Dies ist z. B. der Fall bei dem Übergang des Zustands Idle nach ComputeParam des Kommunikationsverhaltens der PosCalc-Rolle

(siehe Abbildung 3.1). 2. der Definition 17 stellt sicher, dass die internen Aktionen, die das zeitliche Verhalten beeinflussen, ebenfalls durch eine Transition eines Timed Trace berücksichtigt werden.

Wie einleitend beschrieben, soll die Definition der Verfeinerung direkt über Clock Zones erfolgen, da hierdurch direkt der Zusammenhang mit der Implementierung der Verfeinerung ermöglicht wird. Der in Definition 15 beschriebene Timed Trace angelehnt an der Definition von Yi und Jonsson [YJ94] beschreibt, wie für die Verfeinerung benötigt, das extern sichtbare Verhalten. Clock Zones werden in dieser Verfeinerung jedoch noch nicht berücksichtigt.

Nachdem die Zustände und Transitionen eines Timed Trace definiert wurden, stellen wir im Folgenden eine Definition von Timed Traces vor, die auf diesen Definitionen basierend ebenfalls Clock Zones berücksichtigen [HHH10]. In der Literatur sind zwei Definitionen für Clock Zones weit verbreitet, die von Bengtson und Yi [BY03] und Alur [Alu99]. Wir werden im Folgenden aus beiden Definition die Elemente verwenden, die eine für Timed Traces einfache Berechnung der Clock Zones ermöglichen. Wir werden uns daher für die Berechnung der Zones an Bengtson und Yi orientieren und für Clock Resets an der Definition von Alur.

Definition 18 (Timed Trace über Clock Zones)

Sei M ein Timed Automaton mit extern sichtbaren Ereignissen (Nachrichten) $A = A_i \cup A_o$ mit A_i empfangene Nachrichten und A_o gesendete Nachrichten, $A \subseteq \Sigma$ und \mathcal{Z} eine Menge von Clock Zones über die Clocks C des Automaten. Ein Timed Trace $\xi = (S_\xi, R_\xi)$ ist ein Ausführungspfad von M mit Zuständen S_ξ und Transitionen R_ξ für den gilt:

$$\xi = \langle s_0, z_0 \rangle \Rightarrow_{\delta_0} \langle s_0, z_0^\uparrow \rangle \Rightarrow_{a_0} \langle s_1, z_1 \rangle \Rightarrow_{\delta_1} \langle s_1, z_1^\uparrow \rangle \dots$$

mit

1. $z^\uparrow = \{z + d \mid z \in \mathcal{Z}, d \in \mathbb{R}_+\}$
2. $\langle s_0, z_0 \rangle \Rightarrow_{\delta_0} \langle s_0, z_0^\uparrow \rangle$ entspricht $\langle s_0, z_0 \rangle \Rightarrow \langle s_0, z_0^\uparrow \wedge I(s_0) \rangle$ mit $I(s_0)$ Invariante von Zustand s_0 .
3. $\langle s_0, z_0 \rangle \Rightarrow_{a_0} \langle s_1, z_1 \rangle$ entspricht $\langle s_0, z_0 \rangle \Rightarrow \langle s_1, ((z_0 \wedge g)[\lambda := 0]) \wedge I(s_1) \rangle$ mit $I(s_1)$ Invariante von s_1 , g Time Guard der Transition und λ eine Menge von Clocks $\lambda \subseteq C$, die auf 0 zurückgesetzt werden.
4. $\forall s \in S_\xi : s.z$ ist nicht leer

Wie in 17 definiert, können die Transitionen eines Timed Trace über Clock Zones entweder Zeit vergehen lassen (Delay Transition) oder ein Ereignis empfangen bzw. verschicken. Die Berechnung der Clock Zones gibt dabei an wann ein Zustand verlassen und betreten werden darf.

Im Fall einer Delay Transition muss für die Berechnung der Clock Zone entsprechend die Invariante (siehe Definition 1) des Zustands berücksichtigt werden. Eine Invariante eines Zustands gibt dabei für eine Clock eine obere Schranke durch eine Konstante oder eine andere Clock an (siehe Definition 3 zu Clock Constraints). Die resultierende Clock Zone nach Anwendung der Invarianten wird berechnet, in dem die Clock Zone des aktuellen Zustands mit den Invarianten geschnitten wird (siehe Punkt 2 Definition 18).

Eine Transition die ein Ereignis empfängt oder versendet ist durch mehrere interne Operationen beschrieben, wie dies der Fall beim Schalten einer Transition eines Timed Automata nach Definition 1 der Fall ist. Zum Berechnen der Clock Zone müssen folglich die Time Guards der Transition (welche Schranken gelten für die Transition), die Clock Resets (welche Clocks werden auf 0 zurück gesetzt) sowie die Invarianten des Zielzustands (welche Schranken gelten für das Betreten des Zustands) berücksichtigt werden. Punkt 3 der Definition eines Timed Trace über Clock Zones beschreibt dies, in dem zuerst die Clock Zone des aktuellen Zustands mit dem Time Guard g geschnitten wird, anschließend die Clock Resets λ angewandt werden und dann die Invarianten des Zielzustands ebenfalls mit der aktuellen Clock Zone geschnitten werden.

Basierend auf der Definition der Timed Traces kann im Folgenden die Verfeinerung eines einzelnen Traces definiert werden, die Grundlage für die Verfeinerung von zwei Timed Automata sein wird. In der Definition setzen wir voraus, dass die Namen der Nachrichten sowie Clocks eines abstrakten Automaten a und eines verfeinerten Automaten k gleich benannt sind.

Definition 19 (Verfeinerter Trace)

Seien $\xi_a = \langle S_{\xi,a}, R_{\xi,a} \rangle, \xi_k = \langle S_{\xi,k}, R_{\xi,k} \rangle$ Timed Traces über Clock Zones (siehe Definition 18) mit Startzuständen $l_{a,0}, l_{k,0}$ für Timed Automata M_a, M_k . Sei $\Omega \subseteq S_a \times S_k$ eine Abstraktionsfunktion, die eine Location aus M_a mit einer Location aus M_k assoziiert. Sei weiterhin $D(s, c)$ eine Relation, die zu einer Zone s und einer Clock c alle Clock Zones seit der letzten Nachricht vor Zone s liefert, in denen die Clock c zurückgesetzt wurde. ξ_k ist ein verfeinerter Trace zu ξ_a , $\xi_k \leq \xi_a$, falls:

1. $(l_{a,0}.s, l_{k,0}.s) \in \Omega$ und in $l_{a,0}.z, l_{k,0}.z$ sind alle Clocks 0
2. Für jede Transition $t_i \in R_{\xi,a}$ mit $s_a \Rightarrow_{a_o} s'_a$ und Nachricht $a_o \in A_o$ existiert eine Transition $t_j \in R_{\xi,k}$ mit $s_k \Rightarrow_{a_o} s'_k$, wobei $(s_a.s, s_k.s) \in \Omega$, für die gilt
 - $(s'_a.s, s'_k.s) \in \Omega$
 - Für alle Clocks c in $s'_a.z$: $\sum_{\{z|z \in D(s'_a,c)\}} \text{ubound}(z.c) + \text{ubound}(s'_a.z.c) = \sum_{\{z|z \in D(s'_k,c)\}} \text{ubound}(z.c) + \text{ubound}(s'_k.z.c)$
3. Für jede Transition $t_i \in R_{\xi,a}$ mit $s_a \Rightarrow_{a_i} s'_a$ mit $a_i \in A_i$ existiert eine Transition $t_j \in R_{\xi,k}$ mit $s_k \Rightarrow_{a_i} s'_k$, wobei $(s_a.s, s_k.s) \in \Omega$, für die gilt
 - $(s'_a.s, s'_k.s) \in \Omega$
 - Für alle Clocks c in $s'_a.z$: $\sum_{\{z|z \in D(s'_a,c)\}} \text{ubound}(z) + \text{ubound}(s'_a.z) \leq \sum_{\{z|z \in D(s'_k,c)\}} \text{ubound}(z) + \text{ubound}(s'_k.z)$
4. Alle externen Ereignisse (Nachrichten) sind in ξ_a und ξ_k über den gleichen Namensraum definiert.

Die Verfeinerungsdefinition ist so aufgebaut, dass beginnend mit der Startlocation (Bedingung 1.) zu jeder Location des abstrakten Timed Trace eine korrespondierende Location im verfeinerten Timed Trace zugeordnet wird (Bedingung 2. und 3). Bedingung 2. und 3. fordern dies entsprechend für die ein- und ausgehenden Nachrichten. Durch die Bedingung, dass die jeweils

vorherige Location Teil der Funktion Ω ist, die die Location des abstrakten Trace mit dem verfeinerten in Beziehung setzt, wird sukzessive die Relation zwischen den beiden Traces aufgebaut. Voraussetzung für diese Zuordnung ist, dass die Transitionen die gleichen Nachrichten verarbeiten.

Wie zu den Transitionen eines Timed Traces erläutert (siehe Definition 17) ist zwar für das extern sichtbare Verhalten die Nachrichtenkommunikation der Timed Traces relevant, es müssen jedoch auch die internen Transitionen berücksichtigt werden, um die Zeitintervalle mit richtiger oberer Schranke zwischen zwei Zuständen eines Timed Trace zu bestimmen, die durch Clock Resets an internen Transitionen beeinflusst werden können. $ubound$ in Bedingung 2. und 3. liefert die oberen Schranken der Clock Zones zwischen den Zuständen eines Timed Trace zurück, falls an den internen Transitionen eine Clock Reset Operation durchgeführt wurde. Ein Timed Trace ist dann eine gültige Verfeinerung, wenn die oberen Schranken des Sendeintervalls im Bezug zu dem abstrakten Timed Trace gleich bleiben (Bedingung 2.) und das Empfangsintervall mindestens die gleiche obere Schranke besitzt (Bedingung 3.).

Für die Verfeinerung wird das Verhalten des Kommunikationspartners nicht explizit betrachtet. Bedingung 2. über Clocks beim Versenden ist daher notwendig, da eine kleinere oder größere obere Schranke dazu führen kann, dass der Kommunikationspartner die Nachricht(en) nicht mehr rechtzeitig empfangen kann. Grund hierfür ist, dass der Kommunikationspartner potentiell die obere Schranke beim Empfangen von Nachrichten des abstrakten Timed Trace berücksichtigt.

Zur Veranschaulichung betrachten wir folgende Timed Traces unserer Beispielanwendung aus Abschnitt Beispielanwendung auf Seite 65:

$$\xi_{Coordinator}^1 = \langle Idle, c1 \leq 200 \rangle \dots \Rightarrow_{/update(para)} \langle AwaitAck, c1 \leq 25 \rangle \dots$$

und

$$\xi_{PosCalc}^1 = \langle Idle, c1 \leq 200 \rangle \dots \Rightarrow_{/update(CoParam)} \langle AwaitAck, c1 \leq 20 \rangle \dots$$

In Ω sind enthalten $(Idle, Idle)$ und $(AwaitAck, AwaitAck)$. $\sum_{\{z|z \in D(AwaitAck, C1)_{Coordinator}\}} ubound(z.c)$ ergibt 200, da ein Clock Reset zwischen $Idle$ und $SendUpdate$ durchgeführt wird. $\sum_{\{z|z \in D(AwaitAck, C1)_{PosCalc}\}} ubound(z.c)$ ergibt 205, die sich aus den Clock Resets zwischen $Idle$ und $ComputeParam$ und $ComputeParam$ und $AwaitAck$ ergeben. Insgesamt ergibt sich damit für $\xi_{Coordinator}^1$ eine Summe von $\sum_{\{z|z \in D(AwaitAck, C1)_{Coordinator}\}} ubound(z.c) + ubound(AwaitAck, C1) = 200 + 25 = 225$. Für $\xi_{PostCalc}^1$ ergibt sich eine Summe von $\sum_{\{z|z \in D(AwaitAck, C1)_{PostCalc}\}} ubound(z.c) + ubound(AwaitAck, C1) = 205 + 20 = 225$, womit $\xi_{PostCalc}^1$ eine Verfeinerung von $\xi_{Coordinator}^1$ ist.

Beim Empfangen von Nachrichten ist eine Verkleinerung der oberen Grenze ausgeschlossen, da der Kommunikationspartner, wie beim Versenden, potentiell das gesamte Intervall des abstrakten Timed Trace ausnutzen kann. Es muss also auch beim Empfangen gefordert werden, dass die obere Schranke eingehalten wird, wie Bedingung 3. fordert. Zusätzlich wird das Empfangsintervall relaxiert, im dem die obere Schranke durch den verfeinerten Trace überschritten werden darf. Voraussetzung für die Gültigkeit der Relaxierung ist eine asynchrone Echtzeitkommunikation wie in Abschnitt Anforderungen und Voraussetzungen auf Seite 67 beschrieben. Unter der

einfach ableiten, in dem die Obergrenze des Empfangsintervalls des abstrakten Verhaltens nicht nur verpflichtend ist, sondern auch nicht überschritten werden darf.

Im Folgenden werden wir auf der Grundlage der Verfeinerung eines Timed Traces eine Verfeinerung für zwei Timed Automata definieren. Gerade durch Wiederverwendung bereits vorhandener Protokolle, kann es grundsätzlich möglich sein, dass eine Verfeinerung auch mehr externes Verhalten anbietet als ein abstraktes Verhalten. Daher definieren wir vorab einen Schnittstellen-beschränkten Automaten basierend auf [Gie03], den wir in der Verfeinerung für zwei Timed Automata berücksichtigen.

Definition 20 (Schnittstellen-beschränkter Automat)

Sei $M_k = (S_k, S_k^0, T_k, Inv_k, A_k, C_k)$ ein Timed Automaton mit externen Ereignissen (Nachrichten) $A_k = A_{i,k} \cup A_{o,k}$ mit $A_{i,k}$ empfangene Nachrichten und $A_{o,k}$ gesendete Nachrichten. Sei M_a ein Timed Automaton, der eine abstrakte Schnittstelle über Nachrichten $A = A_i \cup A_o$ mit A_i empfangene Nachrichten und A_o gesendete Nachrichten repräsentiert. Falls gilt $A_i \subseteq A_{i,k}$ und $A_o \subseteq A_{o,k}$, dann kann ein Schnittstellen-beschränkter Automat $Int(M_k) = (S_{int}, S_{int}^0, T_{int}, Inv_{int}, A_{int}, C_{int})$ zu M_k gebildet werden mit:

- $S_{int} = S_k$
- $S_{int}^0 = S_k^0$
- $A_{int} = A$
- $Inv_{int} = Inv_k$
- $C_{int} = C_k$
- $T_{int} = T_k \setminus \{(l, a, g, r, l') \mid a \in A_k \setminus A\}$

Ein Schnittstellen-beschränkter Automat ist demnach dadurch charakterisiert, dass die Transitionen eines Automaten entfernt werden, die Nachrichten anbieten, die nicht Teil der Schnittstelle sind.

Als Hilfsmittel definieren wir im Folgenden zudem das extern sichtbare Verhalten über Timed Traces.

Definition 21 (Extern sichtbares Verhalten)

Das extern sichtbare Verhalten eines Timed Automaton M entspricht der Menge seiner Timed Traces $Trace(M)$.

Die Verfeinerung für zwei Timed Automata sei damit wie folgt definiert:

Definition 22 (Verfeinerung)

Seien M_a, M_k Timed Automata mit externem Verhalten $Trace(M_a)$ bzw. $Trace(M_k)$. Sei M_{int} ein Schnittstellen-beschränkter Automat zu M_k . M_k ist eine Verfeinerung von M_a , $M_k \leq M_a$, falls

1. für jeden Trace $\xi_k \in Trace(M_{int})$ ein Trace $\xi_a \in Trace(M_a)$ mit $\xi_k \leq \xi_a$ existiert und

2. $\neg \exists \xi_k \in \text{Trace}(M_{int}) : \text{succ}(s) = \emptyset$ für einen Zustand $s \in \xi_k$ mit Nachfolgezustand $\text{succ}(s)$ und
3. jeder Trace aus $\text{Trace}(M_a)$ überdeckt wurde. Eine Menge von Timed Traces (siehe Definition 18) ist überdeckt, wenn für jeden Trace zwischen je zwei Nachrichten ein Zustand in der Menge der korrespondierenden Zustände enthalten ist.

Die Verfeinerung soll nach Abschnitt Anforderungen und Voraussetzungen auf Seite 71 folgende Anforderungen erfüllen: 1) Erhalt des extern sichtbaren Protokollverhalten, 2) Erhalt der Verifikationsergebnisse und 3) möglichst viele Konkretisierungen zu lassen.

Anforderung 1) wird erfüllt, da Bedingung 3. fordert, dass alle Timed Traces des abstrakten Protokolls durch die Verfeinerung ebenfalls angeboten werden. Durch Bedingung 1. wird zudem sichergestellt, dass die Schnittstellen-beschränkte Verfeinerung auch nicht mehr Timed Traces anbietet als dies der Fall für das abstrakte Verhalten ist. Eine Ordnung über die Nachrichten wird zudem zugesichert.

Lemma 1

Gegeben seien zwei beliebige Timed Traces ξ_a mit Nachrichtenordnung a_1, a_2, \dots, a_n und Timed Trace ξ_k mit $\xi_k \leq \xi_a$, dann gilt, dass ξ_k ebenfalls die Nachrichtenordnung a_1, a_2, \dots, a_n einhält.

Beweis 1

Die sukzessive Konstruktion einer Verfeinerung zweier Timed Traces nach Definition 19 garantiert, dass in Ω nur korrespondierende Zustandspaare aufgenommen werden, die über die gleichen Nachrichtenfolgen erreicht werden. \square

Zusammen durch Bedingung 1. und 3. wird eine Bisimulation durch Nutzung einer Weak Transition Relation definiert [HH11]. Durch eine Weak Transition Relation wird sich auf Transitionsfolgen bezogen, die im Bezug auf das extern sichtbare Verhalten äquivalent sind. Durch eine Bisimulation wird zum einen verboten, dass eine Verfeinerung zusätzliches Verhalten gegenüber der Abstraktion anbietet und zum anderen, dass sämtliches mögliches Verhalten der Abstraktion durch die Verfeinerung unterstützt wird. Hierdurch bleiben CTL Formeln erhalten und die für einen kompositionalen Ansatz erforderlichen ACTL Formeln [CGP00]. Eine Timed Bisimulation wird für die spätesten Zeitpunkte zu denen eine Nachricht verschickt definiert und damit bleiben hierfür TCTL Formeln erhalten [TY01] und entsprechend auch ACCTL Formeln [Gie03]. Für zu empfangende Nachrichten können im Allgemein Zeit-Formeln nicht erhalten bleiben, da die Schranke hierfür nach oben erhöht werden kann. Um Time Stopping Deadlocks auszuschließen, wird Bedingung 2. gefordert.

Anforderung 3) wird durch die Definition einer relaxierten Bisimulation adressiert. Hierdurch wird im Vergleich zu den bisherigen Timed (Bi-) Simulationen eine Verschiebung des Empfangsintervalls erlaubt, ohne dabei den Erhalt des externen Protokollverhaltens zu verletzen.

3.1.2 Timed Story Charts

Im vorherigen Abschnitt wurde eine Verfeinerung für Protokollverhalten basierend auf REAL-TIME STATECHARTS vorgestellt, mit denen Ports der Multiplizität eins betrachtet werden können. Um Multielemente zu betrachten, die eine dynamische Änderung der Kommunikationsstruktur ermöglichen, müssen wir neben dem Echtzeitverhalten auch mögliche Strukturanpassungen in einer Verfeinerung berücksichtigen. Dies ist notwendig, da eine korrekte Verfeinerung des Statechartverhaltens nicht ausreichend ist, wenn die (kompositionale) Strukturanpassung der Kommunikationsstruktur in der Verfeinerung zu spät ausgeführt wird.

Wir werden in diesem Abschnitt die Verfeinerung für Einfachelemente auf Multielemente erweitern. Technisch soll die Verfeinerung von REAL-TIME STATECHARTS auf TIMED STORY CHARTS (siehe Abschnitt 2.6.4) übertragen werden, die für eine dynamische Änderung der Kommunikationsstruktur ausgelegt sind.

Um eine Verfeinerung für TIMED STORY CHARTS zu beschreiben definieren wir Analog zu der Verfeinerung für REAL-TIME STATECHARTS Timed Traces über Clock Zones für TIMED STORY CHARTS. Die Semantik der TIMED STORY CHARTS wird über zeitbehaftete Graphtransformationssysteme nach Hirsch [Hir08] beschrieben. Im Folgenden erweitern wir die Definition von Hirsch für zeitbehaftete Graphtransformationssysteme um die explizite Betrachtung von Clock Zones. Zuerst beginnen wir mit der Definition eines zeitbehafteten Graphen.

Definition 23 (Zeitbehafteter Graph)

Ein zeitbehafteter Graph $G_t := (G, C, \mathcal{Z})$ ist ein Tripel bestehend aus einem Objektgraphen G , einer Anzahl von Clock-Instanzen C und einer Menge von Clock Zones \mathcal{Z} über die Elemente aus C .

Die Definition berücksichtigt zum einen direkt Clock Zones und zum anderen wie in Definition 10 beschrieben eine Typisierung der Objektgraphen über ein Klassendiagramm. Die Clock Zones beschreiben über Clock-Instanzen Bedingungen über die Clocks, wie dies Analog für Zustände eines Timed Transition Systems der Fall ist. Ein zeitbehaftetes Graphtransformationssystem lässt sich damit wie folgt definieren.

Definition 24 (Zeitbehaftetes Graphtransformationssystem)

Ein zeitbehaftetes Graphtransformationssystem $\mathcal{G}_t = (G_t, G^0, TR, IR)$ besteht aus einer Menge an zeitbehafteten Graphen G_t , einem Startgraphen G^0 , einer Menge von Schaltregeln TR und einer Menge von Invariantenregeln IR . Die Menge $GRAPH_{G_t}$ beschreibt die Menge aller zulässiger zeitbehafteter Graphen.

Für die hier betrachteten TIMED STORY CHARTS wird als Typgraph das Metamodell der Komponenten aus Abbildung 2.20 sowie dessen Statechart Metamodell aus Abbildung 2.27 verwendet. Wie in Abschnitt 2.6.4 beschrieben, werden PARAMETERIZED REAL-TIME STATECHARTS auf TIMED STORY CHARTS abgebildet. Diese Abbildung beschreibt unter anderem, wie Nachrichten und Clocks auf Objekte der Statechart Metamodell Klasse abgebildet werden. Hierüber

wird es ermöglicht im Folgenden analog zu der Verfeinerungsdefinition für REAL-TIME STATE-CHARTS ebenfalls über Nachrichten und Clocks zu argumentieren. Damit ist die Grundlage geschaffen, um zunächst einen Zustand eines Timed Trace für TIMED STORY CHARTS zu definieren.

Definition 25 (Zustand eines Timed Trace für TIMED STORY CHARTS)

Sei $\mathcal{G}_t = \langle G_t, G^0, TR, IR \rangle$ ein zeitbehaftetes Graphtransformationssystem gemäß Definition 24. Ein Zustand S eines Timed Trace zu \mathcal{G}_t ist eine Zone $\langle g, z \rangle$ mit $g \in GRAPH_{G_t}$ und z die dazugehörige Clock Zone. Es bezeichnet $S.g$ den Objektgraphen und $S.z$ die Clock Zone von S . Es bezeichnet weiterhin $z.c$ die Menge der Clock Constraints über die Clock c in z .

Nach Definition 16 werden die Zustände eines Timed Traces ξ eines Timed Automaton M über eine Zone $\langle s, z \rangle$ beschrieben, wobei s eine Location aus M ist und z eine Clock Zone. Im Vergleich hierzu ist ein Zustand eines Timed Trace für TIMED STORY CHARTS über einen Objektgraphen $g \in GRAPH_{G_t}$ definiert. Da sich die Definition einer Transition hierdurch nicht ändert, wird im Folgenden für Transitionen eines Timed Traces Definition 17 benutzt. Weiterhin bezeichne im Folgenden $A = A_i \cup A_o$, mit A_i empfangene Nachrichten und A_o gesendete Nachrichten, die extern sichtbaren Nachrichten eines TIMED STORY CHARTS, die über das Metamodell aus Abbildung 2.27 definiert wurden. Ein Timed Trace für TIMED STORY CHARTS ist damit wie folgt definiert.

Definition 26 (Timed Trace eines Timed Story Charts)

Sei $\mathcal{G}_t = \langle G_t, G^0, TR, IR \rangle$ ein zeitbehaftetes Graphtransformationssystem gemäß Definition 24 mit extern sichtbaren Ereignissen (Nachrichten) $A = A_i \cup A_o$ des TIMED STORY CHARTS mit A_i empfangene Nachrichten und A_o gesendete Nachrichten und \mathcal{Z} eine Menge von Clock Zones über Clock-Instanzen C . Ein Timed Trace $\xi = (S_\xi, R_\xi)$ ist eine Folge von Regelanwendungen aus TR mit Zuständen S_ξ und Transitionen R_ξ für den gilt:

$$\xi = \langle g_0, z_0 \rangle \Rightarrow_{\delta_0} \langle g_0, z_0^\uparrow \rangle \Rightarrow_{a_0} \langle g_1, z_1 \rangle \Rightarrow_{\delta_1} \langle g_1, z_1^\uparrow \rangle \dots$$

mit

- $z^\uparrow = \{z + d \mid z \in \mathcal{Z}, d \in \mathbb{R}_+\}$
- $\langle g_0, z_0 \rangle \Rightarrow_{\delta_0} \langle g_0, z_0^\uparrow \rangle$ entspricht $\langle g_0, z_0 \rangle \Rightarrow \langle g_0, z_0^\uparrow \wedge I(g_0) \rangle$ mit $I(g_0)$ ist eine auf g_0 anwendbare Invariante
- $\langle g_0, z_0 \rangle \Rightarrow_{a_0} \langle g_1, z_1 \rangle$ entspricht $\langle g_0, z_0 \rangle \Rightarrow \langle g_1, ((z_0 \wedge g)[\lambda := 0]) \wedge I(g_1) \rangle$ mit $I(g_1)$ ist eine auf g_1 anwendbare Invariante, g ist ein Time Guard der Transition und λ eine Menge von Clock-Instanzen $\lambda \subseteq C$, die auf 0 zurückgesetzt werden.
- $\forall s \in S_\xi : s.z$ ist nicht leer

Nach Definition 18 ist ein Timed Trace $\xi = (S_\xi, R_\xi)$ eines Timed Automaton M definiert als ein Ausführungspfad von M mit Zuständen S_ξ und Transitionen R_ξ . Im Unterschied zu dieser Definition wird ein Timed Trace eines TIMED STORY CHARTS über eine Folge von Regelanwendungen aus TR definiert, die das Schalten einer Transition und das Vergehen von Zeit beschreiben. Zudem wird eine Invariante im Vergleich zu der Definition für Timed Automata in

Form eines Graphen definiert (siehe Abschnitt A.1.7) und Clock Resets werden über Objekte vom Typ ClockReset definiert (siehe Abschnitt A.1.9). Das extern sichtbare Verhalten eines TIMED STORY CHARTS können wir damit wie folgt beschreiben.

Definition 27 (Extern sichtbare Verhalten)

Das extern sichtbare Verhalten eines zeitbehafteten Graphtransformationssystems \mathcal{G} , dessen Transformationsregeln über ein TIMED STORY CHART beschrieben sind, entspricht der Menge seiner Timed Traces $\text{Trace}(\mathcal{G})$.

In der Definition werden explizit die Transformationsregeln eines TIMED STORY CHARTS berücksichtigt. Da PARAMETERIZED REAL-TIME STATECHARTS auf TIMED STORY CHARTS abgebildet werden können, ist hierdurch direkt ein Bezug zu der Protokollbeschreibung von Multielementen gegeben. Ein verfeinerter TIMED STORY CHART Trace lässt sich damit wie folgt definieren.

Definition 28 (Verfeinerter Trace)

Seien $\xi_a = \langle S_{\xi,a}, R_{\xi,a} \rangle$, $\xi_k = \langle S_{\xi,k}, R_{\xi,k} \rangle$ Timed Traces für zeitbehaftete Graphtransformationssysteme $\mathcal{G}_t^a = \langle G_t^a, C_a^0, TR_a, IR_a \rangle$ und $\mathcal{G}_t^k = \langle G_t^k, C_k^0, TR_k, IR_k \rangle$. Sei $\text{abs} : \mathcal{G}_t^k \rightarrow \mathcal{G}_t^a$ eine Abstraktionsfunktion, die Objekte aus \mathcal{G}_t^k mit Objekten aus \mathcal{G}_t^a assoziiert. Sei weiterhin $D_{\text{reset}}(s, c)$ eine Relation, die zu einer Zone s und einer Clock c alle Clock Zones seit der letzten Nachricht vor Zone s liefert, in denen die Clock c zurückgesetzt wurde. ξ_k ist ein verfeinerter Trace zu ξ_a , $\xi_k \leq \xi_a$, falls:

1. $G_a^0 \subseteq \text{abs}(G_k^0)$ und in $s_{a,0}.z, s_{k,0}.z$ sind alle Clocks 0
2. Für jede Transition $t_i \in R_{\xi,a}$ mit $s_a \Rightarrow_{a_o} s'_a$ mit Nachricht $a_o \in A_o$ existiert eine Transition $t_j \in R_{\xi,k}$ mit $s_k \Rightarrow_{a_o} s'_k$, wobei $s_a.g \subseteq \text{abs}(s_k.g)$, für die gilt
 - $s'_a.g \subseteq \text{abs}(s'_k.g)$
 - Für alle Clocks c in $s'_a.z$: $\sum_{\{z|z \in D(s'_a,c)\}} \text{ubound}(z.c) + \text{ubound}(s'_a.z.c) = \sum_{\{z|z \in D(s'_k,c)\}} \text{ubound}(z.c) + \text{ubound}(s'_k.z.c)$
3. Für jede Transition $t_i \in R_{\xi,a}$ mit $s_a \Rightarrow_{a_i} s'_a$ mit $a_i \in A_i$ existiert eine Transition $t_j \in R_{\xi,k}$ mit $s_k \Rightarrow_{a_i} s'_k$, wobei $s_a.g \subseteq \text{abs}(s_k.g)$, für die gilt
 - $s'_a.g \subseteq \text{abs}(s'_k.g)$
 - Für alle Clocks c in $s'_a.z$: $\sum_{\{z|z \in D(s'_a,c)\}} \text{ubound}(z) + \text{ubound}(s'_a.z) \leq \sum_{\{z|z \in D(s'_k,c)\}} \text{ubound}(z) + \text{ubound}(s'_k.z)$
4. Alle externen Ereignisse (Nachrichten) sind in ξ_a und ξ_k über den gleichen Namensraum definiert.

Im Unterschied zu Definition 19 (verfeinerter Trace eines Timed Automaton) wird hier basierend auf Definition 25 die Korrespondenz zwischen zwei Zuständen über eine Abstraktionsfunktion abs definiert, die die Objekte aus \mathcal{G}_t^k mit denen der Objekte aus \mathcal{G}_t^a verbindet. Über die Teilmengenrelation wird sichergestellt, dass alle strukturellen Elemente, wie Ports oder auch PARAMETERIZED REAL-TIME STATECHARTS Instanzen, sowohl im abstrakten wie auch konkreten

Verhalten existieren. Eine strukturelle Verfeinerung über eine Teilmengenrelation zu beschreiben wurde bereits durch Heckel und Thöne in [HT04, HT05] vorgestellt. Die dort definierte Verfeinerung basiert jedoch rein auf der Struktur und der Abfolge der erzeugten Graphen, ohne Betrachtung von (Echtzeit-) Verhalten oder den Erhalt von Verifikationsergebnissen.

In der MECHATRONIC UML wird eine Beziehung zwischen den strukturellen Elementen durch eine Delegation, die Ports der äußeren Komponente mit den Ports der eingebetteten Parts verbindet, beschrieben. Da die Rollen der Muster ebenfalls eindeutig in Beziehung mit deren Anwendung durch Komponenten-Ports stehen, wird hier ebenfalls eine Korrespondenz zwischen den Strukturen beschrieben. Eine gültige Strukturverfeinerung liegt dann vor, wenn zu jedem Port eines abstrakten Protokolls ein verfeinerter Port existiert. Ein Beispiel zeigt Abbildung 3.7. Nach der gegebenen Definition entspricht damit Instanzsituation (a) einer gültigen Verfeinerung und (b) nicht.

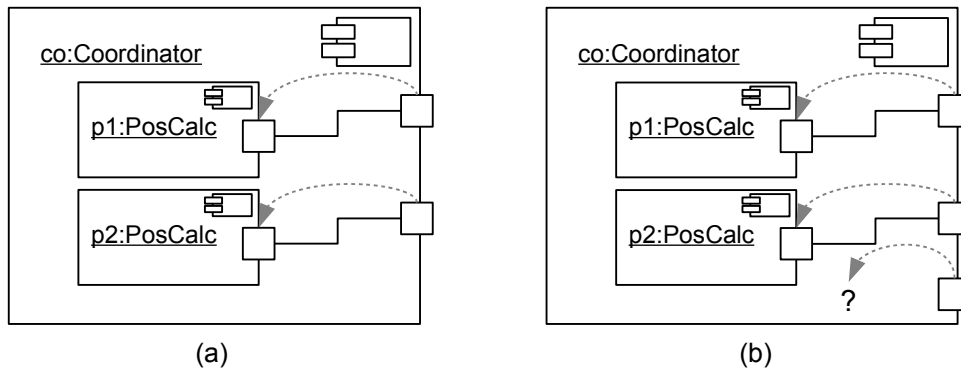


Abbildung 3.7: Beispiel für eine Strukturverfeinerung: (a) zeigt eine gültige Strukturverfeinerung, (b) eine ungültige

Analog zu der Verfeinerung für REAL-TIME STATECHARTS definieren wir im Folgenden basierend auf der Definition 28 eines verfeinerten Timed Trace eine Verfeinerung für TIMED STORY CHARTS, die entsprechend über alle Timed Traces argumentiert.

Definition 29 (Verfeinerung)

Seien $\mathcal{G}_t^a, \mathcal{G}_t^k$ zeitbehaftete Graphtransformationssysteme mit externem Verhalten $Trace(\mathcal{G}_t^a)$ bzw. $Trace(\mathcal{G}_t^k)$. \mathcal{G}_t^k ist eine Verfeinerung von \mathcal{G}_t^a , $\mathcal{G}_t^k \leq \mathcal{G}_t^a$, falls

1. für jeden Trace $\xi_k \in Trace(\mathcal{G}_t^k)$ ein Trace $\xi_a \in Trace(\mathcal{G}_t^a)$ mit $\xi_k \leq \xi_a$ existiert und
2. $\neg \exists \xi_k \in Trace(\mathcal{G}_t^k) : succ(s) = \emptyset$ für einen Zustand $s \in \xi_k$ mit Nachfolgezustand $succ(s)$ und
3. jeder Trace aus $Trace(\mathcal{G}_t^a)$ überdeckt wurde.

Verfeinerungsdefinition 29 ist aufgebaut wie die Verfeinerungsdefinition 22 für REAL-TIME STATECHARTS. Zusammen durch Bedingung 1 und 3 wird ebenfalls eine Bisimulation unter Berücksichtigung einer Weak Transition Relation definiert [HH11]. Ebenfalls analog zu der Verfei-

nerungsdefinition für REAL-TIME STATECHARTS kann durch Anwendung einer Schnittstellen-Beschränkung (siehe Definition 20) nicht ausgeschlossen werden, dass Deadlocks entstehen. Es muss entsprechend sichergestellt werden, dass jeder Zustand im Trace einen Nachfolgezustand hat. Bedingung 2 stellt dies sicher. Wie zu der Verfeinerungsdefinition für REAL-TIME STATECHARTS argumentiert, werden durch diese Verfeinerung ebenfalls die Anforderungen wie in Abschnitt Anforderungen und Voraussetzungen auf Seite 71 beschrieben erfüllt. Für Details sei der Leser auf die Diskussion zu Definition 22 auf Seite 79 verwiesen. Darüber hinaus wird durch Definition 28 eine strukturelle Verfeinerung beschrieben, womit zudem eine dynamische Änderung der Kommunikationsstruktur durch die Verfeinerung berücksichtigt wird.

3.1.3 Diskussion

Die in den vorherigen Abschnitten vorgestellte Verfeinerung für REAL-TIME STATECHARTS und TIMED STORY CHARTS erhält Verifikationsergebnisse des abstrakten Verhaltens sowie das extern sichtbare Echtzeitkommunikationsverhalten. Bisherige Ansätze unterstützen keine Strukturanpassungen (z. B. [JLS00]), keine explizite Zeitbetrachtung (z. B. [JLS00, GRPS02, HT05, Gie07]) oder ermöglichen nur eine statische Analyse (z. B. [GRPS02, Gie07]), die keine Relaxierung ermöglicht. Der vorgestellte Ansatz ermöglicht durch die explizite Betrachtung von Zeit eine Verfeinerung, in der Zeitintervalle relaxiert werden können. Da der TIMED STORY CHARTS Formalismus sowohl Zeit, wie auch Strukturanpassungen berücksichtigt, werden die geforderten Anforderungen selbstoptimierender, mechatronischer Systeme erfüllt. Das Connector-Verhalten wurde in den Definitionen nicht explizit berücksichtigt. Dies kann jedoch einfach parallel zu dem Rollenverhalten geschaltet werden, wodurch die Verfeinerungsdefinition nicht beeinflusst wird.

Wie in [ACH94, AD90] beschrieben, ist im allgemeinen Fall die Verifikation der Verfeinerung über eine Teilmenge der Traces für Timed Automata nicht entscheidbar. Nicht-Determinismus sowie eine potentiell unendliche Menge an Timed Traces ([AD90, ACH94, YJ94]) sind die Gründe hierfür. Durch die beschriebenen Voraussetzungen aus dem modellbasierten Ansatz der MECHATRONIC UML (vgl. Abschnitt Anforderungen und Voraussetzungen auf Seite 67) kann aber eine Entscheidbarkeit erreicht werden. Nach Voraussetzung gibt es zu einem Automaten (oder TIMED STORY CHART) mindestens eine gültige deterministische Verfeinerung oder das abstrakte Verhalten ist deterministisch. Weiterhin ist die Menge (Anzahl) der Timed Traces und Zustände endlich. Wie in [ACH94, AD90] beschrieben kann unter diesen Voraussetzungen die Entscheidbarkeit der Verfeinerung gefolgert werden.

In den bisherigen untersuchten Anwendungen des RailCabs (siehe auch [May09]), können die hier beschriebenen Verfeinerungskonzepte angewendet werden. Hieraus ist aber auch zu erkennen, dass es generell nicht die eine „ideale“ Verfeinerung für eine Anwendungsdomäne gibt. In der Masterarbeit von Christan Brenner [Bre10] wurde daher eine Generalisierung der hier vorgestellten Verfeinerungen zu einer parametrisierten Verfeinerung vorgeschlagen. Idealerweise kann hiermit werkzeuggestützt den konkreten Anforderungen entsprechend eine Verfeinerungsbeziehung vorgeschlagen und überprüft werden.

3.2 Verfeinerungsüberprüfung

Nachdem wir im vorherigen Abschnitt eine Verfeinerung für REAL-TIME STATECHARTS und TIMED STORY CHARTS definiert haben, werden wir in diesem Abschnitt eine Überprüfung der Verfeinerung vorstellen. Wie bereits in Abschnitt 3.1.3 diskutiert, wurde die Verfeinerung möglichst flexibel durch eine Relaxierung von Zeitintervallen ausgelegt, um eine große Anzahl an Wiederverwendungen zu ermöglichen. Damit ist allerdings auch keine statische Analyse möglich, wie dies z. B. in [GRPS02] vorgestellt wurde. Für unseren Ansatz müssen wir den möglichen Konfigurationsraum aufbauen. Dies wird klassisch durch eine Erreichbarkeitsanalyse ermöglicht (siehe z.B. [BK08]), die wir im Folgenden vorstellen. Anschließend werden in Abschnitt 3.2.2 eine Verifikation der Verfeinerung basierend auf der Erreichbarkeitsanalyse vorstellen. Im Folgenden werden wir die Verfeinerungsüberprüfung für TIMED STORY CHARTS zeigen. Diese lässt sich ebenfalls auf die der REAL-TIME STATECHARTS anwenden, da diese lediglich den Sonderfall einer eins zu eins Multiplizität zwischen den Strukturen (Ports und Statechart-Instanzen) darstellen.

3.2.1 Erreichbarkeitsanalyse

Um den möglichen Konfigurationsraum für TIMED STORY CHARTS zu berechnen, nutzen wir eine Erreichbarkeitsanalyse aus. Da unser Ansatz kompositionale Strukturanpassungen berücksichtigt, können wir nicht direkt auf klassische Ansätze für rein zeitbehaftetes Verhalten zurückgreifen, wie z.B. in [BY03] vorgestellt. In [Hir08] wurde eine Erreichbarkeitsanalyse für zeitbehaftete Graphtransformationssysteme (Timed Graph Transformation Systems - TGTS) vorgestellt, die wir für die Erreichbarkeitsanalyse von TIMED STORY CHARTS anwenden. Dies ist möglich, da wie in Abschnitt 2.6.4 und 3.1.2 beschrieben TIMED STORY CHARTS auf der Semantik von TGTS basieren.

Ausgangspunkt für die Erreichbarkeitsanalyse ist ein Timed Transition System, welches die erreichbaren Zustände unter Berücksichtigung der zeitlichen Bedingungen darstellt. Definition 4 beschreibt dieses für Timed Automata. Hirsch hat dieses Konzept in [Hir08] auf TGTS übertragen. Im Folgenden geben wir die notwendigen Definitionen für die Erreichbarkeitsanalyse basierend auf der von TGTS wieder. Im Unterschied zu der Verfeinerungsdefinition, in der die Betrachtung des extern sichtbaren Verhalten inhärent ist und dadurch über Timed Traces definiert wurde, bietet sich für die Verfeinerungsüberprüfung die Darstellung des erreichbaren Verhaltens über ein Timed Transition System an. Grund hierfür ist, dass im Vergleich zu Timed Traces isomorphe Zustände, womit bereits identifizierte Zustände identifiziert werden, nur einmal dargestellt werden. Im Folgenden definieren wir zuerst einen Zustand eines zeitbehafteten Transitionssystems.

Definition 30 (Zustand eines zeitbehafteten Transitionssystems)

Sei $\mathcal{G}_t = \langle G_t, G^0, TR, IR \rangle$ ein zeitbehaftetes Graphtransformationssystem gemäß Definition 24. Ein Zustand eines zeitbehafteten Transitionssystems zu \mathcal{G}_t ist ein Tupel $s = \langle g, z \rangle$ mit $g \in GRAPH_{TG}$ und einer Clock Zone z .

Um bereits erreichte Zustände zu identifizieren, werden für TGTS isomorphe Zustände definiert. Dies ist analog zu gleichen Zuständen eines Timed Transition Systems für Timed Automata der Fall, wenn die Graphen, die eine Konfiguration präsentieren, isomorph sind und sie die gleichen Clock Zones besitzen.

Definition 31 (Isomorphe Zustände)

Zwei Zustände $S_1 = \langle G_1, Z_1 \rangle, S_2 = \langle G_2, Z_2 \rangle$ eines zeitbehafteten Transitionssystems sind isomorph, gdw. $G_1 \equiv G_2$ und $Z_1 = Z_2$.

Ein erreichbares Timed Transition System für TGTS ist damit wie folgt definiert.

Definition 32 (Erreichbares zeitbehaftetes Transitionssystem)

Sei $\mathcal{G}_t = \langle TG, G^0, TR, IR \rangle$ ein zeitbehaftetes Graphtransformationssystem gemäß Definition 24 und \mathcal{Z} eine Menge von Clock Zones. Das erreichbare zeitbehaftete Transitionssystem ETTTS mit Startzustand s_0 zu \mathcal{G}_t ist ein 2-Tupel (V, E) mit

- $V = \{s' = \langle g, z \rangle \mid g \in GRAPH_{TG} \wedge s_0 \xrightarrow{*} s', z \in \mathcal{Z}\}$ ist eine Menge von Zuständen, die vom Startzustand aus erreichbar sind.
- $s_0 \in V = \langle G^0, z_0 \rangle$, wobei z_0 die Clock Zone ist, in der alle Clocks den Wert 0 haben.
- $E = \{(s_1, s_2) \mid s_1, s_2 \in V \wedge s_1 \xrightarrow{r} s_2 \wedge r \in TR\} \cup \{(s_1, s_2) \mid s_1, s_2 \in V \wedge s_1 \xrightarrow{\delta} s_2\}$ ist eine Menge von Transitionen.

Damit kann eine Erreichbarkeitsanalyse analog zu [Hir08] durchgeführt werden. Diese nutzt jedoch die Definition von Alur [Alu99] aus, um Folgezustände zu berechnen. Hiermit ist es nicht möglich die Reihenfolge in der Clock Zones erreicht werden bei gleichen Zuständen zu unterscheiden (diese werden in einem solchen Fall über Federations vereinigt). Zudem ist es hiermit auch nicht möglich explizit zwischen Delay- und Action-Transitionen zu differenzieren. Die Verfeinerungsdefinition 32 für TIMED STORY CHARTS setzt dies jedoch voraus. Da die Definition von Bengtsson und weitere [BY03] genau diese Unterscheidungen ermöglichen, nutzen wir diese im Unterschied zu [Hir08] in der Erreichbarkeitsanalyse für TIMED STORY CHARTS zur Berechnung von Folgezuständen (Konfigurationen) aus.

Da die Definition der Verfeinerung auf Timed Traces basiert müssen wir folglich noch zeigen, dass die Menge der Pfade eines Timed Transition Systems für TIMED STORY CHARTS der Menge seiner Timed Traces entspricht.

Theorem 1

Für ein TIMED STORY CHART gilt, dass die Menge der Pfade seines zeitbehafteten Transitionssystems genau der Menge der Timed Traces entspricht.

Beweis 2

Sei $\mathcal{G}_t = \langle G_t, G^0, TR, IR \rangle$ ein zeitbehaftetes Graphtransformationssystem gemäß Definition 24. Ein Timed Trace $\xi = (S_\xi, R_\xi)$ ist nach Definition 26 eine Folge von Regelanwendungen aus TR mit Zuständen S_ξ und Transitionen R_ξ . Nach Definitionen 25 und 30 sind die Zustände

von *Traces* und zeitbehafteten Transitionssystemen gleich definiert. Gegeben durch die Definition einer Transition eines zeitbehafteten Transitionssystems für TGTS mit $E = \{(s_1, s_2) \mid s_1, s_2 \in V \wedge s_1 \xrightarrow{r} s_2 \wedge r \in TR\} \cup \{(s_1, s_2) \mid s_1, s_2 \in V \wedge s_1 \xrightarrow{\delta} s_2\}$, werden alle möglichen Anwendungen von Transformationsregeln auf jeden Zustand berechnet. Damit werden durch ein solches Transitionssystem alle möglichen Ausführungspfade dargestellt. Ein Timed Trace beschreibt genau einen Ausführungspfad. Damit entspricht die Menge aller Timed Traces der Menge aller Ausführungspfade eines zeitbehafteten Transitionssystems. \square

Neben der oben dargestellten Anpassung der Erreichbarkeitsanalyse von Hirsch zur Berechnung eines Folgezustands (Folgekonfiguration) müssen wir zudem zwischen Graphtransformationen unterscheiden, 1) die eine Transition eines TIMED STORY CHARTS ausführen und den, 2) die Seiteneffekte, Actions oder Hilfsfunktionen ausführen. Hierdurch wird es ermöglicht, wie nach der Verfeinerungsdefinition für TIMED STORY CHARTS gefordert, dass zum einen durch 1) das Zustandsverhalten überprüft werden kann und dass die Elemente aus 2), die z.B. durch einen Seiteneffekt eine (kompositionale) Strukturanpassung ausführen können, nur aufgerufen werden, wenn diese auch tatsächlich in Folge einer Zustandstransformation aus 1) aufgerufen werden.

Die Folgezustandsberechnung von Hirsch muss zudem von der Berechnung einer Graphtransformationsregel auf mehrere erweitert werden, da TIMED STORY CHARTS durch (Timed) Story Diagramme beschrieben werden (siehe Abschnitt 2.6.4), die mehrere Graphtransformationen durch Stories nacheinander beschreiben können. Der Folgegraph ergibt sich damit aus einer Menge von Graphtransformationen.

3.2.2 Verifikation der Verfeinerung

In diesem Abschnitt stellen wir einen Algorithmus zur Verifikation der Verfeinerung von TIMED STORY CHARTS vor. Gemäß der Verfeinerungsdefinition 29 berechnen wir in einem ersten Schritt die Menge der Traces für das abstrakte ($Trace(\mathcal{G}_t^a)$) und verfeinerte Verhalten ($Trace(\mathcal{G}_t^k)$). Im nächsten Schritt wird überprüft, ob für jeden Trace $\xi_k \in Trace(\mathcal{G}_t^k)$ ein Trace $\xi_a \in Trace(\mathcal{G}_t^a)$ existiert, so dass ξ_k gemäß Definition 28 eine Verfeinerung von ξ_a ist. Dies entspricht Bedingung 1 nach Definition 29. Hierbei wird ebenfalls die Verfeinerung auf Deadlockfreiheit überprüft, wie dies durch Bedingung 2 gefordert wird. Als letztes wird überprüft, ob alle Pfade $Trace(\mathcal{G}_t^a)$ überdeckt wurden, womit Bedingung 3 adressiert wird.

Für die Verfeinerungsüberprüfung wird angenommen, dass die Verifikation des Protokollverhaltens erfolgreich war. Das Verhalten des Kommunikationspartners wird daher in der Verfeinerungsüberprüfung nicht betrachtet. Um Transitionen auszuführen, die durch eine Nachricht des Kommunikationspartners aktiviert werden, legen wir eine zusätzliche Transition an, die zu jedem Zeitpunkt schalten kann, an dem ihre Time Guards erfüllt sind. Weiterhin gehen wir davon, dass das verfeinerte TIMED STORY CHART nach Definition 20 Schnittstellen-beschränkt ist.

Algorithmus 3.1 zeigt den Pseudocode zum Überprüfen einer Verfeinerung nach Definition 29. Implementiert ist der Algorithmus in Form einer Tiefensuche. Wie einleitend erläutert startet der

Algorithmus mit der Berechnung der erreichbaren Traces für ein abstraktes und konkretes TIMED STORY CHART in Zeile 2 und 3 (siehe hierzu Abschnitt 3.2.1). Anschließend wird in Zeile 4 überprüft, ob eine Strukturverfeinerung für die initialen Strukturen gilt. Durch die Variable *success* wird der Zustand der Überprüfung in Form eines Boolean-Wertes (*true* oder *false*) implementiert. Die while-Schleife in Zeile 6, die die Tiefensuche zur Überprüfung der Verfeinerung umsetzt, terminiert entsprechend, wenn eine der notwendigen Überprüfungen für die Verfeinerung fehlschlägt, wodurch $success == false$ gilt oder alle Knoten des TIMED STORY CHART Graphen untersucht wurden. Wurden alle Knoten untersucht, schlägt die Überprüfung auf einen Nachfolger in Zeile 8 fehl. Innerhalb der while-Schleife wird das nach Abschnitt Anforderungen und Voraussetzungen auf Seite 67 endliche Transitionssystem des verfeinerten TIMED STORY CHARTS expandiert (Zeile 9). Da es sich um ein endliches Transitionssystem handelt, bricht die Schleife irgendwann ab. Für den Fall, dass alle Überprüfungen erfolgreich waren, wird als letzter Schritt überprüft, ob das abstrakte System überdeckt wurde.

In der while-Schleife wird für jeden Nachfolger n' des aktuell expandierten Knotens n geprüft, ob der Knoten bekannt ist. Ist dies nicht der Fall, wird überprüft, ob der Übergang zwischen diesen beiden Knoten mit einer Nachricht versehen ist, womit überprüft werden muss, ob es hierzu einen korrespondierenden Zustand in dem abstrakten System gibt (siehe Abschnitt 3.2.2.1). Andernfalls, wird ein Kreis im Transitionssystem geschlossen oder es werden zwei Pfade nach Definition 31 vereinigt, da die Zustände isomorph sind. Für den Fall das zwischen dem Übergang von n nach n' eine Nachricht annotiert ist, wird analog zu Fall 1 ein korrespondierender Pfad gesucht. Handelt es sich um einen Kreis, wird dieser auf Wohlgeformtheit überprüft (siehe Abschnitt 3.2.2.1).

3.2.2.1 Überprüfung von Pfaden

Kann eine Kante des verfeinerten Systems eine Nachricht empfangen oder versenden, so muss nach Definition der Verfeinerung ein korrespondierender Pfad im Abstrakten System existieren, der ebenfalls eine solche Nachricht empfangen oder versenden kann. Die Funktion CHECKPATH (siehe Algorithmus 3.2) überprüft ausgehend von einer Kante (n, n') des verfeinerten Transitionssystems, die eine Nachricht trägt, auf allen Pfaden beginnend in einem korrespondierend Zustand, die diese Kante enthalten, ob es hierzu einen korrespondierenden Pfad im abstrakten Transitionssystem gibt (siehe Zeile 10). Die Überprüfung schlägt fehl, wenn kein Pfad im abstrakten Transitionssystem gefunden wird.

Korrespondierende Zustände Für einen abstrakten und konkreten Pfad des Transitionssystems werden durch die Funktion FINDCORRESPONDINGSTATES die korrespondierenden Zustände ermittelt (siehe Algorithmus 3.3). Durch die Relaxierung der Zeitintervalle durch die in Abschnitt 3.1.2 eingeführte Verfeinerungsdefinition, müssen wir zwischen gesendeten und empfangenen Nachrichten unterscheiden (Zeile 2). Die Funktion CHECKTIMECONSTRAINTS (siehe Abschnitt Prüfen der zeitlichen Bedingungen auf Seite 91) überprüft für jeden Teilpfad s und t nach Definition 28 die oberen Schranken der Clocks. Für jeden dieser Teilpfade wird ebenfalls

Algorithmus 3.1 Überprüfen der korrekten Verfeinerung

```

1: function CHECKCORRECTREFINEMENT(TimedStoryChart abs, TimedStoryChart ref)
2:   absReach = STARTREACHABILITYANALYSIS(abs)
3:   refReach = STARTREACHABILITYANALYSIS(ref)
4:   success := CHECKSTRUCTUREREFINEMENT(absReach.initial, refReach.initial)
5:   OPEN.PUSH(refReach.initial)
6:   while OPEN  $\neq \emptyset \wedge$  success do                                ▷ Untersuche alle erreichbaren Knoten
7:     n := OPEN.POP()
8:     success := refReach.HASSUCCESSOR(n)
9:     for all  $n' \in$  refReach.EXPAND(n) do
10:      if  $n'$  is not known then                                       ▷ Fall 1: Neuer Knoten
11:        OPEN.PUSH( $n'$ )
12:        if ( $n, n'$ ) has event  $e$  then
13:          success := CHECKPATH( $(n, n')$ )
14:        end if
15:      else                                                             ▷ Fall 2: Knoten schon bekannt
16:        if ( $n, n'$ ) closed cycle then                                  ▷ a) Kante schließt einen Kreis
17:          if ( $n, n'$ ) has event  $e$  then
18:            success := CHECKPATH( $(n, n')$ )
19:          end if
20:          success := ISWELLFORMEDCYCLE( $n'$ )
21:        else                                                           ▷ b) Verschmelzung von zwei Pfaden
22:          if ( $n, n'$ ) has event  $e$  then                                 ▷ Identisch zu Fall 1
23:            success := CHECKPATH( $(n, n')$ )
24:          end if
25:        end if
26:      end if
27:    end for
28:  end while
29:  if success then
30:    success := CHECKCOVERAGE(absReach)
31:  end if
32:  return success
33: end function

```

Algorithmus 3.2 Pfade überprüfen

```

1: function CHECKPATH(Transition ( $n, n'$ ))
2:   for all  $cs \in$  PrecedingCorrespondingStates do
3:     Path refPath := ( $cs \Rightarrow n$ )
4:     Transition ( $a, b$ ) := getEventTrans( $cs, n$ )
5:     if ( $a, b$ )  $\neq$  null then
6:       Path absPath := GETEQUIVALENTPATH( $cs, (a, b).event, (n, n').event$ )
7:       if (absPath = null) then
8:         success := false
9:       else
10:        success := FINDCORRESPONDINGSTATES(absPath, refPath)
11:      end if
12:    end if
13:    if not success then
14:      return false
15:    end if
16:  end for
17:  return true
18: end function

```

nach der Verfeinerungsdefinition eine Strukturverfeinerung überprüft (Funktion CHECKSTRUCTUREREFINEMENT, siehe Abschnitt Strukturverfeinerung überprüfen auf Seite 91). Eine Korrespondenz zwischen den beiden Zuständen liegt nur dann vor, wenn beide Bedingungen erfüllt sind.

Prüfen der zeitlichen Bedingungen Zum Überprüfen der zeitlichen Bedingungen müssen wir gemäß der Verfeinerungsdefinition für TIMED STORY CHARTS (siehe Abschnitt 3.1.2) zwischen Nachrichten unterscheiden, die empfangen oder versendet werden. Der Algorithmus zur Überprüfung der zeitlichen Bedingung (siehe Algorithmus 3.4) überprüft als erstes, ob der zu überprüfende Zustand des verfeinerten Pfades über die gleichen Clocks wie der korrespondierende abstrakte Zustand verfügt. Ist dies nicht der Fall, liegt nach der Verfeinerungsdefinition für TIMED STORY CHARTS eine Verletzung vor. Anschließend wird nach Definition 28 für gesendete und empfangene Nachrichten die Summen über Clock Resets für jede Clock einzeln berechnet. Nur falls jede der Bedingungen erfüllt sind, ist die Überprüfung erfolgreich.

Strukturverfeinerung überprüfen Eine Strukturverfeinerung muss auf den initialen Strukturen des verfeinerten und abstrakten Transitionssystems durchgeführt werden sowie bei der Überprüfung auf korrespondierende Zustände (siehe Abschnitt Korrespondierende Zustände auf Seite 89). Der Algorithmus 3.5 basiert auf dem Ansatz nach [HT04] zur Überprüfung von Strukturverfeinerungen. Durch Anwendung einer Abstraktionsfunktion (Zeile 3) wird der abstrakte Graph (Zustand s) in den Namensraum des verfeinerten Graphen (Zustand t) übersetzt. Über

Algorithmus 3.3 Korrespondierende Zustände überprüfen

```

1: function FINDCORRESPONDINGSTATES(Path absPath, Path refPath)
2:   sent := ISSENEVENT(refPath.lastEvent)
3:   absSubPath := GETSUBPATHBETWEENEVENTS(absPath)
4:   refSubPath := GETSUBPATHBETWEENEVENTS(refPath)
5:   for s ∈ absSubPath do
6:     for t ∈ refSubPath do
7:       if !CHECKTIMECONSTRAINTS(s, t, absPath, refPath, sent) then
8:         continue
9:       end if
10:      if !CHECKSTRUCTUREREFINEMENT(s, t) then
11:        continue
12:      end if
13:      MARKCORRESPONDING(s, t)           ▷ Struktur und Zeit sind passend
14:      return true
15:    end for
16:  end for
17:  return false
18: end function

```

eine Delegation zwischen abstrakten und verfeinerten Port kann die Bestimmung der Abbildung automatisch erfolgen. Die Überprüfung der Strukturverfeinerung wird durch das Finden eines Matchings implementiert (Zeile 4).

Überprüfung von Kreisen Für die in dieser Arbeit betrachteten Systeme enden Pfade eines zeitbehafteten Transitionssystems eines TIMED STORY CHARTS in einem Kreis (siehe Einleitung Abschnitt 3.2.2). Die Funktion 3.6 zur Überprüfung von Kreisen wird immer dann aufgerufen, wenn zwei Pfade verschmolzen werden können (siehe Algorithmus 3.1). Der Algorithmus 3.6 zur Überprüfung von Kreisen muss sicherstellen, dass zwischen allen Nachrichten des Kreises korrespondierende Zustände zwischen dem abstrakten und verfeinerten Pfad existieren.

Ausgangspunkt für den Algorithmus ist, dass der Kreis bereits einmal durchlaufen wurde (siehe Zeile 20 von Algorithmus 3.1). Es reicht allerdings nicht aus den Kreis nur durch einen Lauf zu überprüfen, da auch für alle zukünftigen Ausführungen des Kreises korrespondierende Zustände zwischen aufeinander folgende Nachrichten existieren müssen. Um dies sicherzustellen überprüfen wir den Kreis ein weiteres mal. Wir können dabei zwischen den in Abbildung 3.8 dargestellten vier verschiedenen Arten von Kreisen unterscheiden, wobei (a) und (b) zulässige Kreise darstellen und (c) und (d) unzulässige. Die in a und b dargestellten Kreise sind zulässig, da zwischen jedem auftreten von Nachrichten innerhalb des Kreises korrespondierende Zustände identifiziert wurden (in grau). Die in a dargestellten Kreise betrachten die triviale Situation, dass keine Nachricht in dem Kreis enthalten ist. Die in c dargestellte Situation ist unzulässig, da in dem verfeinerten Pfad der korrespondierende Zustand außerhalb der Schleife liegt. Situa-

Algorithmus 3.4 Zeitliche Bedingungen überprüfen

```

1: function CHECKTIMINGCONSTRAINTS(State s, State t, Path absPath, Path refPath, bool
   sent)
2:   if ! s.clocks  $\subseteq$  t.clocks then
3:     return false
4:   end if
5:    $s' := \text{SUCC}(\text{absPath.initial})$ 
6:   repeat
7:     for all  $c \in s'$  do                                      $\triangleright$  Summe der Resets für absPath berechnen
8:        $\triangleright$  Schranke kleiner als im vorherigen Zustand  $\Rightarrow$  Reset durchgeführt
9:       if  $\text{ubound}(s'.c) < \text{ubound}(\text{prev}(s').c)$  then
10:         $\text{absUBounds}(c) += \text{ubound}(\text{prev}(s).c)$ 
11:      end if
12:    end for
13:  until ( $s \neq s'$ )
14:  ...                                                          $\triangleright$  Gleiche Berechnung für refPath
15:  for all ( $\text{doc} \in s$ )
16:    if  $\text{sent} \wedge \text{not} (\text{absUBounds}(c) + \text{ubound}(s.c) = \text{refUBounds}(c) + \text{ubound}(t.c))$  then
17:      return false
18:    end if
19:    if not  $\text{sent} \wedge \text{not} (\text{absUBounds}(c) + \text{ubound}(s.c) \geq \text{refUBounds}(c) + \text{ubound}(t.c))$ 
   then
20:      return false
21:    end if
22:  end for
23:  return true
24: end function

```

Algorithmus 3.5 Strukturverfeinerung überprüfen

```

1: function CHECKSTRUCTUREREFINEMENT(State s, State t)
2:   sCopy := MAKECOPY(s)
3:   sCopy := APPLYABS(sCopy)
4:   if FINDMATCHING(sCopy, t) then                              $\triangleright$  Berechne Matching von sCopy in t
5:     return true
6:   else
7:     return false
8:   end if
9: end function

```

Algorithmus 3.6 Kreise überprüfen

```
1: function ISWELLFORMEDCYCLE(State n)
2:   refCycle := refReach.GETCYCLE(n)
3:   absCycle := absReach.FINDCORRESPONDINGCYCLE(n, refCycle)
4:   correctOrder := HAVESAMEEVENTORDER(absCycle, refCycle)
5:   refWellFormed := true
6:   for all consecutive events a, b in refCycle do
7:     if not  $\exists$  correspondingState between a and b then
8:       refWellFormed := false
9:       break
10:    end if
11:  end for
12:  absWellFormed := true
13:  for all consecutive events a, b in absCycle do
14:    if not  $\exists$  correspondingState between a and b then
15:      absWellFormed := false
16:      break
17:    end if
18:  end for
19:  return correctOrder  $\wedge$  refWellFormed  $\wedge$  absWellFormed
20: end function
```

tion d zeigt eine Inkonsistenz, da im abstrakten Pfad kein korrespondierender Zustand zwischen Nachricht a und b identifiziert wurde. Die for-Schleifen in den Zeilen 6 und 13 des Algorithmus überprüfen die dargestellten Situationen.

Prüfen der Überdeckung Nachdem alle Pfade des verfeinerten Transitionssystems erfolgreich überprüft wurden, muss noch sichergestellt werden, dass alle Pfade des abstrakten Transitionssystems überdeckt wurden, um die notwendige Bedingung der Bisimulation zu erfüllen (siehe Bedingung 3 Verfeinerungsdefinition 29 eines TIMED STORY CHARTS). Algorithmus 3.7 zeigt den Pseudocode für die Überprüfung der Pfadüberdeckung. Es wird dabei das gesamte abstrakte Transitionssystem expandiert. Die for-Schleifen ab Zeile 4 stellen dabei sicher, dass zwischen zwei Nachrichten ein Zustand als korrespondierend zu einem Zustand des verfeinerten Transitionssystems identifiziert wurde.

3.2.3 Diskussion

Der vorgestellte Algorithmus ermöglicht die Verifikation der Verfeinerung nach Definition 29 in Form einer Erreichbarkeitsanalyse. Im Rahmen aktueller Arbeiten [Bre10] wird eine Variante umgesetzt, die das Konzept der Testautomaten nutzt, da hierüber elegant die Verfeinerungsüberprüfung zu einer formalen Verifikation von Sicherheits- und Lebendigkeitseigenschaften erwei-

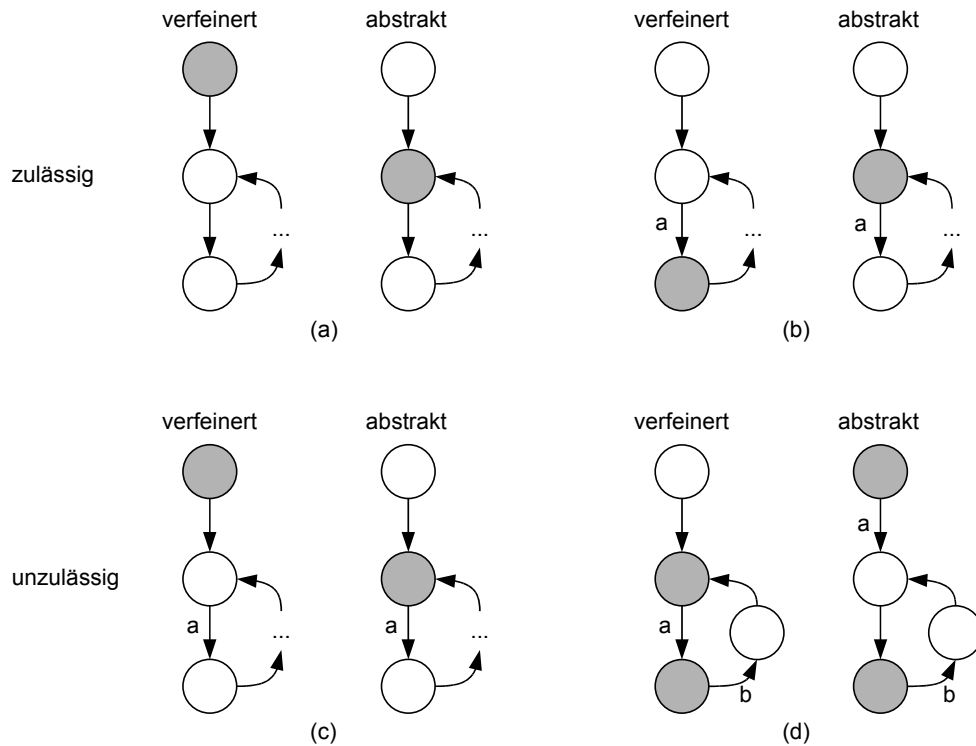


Abbildung 3.8: Überprüfung Kreise

Algorithmus 3.7 Überdeckung des abstrakten Systems prüfen

```

1: function CHECKCOVERAGE(TTS absReach)
2:   absReach.FULLEXPAND()
3:   success := true
4:   for all path  $\in$  absReach do
5:     for all consecutive events a, b on path do
6:       if not  $\exists$  correspondingState between a and b then
7:         success := false
8:         break
9:       end if
10:    end for
11:  end for
12:  return success
13: end function

```

tert werden kann. In [ABBL03] wurde bereits ein Ansatz vorgestellt, der aus einer eingeschränkten Klasse von TCTL-Formeln einen Testautomaten synthetisiert, der dann wiederum mit dem zu überprüfenden Modell parallel geschaltet werden kann. In der Erreichbarkeitsanalyse muss dann gezeigt werden, dass dieser Zustand erreichbar ist. Damit müssen die Algorithmen nur hinsichtlich der Testautomaten-Synthese erweitert werden.

Kapitel 4

Integration von Altkomponenten

In der industriellen Praxis kann es häufig vorkommen, dass Altkomponenten wiederverwendet werden, um zum einen den Entwicklungsprozess zu beschleunigen und zum anderen auf bewährte Qualität zurückzugreifen. Die Integration von Altkomponenten in eine MECHATRONIC UML Architektur stellt einen weiteren Anwendungsfall einer möglichen Konkretisierung dar (siehe Abschnitt 2.1 auf Seite 13). Unser Ansatz unterstützt eine Integration von Altkomponenten, indem das Verhaltensmodell für die Integration iterativ erlernt wird und auf dessen Basis dann formal die Integration überprüft werden kann.

Das Protokollverhalten der Komponente, mit welcher die Altkomponente interagieren soll, nennen wir Kontext. Eine Integration ist dann erfolgreich, wenn die Kommunikation zwischen Kontext und Altkomponente fehlerfrei ist. Dies wird durch Sicherheits- und (begrenzte) Lebendigkeitseigenschaften spezifiziert (siehe Abschnitt 2.4.1). Darüber hinaus ist es wichtig zu überprüfen, dass in Abhängigkeit vom Kommunikationsverhalten das erwartete Reglerverhalten ausgeführt wird. Es wird daher ein Ansatz für hybride Systeme unter Berücksichtigung von Sicherheits- und begrenzten Lebendigkeitseigenschaften benötigt.

Im Vergleich zu bisherigen Ansätzen (z. B. [HNS03b, BJR06, PVY99]) unterstützen wir eine Integration unter Berücksichtigung von Sicherheits- und begrenzten Lebendigkeitseigenschaften. Um dies zu ermöglichen wird iterativ das Modell des Verhaltens für die Integration der Altkomponente gelernt. Nach jedem Iterationsschritt wird überprüft, ob das erlernte Modell den Eigenschaften standhält. Der Kontext wird in der Analyse berücksichtigt, um nur die hierfür spezifischen Kommunikationen zu betrachten.

Zusätzlich zu dem iterativen Erlernen des Zustandsverhaltens integrieren wir die Möglichkeit, regelungstechnisches Verhalten für einen bekannten Zustand mit Hilfe klassischer Verfahren der Systemidentifikation zu identifizieren. So wird das Ein-/ Ausgangsverhalten linearer Systeme etwa durch Übertragungsfunktionen beschrieben [Ise92]. Sind die Übertragungsfunktionen bekannt, können auch Rekonfigurationen identifiziert werden.

Dieser Ansatz stellt damit ein Hilfsmittel für den Ingenieur dar, um bereits früh Konflikte auf Modellebene zu identifizieren. Derzeit wird die Integration von Altkomponenten mit Reglerverhalten am Ende des Entwicklungsprozesses während der Systemintegrationsphase durchgeführt. Typischerweise testet dabei der Ingenieur die Altkomponente (nur) in Hardware-In-The-Loop

Szenarien oder direkt in der realen Anwendung. Hier entdeckte Fehler sind nur unter großem Ressourceneinsatz zu beheben und daher teuer (z.B. [BN03]).

Anforderungen und Voraussetzungen Zur Veranschaulichung betrachten wir wieder einen Ausschnitt des einleitenden Konvoi-Beispiels (siehe Abbildung 1.2). Wie bereits in den Abschnitten 2.3 und 2.6.1 beschrieben, ist die Architektur der MECHATRONIC UML durch Komponenten (siehe Abbildung 4.1, RailCab und LegacyRailCab), Ports und den Verbindungen zwischen Ports gegeben.

Die Kommunikation zwischen den Komponenten ist definiert durch PARAMETERIZED REAL-TIME STATECHARTS, bzw. REAL-TIME STATECHARTS (vgl. Abschnitt 2.4.1). Da wir hier eine konkrete Integrationssituation betrachten, müssen wir im Folgenden auch nur REAL-TIME STATECHARTS berücksichtigen. In Abbildung 4.1 wird das DistanceCoordination-Muster gezeigt. Die Kommunikationsmuster werden in dem Beispiel durch die Rollen front und rear spezifiziert. Das Kommunikationsverhalten wird mit REAL-TIME STATECHARTS beschrieben. In Abschnitt 2.4.1 wurden bereits die Rollenverhalten für die front- und rear-Rolle spezifiziert.

Die Integration einer Altkomponente, wie dem LegacyRailCab, ohne eine Zustandsspezifikation des Kommunikationsverhaltens fordert eine Herleitung eines solchen Modells von der Komponentenschnittstelle und gegebenenfalls vom Quellcode der Altkomponente, um die benötigten Analysen für eine Integration durchführen zu können.

In dem in Abbildung 4.1 gezeigten Beispiel ist nur das Verhalten der front-Rolle bekannt. Es muss gezeigt werden, dass das unbekannte Kommunikationsverhalten der LegacyRailCab-Komponente mit dem erwarteten Verhalten des DistanceCoordination-Musters die Spezifikation (Sicherheits- und Lebendigkeitseigenschaften, wie front.convoy implies rear.convoy und Deadlock-Freiheit (A[] not deadlock)) nicht verletzt.

Grundsätzlich muss es sich hierbei nicht um ein Koordinationsmuster handeln. Es ist auch möglich, dass hier ein Protokollverhalten ohne vorherige Musterspezifikation mit einer Altkomponente verbunden wird oder wie Abbildung 2.1 darstellt ein Protokollverhalten an eine eingebettete Altkomponente delegiert wird.

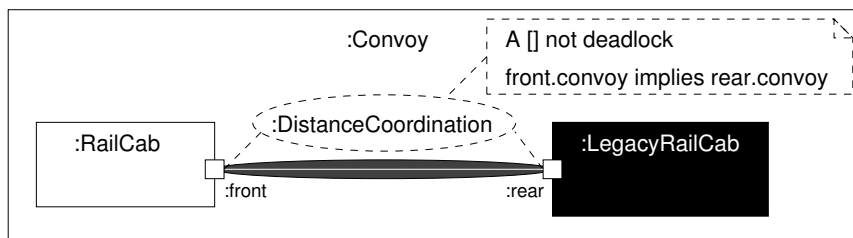


Abbildung 4.1: Architektur mit LegacyRailCab

Um überhaupt eine Altkomponente modellbasiert integrieren zu können, gehen wir davon aus, dass einige Informationen der Altkomponente zur Verfügung gestellt werden. Eine Vorausset-

zung ist, dass die Altkomponente eine Schnittstelle zur Verfügung stellt, die alle eingehenden und ausgehenden Nachrichten für die Kommunikation definiert, alle Signale definiert die durch die eingebetteten Regler benötigt werden sowie alle relevanten Informationen für die Ausführung spezifiziert (wie z. B. die Ausführungsperiode).

Im Bereich sicherheitskritischer Systeme, sind diese Voraussetzungen keine Einschränkung, da sie elementar für eine Integration sind (z. B. [HMSN10a]). Ein Automobilhersteller kann z. B. keine Altkomponente ohne Angabe der Ausführungsperiode, unter der die Altkomponente die spezifizierten Eigenschaften erfüllt, integrieren.

Die Informationen, die durch eine Altkomponente zur Verfügung gestellt werden können stark variieren. Grundsätzlich können wir allerdings zwei Fälle unterscheiden. 1) die Altkomponente bietet zusätzliche Schnittstellenoperationen an, um den aktuellen Zustand zu erfragen¹ oder 2) dies ist nicht der Fall. Für den zweiten Fall können wir zudem unterscheiden zwischen einer a) Black Box (der Quellcode ist nicht zugreifbar) und b) einer White Box (auf den Quellcode kann zugegriffen werden). Wir haben entsprechend für diese drei Fälle Algorithmen entwickelt: Gray Box Checking für Fall 1), Black Box Checking für Fall 2a) und White Box Checking für Fall 2b).

Eine Integration ist dann erfolgreich, wenn die Spezifikation nicht verletzt wird. Dies kann dabei 1) durch eine formale Verifikation der Altkomponente mit dem Kontext (z. B. front-Rolle) gezeigt werden oder 2) durch eine Verifikation der Verfeinerung des abstrakten Rollen Verhaltens (z. B. rear-Rolle) gegeben durch ein REAL-TIME COORDINATION PATTERN. Es muss also gezeigt werden, dass: $\text{Kontext (front Rolle)} \parallel \text{erlerntes Verhalten (erlernte rear Rolle)} \models \phi$ ($\text{front.convoy implies erlerntes rear.convoy}$) $\wedge \neg \delta$ ($\text{Deadlock - Freiheit}$) für 1) und $\text{erlerntes Verhalten (erlernte rear Rolle)} \leq \text{abstraktes Rollenverhalten (rear Rolle)}$ für 2).

Eine Verfeinerung kann auch durch eine parallele Komposition gezeigt werden, in dem aus dem abstrakten Rollenverhalten (z. B. front-Rolle) ein Testautomat erstellt wird [JLS00]. Ein Testautomat wird durch Komplementbildung erstellt. Zusätzlich wird das Verhalten, welches nicht durch das abstrakte Verhalten erfüllt wird, durch einen extra Fehlerzustand und entsprechenden Transitionen, die in diesen Fehlerzustand führen, dargestellt. Die Analyse muss dann folglich zeigen, ob dieser Fehlerzustand erreicht werden kann. Dies wird, wie oben für 1) beschrieben, durch eine parallele Komposition des Testautomaten mit dem verfeinerten Verhalten erreicht. Wir werden daher im Folgenden die Integration nur für 1) zu zeigen.

Übersicht Im Folgenden stellen wir als erstes den Gray Box Checking Ansatz vor, da hier der grundlegende Ansatz des iterativen Erlernens des Kommunikationsverhaltens einer Altkomponente sowie der schrittweisen Überprüfung des erlernten Verhaltens vorgestellt wird. Anschließend betrachten wir, wie Zustände im Falle der Black- und White Box erlernt werden können. Dann diskutieren wir in Abschnitt 4.4, wie wir das regelungstechnische Verhalten zu einem Zustand identifizieren können. Der Gray Box Checking Ansatz wurde grundlegend in den Arbeiten

¹AUTOSAR-Komponenten bieten z. B. diese Informationen an.

[HH07, HH08a, GHH08a, Bre08, BGH⁺08] vorgestellt. Eine geeignete Testumgebung wurde in [GHHP07, Pri07] beschrieben. Der Gesamtansatz inklusive des Black- und White Box Checking wurde in [HBB⁺09, HMSN10b, HMSN10a, BBB⁺09, HMS⁺10] vorgestellt.

4.1 Gray Box Checking

In diesem Abschnitt betrachten wir die Integration von Altkomponenten, deren Schnittstelle Operationen zur Verfügung stellen, um den aktuellen Zustand der Altkomponente zu erfragen. Wie bereits in Paragraph Anforderungen und Voraussetzungen auf Seite 98 im übergeordneten Abschnitt erläutert, bieten einige Klassen von Altkomponenten diese Informationen an. Gerade im Bereich sicherheitskritischer Systeme ist die Bereitstellung aktueller Zustandsinformationen durchaus üblich, um (eindeutig) zu identifizieren, welche Aktionen das System ausführt [Sto96, Pul01, Dun02].

Eine Reihe von Ansätzen existieren, die entweder einen reinen Black-Box-Ansatz und Automaten-Lernen verfolgen (z. B. [HNS03a]) oder einen White-Box-Ansatz propagieren, die ein Modell aus Quellcode extrahieren [DKU06, CDH⁺00, HS99]. Keiner dieser Ansätze betrachtet allerdings Echtzeitsysteme oder nutzt das Wissen eines Kontextes und Komponenten aus, um auch für größere Systeme skalieren zu können. Weiterhin vermeiden diese Ansätze keine falsch positive oder falsch negative Ergebnisse. Dies gilt übrigens auch für den gesamten klassischen Reverse Engineering Bereich, der sich im Wesentlichen auf die Unterstützung der Dokumentation von Altkomponenten fokussiert (siehe z. B. [MJS⁺00]). Damit sind all diese Ansätze nicht für die hier betrachteten sicherheitskritischen Systeme geeignet.

Abbildung 4.2 gibt eine Übersicht über unseren Ansatz. Initial nehmen wir an, dass die Altkomponente in einen Startzustand oder allgemein in einem sogenannten Quiescent Zustand ist (siehe [KM98, ZC06]). Informationen über die Initialisierung der Altkomponente, die sie in einen solchen Zustand versetzt, nehmen wir als bekannt an.

Unser Ansatz erweitert das aktuelle Wissen über die Altkomponente mit chaotischem Verhalten. Dieses chaotische Verhalten spezifiziert jedes mögliche Kommunikationsverhalten auf der einen Seite und auf der anderen Seite, kann die Altkomponente zu jedem Zeitpunkt in einen Deadlock eintreten. Dieses Verhalten stellt damit eine Überapproximation des Kommunikationsverhaltens der Altkomponente dar: zu jedem Zeitpunkt wird das mögliche Gesamtverhalten spezifiziert, jedoch muss nicht das mögliche Gesamtverhalten tatsächlich durch die Altkomponente unterstützt sein.

Das chaotische Verhalten wird dann in Kombination mit dem Kontextverhalten via Model Checking formal verifiziert unter Berücksichtigung von Sicherheits- und begrenzten Lebendigkeitseigenschaften (Schritt 1 und 2 in Abbildung 4.2). Wenn die Überprüfung zu einem Gegenbeispiel führt, dient dieses als Testeingabe für die Altkomponente (Schritt 3). Wenn das Gegenbeispiel durch die Altkomponente bestätigt wird, haben wir ein wirkliches Gegenbeispiel gefunden. Ist dies nicht der Fall, nutzen wir das beobachtete Verhalten der Altkomponente aus, um das

bisher erlernte Verhalten zu verfeinern (Schritt 4). Dieser Vorgang wird solange fortgesetzt, bis entweder ein Gegenbeispiel gefunden wird oder alle möglichen Traces des Kontexts betrachtet wurden.

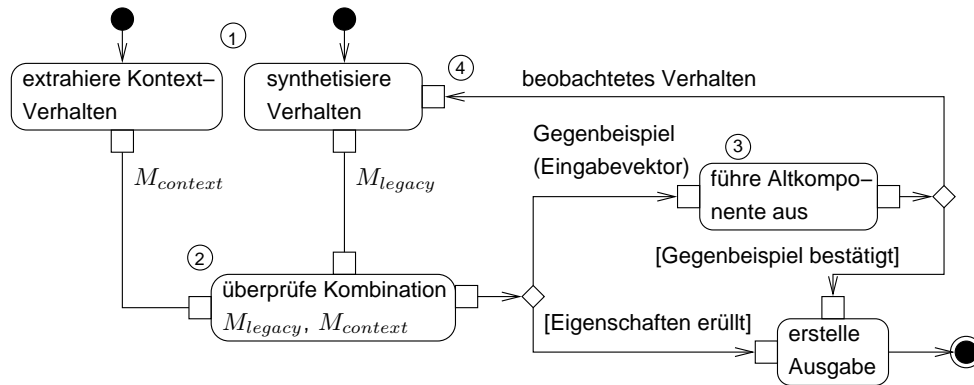


Abbildung 4.2: Iteratives Lernen und Überprüfen: Gray Box Checking

Im Folgenden stellen wir die Grundlagen für den Lernansatz vor. Anschließend präsentieren wir in den Abschnitten 4.1.2 und 4.1.3 unseren iterativen Lern- und Überprüfungs-Ansatz.

4.1.1 Formalisierungen

Da eine Implementierung aufgrund der aktuellen Hardwarearchitekturen nur diskret erfolgen kann, betrachten wir hier entsprechend auch nur diskrete Zeit. Der Kontext muss also in einer entsprechenden plattformspezifischen Verfeinerung vorliegen (siehe 3.1.1 und 6.1). Dies ist aber ohnehin notwendig, um eine Ausführung zusammen mit der Altkomponente zu ermöglichen. Hieraus folgern wir, dass ein Automatenmodell mit diskreter Zeit ausreichend ist, um das Verhalten der Altkomponente zu erlernen und um eine kompositionelle Verifikation kombiniert mit Testen und Beobachtung zu ermöglichen (siehe Anforderungen und Voraussetzungen auf Seite 98).

Die Vereinfachung ist gerechtfertigt mit den folgenden Annahmen, die gültig für die betrachteten Systeme hat: 1) die Uhren laufen ausreichend synchron. Dies ist gewöhnlich für sicherheitskritische Systeme und bedeutet, dass die Zeit in den Komponenten gleich schnell verläuft, bzw. eine Zeitverschiebung bekannt ist (siehe Abschnitt 2.4.6). 2) ein diskretes Zeitmodell ist ausreichend um alle Zeiteigenschaften zu spezifizieren, da die Infrastruktur nicht unendlich schnell reagieren kann.

Das vereinfachte Echtzeit-Automatenmodell und dessen Echtzeitverarbeitung ist wie folgt definiert:

Definition 33 (Diskreter Echtzeit-Automat)

Ein diskreter Echtzeit-Automat ist ein 5-Tupel $M = (S, I, O, T, Q)$ mit einer endlichen Menge S von Zuständen, Eingangs-Nachrichten I , Ausgangs-Nachrichten O , einer Menge von Tran-

sitionen $T \subseteq S \times \wp(I) \times \wp(O) \times S$, wobei $\wp(x)$ die Potenzmenge der Eingangs-/Ausgangs-Nachrichten angibt und Q die initiale Zustandsmenge. Das Schalten einer Transition entspricht genau einer Zeiteinheit.

Das Verhalten ist charakterisiert durch Ausführungspfade.

Definition 34 (Regulärer- und Deadlock-Ausführungspfad)

Ein regulärer Ausführungspfad ist eine Sequenz von Zuständen $s_i \in S$ und Eingangs-/Ausgangs-Nachrichten $A_i/B_i \in I/O$ mit $\pi = s_1, A_1/B_1, s_2, \dots$, wobei für jede Nachricht $i \geq 1$ eine Transition $(s_i, A_i, B_i, s_{i+1}) \in T$ existiert. Ein Deadlock-Ausführungspfad ist eine Sequenz von Zuständen $s_i \in S$ und Eingangs-/Ausgangs-Nachrichten $A_i/B_i \in I/O$ mit $\pi = s_1, A_1/B_1, s_2, \dots, s_n, A_n/B_n$, wobei für jede Nachricht $1 \leq i \leq n$ eine Transition $(s_i, A_i, B_i, s_{i+1}) \in T$ existiert und weiterhin $\nexists s_{n+1} \in S$ mit $(s_n, A_n, B_n, s_{n+1}) \in T$. $[M]$ beschreibt alle regulären und Deadlock-Ausführungspfade. Wir schreiben $\pi|_{I/O}$, um einen Ausführungspfad zu einer beobachtbaren Sequenz einzuschränken und $\pi|_S$, um die Zustandssequenz zu beschreiben.

Diese vorgestellte Definition hat Ähnlichkeiten mit dem Konzept von Prozessalgebren. Während reguläre beobachtbare Sequenzen Ausführungspfade in CSP [Hoa85] oder anderen Prozessalgebren sind, sind Deadlock-Sequenzen vergleichbar mit Fehlern in CSP. Prozessalgebren abstrahieren allerdings von Zuständen.

Die Zeitsemantik eines Automaten ist dadurch definiert, dass eine Transition genau eine Zeiteinheit benötigt. Aus Vereinfachungsgründen werden wir allerdings in Beispielen, die in Abschnitt 2.6 benutzte Syntax $clock > Konstante$ verwenden. Mit der hier vorgestellten Zeitsemantik müssten entsprechenden der Konstanten viele Zwischen-Transitionen oder ein Zähler über Selbsttransitionen benutzt werden.

Aus Vereinfachungsgründen bezeichne im Folgenden S_i, I_i, O_i, T_i , und Q_i Elemente des diskreten Echtzeit-Automaten M_i . Zwei Automaten M und M' mit unterschiedlichen Eingabe- und Ausgabe-Mengen ($I \cap I' = \emptyset$ und $O \cap O' = \emptyset$) bezeichnen wir komponierbar (engl. composable). Wenn zudem $I \cap O' = \emptyset$ und $O \cap I' = \emptyset$ gilt, dann sind sie orthogonal zueinander.

Spezifikation von Eigenschaften Eigenschaften werden, wie in Abschnitt 2.4.6 beschrieben, mit Timed CTL (TCTL) Bedingungen (ϕ) und Invarianten (ψ) spezifiziert. Ein Echtzeit-Automat M wird um eine Propositionsmenge P erweitert und ein beliebiger Zustand $s \in S$ wird mit allen Propositionen aus P durch eine Markierungsfunktion $L : S \rightarrow \wp(P)$ annotiert welche diese erfüllen. Ein Echtzeit-Automat $M = (S, I, O, T, Q)$ wird entsprechend zu einem 6-Tupel $M = (S, I, O, T, L, Q)$ erweitert. Die Markierungsmenge $\mathcal{L}(M)$ bezeichnet die Menge aller Propositionen aus P , die durch die Markierung betrachtet werden.

In den folgenden Formel-Definitionen lassen wir aus Vereinfachungsgründen jegliche syntaktischen Details von TCTL wegfallen und schreiben $M \models \phi$ wenn ein Automat M eine Bedingung oder Invariante ϕ erfüllt.

Das spezielle Symbol δ wird benutzt um auszudrücken, dass ein *deadlock* erreicht werden kann. Ein Deadlock ist ein Zustand ohne jegliche ausgehende Transition. $M \models \neg\delta$ drückt damit aus, dass M keinen Deadlock beinhaltet.

Parallele Komposition Werden mehrere Komponenten komponiert, so werden diese parallel ausgeführt. Die Kommunikation wird durch eine *synchrone Kommunikation* formalisiert, so dass Senden und Empfangen von Nachrichten innerhalb des gleichen Zeitschritts stattfinden. Da wir allerdings grundlegend eine asynchrone Nachrichten-Semantik verfolgen, führen wir explizite Nachrichtenpuffer durch einen extra Automaten ein. Diese expliziten Automaten sind zudem gefordert, um Verbindungscharakteristiken, wie Nachrichten-Ausfall, zu spezifizieren (siehe Abschnitt 2.4).

Die Kombination von zwei komponierten Automaten wird durch eine Verbindung der Eingabe und Ausgabe-Nachrichten erreicht. Im Unterschied zu Definition 2 stellen wir im Folgenden eine angepasste parallele Komposition für die diskreten Echtzeit-Automaten vor. Dies ist notwendig, da Zeit nicht explizit über Clocks, sondern Transitionen kodiert ist.

Definition 35 (Parallele Komposition)

Für zwei diskrete Echtzeit-Automaten $M = (S, I, O, T, L, Q)$ und $M' = (S', I', O', T', L', Q')$, welche komponierbar zueinander sind ($I \cap I' = \emptyset$ und $O \cap O' = \emptyset$), definieren wir ihre parallele Komposition ausgedrückt durch $M \parallel M'$ als Automat $(S'', I'', O'', T'', L'', Q'')$ mit $S'' = S \times S'$, $I'' = I \cup I'$, $O'' = O \cup O'$, $Q'' = Q \times Q'$, und $((s_1, s'_1), A'', B'', (s_2, s'_2)) \in T''$, wenn $(s_1, A, B, s_2) \in T$ und $(s'_1, A', B', s'_2) \in T'$ existiert mit $A'' = A \cup A'$ und $B'' = B \cup B'$. Zusätzlich muss $(A \cap O') = B'$ und $(A' \cap O) = B$ gelten. Die Markierung L'' für $(s, s') \in S''$ wird hergeleitet aus $L''((s, s')) = L(s) \cup L'(s')$.

Eine Transition in T'' ist damit eine Kombination von zwei Transitionen in jedem Automaten, wenn alle benötigten lokalen Eingaben durch den anderen Automaten erfüllt werden ($(A \cap O') = B'$ und $(A' \cap O) = B$). Die nicht lokalen Eingaben und Ausgaben sind einfach eine Vereinigung beider Automaten.

Verfeinerung Die Definition einer Verfeinerung ist essentiell, um zu beschreiben, dass ein Protokollverhalten einer Altkomponente eine korrekte Implementierung eines abstrakten Rollenverhaltens ist. Im Unterschied zu der Verfeinerung definiert in Abschnitt 3.1.1 müssen wir hier Automaten mit diskretem Zeitformalismus betrachten. Die Verfeinerung ist wie folgt definiert.

Definition 36 (Verfeinerung)

Ein diskreter Echtzeit-Automat $M = (S, I, O, T, L, Q)$ ist eine Verfeinerung des diskreten Echtzeit-Automaten $M' = (S', I', O', T', L', Q')$ ($M \sqsubseteq M'$) wenn gilt:

$$\forall \pi = \dots s \in [M] \exists \pi' = \dots s' \in [M'] : \pi|_{I/O} = \pi'|_{I'/O'} \wedge L(s) = L'(s') \quad (4.1)$$

$$\forall \pi = \dots s, A/B \in [M] \exists \pi = \dots s', A/B \in [M'] : \pi|_{I/O} = \pi'|_{I'/O'} \quad (4.2)$$

Damit gilt, dass für jeden Pfad in einer Altkomponente M Gleichung 4.1 zusichert, dass es einen zugehörigen Pfad in dem abstrakten Automaten M' existiert. Aus Gleichung 4.2 folgt weiterhin, dass für jeden Deadlock-Pfad in M ebenfalls ein möglicher Deadlock-Pfad in M' existiert. Damit wird eine Ordnung hergestellt, wie für Simulationsbeziehungen gefordert [CGP00]. Womit gilt, dass aus \sqsubseteq eine Simulationsbeziehung (\preceq) folgt. Entsprechend der Definition bleibt das extern sichtbare Echtzeitverhalten $\pi|_{I/O}$ erhalten und zudem aufgrund des Erhalts einer Simulationsbeziehung auch die geforderten Sicherheits- und begrenzten Lebendigkeitseigenschaften [BK08]. $\pi|_{I/O}$ können wir zudem einfach relaxieren, wie in Abschnitt 3.1.1 beschrieben, um weitere Verfeinerungen zu erlauben. Für die folgenden Definitionen und den in den Abschnitten 4.1.2 und 4.1.3 beschriebenen Lernansatz hat das allerdings keine Auswirkung.

Kompositionelle Bedingungen Im Folgenden betrachten wir ausführlicher, welche Klassen von Bedingungen durch Komposition und Verfeinerung für unseren spezifischen diskreten Echtzeitautomaten erhalten bleiben.

Definition 37 (Kompositionelle Bedingungen)

Eine Bedingung ϕ ist kompositionell, wenn für jeden diskreten Echtzeit-Automaten M_1 , M'_1 , und M_2 mit $\mathcal{L}(M_2) \cap \mathcal{L}(\phi) = \emptyset$ gilt

$$(M_1 \models \phi) \Rightarrow ((M_1 \parallel M_2 \models \phi) \vee (M_1 \parallel M_2 \models \delta)) \text{ und} \quad (4.3)$$

$$((M_1 \sqsubseteq M'_1) \wedge (M'_1 \models \phi)) \Rightarrow (M_1 \models \phi) \quad (4.4)$$

Allgemein gilt, dass CTL-Formeln durch Bisimulations-Beziehungen erhalten bleiben. ACTL-Formeln bleiben durch Simulations-Beziehungen erhalten (\preceq) [CGP00]. Die vorgestellte Verfeinerung impliziert eine Simulations-Beziehung und erhält daher ACTL-Formeln und zusätzlich Deadlock-Freiheit.

Lemma 2

Für einen diskreten Echtzeit-Automaten M und M' mit $M \sqsubseteq M'$ gilt $M' \models \neg\delta \Rightarrow M \models \neg\delta$.

Beweis 3

Bedingung 4.1 sichert zu, dass für jeden Zustand $s \in S$ wenigstens ein Zustand $s' \in S'$ mit $(s, s') \in \Omega$ existiert. Ist M' ohne Deadlock, dann folgt daraus, dass s' mindestens eine ausgehende Transition hat und Bedingung 4.2 stellt sicher, dass dieses auch für s gilt. Damit gilt, dass auch M frei von einem Deadlock ist.

Invarianten, untere und obere Zeitschranken sowie ACTL-Formeln sind im generellen Bedingungen, die sich nur auf alle möglichen Pfade beziehen. Durch die Bedingung, dass die Zustandsmarkierungen disjunkt sind, kann die Anzahl der Zustandssequenzen mit gleicher Markierung nicht erhöht werden und damit sind diese kompositionell.

Deadlock-Freiheit ist ebenfalls kompositionell. Dies folgt aus Konstruktion von Bedingung 4.3 und durch Lemma 2 für Bedingung 4.4.

Kompositionalität gilt also für Deadlock-Freiheit, obere Schranken für maximale Nachrichten-Verzögerungen, untere Schranken für die minimale Verzögerung von Nachrichten und für Invarianten. Formeln der Form $AG(\neg p_1 \vee (AF_{[1,d]} p_2))$ können damit erfüllt werden. Formeln, die z. B. Lebendigkeitseigenschaften ohne Zeitbeschränkung definieren, können nicht erhalten werden. Im Allgemeinen ist dies allerdings für harte Echtzeitsysteme nicht notwendig, da zu einer Bedingung auch immer eine zeitliche Einschränkung per Definition gefordert wird.

Parallele Komposition & Verfeinerung Nun müssen wir noch zeigen, dass die parallele Komposition unserer diskreten Echtzeit-Automaten auch die Verfeinerung nach Definition 36 erhält.

Lemma 3

Für einen beliebigen diskreten Echtzeit-Automaten M_1 , einen Automaten M_2 und einer Verfeinerung $M_2 \sqsubseteq M'_2$ gilt $M_2 \sqsubseteq M'_2 \Rightarrow (M_1 \parallel M_2 \sqsubseteq M_1 \parallel M'_2)$.

Beweis 4

Für $M_1 \parallel M'_2$ können wir aus der Konstruktion der parallelen Komposition folgern, dass nur Pfade und Deadlock-Pfade resultieren, die auch in $M_1 \parallel M_2$ existieren. Daher sind Bedingung 4.1 und 4.2 durch $M_1 \parallel M_2$ und $M_1 \parallel M'_2$ erfüllt.

Weiterhin müssen wir zeigen, dass kompositionelle Bedingungen und Deadlock-Freiheit erhalten bleiben.

Lemma 4

Für M_1, M_2 und M'_2 mit $M_2 \sqsubseteq_{I/O} M'_2, I_1 \cap (O_2 - O'_2) = \emptyset, O_1 \cap (I_2 - I'_2) = \emptyset$, und $\mathcal{L}(M_1) \cap (\mathcal{L}(M_2) - \mathcal{L}(M'_2)) = \emptyset$ und einer beliebigen kompositionellen Bedingung ϕ gilt

$$(M_1 \parallel M'_2 \models \phi \wedge \neg\delta) \Rightarrow (M_1 \parallel M_2 \models \phi \wedge \neg\delta) \quad (4.5)$$

Beweis 5

Da ϕ und $\neg\delta$ kompositionell sind und aus Definition 37 können wir für $M''_2 = M_2|_{I'_2/O'_2/\mathcal{L}(M'_2)}$ folgern, dass $M_1 \parallel M''_2 \models \phi \wedge \neg\delta$ oder $M_1 \parallel M''_2 \models \delta$ gilt. Durch Lemma 2 und 3 gilt zudem $M_1 \parallel M'_2 \models \phi \wedge \neg\delta$. Durch $I_1 \cap (O_2 - O'_2) = \emptyset$ und $O_1 \cap (I_2 - I'_2) = \emptyset$ folgt weiterhin, dass M_2 M''_2 nur I/O-Nachrichten hinzufügt, welche nicht M_1 behindern und daher gilt, dass $M_1 \parallel M_2$ die gleiche erreichbare Zustandsmenge und Transitionen haben, womit $M_1 \parallel M_2 \models \neg\delta$ gilt. Da ϕ nur über Zustände interpretiert wird und Markierungen identisch für $\mathcal{L}(\phi) \subseteq \mathcal{L}(M'_2)$ und ϕ sind, ist damit Bedingung 4.5 bewiesen.

Unvollständiger Automat Um inkrementell die Genauigkeit eines Verhaltensmodells zu verbessern, führen wir das Konzept von unvollständigen Automaten ein.

Definition 38 (Unvollständiger Automat)

Ein unvollständiger Automat ist ein 6-Tupel $M = (S, I, O, T, \bar{T}, Q)$ mit $M = (S, I, O, T, Q)$ ist ein Automat und $\bar{T} \subseteq S \times \wp(I) \times \wp(O)$ beschreibt die bekannten nicht unterstützten Kommunikationen. Um sicherzustellen, dass T und \bar{T} konsistent sind, verlangen wir

$$\neg(\exists s, A, B, s' : (s, A, B, s') \in T \wedge (s, A, B) \in \bar{T}).$$

Das Verhalten ist ebenfalls durch Ausführungspfade charakterisiert.

Definition 39 (Unvollständige Ausführungspfade)

Ein regulärer Ausführungspfad eines unvollständigen Automaten ist eine Sequenz von Zuständen $s_i \in S$ und Eingangs-/Ausgangs-Nachrichten $A_i/B_i \in I/O$ mit $\pi = s_1, A_1/B_1, s_2, \dots$, wobei für jede Nachricht $i \geq 1$ eine Transition $(s_i, A_i, B_i, s_{i+1}) \in T$ existiert. Ein Deadlock-Ausführungspfad ist eine Sequenz von Zuständen $s_i \in S$ und Eingangs-/Ausgangs-Nachrichten $A_i/B_i \in I/O$ mit $\pi = s_1, A_1/B_1, s_2, \dots, s_n, A_n/B_n$, wobei für jede Nachricht $1 \leq i \leq n$ eine Transition $(s_i, A_i, B_i, s_{i+1}) \in T$ existiert und weiterhin $(s_n, A_n, B_n) \in \bar{T}$ gilt. $[M]$ beschreibt alle regulären und Deadlock-Ausführungspfade.

Die Definition der Ausführungssequenz hebt hervor, dass Deadlock-Ausführungspfade eines unvollständigen Automaten nur angenommen werden, wenn diese explizit durch \bar{T} definiert wurden und nicht implizit, wenn keine Transition in T gegenwärtig ist.

Ein Automat ist *deterministisch*, wenn für jeden Zustand s sowie Nachrichten A und B gilt, dass $|\{(s, A, B, s') \in T\}| \leq 1$. Ein unvollständiger Automat ist *deterministisch*, wenn für jeden Zustand s sowie Nachrichten A und B gilt, dass $|\{(s, A, B, s') \in T\} \cup \{(s, A, B) \in \bar{T}\}| \leq 1$.

Für einen unvollständigen Automaten beschreiben wir einen Vervollständigungsschritt als eine beliebige Erweiterung von S , T oder \bar{T} , welche wieder in einen unvollständigen Automaten resultiert. Letztendlich wird ein unvollständiger Automat *vollständig*, wenn jede mögliche Kommunikation entweder durch \bar{T} verboten ist oder in T ist:

$$\forall s \in S, A \in \wp(I), B \in \wp(O) : (\exists s' \in S : (s, A, B, s') \in T \text{ xor } (s, A, B) \in \bar{T}).$$

Chaotischer Automat und Hülle Betrachten wir die Verfeinerungsdefinition 36, können wir ein maximales Verhalten identifizieren, welches wir durch einen chaotischen Automaten definieren. Dieser chaotische Automat ist eine Abstraktion von allen möglichen Verhalten, da jede mögliche Eingabe-Sequenz akzeptiert wird, genauso wie Deadlocks.

Definition 40 (Chaotischer Automat)

Für eine gegebene Eingabe- und Ausgabemenge I und O , wird der chaotische Automat $M_c = (S_c, I, O, T_c, Q_c)$ wie folgt definiert: Die Zustandsmenge S_c ist definiert durch $S_c = \{s_\delta, s_\forall\}$. Die Transitionen sind definiert durch $T_c = \{(s_\forall, A, B, s_\forall) | A \in \wp(I), B \in \wp(O)\} \cup \{(s_\forall, A, B, s_\delta) | A \in \wp(I), B \in \wp(O)\}$ und $Q_c = \{s_\delta, s_\forall\}$.

Abbildung 4.3 stellt einen chaotischen Automat nach Definition 40 dar. Aus dieser Abbildung ist zu sehen, dass s_\forall und s_δ mögliche initiale Zustände sind. Während s_δ jede Kommunikation blockiert, akzeptiert s_\forall jede mögliche Kommunikation. Die möglichen Eingabe- und Ausgabe-Kombinationen werden mit '*' gekennzeichnet.

Wenn zudem Bedingungen relevant sind, fügen wir weitere Zustände s_\forall und s_δ für jede mögliche Bedingung P' von P ein. Es ist jedoch effizienter s_\forall und s_δ mit einer neuen Proposition p' zu markieren, da hierdurch keine weiteren Zustände eingeführt werden müssen. Weiterhin müssen

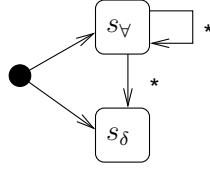


Abbildung 4.3: Maximal chaotisches Verhalten: der chaotische Automat

dann alle Propositionen $p \in P$ in denen p auftritt durch $(p \vee p')$ ersetzt werden, genauso wie alle $\neg p$ durch $(\neg p \vee p')$ ersetzt werden.

Da wir an einer sicheren Abstraktion des Protokollverhaltens der Altkomponente interessiert sind, führen wir eine besondere Vervollständigung ein, die sogenannte chaotische Vervollständigung. Die chaotische Vervollständigung resultiert in einen willkürlichen chaotischen Verhalten.

Definition 41 (Chaotische Hülle)

Gegeben sei ein unvollständiger Automat $M = (S, I, O, T, \bar{T}, Q)$, dann leiten wir die dazugehörige chaotische Hülle $M' = (S', I, O, T', Q')$ wie folgt ab:

1. verdoppele die Zustandsmenge und berücksichtige den chaotischen Automaten: $S' = (S \times \{0\}) \uplus (S \times \{1\}) \uplus S_c$ und
2. passe die Transitionsmenge der Verdoppelung so an, dass alle nicht spezifizierten Kommunikationen entweder nicht unterstützt werden oder in den hinzugefügten chaotischen Automaten führen: $T' = \{((s, 0), A, B, (s', 0)) \mid (s, A, B, s') \in T\} \uplus \{((s, 0), A, B, (s', 1)) \mid (s, A, B, s') \in T\} \uplus \{((s, 1), A, B, (s', 0)) \mid (s, A, B, s') \in T\} \uplus \{((s, 1), A, B, (s', 1)) \mid (s, A, B, s') \in T\} \uplus \{((s, 1), A, B, s_v) \mid s \in S, a \in \varnothing(I), B \in \varnothing(O), (s, A, B) \notin \bar{T}\} \uplus \{((s, 1), A, B, s_\delta) \mid s \in S, a \in \varnothing(I), B \in \varnothing(O), (s, A, B) \notin \bar{T}\} \uplus T_c$.

Wir beschreiben die chaotische Hülle von M mit $\text{chaos}(M)$.

In dieser Konstruktion gilt: $Q' = \{(s, 0) \mid s \in Q\} \uplus \{(s, 1) \mid s \in Q\}$. Die Zustände $(s, 0)$ repräsentieren den Fall, dass keine Erweiterung angenommen wird, welche in einen Deadlock führt. Die Zustände $(s, 1)$ repräsentieren den Fall, dass alle möglichen Erweiterungen angenommen werden, welche entsprechend in das Chaos führen. Dies wird repräsentiert durch s_δ und s_v . Aus dieser Definition ist zudem ersichtlich, dass das chaotische Verhalten (hochgradig) nichtdeterministisch ist, während das reale Altkomponenten-Verhalten deterministisches Verhalten aufweist.

Beobachtungs-Konformität & Verfeinerung

Definition 42 (Beobachtungs-Konformität)

Ein unvollständiger Automat M ist beobachtungs-konform bezüglich eines Automaten M_r , wenn $[M] \subseteq [M_r]$.

In unserem Fall haben wir die Beobachtung über Zustände definiert. Typischerweise wird dies über Pfade definiert. Für den hier betrachteten Gray-Box-Checking-Ansatz ist das allerdings unpraktisch, da wir die Zustände der Altcomponente kennen.

Theorem 2

Wenn M ein beobachtungs-konformer unvollständiger Automat bezüglich einer konkreten deterministischen Altcomponente M_r ist, dann gilt, dass $M_r \sqsubseteq \text{chaos}(M)$.

Beweis 6

Aus der Verfeinerungs-Bedingung 4.1 folgt direkt $[M] \subseteq [M_r]$, da s_δ und s_\forall alle positiven und negativen Propositionen durch die diskutierten Anpassungen der Formeln erfüllen. Bedingung 4.2 ist erfüllt, da die chaotische Hülle per Konstruktion garantiert, nur zusätzliches Verhalten hinzuzufügen, welches immer in einen Deadlock führen kann. \square

4.1.2 Initiale Verhaltenssynthese

Gegeben sei ein konkreter Kontext M_r^c mit abstraktem Modell M_a^c , für das gilt, dass der konkrete Kontext eine Verfeinerung des abstrakten ist ($M_r^c \sqsubseteq M_a^c$). Weiterhin sei eine konkrete Altcomponente M_r mit versteckten internen Details gegeben. Die generelle Frage, die unser Gray-Box-Checking-Ansatz zu beantworten hat ist, ob eine gegebene Bedingung ϕ so wie Deadlock-Freiheit ($\neg\delta$) erfüllt sind.

Da wir sicherheitskritische Systeme betrachten, muss unser Ansatz entweder eine Garantie geben, dass beide Eigenschaften erfüllt sind oder ein Gegenbeispiel liefern, welches eine Verletzung der Bedingungen aufweist. Allerdings kann gewöhnlich nicht der gesamte Zustandsraum von M_r traversiert werden, da der Zustandsraum der parallelen Ausführung mit $M_a^c \parallel M_r$ zu groß ist, um vollständig alle Eigenschaften zu überprüfen.

Um dieses Problem zu lösen, erstellen wir eine Serie von Abstraktionen M_a^i von M_r . Es handelt sich hierbei um sichere Abstraktionen (siehe Definition 41), um entsprechend zuverlässige Verifikationsergebnisse zu ermöglichen. Die Abstraktionen werden nach und nach akkurater, so dass letztendlich gezeigt werden kann, dass entweder die Integration korrekt ist oder ein entsprechendes Gegenbeispiel gefunden wurde. Dabei ist immer die folgende Relation erfüllt:

$$M_r \sqsubseteq M_a^i \quad (\forall i \geq 0). \tag{4.6}$$

Unser Ansatz startet mit der initialen Synthese eines Modells der Altcomponente basierend auf den bekannten Schnittstellen-Informationen. Wir gehen dabei davon aus, dass die ausgetauschten Nachrichten entweder direkt zueinander passen oder eine Abbildung, die z.B. den Namensraum oder auch die Typen aufeinander abbildet, bekannt ist.

In einem ersten Schritt erstellen wir einfach M_a^0 basierend auf den bekannten Informationen der Schnittstelle von M_r . M_a^0 wird erstellt durch Bestimmung des initialen Zustands s_0 von M_r und durch Herleitung eines Automaten: $M_a^0 = (\{s_0, I, I, \emptyset, \{s_0\}\})$. Wie in Paragraph Anforderungen

und Voraussetzungen auf Seite 98 beschrieben, setzen wir voraus, dass die Altkomponente Initial in einem wohldefinierten Zustand ist.

Unter Ausnutzung der chaotischen Hülle (siehe Definition 41) leiten wir eine erste Approximation her: $M_a^0 = \text{chaos}(M_l^0)$. Mit Theorem 2 stellen wir sicher, dass M_a^0 eine sichere Abstraktion von M_r ist ($M_r \sqsubseteq M_a^0$).

Lemma 5

Für das initiale Modell $M_a^0 = \text{chaos}(M_l^0)$ für M_l^0 erstellt für den initialen Zustand s_0 von M_r ausgedrückt durch den Automat $M_l^0 = (\{s_0, I, I, \emptyset, \{s_0\}\})$ gilt $M_r \sqsubseteq M_a^0$.

Beweis 7

Aufgrund des Theorems 2 können wir folgern, dass M_a^0 eine sichere Abstraktion von M_r ist, da M_l^0 beobachtungs-konform zu M_r ist. \square

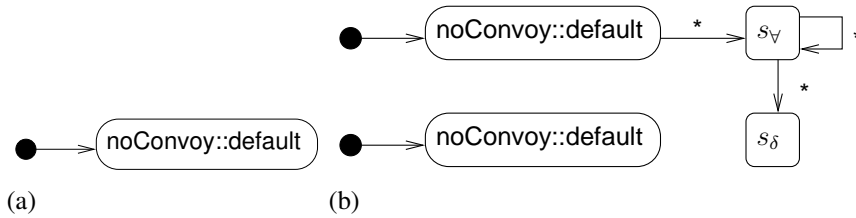


Abbildung 4.4: Trivialer initialer Automat, der den bekannten initialen Zustand berücksichtigt (4.4(a)) und das initiale Verhalten einer Altkomponente (4.4(b))

In Abbildung 4.4(a) wird der initiale, triviale Automat dargestellt. Der Automat besteht aus einem initialen Zustand noConvoy::default.

Der resultierende Automat nach Anwendung der chaotischen Hülle auf den trivialen unvollständigen Automaten dargestellt in Abbildung 4.4(a), welcher nur den bekannten initialen Zustand noConvoy::default darstellt, ist in Abbildung 4.4(b) dargestellt. Die Abbildung verdeutlicht, dass der initiale Zustand verdoppelt wurde und dass einer dieser Zustände verbunden ist mit jeglichen möglichen Kommunikationen durch die beiden chaotischen Zustände s_{\forall} and s_{δ} . Wie bereits weiter oben erläutert, stellt '*' alle möglichen Eingabe- und Ausgabekombination dar.

Abbildung 4.5 zeigt das bekannte Kontextverhalten, die front-Rolle, die wir im Folgenden für die iterative Verhaltenssynthese berücksichtigen. Da der hier beschriebene Gray-Box-Checking-Ansatz aus Vereinfachungsgründen nicht für hierarchisches Kontextverhalten definiert wurde, beschreibt 4.5 im Wesentlichen nur eine flache Repräsentation des front-Rollenverhaltens (siehe Abschnitt 2.4.2). Dies stellt keine Einschränkung für die Einsatzmöglichkeit des Ansatzes dar. Die Automaten können lediglich komplexer werden, da Hierarchien durch mehrere Zustände und Transitionen flach ausgedrückt werden.

Der Startzustand des Statecharts ist der noConvoy-Zustand. Der Automat verweilt in diesem Zustand bis eine convoyProposal-Nachricht empfangen wird. Danach wird in den answer-Zustand gewechselt. In diesem Zustand entscheidet der Automat nichtdeterministisch die Konvoianfrage

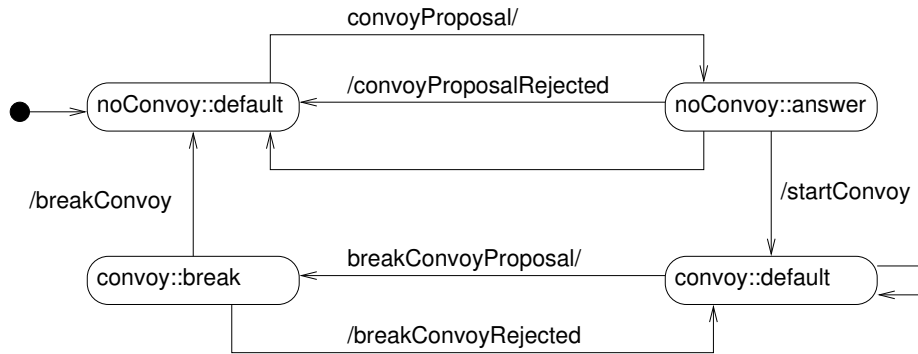


Abbildung 4.5: Bekanntes Kontextverhalten

abzulehnen (`convoyProposalRejected`) oder den Konvoi zu starten (`startConvoy`). Ist letzteres der Fall, schaltet der Automat in den `convoy`-Zustand und verweilt dort, so lange eine `breakConvoyProposal`-Nachricht empfangen wird. Der Automat entscheidet dann wieder nichtdeterministisch diese Anfrage anzunehmen oder abzulehnen.

4.1.3 Iterative Verhaltenssynthese

Auf Basis der initialen Verhaltenssynthese (siehe Abschnitt 4.1.2), beschreiben wir in diesem Abschnitt unseren iterativen Verhaltenssynthese-Ansatz. Als erstes überprüfen wir, ob die gegebenen Bedingungen für das initial synthetisierte Verhalten erfüllt sind. Ist dies nicht der Fall, testen wir die Altkomponente basierend auf dem Gegenbeispiel, welches die Verletzung der Bedingung aufzeigt. Während der Testdurchführung beobachten wir die Altkomponente. Wenn das Gegenbeispiel durch die Altkomponente bestätigt wird, ist die Integration fehlgeschlagen. Andernfalls verwenden wir den beobachteten Trace, um das Verhalten zu erlernen. Das neu synthetisierte Verhalten ist dann der Startpunkt für die nächste Iteration.

Formaler Verifikationsschritt Die iterative Verhaltenssynthese startet mit der Überprüfung der hergeleiteten Abstraktion aus der initialen Verhaltenssynthese (siehe Abschnitt 4.1.2). Überprüft wird, ob ein Gegenbeispiel für die geforderte Bedingung ϕ existiert. Wir überprüfen daher für $i \geq 0$

$$M_a^c \parallel M_a^i \models \phi \wedge \neg \delta. \quad (4.7)$$

Wenn die Überprüfung erfolgreich ist, haben wir tatsächlich bewiesen, dass die Bedingungen auch für $M_a^c \parallel M_r$ und $M_r^c \parallel M_r$ gelten müssen.

Lemma 6

Gegeben sei ein konkreter Kontext M_r^c mit abstraktem Modell M_a^c und eine konkrete Altkomponente M_r mit hergeleiteter Abstraktion M_a^i , so dass der konkrete Kontext eine Verfeinerung des

abstrakten Kontextes ist $(M_r^c \sqsubseteq M_a^c)$ und das M_a^i eine gültige Abstraktion der Altkomponente M_r ist $(M_r \sqsubseteq M_a^i)$, dann gilt für jede kompositionelle Bedingung ϕ :

$$M_a^c \parallel M_a^i \models \phi \quad \Rightarrow \quad M_r^c \parallel M_r \models \phi. \quad (4.8)$$

Beweis 8

Wie in Abschnitt 4.1.1 gezeigt, bleibt eine Verfeinerung (\sqsubseteq) durch eine parallele Komposition (\parallel) erhalten. Wenn also $M_r^c \sqsubseteq M_a^c$ gilt, dann können wir daraus schließen, dass $M_r^c \parallel M_a^i \sqsubseteq M_a^c \parallel M_a^i$ ebenso gilt. Gilt $M_r \sqsubseteq M_a^i$, dann können wir damit ebenso folgern, dass $M_r^c \parallel M_r \sqsubseteq M_a^c \parallel M_a^i$ gilt. Da die Verfeinerung die Bedingung ϕ erhält, können wir aus $M_a^c \parallel M_a^i \models \phi$ folgern, dass $M_r^c \parallel M_r \models \phi$ ebenfalls gilt. \square

Wenn die Überprüfung allerdings nicht erfolgreich ist, wird ein Gegenbeispiel erstellt. Das Gegenbeispiel ist ein Pfad π für $M_a^c \parallel M_a^i$, welcher eine Verletzung aufzeigt, dass ϕ nicht für die Abstraktion erfüllt ist. Dieses Gegenbeispiel eingeschränkt auf M_a^i wird benutzt, um die Altkomponente zu testen.

Beobachtung 4.1 zeigt das Gegenbeispiel der Überprüfung des initialen chaotischen Verhaltens. Das Gegenbeispiel ist relativ lang bezogen auf die Größe des initialen Automaten. Als erstes schickt die chaotische Hülle eine `convoyProposal`-Nachricht dem Kontext zu. Danach schickt der Kontext eine `convoyProposalReject`-Nachricht. Dann schickt die Hülle wieder eine `convoyProposal`-Nachricht und der Kontext entscheidet sich einen Konvoi zu erstellen, in dem er eine `startConvoy`-Nachricht verschickt. Nachdem der Konvoi erstellt wurde, versucht der Kontext den Konvoi aufzulösen, während die Hülle in den s_δ Zustand wechselt und damit ein Deadlock manifestiert ist.

Beobachtung 4.1: Initiales Gegenbeispiel

```

1  rcl.noConvoy, legacyRC.s_all,
2  legacyRC.convoyProposal!, shuttle1.convoyProposal?
3  rcl.answer, legacyRC.wait,
4  rcl.convoyProposalRejected!, legacyRC.convoyProposalRejected?
5  rcl.noConvoy, legacyRC.s_all
6  legacyRC.convoyProposal!, shuttle1.convoyProposal?
7  rcl.answer, legacyRC.wait
8  rcl.startConvoy!, legacyRC.startConvoy?
9  rcl.convoy, legacyRC.s_all
10 legacyRC.breakConvoyProposal!, rcl.breakConvoyProposal?
11 rcl.break, legacyRC.s_delta
```

Testschritt Wenn der Test der Altkomponente mit dem Gegenbeispiel aufdeckt, dass der Pfad π auch in der Altkomponente möglich ist, können wir daraus schließen, dass wir einen Fehler in der Integration gefunden haben.

Lemma 7

Gegeben sei ein konkreter Kontext M_r^c mit abstrakten Modell M_a^c und eine Alt Komponente M_r mit abgeleiteter Abstraktion M_a^i , so dass der konkrete Kontext den abstrakten verfeinert ($M_r^c \sqsubseteq M_a^c$) und dass die Abstraktion der Alt Komponente gültig ist ($M_r \sqsubseteq M_a^0$), dann gilt:

$$\left(M_a^c \parallel M_a^i, \pi \not\models \phi \quad \wedge \quad \pi \in M_r^c \parallel M_r \right) \Rightarrow M_r^c \parallel M_r \not\models \phi \quad (4.9)$$

Beweis 9

Da π eine Verletzung von $\neg\phi$ nachweist und ϕ ein Ausführungspfad von $M_r^c \parallel M_r$ ist, können wir daraus folgern, dass $M_r^c \parallel M_r \not\models \phi$ gilt. \square

Im Folgenden wenden wir unsere Vereinfachungen an, um die Bedingungen P' anzupassen, anstatt diese über die chaotische Hülle auszudrücken, welche alle möglichen unterschiedlichen Teilmengen der atomaren Bedingungen P unterscheiden würde (siehe Definition 40). Unter Anwendung dieser Vereinfachung müssen wir nur ϕ auf $M_r^c \parallel M_r, \pi$ auswerten, um zu überprüfen, dass das Gegenbeispiel durch die Alt Komponente bestätigt wird. Dies kann nur passieren, wenn π Zustände in der chaotischen Hülle (s_\forall or s_δ) aufsucht.

Daher ist garantiert, dass in diesen Fällen π nicht ein realer Ausführungspfad von $M_r^c \parallel M_r$ ist, da die konkreten Zustände niemals Zustände der chaotischen Hülle beinhalten. Dies ist notwendig, da ansonsten durch die künstlichen Deadlocks ermöglicht durch die chaotische Hülle, um Verhalten zu lernen, keine sichere Abstraktion gewährleistet wäre.

In dem hier betrachteten Fall von Gray Box Checking, wo wir die konkreten Zustände der Alt Komponente beobachten können, können wir davon ausgehen, dass für Ausführungspfade die Kodierung (s, i) mit $i \in \{0, 1\}$ äquivalent zu einem Zustand s sind (siehe Definition 41). Damit sind Ausführungspfade, die nur diese Zustände besuchen, auf Ausführungspfade der Alt Komponente abbildbar. Diese Ausführungspfade können damit reale Gegenbeispiele auffinden.

Wenn der Ausführungspfad nicht durch Testen der Alt Komponente bestätigt wird, nutzen wir die beobachtete Differenz zwischen π und der Beobachtung π' der Alt Komponente, um ein verbessertes M_a^{i+1} herzuleiten.

Wenn wir die Alt Komponente in unserem Beispiel basierend auf dem Gegenbeispiel in Beobachtung 4.1 mit Hilfe der Techniken beschrieben in Abschnitt 6.1.1 testen, beobachten wir den Pfad dargestellt in Beobachtung 4.2.

Die in Abschnitt 6.1.1 beschriebenen Techniken ermöglichen uns während der Testausführung einer Alt Komponente nur die relevanten Ereignisse aufzunehmen, die Notwendig für eine deterministische Wiederholung sind. Dies ist für die betrachteten Systeme wichtig, da hiermit eine gleichbleibende Instrumentierung realisiert wird, die einen sogenannten Probe-Effekt für Echtzeitsysteme verhindern kann.

Ein Probe-Effekt kann zu unterschiedlichen Verhalten während der Test und realen Ausführung der Alt Komponente führen, da sich durch die Instrumentierung z.B. das zeitliche Verhalten verändern kann. Unser Ansatz ermöglicht eine deterministische Wiederholung auf Basis der Be-

obachtung der ausgehenden Nachricht `convoyProposal` an Port `rearRole` und der eingehenden Nachricht `convoyProposalRejected` am gleichen Port.

Wenn wir detaillierter das Verhalten der Altkomponente während der deterministischen Wiederholung beobachten, mit allen relevanten Instrumentierungen für die Beobachtung der Zustände und Zeit, zeigt der Pfad einen Konflikt mit dem erwarteten Verhalten basierend auf dem initialen Gegenbeispiel (siehe Beobachtung 4.3).

Im nächsten Abschnitt werden wir zeigen, wie wir einen Konflikt manifestieren, während wir das synthetisierte Verhalten basierend auf den beobachteten Pfaden überprüfen.

Beobachtung 4.2: Beobachtete relevante Ereignisse für die deterministische Wiederholung: blockierender Zustand

-
- 1 [Message] name="convoyProposal", portName="rearRole", type="outgoing"
 - 2 [Message] name="convoyProposalRejected", portName="rearRole", type=incoming
-

Beobachtung 4.3: Beobachtung aller relevanter Ereignisse: blockierender Zustand

-
- 1 [CurrentState] name="noConvoy"
 - 2 [Message] name="convoyProposal", portName="rearRole", type="outgoing"
 - 3 [Timing] count=1
 - 4 [CurrentState] name="convoy",
 - 5 [Message] name="convoyProposalRejected", portName="rearRole", type=incoming
-

Lernschritt In dem hier gezeigten Lernschritt wenden wir die beobachtete Differenz zwischen π und dem beobachteten Verhalten der Altkomponente π' an, um ein verbessertes M_l^{i+1} herzuleiten. M_a^{i+1} wird wieder aus $\text{chaos}(M_l^{i+1})$ hergeleitet. Durch Theorem 2 gilt per Konstruktion:

$$M_r \sqsubseteq M_a^{i+1}. \quad (4.10)$$

Dies gilt, da π' ein beobachtbares Verhalten von M_r ist und alle weiteren Verhalten von M_l^{i+1} auch in M_l^i vorhanden sind.

Für das Lernen können wir zwei Schritte unterscheiden. Zum einen kann ein noch nicht beobachtetes Verhalten π' aufgenommen worden sein. Dann können wir wie folgt vorgehen:

Definition 43 (Lernen)

Gegeben sei ein unvollständiger deterministischer Automat $M = (S, I, O, T, \bar{T}, Q)$ und ein regulärer Ausführungspfad π , dann können wir den deterministischen unvollständigen Automaten $M' = (S', I, O, T', \bar{T}', Q')$ herleiten, welcher sich aus dem Lernen von π (beschrieben durch $\text{learn}(M, \pi)$) wie folgt ergibt: $S' = S \cup \{s \notin S \mid \pi = \dots s \dots\}$, $T' = T \cup \{(s, A, B, s') \notin T \mid \pi = \dots s(A, B)s' \dots\}$, und $Q' = Q \cup \{s \notin Q \mid \pi = s \dots\}$.

Zum anderen können wir den Fall betrachten, dass der Test blockiert wird. In diesem Fall haben wir einen Deadlock-Ausführungspfad π der Form $\dots s(A, B)$ mit (A, B) wurde in Zustand s blockiert. Das Lernen wird dann wie folgt ermöglicht:

Definition 44 (Lernen - Deadlock)

Gegeben sei ein deterministischer unvollständiger Automat $M = (S, I, O, T, \bar{T}, Q)$ und ein Deadlock-Ausführungspfad $\pi = \dots s(A, B)$, wobei die letzte Kommunikation blockierend ist. Wir leiten dann den deterministischen unvollständigen Automaten $M' = (S, I, O, T, \bar{T}', Q)$ her. Dieser resultiert aus dem Lernen auf Basis von π (bezeichnet durch $\text{learn}(M, \pi)$) wie folgt: $\bar{T}' = \bar{T} \cup \{(s, A, B)\}$.

In beiden Fällen ist das erlernte Verhalten eine sichere Abstraktion, wie im folgenden Lemma beschrieben.

Lemma 8

Gegeben sei ein konkreter Kontext M_r^c mit abstraktem Modell M_a^c und einer konkreten Alt Komponente M_r mit abgeleiteter Abstraktion M_a^i . Weiterhin sei der konkrete Kontext eine Verfeinerung des abstrakten Kontext ($M_r^c \sqsubseteq M_a^c$) und das erlernte Verhalten sei gültig (M_a^0 ist beobachtungs-konform zu M_r), dann gilt für jeden möglichen Ausführungspfad π von $M_r^c \parallel M_r$:

$$M_r \sqsubseteq M_a^{i+1} \text{ für } M_a^{i+1} = \text{chaos}(\text{learn}(M_a^i, \pi)). \tag{4.11}$$

Beweis 10

Es folgt aus der Konstruktion, dass $\text{learn}(M_a^i, \pi)$ wie M_a^i beobachtungs-konform zu M_r ist. Durch Theorem 2 folgt die Verfeinerung für $\text{chaos}(\text{learn}(M_a^i, \pi'))$. \square

Um in der Lage zu sein einen Pfad zu erstellen, der die Abstraktion verbessert, nutzen wir aus, dass die Implementierung M_r deterministisch ist, während M_a^i möglicherweise Nichtdeterminismen beinhaltet. Dies ist keine Einschränkung für sicherheitskritische Systeme, da hier Nichtdeterminismen oder pseudo Nichtdeterminismen nicht erlaubt sind. Ansonsten würden wichtige Eigenschaften dieser Systeme, wie Vorhersagbarkeit, verletzt werden.

Bezogen auf unser Beispiel, zeigt Abbildung 4.6 das synthetisierte Verhalten. Zuerst ist die Alt Komponente in Zustand `noConvoy::default`. Nachdem die Nachricht `convoyProposal` verschickt wurde, wechselt die Alt Komponente in Zustand `noConvoy::wait`.

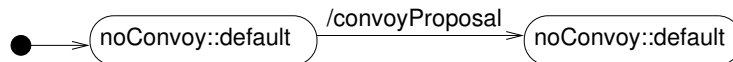


Abbildung 4.6: Synthetisiertes Verhalten: Konflikt mit der Umgebung

Mehrfache Iterationen Mit dem in den vorherigen Paragraphen vorgestellten Vorgehen, können wir systematisch eine Serie von Abstraktion $M_a^0, M_a^1, \dots, M_a^n$ herleiten, so dass wir schrittweise unser Wissen über die Altkomponente M_r verbessern. Im Unterschied zu bisherigen Lernansätzen garantiert die Serie von Abstraktion immer eine gültige Verfeinerung, so dass unser Verfahren terminiert, sobald wir ein erstes n gefunden haben mit $M_a^c \| M_a^n \models \phi$. Diese Aussage impliziert, dass ϕ zudem für das reale System, die Altkomponente, gilt ($M_r^c \| M_r \models \phi$). Wenn, im Gegensatz, wir ein n erreichen, für welches das zugehörige Gegenbeispiel π_n zudem in der realen Implementierung $M_r^c \| M_r$ festgestellt werden kann, weist das Gegenbeispiel auf einen realen Konflikt in der Integration hin.

Theorem 3

Gegeben sei ein konkreter Kontext M_r^c mit abstraktem Modell M_a^c , so dass der konkrete Kontext eine Verfeinerung des abstrakten ist ($M_r^c \sqsubseteq M_a^c$) und eine konkrete Altkomponente M_r mit einer Reihe von hergeleiteten Abstraktionen $\{M_a^i | 0 \leq i \leq n\}$. Diese Abstraktionen seien konstruiert, wie unter Lemma 8 vorgestellt. Hieraus können wir entscheiden, ob eine Bedingung ϕ für $M_r^c \| M_r$ gilt oder wir können die Serie der Abstraktionen fortsetzen.

Beweis 11

Wir können via Induktion zeigen, dass M_i^i beobachtungs-konform zu $M_r, \forall 0 \leq i \leq n$ ist. Der erste Schritt der Induktion ist (Induktionsanfang): Lemma 5 garantiert, dass wir immer wenigstens ein erstes Element M_1^0 in der Serie der Abstraktion finden. Daher ist die Bedingung für $n = 0$ gegeben. Der Induktionsschritt wird mit Hilfe von Lemma 8 gezeigt. Es gilt, dass wenn eine Serie von Abstraktionen für i fortgesetzt werden kann, Lemma 8 garantiert, dass die Bedingungen auch für $i + 1$ gelten.

Wenn wir die Serie nicht fortsetzen können, dann haben wir entweder bewiesen, dass ϕ für $M_a^c \| M_a^n$ gilt oder wir haben gezeigt, dass das Gegenbeispiel π_n auch in $M_r^c \| M_r$ enthalten ist. Durch Lemma 6 haben wir bewiesen, dass die Bedingung ϕ für $M_r^c \| M_r$ gilt. Lemma 7 garantiert zudem, dass die Bedingung ϕ stets durch $M_r^c \| M_r$ verletzt wird.

Daher garantiert unser Ansatz entweder die Reihe von Abstraktionen fortzusetzen oder wir haben einen Beweis, dass die Bedingung ϕ erfüllt oder nicht erfüllt ist. \square

Für Altkomponenten mit endlich vielen Zuständen, können wir zudem garantieren, dass der Lernprozess terminiert. Gehen wir also davon aus, dass die Altkomponente eine endliche Anzahl an Zuständen und Transitionen hat sowie deterministisches Verhalten aufweist. Hierfür können wir zeigen, dass jedes Mal wenn wir ein Gegenbeispiel nicht in der Testphase beobachten können, chaotisches Verhalten durch vorher unbekannte Zustände und Transitionen ersetzt werden. Daher ist die Anzahl der noch nicht bekannten Zustände und Transitionen mit jeder Iterationsrunde strikt monoton abnehmend. Da die Anzahl nicht kleiner Null sein kann, ist eine Terminierung garantiert.

Im Folgenden zeigen wir einen weiteren Iterationsschritt an unserem Beispiel. Basierend auf dem synthetisierten Verhalten gezeigt in Abbildung 4.6, berechnen wir eine chaotische Hülle und überprüfen diese mit dem Kontext. Beobachtung 4.4 zeigt das Gegenbeispiel. Die Bedingung $A[\text{not}(\text{rearRole.Convoy and frontRole.noConvoy})]$ ist verletzt. Der Pfad zeigt, dass die Verletzung

nur im synthetisierten Teil des Modells ist. Damit haben wir bewiesen, dass die Altkomponente im Konflikt mit dem Kontext ist. Darüber hinaus zeigt dieses Beispiel, dass unser Ansatz in der Lage ist, schnell, nach wenigen Iterationsschritten, einen Fehler in der Integration aufzudecken.

Beobachtung 4.4: Gegenbeispiel mit Konflikt im synthetisierten Verhalten

```
1 shuttle1.noConvoy, shuttle2.noConvoy
2 shuttle2.convoyProposal!, shuttle1.convoyProposal?
3 shuttle1.answer, shuttle2.convoy
```

Der Ansatz unterstützt neben einer schnellen Konflikterkennung zudem eine systematische und gleichzeitig automatische Vorgehensweise, um alle relevanten Eingabekombinationen in Bezug auf den Kontext und den gestellten Bedingungen zu testen. Die Eingabe für das Testen ist die gleiche, wie durch das Gegenbeispiel dargestellt in Beobachtung 4.5. Der Beobachtungspfad zeigt, dass alle Kommunikationen durch die Altkomponente ausgeführt wurden, wie durch die Testeingabe erwartet. Das in Abbildung 4.7 gezeigte synthetisierte Verhalten bestätigt diese Beobachtung. Die Überprüfung des synthetisierten Verhaltens zusammen mit der chaotischen Hülle manifestiert einen Deadlock in der chaotischen Hülle und nicht nur in dem synthetisierten Verhaltensteil. Daher können wir das Gegenbeispiel in der nächsten Iteration für Testeingaben ausnutzen.

Beobachtung 4.5: Erfolgreicher Lernschritt: Beobachtung aller relevanter Ereignisse

```
1 [CurrentState] name="noConvoy::default"
2 [Message] name="convoyProposal", portName="rearRole", type="outgoing"
3 [Timing] count=1
4 [CurrentState] name="noConvoy::wait"
5 [Message] name="convoyProposalRejected", portName="rearRole", type=
  incoming
6 [Timing] count=2
7 [CurrentState] name="noConvoy"
8 [Message] name="convoyProposal", portName="rearRole", type="outgoing"
9 [Timing] count=3
10 [CurrentState] name="noConvoy::wait"
11 [Message] name="startConvoy", portName="rearRole", type=incoming
12 [Timing] count=4
13 [CurrentState] name="convoy"
```

4.2 Black Box Checking

Im vorherigen Abschnitt haben wir gezeigt, wie wir für Altkomponenten ein Verhalten iterativ erlernen und überprüfen können, wenn der aktuelle Zustand des Systems beobachtbar ist. Generell können wir nicht voraussetzen, dass der aktuelle Zustand des Systems beobachtbar ist. Wie

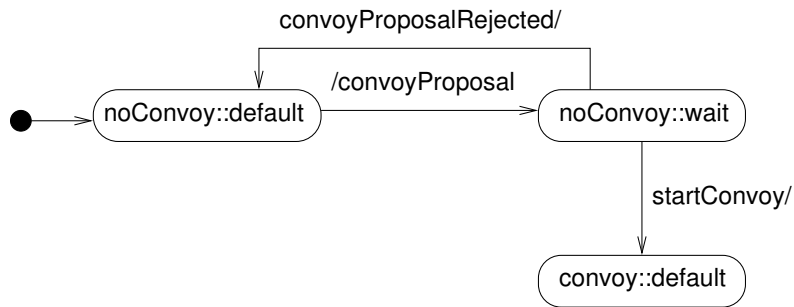


Abbildung 4.7: Korrekt synthetisiertes Verhalten in Bezug auf den Kontext

in Paragraph Anforderungen und Voraussetzungen auf Seite 98 erläutert, betrachten wir daher in diesem Abschnitt zudem den Fall, dass der Zustand nicht beobachtbar ist und auch nicht auf den Quellcode zurückgegriffen werden kann. Im Folgenden Abschnitt 4.3 betrachten wir den Fall, dass wir auf den Quellcode zugreifen können.

Wie unter anderem in Abschnitt 4.1.1 erläutert, können wir die betrachtete Klasse an Altkomponenten auf deterministische Systeme eingrenzen. Wie in Definition 33 gezeigt, müssen wir daher einen deterministischen Automaten der Altkomponente erlernen.

In Abschnitt 4.1 haben wir diskutiert, dass im Allgemeinen Lernalgorithmen für deterministische endliche Automaten nicht für die hier betrachteten sicherheitskritischen Systeme geeignet sind, da diese keine Korrektheitsaussagen treffen können. Wir zeigen im Folgenden, wie wir den L^* Algorithmus von Angluin [Ang87] erweitern können, damit dies doch ermöglicht wird.

Wir werden im Folgenden zeigen wie wir eine kompositionelle Analyse für die Betrachtung von Black-Box-Komponenten durchführen können. Generelle Idee ist dabei das Verfahren von [Ang87] für die Domäne mechatronischer Systeme anzupassen. Das heißt insbesondere, dass wir ein- und ausgehende Nachrichten, Kontextverhalten und Zeit berücksichtigen.

Der Ansatz erweitert im Wesentlichen den Gray-Box-Checking-Ansatz um das zusätzliche Erlernen eines Zustands. Dazu führen wir nach [Ang87] so genannte *Zugehörigkeitsanfragen* (engl. *Membership Queries*) und *Äquivalenzanfragen* (engl. *Equivalence Queries*) ein, die wir im Folgenden Abschnitt näher erläutern. Weiterhin verwenden wir einen Konformitätstest nach Vasilevskii und Chow [Vas73, Cho78], der eine Korrektheitsaussage einer Äquivalenzanfrage unterstützt. Der Black-Box-Checking-Ansatz besteht damit aus folgenden Schritten (siehe Abbildung 4.8):

1. lernen eines Kandidaten des Protokollverhaltens der Altkomponente (mit erweitertem Angluin Ansatz),
2. kompositionelle Überprüfung des Kandidaten aus 1. unter Berücksichtigung des Kontextverhaltens und Sicherheits-/Lebendigkeitseigenschaften an die Integration.
 - a) Wenn aus 2. ein Gegenbeispiel erfolgt, dann wird überprüft, ob dieses Gegenbeispiel durch die Altkomponente bestätigt wird. Ist dies der Fall, ist die Integration fehler-

haft. (Das Verhalten kann und sollte trotzdem vollständig erlernt werden, um eine Anpassung des Kontextes zu ermöglichen.). Ist dies nicht der Fall, wird mit 1. unter Berücksichtigung des Gegenbeispiels fortgefahren.

b) Folgt aus 2. kein Gegenbeispiel, dann wird überprüft, ob das erlernte Verhalten äquivalent zur Altkomponente ist (Konformitätstest mit Vasilevskii und Chow [Vas73, Cho78]).

3. Folgt aus der Äquivalenzprüfung (2.(b)), dass der Automat äquivalent ist, ist die Lernphase abgeschlossen und die Integration korrekt. Ist dies nicht der Fall, wird auf Basis der Ergebnisse der Äquivalenzprüfung mit 1. fortgefahren.

Der Black-Box-Checking-Ansatz führt damit, wie der Gray-Box-Checking-Ansatz ebenfalls, iterativ ein Lernen und Überprüfen durch. Der wesentliche Unterschied ist, dass zum einen ein Lernansatz integriert werden muss, um auch Zustände zu identifizieren, die im Gray-Box-Checking-Ansatz an der Schnittstelle beobachtbar waren. Ein weiterer Unterschied ist der Konformitätstest, der eine Äquivalenz zwischen erlernten Verhalten und Altkomponente durchführt.

In [HNS03b] wurde der Ansatz von Angluin um Mealy-Automaten erweitert. Entsprechend ist die grundlegende Idee, die Eingaben und Ausgaben auf einen akzeptierenden Automaten abzubilden nicht neu. Zeit, Sicherheits- und Lebendigkeitseigenschaften sowie Kontextverhalten werden durch den Ansatz allerdings nicht betrachtet.

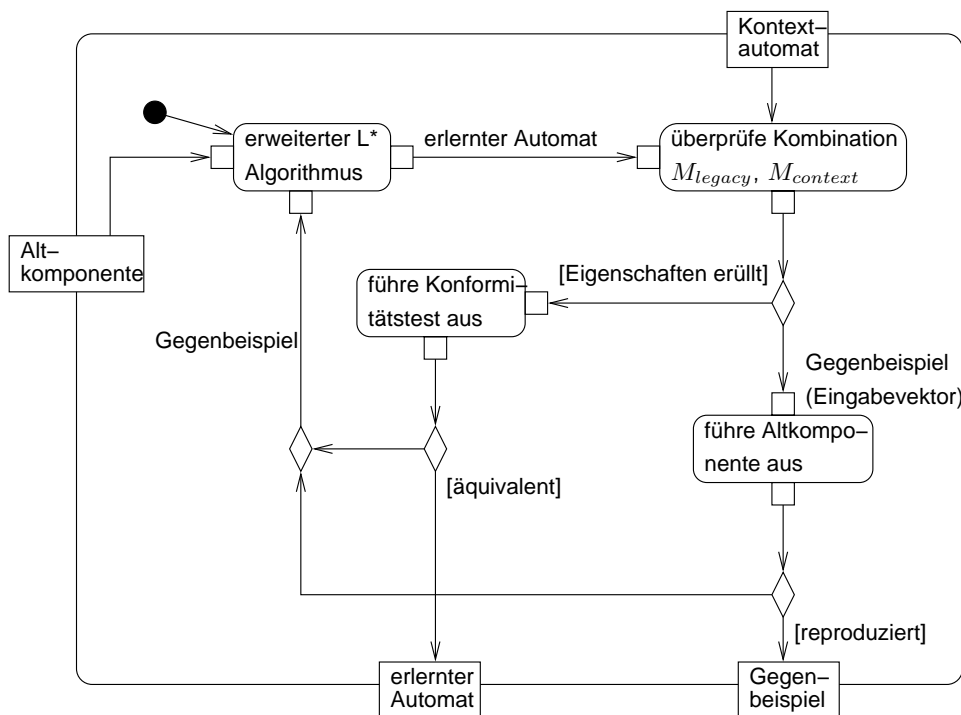


Abbildung 4.8: Iteratives Lernen und Überprüfen: Black Box Checking

4.2.1 L* Lernalgorithmus

Ein Problem vieler Lernalgorithmen ist, dass diese selbst für einen deterministischen endlichen Automaten nicht polynomielle Laufzeiten aufweisen, wie Gold in [Gol78] gezeigt hat. Pitt [Pit89] hat zudem gezeigt, dass diese Klasse von Automaten auch nicht in polynomieller Laufzeit lösbar ist, wenn Äquivalenzanfragen benutzt werden. Äquivalenzanfragen nehmen dem Lernalgorithmus die Überprüfung ab, ob der erlernte Automat äquivalent zur Black Box (Alt Komponente) ist. Erst mit der Einführung von zusätzlichen Zugehörigkeitsanfragen, wie dies Angluin [Ang87] gezeigt hat, können Algorithmen in polynomieller Laufzeit für das Lernen eines deterministischen endlichen Automaten beschrieben werden.

In [Ang87] beschreibt Angluin einen Algorithmus, der einen Automaten zu einer regulären Sprache mit Hilfe von Anfragen und Gegenbeispielen konstruiert. Dieser Algorithmus wird unter dem Namen L^* eingeführt. Ausgabe dieses Ansatzes ist ein minimaler deterministischer endlicher Automat.

Voraussetzung für den Algorithmus ist die Möglichkeit Anfragen stellen zu können. Es muss also geprüft werden können, ob ein vom Algorithmus gegebenes Wort in der Sprache der Alt Komponente enthalten ist. Des Weiteren muss eine Äquivalenzprüfung durchgeführt werden können. Dabei muss geprüft werden, ob die Sprache, die durch einen vorläufigen erlernten Automaten, auch *Kandidat* genannt, repräsentiert wird, äquivalent zu der gesuchten Sprache ist. Sollte dies nicht der Fall sein, so muss ein Gegenbeispiel zurückgeliefert werden. Im Allgemeinen wird das Erlernen eines deterministischen endlichen Automaten aus einer Alt Komponente mit regulärer Inferenz bezeichnet.

Reguläre Inferenz Reguläre Inferenz betrachtet das System als Black Box. Es wird angenommen, dass die betrachtete Black-Box-Alt Komponente durch einen endlichen deterministischen Automaten modelliert werden kann. Das sich daraus ergebende Problem ist die Identifizierung der regulären Sprache $\mathcal{L}(M)$ der Alt Komponente M .

Ein *Lerner*, der zu Beginn nur das Alphabet Σ^* von M kennt, versucht die Sprache $\mathcal{L}(M)$ dadurch zu erlernen, dass er Anfragen an einen *Lehrer* und an ein *Orakel* stellt. $\mathcal{L}(M)$ wird durch Zugehörigkeitsanfragen (membership queries) an den Lehrer erlernt. Es wird überprüft, ob eine Zeichenkette $w \in \Sigma^*$ in $\mathcal{L}(M)$ enthalten ist.

Weiterhin ist zum Erlernen der Sprache noch eine *Äquivalenzanfrage* (*equivalence query*) notwendig. Diese fragt das *Orakel*, ob der erlernte Automat \mathcal{A} korrekt ist. Dies ist der Fall, wenn $\mathcal{L}(\mathcal{A}) = \mathcal{L}(M)$. Ansonsten wird ein Gegenbeispiel gegeben. Typischerweise fragt der *Lerner* eine Sequenz von *Zugehörigkeitsanfragen* und nutzt die Antworten, um einen vermuteten Automaten (*Kandidat*) zu erstellen. Ist der Kandidat stabil (verändert sich nicht gegenüber vorherigen Iterationen), wird eine Äquivalenzanfrage an das Orakel gestellt, um herauszufinden, ob das Verhalten dem der Black-Box-Alt Komponente entspricht. Ist dies der Fall, terminiert der Algorithmus. Andernfalls wird ein Gegenbeispiel zurückgegeben, um \mathcal{A} zurückzuweisen. In diesem

Fall werden weitere *Zugehörigkeitsanfragen* gestellt, bis der nächste mögliche Automat erstellt ist und so weiter.

Der reguläre Inferenzalgorithmus von Angluin organisiert die Informationen, die durch Anfragen und Antworten erlangt werden, in einer so genannten Beobachtungstabelle. Die in dieser Tabelle gespeicherten Zeichenketten bestehen dabei aus einem Präfix und einem Suffix. Die Präfixe sind Indizes für die Reihen, während die Suffixe die Spalten der Tabelle indizieren. Ein Präfix ist eine Zeichenkette, die zu einem Zustand des Systems führt. Ein Suffix wird genutzt, um die Präfixe auseinanderzuhalten, die zu verschiedenen Zuständen führen.

Zusätzlich führen wir noch den Konkatenationsoperator \cdot für Wörter ein: $S \cdot T = \{s \cdot t : s \in S, t \in T\}$.

Darüber hinaus führen wir zwei Mengen S und E ein. Die Menge S wird Präfix-Menge genannt. E heißt Suffix-Menge. S und E werden jeweils mit dem leeren Wort ($\{\epsilon\}$) initialisiert.

Nun können wir die Beobachtungstabelle an sich definieren.

Definition 45 (Beobachtungstabelle)

Wir bezeichnen eine Beobachtungstabelle mit T . Die Zeilen werden aus der Menge $S \cup S \cdot A$ gebildet, die Spalten aus E . Die Einträge der Zellen geben an, ob die Konkatenation des Zeilen- und des Spalten-Wortes in der gesuchten Sprache enthalten ist. Wir definieren also $T(s, e) = 1$ wenn $s \cdot e \in \mathcal{L}(M)$, sonst 0.

Um eine Zeile der Tabelle zu bezeichnen, führen wir noch die *row-Funktion* ein. $\text{row}(s)$ bezeichnet dabei das Tupel aus allen Werten der Tabelle in der Zeile zu s . Wenn zwei Zeilen (s_1 und s_2) die gleichen Werte enthalten gilt also $\text{row}(s_1) = \text{row}(s_2)$.

Konstruktion eines Automaten Auf Basis der Definition der Beobachtungstabelle betrachten wir im Folgenden, wie wir einen Automaten aus dieser Tabelle bilden können.

Der grundlegende Ansatz von Angluin beschreibt dies für akzeptierende Automaten, wie im Folgenden definiert. In Abschnitt 4.2.2 werden wir diesen entsprechend für die in Definition 33 beschriebenen einfachen Echtzeit-Automaten erweitern.

Definition 46 (Akzeptierender Automat)

Ein endlicher akzeptierender Automat M wird als 5-Tupel $(Q, q_0, A, \Sigma, \delta)$ definiert. Q ist eine Menge von Zuständen. $q_0 \in Q$ ist der Start-Zustand. $A \subseteq Q$ ist die Menge der akzeptierenden Zustände. Σ ist das Eingabe-Alphabet und δ ist die Übergangsfunktion mit $Q \times \Sigma \times Q$.

Unseren Automaten definieren wir nun aus der Beobachtungstabelle wie folgt:

Definition 47 (Akzeptierender Automat abgeleitet aus Beobachtungstabelle)

- $Q = \{\text{row}(s) : s \in S\}$
- $q_0 = \text{row}(\epsilon)$
- $\delta(\text{row}(s), a) = \text{row}(s \cdot a)$

$$\bullet F = \{\text{row}(s) : s \in S, T(s) = 1\}$$

Dieser Automat ist wohldefiniert, wenn die Voraussetzung erfüllt ist, dass er abgeschlossen und konsistent ist.

Abgeschlossenheit In der Übergangsfunktion wird als Funktionsterm $\text{row}(s \cdot a)$ verwendet. Nach der Definition für die Zustände $Q = \{\text{row}(s) : s \in S\}$ ist jedoch nicht garantiert, ob ein passender Zustand existiert. Es muss also ein Element $t \in S$ geben, für das die row-Funktion den gleichen Funktionswert hat wie $\text{row}(s \cdot a)$. Diese Bedingung wird Abgeschlossenheit genannt.

Definition 48 (Abgeschlossenheit)

Eine Tabelle T ist genau dann abgeschlossen, wenn gilt:

$$\forall s \in S, a \in A : \exists t \in S : \text{row}(t) = \text{row}(s \cdot a).$$

Konsistenz Aus der Definition der Übergangsfunktion folgt aber noch ein weiteres Problem. Nach den bisherigen Definitionen könnte sie für einen Kombination aus $\text{row}(s)$, a mehrere Funktionswerte haben, wenn zwei unterschiedliche Werte für s existieren, für die die row-Funktion aber den gleichen Wert ergibt. Dies folgt aus der Konkatenation mit einem beliebigen, aber festen Element a eingesetzt in die row-Funktion, die dann entsprechend zwei unterschiedliche Werte ergibt. Dann würde der zu $\text{row}(s)$ zugehörige Zustand mit der Eingabe a Transitionen auf mehrere unterschiedliche Zustände haben und wäre nicht mehr deterministisch. Wir definieren also als Bedingung, dass dieses nicht auftreten darf und nennen diese Bedingung *Konsistenz*.

Definition 49 (Konsistenz)

Eine Tabelle T ist genau dann konsistent, wenn gilt:

$$\forall s_1, s_2 \in S, a \in A : \text{row}(s_1) = \text{row}(s_2) \Rightarrow \text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a).$$

Lernansatz Der Lern-Algorithmus geht iterativ vor (siehe Algorithmus 4.1). In jedem Durchlauf wird zunächst die Konsistenz und Abgeschlossenheit sichergestellt. Dabei wird die Tabelle gegebenenfalls um neue Zeilen oder Spalten ergänzt, die mit Hilfe von Zugehörigkeitsanfragen an den Lehrer gefüllt werden.

Ist die Tabelle nicht abgeschlossen, so existiert offenbar ein $s \in S$ und ein $a \in A$, für das kein $t \in S$ existiert, für das gilt: $\text{row}(t) = \text{row}(s \cdot a)$. Um das zu verhindern, wird das Wort $s \cdot a$ zu der Menge S hinzugefügt. Damit können wir einfach $t = s \cdot a$ wählen und die Bedingung der Abgeschlossenheit ist zumindest für das gewählte s und a wieder hergestellt.

Wenn die Tabelle nicht konsistent ist, muss es nach der Definition ein $s_1 \in S$, ein $s_2 \in S$ und ein $a \in A$ geben, für die gilt: $\text{row}(s_1) = \text{row}(s_2)$, aber nicht $\text{row}(s_1 \cdot a) = \text{row}(s_2 \cdot a)$. Da sich die Zeilen unterscheiden, muss es mindestens einen unterschiedlichen Eintrag in einer Spalte geben. Das zur Spalte gehörende Suffix $e \in E$ wird ausgewählt. Damit wissen wir, dass $T(s_1 \cdot a \cdot e) \neq T(s_2 \cdot a \cdot e)$.

Nun können wir die Voraussetzung $\text{row}(s_1) = \text{row}(s_2)$ vermeiden, in dem wir die Suffix-Menge E um $a \cdot e$ erweitern. Dadurch enthalten beide Zeilen $\text{row}(s_n)$ die Werte $T(s_n \cdot a \cdot e)$, von denen wir eben gezeigt haben, dass sie ungleich sind. Damit gilt auch $\text{row}(s_1) \neq \text{row}(s_2)$ und die Konsistenz ist für diesen Fall wiederhergestellt.

Ist die Tabelle dann sowohl konsistent als auch abgeschlossen, so kann ein Automat konstruiert werden. Dieser so konstruierte Automat akzeptiert alle bereits gelernten Wörter. Da er aber noch nicht zwangsweise genau die gesuchte Sprache akzeptiert, wird eine Äquivalenzanfrage an das Orakel gestellt. Wenn das Orakel die Äquivalenz bestätigt, haben wir einen Automaten gefunden, der die Sprache akzeptiert. Andernfalls wird ein Gegenbeispiel erzeugt, um das wir die Tabelle erweitern. Dann wird mit dem nächsten Durchlauf begonnen.

Algorithmus 4.1 L^*

```
1: loop
2:   repeat
3:     if  $T$  ist nicht konsistent then
4:       Ergänze  $E$  um gefundenes  $a \cdot e$ 
5:       Fülle neue Spalten mit Hilfe des Lehrers
6:     end if
7:     if  $T$  ist nicht abgeschlossen then
8:       Ergänze  $S$  um gefundenes  $s \cdot a$ 
9:       Fülle neue Zeilen mit Hilfe des Lehrers
10:    end if
11:    until  $T$  ist abgeschlossen und konsistent
12:    Erzeuge Automat
13:    Prüfe mit dem Orakel, ob der Automat korrekt ist
14:    if Automat ist korrekt then
15:      Beende äußere Schleife und gebe den Automaten zurück
16:    else
17:      Füge Gegenbeispiel und alle seine Präfixe zu der Menge  $S$  hinzu
18:    end if
19:  end loop
```

Die Komplexität des L^* Algorithmus ist wie folgt. Die obere Grenze für die Anzahl der Äquivalenzanfragen beträgt n (n ist die Anzahl der Zustände von \mathcal{M}). Die obere Grenze für die Anzahl der Zugehörigkeitsanfragen beträgt $\mathcal{O}(|\Sigma|n^2m)$.

Beispiel Angluin Um den Algorithmus zu demonstrieren, wählen wir als ein einfaches Beispiel einen Ausschnitt des REAL-TIME COORDINATION PATTERNS Convoy aus. Wir betrachten den Fall, dass eine ConvoyRequest-Nachricht verschickt wird und auf eine Antwort gewartet wird. Das Warten wird durch eine spezielle Nachricht simuliert, wie in Abschnitt 4.2.2 vorgestellt. Das Schalten einer Transition entspricht einem Zeitschritt. Für jeden zu wartenden Zeitschritt wird eine solche Nachricht verschickt. Konkret ergibt sich damit die Sprache

$U = \text{convoyRequest}, \epsilon_t^*$. Um die Beobachtungstabellen möglichst klein zu halten ersetzen wir convoyRequest durch a und ϵ_t durch b . Das Eingabealphabet ist damit $A = \{a, b\}$. Wie oben beschrieben werden die Mengen S und E jeweils mit $\{\epsilon\}$ initialisiert². Der initiale Zustand ist also folgender:

- $A = \{a, b\}$
- $U = ab^*$
- $S = \{\epsilon\}$
- $E = \{\epsilon\}$

Die Beobachtungstabelle sieht damit wie folgt aus:

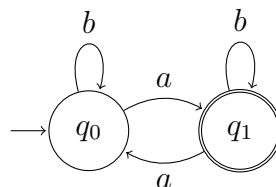
T_0	ϵ
ϵ	0
a	1
b	0

Diese Tabelle ist zwar konsistent, aber offensichtlich nicht abgeschlossen, da es für die Konkatination aus $\epsilon \cdot a$ kein Element $s \in S$ gibt, für das $\text{row}(\epsilon \cdot a) = \text{row}(s)$ gilt. Daher ergänzen wir die Menge S um eben dieses Element $\epsilon \cdot a = a$.

Damit ist nun $S = \{\epsilon, a\}$. Wenn wir die Tabelle damit erweitern und die neuen Zeilen mit Hilfe des Lehrers füllen, sieht sie wie folgt aus:

T_1	ϵ
ϵ	0
a	1
b	0
aa	0
ab	1

Diese Tabelle ist nun sowohl abgeschlossen, als auch konsistent. An dieser Stelle wird also ein vorläufiger Automat konstruiert und dann auf Äquivalenz zu der gesuchten Sprache überprüft. Der konstruierte Automat sieht wie folgt aus:



²Im Vergleich zu der bisherigen visuellen Syntax von Zuständen in dieser Arbeit (siehe z. B. Abbildung 2.5), werden wir im Folgenden, wie dies üblich für akzeptierende Automaten ist, die Zustände als Kreise und nicht als Rechtecke mit abgerundeten Ecken darstellen.

Offensichtlich ist er jedoch nicht äquivalent mit der gesuchten Sprache L . Das Orakel wird daher ein Gegenbeispiel zurückliefern. Dieses könnte z.B. das Wort ba sein, dass der Automat akzeptiert, aber nicht in der Sprache L enthalten ist.

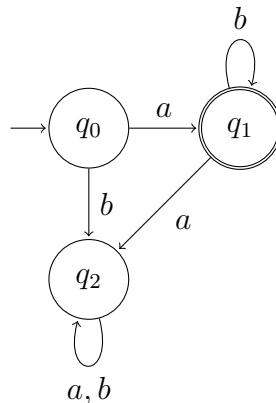
Daher erweitern wir die Menge S um das Gegenbeispiel und alle seine Präfixe, in diesem Fall $\{ba, b\}$. Nun ist also $S = \{\epsilon, a, b, ba\}$. Die neue (und wieder über den Lehrer gefüllte) Beobachtungstabelle sieht wie folgt aus:

T_2	ϵ
ϵ	0
a	1
b	0
ba	0
aa	0
ab	1
bb	0
baa	0
bab	0

Diese Tabelle ist nun zwar abgeschlossen, aber nicht konsistent. Als Gegenbeispiel können wir $\text{row}(\epsilon) = \text{row}(b)$ finden, da $\text{row}(\epsilon \cdot a) \neq \text{row}(b \cdot a)$. Der Unterschied besteht in der bisher einzigen Spalte ϵ , also ergänzen wir die Menge E um $\epsilon \cdot a \cdot \epsilon = a$. Also ist $E = \{\epsilon, a\}$. Die um die neue Spalte erweiterte Tabelle sieht nun so aus:

T_0	ϵ	a
ϵ	0	1
a	1	0
b	0	0
ba	0	0
aa	0	0
ab	1	0
bb	0	0
baa	0	0
bab	0	0

Da diese Tabelle nun wieder abgeschlossen und konsistent ist, wird wieder ein Automat erzeugt. Dieser sieht nun so aus:



Dieser akzeptiert nun genau die Sprache L , das Orakel wird also die Äquivalenz bestätigen und der Algorithmus terminiert.

4.2.2 L^* für mechatronische Systeme

Zum einen müssen wir jetzt noch zeigen, wie wir den grundlegenden Algorithmus von Angluin für einfache diskrete Echtzeit-Automaten, wie in Definition 33 beschrieben, erweitern. Zum anderen müssen wir die Zugehörigkeitsanfragen und Äquivalenzanfragen umsetzen, da diese nicht durch den Angluin Ansatz vorgegeben sind.

Erweiterung von L^* für diskrete Echtzeitautomaten Die beschriebenen diskreten Echtzeitautomaten unterscheiden sich von den akzeptierenden Automaten in der Hinsicht, dass diese Eingangs-Nachrichten I und Ausgangs-Nachrichten O empfangen, bzw. verschicken können, durch das Schalten einer Transition.

Eine Transition eines akzeptierenden Automaten ist markiert durch ein Wort aus Σ . Offensichtlich können wir daher eine Kombination aus Eingaben und den korrespondierenden Ausgaben durch ein Wort aus Σ darstellen. Entsprechend muss für jede Kombination von Eingaben und Ausgaben ein Wort spezifiziert werden.

Dies führt zum einen zu einer höheren Laufzeit des Angluin Ansatzes, da die Anzahl der Wörter durch die Kombinationen ansteigt. Aufgrund der Zeitsemantik von den diskreten Echtzeit-Automaten, wo das Schalten einer Transition einen Zeitschritt (Tick) entspricht, ist die Darstellung einer Eingabe/Ausgabe-Kombination an einer Transition für die betrachteten Echtzeitsysteme nicht realistisch, da nicht in Nullzeit Nachrichten gleichzeitig empfangen und verschickt werden können. Daher stellen wir Eingabe/Ausgabe-Kombination durch aufeinander folgende Transitionen, getrennt durch einen Zustand, mit entsprechenden Wörtern aus Σ dar, die die Eingabe- und Ausgabe-Nachrichten repräsentieren.

Um die Zeitsemantik von diskreten Echtzeit-Automaten vollständig zu berücksichtigen, führen wir zudem spezielle Leere-Wörter ϵ_t ein, die ein Zeitvergehen repräsentieren, wie wir dies Be-

reits schon in der Beispielanwendung illustriert haben (siehe Paragraph 4.2.1 auf Seite 122). Für den Lernalgorithmus führt dies ebenfalls zu keiner Veränderung, da die Wörter für den Algorithmus nicht gesondert betrachtet werden müssen. Lediglich die Testumgebung muss für jede geschaltete Periode ein Leeres Wort ϵ_t als ausgehende Nachricht der Alt Komponente erstellen. Im Folgenden definieren wir den erweiterten akzeptierenden Automaten für den Lernansatz:

Definition 50 (Erweiterter akzeptierender Automat)

Ein endlicher erweiterter akzeptierender Automat M ist ein akzeptierender Automat mit $(Q, q_0, A, \Sigma_{in}, \Sigma_{out}, \epsilon^t, \delta)$. Q, q_0 sowie A ist definiert wie in Definition 46 gezeigt. Die Menge Σ_{in} beschreibt das Eingabealphabet, Σ_{out} das Ausgabealphabet und ϵ^t ist eine spezielle Nachricht, die einen Zeittick darstellt. δ ist die Übergangsfunktion mit $Q \times (\Sigma_{in} \oplus \Sigma_{out} \oplus \epsilon^t) \times Q$.

Um den Algorithmus weiterhin terminieren zu lassen, muss eine obere Grenze für die Anzahl der zu warteten Transitionen bekannt sein. Dies lässt sich aus der bekannten Ausführungsperiode herleiten, da typischerweise pro Periode eine Transition geschaltet wird. Weiterhin müssen wir von der Alt Komponente die Information erhalten, wie lange/wie viele Perioden auf eine Antwort gewartet werden muss. Unter diesen Voraussetzungen, ermöglicht das hier vorgestellte Verfahren einen minimalen deterministischen Echtzeit-Automaten zu erlernen.

Zugehörigkeitsanfragen Wie in Abschnitt 4.2.1 dargestellt, beantwortet der Lehrer Zugehörigkeitsanfragen (-tests) an die Alt Komponente und stellt Gegenbeispiele zur Verfügung, wenn die Äquivalenz nicht gegeben ist. Zugehörigkeitsanfragen werden allgemein durch Testen des Systems realisiert. Die durch den L^* Algorithmus ermittelten Eingabefolgen dienen entsprechend als Testeingaben für die Alt Komponente.

Um die Anzahl der Zugehörigkeitsanfragen möglichst gering zu halten, können wir zudem ausnutzen, dass die betrachteten Systeme Präfix-Abgeschlossen sind. Dies gilt im Allgemeinen für reaktive Systeme, zu denen auch mechatronische Systeme gehören.

Ist daher ein Wort α in der Sprache enthalten und β ist ein Präfix von α , dann ist auch β Element der Sprache. Damit werten wir jeden Präfix eines Eintrags der Beobachtungstabelle ebenfalls mit *true* aus, wenn denn ein entsprechender Eintrag mit *true* ausgewertet wurde und damit Teil der Zielsprache ist: $\exists o' \in prefix(o) \wedge T(o) = 1 \Rightarrow T(o') = 1$. Andersherum können wir auch die Anfragen ausschließen, für die wir bereits gezeigt haben, dass ein Präfix *false* ist: $\exists o' \in prefix(o) \wedge T(o') = 0 \Rightarrow T(o) = 0$.

Wir erweitern zudem Angluin's Algorithmus um Gegenbeispiele des Model Checking, des schrittweise erlernten Verhaltens der Alt Komponente mit dem Kontext.

Wenn ein Gegenbeispiel durch die Alt Komponente bestätigt wird, ist die Integration fehlerhaft. Andernfalls nutzen wir dieses Gegenbeispiel, um den folgenden Kandidaten zu erlernen. Die Berücksichtigung des Kontextes ermöglicht uns gezielter das relevante zu erlernende Verhalten zu ermitteln und frühzeitig Fehler in der Integration zu ermitteln.

Abbildung 4.8 stellt dieses Vorgehen dar. Die Formalisierung sowie die Terminierung im Fehlerfall ist identisch zu dem in Abschnitt 4.1 vorgestellten Gray-Box-Checking-Ansatz. Womit der

Black-Box-Checking-Ansatz ebenfalls iterativ das Verhalten überprüfen kann und reale Fehler identifizieren kann.

Falls die Überprüfung des Kontextes mit dem Kandidaten der Altkomponente ergibt, dass die gestellten Sicherheits- und Lebendigkeitseigenschaften erfüllt sind, müssen wir, wie in Abschnitt 4.2.1 beschrieben, noch eine Äquivalenzprüfung durchführen. Diese beschreiben wir im nächsten Abschnitt.

Im Idealfall muss damit eine Äquivalenzprüfung nur einmal durchgeführt werden. Wir ermöglichen mit diesem Ansatz daher nicht nur eine frühe Fehlererkennung, sondern auch im Vergleich zu bisherigen Lernansätzen eine Vermeidung der Äquivalenzanfragen, die vergleichsweise zu den Zugehörigkeitsanfragen und dem Model Checking teuer ist.

Äquivalenzanfragen Das beschriebene Lernverfahren benötigt ein Orakel, welches überprüft, ob der erlernte Kandidat äquivalent zu der Black-Box-Altkomponente ist. Ist dies nicht der Fall, wird ein Gegenbeispiel angegeben.

Der Konformitätstest ist ein weitverbreiteter Ansatz, um die Äquivalenz zu überprüfen. Konformitätstests bieten ein systematisches Verfahren an, um Antworten für Äquivalenzanfragen zu ermitteln. [PVY99] oder [Ber06] sind z.B. Lernansätze die hierauf basieren.

Wie auch diese Ansätze, basieren die meisten Konformitätstest-Ansätze auf dem Verfahren von Vasilevskii und Chow [Vas73, Cho78], aufgrund der guten Laufzeiteigenschaften.

Entsprechend Vasilevskii und Chow existiert eine obere Grenze für die Gesamtlänge einer Testsequenz. Diese ist $\mathcal{O}(k^2 l |\Sigma|^{l-k+1})$. Die Laufzeit ergibt sich damit quadratisch aus der Anzahl der Zustände k , aus der Anzahl der erlernten Zustände l sowie aus der exponentiellen Differenz der Anzahl der Zustände k des Systems und des erlernten Kandidaten l über dem Alphabet.

Falls in unserem Fall das Model Checking kein Gegenbeispiel liefert, wird eine Äquivalenzanfrage mittels des Verfahrens von Vasilevskii und Chow durchgeführt. Eingabe in das Verfahren ist der Kandidat der Altkomponente $M = (S, \sigma(I/O), T, Q)$, die Altkomponente M_r sowie die obere Anzahl an erwarteten Zuständen der Altkomponente k . Der Algorithmus bestätigt entweder die Äquivalenz oder gibt ein Gegenbeispiel zurück, welches wiederum als Eingabe für den erweiterten L* Algorithmus verwendet werden kann (siehe Abbildung 4.8).

Um eine Äquivalenz zu bestimmen, wird ein Spannbaum für M und seinen korrespondierenden Pfaden π bestimmt. In einem ersten Schritt wird auf Basis eines solchen Spannbaums beginnend von dem initialen Zustand jeder Zustand durch eine abgeleitete Sequenz aus dem Spannbaum erreicht. Im Folgenden wird iterativ versucht auf Basis dieses Spannbaums, dem Eingabealphabet sowie einer Separations-Funktion sf ein Gegenbeispiel zu finden. Die Funktion garantiert dabei Pfade zu finden, die in M_r und nicht in M sind, wenn M_r Zustände enthält, die noch nicht in M enthalten sind [GPY02].

Unter der Annahme, dass eine obere Schranke für die Zustände bekannt ist, kann durch das vorgestellte Verfahren tatsächlich sichergestellt werden, dass entweder ein Fehler in der Integration

festgestellt wird oder die Integration unter Berücksichtigung der Sicherheits- und Lebendigkeitseigenschaften durch die oben beschriebene iterative Überprüfung korrekt ist.

Ein Problem des beschriebenen Black-Box-Checking-Ansatzes ist, dass die obere Anzahl an Zuständen typischerweise nicht bekannt ist. Um die obere Anzahl an Zuständen zu ermitteln, können wir uns, aufgrund der Eigenschaften der betrachteten Systeme, an dem Kontextverhalten orientieren. Grund hierfür ist, dass die betrachteten Systeme häufig eine Art Watchdog-Muster in dem Kommunikationsverhalten implementieren (siehe Kapitel 3 auf Seite 67). Die damit geforderten Sende- und Empfangs-Sequenzen lassen auf einen Zustand jeweils zwischen dem Senden und Empfangen schließen. Eine sichere obere Anzahl an Zuständen ist damit natürlich nicht gegeben. Eine Möglichkeit, um sicher zu sein, dass eine passende obere Grenze gefunden wurde, ist das Mehrfachausführen des Black-Box-Checking mit unterschiedlichen Obergrenzen. Da der ermittelte Automat ein minimaler ist, folgt aus einem gleichen erlernten Verhalten, dass die kleinere Obergrenze ausreichend ist.

Der Black-Box-Checking-Ansatz wurde im Rahmen der Projektgruppe ReCab [BBB⁺09] durch die *FRiTS^{Cab}* Tool Suite umgesetzt. Hierbei wurden zum einen Anwendungsbeispiele aus dem RailCab Projekt umgesetzt [BBB⁺09, HBB⁺09] sowie auch ein Beispiel aus der industriellen Praxis [HMSN10a, HMSN10b, HMS⁺10]. Detaillierter werden wir die Umsetzung sowie die Evaluierungen in Abschnitt 6.3.1.3 betrachten.

4.3 White Box Checking

Im Vergleich zu dem Gray-Box-Checking-Ansatz und Black-Box-Checking-Ansatz, beschreiben wir für den hier vorgestellten White-Box-Checking-Ansatz keine expliziten Iterationen, um das Verhalten zu erlernen und zu überprüfen. Dies liegt im Wesentlichen daran, dass bereits eine Reihe von Ansätzen für Quellcode-Analysen existieren, die grundsätzlich gut anwendbar sind, da diese bereits iterative Ansätze umsetzen. Unterliegend basieren diese Verfahren ebenfalls auf der Idee einer Abstraktion, die nach und nach durch Gegenbeispiele konkretisiert wird.

Ziel unseres White-Box-Verifikationsverfahrens ist es, die Kommunikation zwischen einer Altkomponente, für die Quellcode aber kein Modell vorliegt und einer mit MECHATRONIC UML entwickelten Komponente zu verifizieren. Dazu wird zunächst C-Code für die als Modell vorliegende Komponente (den Kontext des Altsystems) generiert. Dieser wird zusammen mit dem des Altsystems in ein spezielles Framework eingebettet, welches Scheduling, Nachrichtenaustausch und Zeitverhalten simuliert. Das resultierende Gesamtsystem schließlich wird mittels (Quellcode) Model Checking verifiziert.

Aufgabe des Verifikationsframeworks ist es, ein generisches Kommunikationsszenario zwischen den beiden Komponenten zu beschreiben, in dem bestimmte Abläufe als unsicher identifiziert werden können. Um diese Abläufe zu finden, verwenden wir vorhandene Quellcode-Model Checker. Diese unterstützen jedoch weder die Verifikation von Echtzeitverhalten noch eine parallele Ausführung der beiden Komponenten. Beides wird allerdings von den betrachteten Systemen

gefordert. Aus diesem Grund werden Zeit und parallele Ausführung explizit durch das Framework simuliert, wodurch indirekt eine Berücksichtigung derartigen Verhaltens durch den Model Checker ermöglicht wird.

Die Aufgaben des Frameworks sind die periodische Ausführung der für die Kommunikation relevanten Prozeduren innerhalb jeder Periode jeweils für die Altkomponente und dessen Kontext. Dabei werden Perioden und Zeit aktualisiert und die ausgetauschten Nachrichten zum passenden (virtuellen) Zeitpunkt vom Sendepuffer der sendenden in den Empfangspuffer der empfangenden Komponente verschoben. Die übrige Funktionalität, wie die Verwaltung dieser Puffer, die Simulation von Zeit innerhalb der Perioden sowie die Betrachtung von Zeitbedingungen wird über Funktionen bereitgestellt, die durch die Komponenten (Altkomponente und Kontext) aufgerufen werden.

Hauptaufgabe für unseren Ansatz ist es folglich, die zur Verfügung stehenden Informationen geeignet in die Eingaben der möglichen Modell Checker zu transformieren, ohne die Semantik unserer Modelle zu verletzen. Dies ist notwendig, da ansonsten die Analysen nicht gültig sind. Im Besonderen müssen wir daher eine Simulation der Zeit, der Perioden sowie eine geeignete Nachrichtenkommunikation auf Quellcode-Ebene umsetzen.

Wir werden im Folgenden zwei Ansätze diskutieren, die einen iterativen Analysansatz auf Quellcode-Ebene unterstützen. Anschließend werden wir einen Ansatz einer Verifikationsumgebung diskutieren. Vorher werden wir noch auf einige Voraussetzungen und Annahmen eingehen.

Voraussetzungen und Annahmen Für die Durchführung der Verifikation werden neben dem C-Quellcode der Altkomponente auch die Namen der durch das Framework aufzurufenden Funktionen (Initialisierung, periodisch auszuführende Haupt-Prozedur) benötigt. Dies beinhaltet die aufzurufenden Funktionen zum Nachrichtenaustausch sowie die Funktionen zum Ermitteln der Systemzeit. Weiterhin muss eine Zuordnung der Nachrichten im Kontext zu der entsprechenden internen Codierung auf der Seite der Altkomponente gegeben sein.

Um den Zeitbedarf von Prozeduren im Altsystem berücksichtigen zu können, müssen diese zuvor instrumentiert werden: Bei Ausführung jeder Prozedur mit Zeitbedarf muss die Funktion „consumeLegacyTime(int BCET, int WCET)“ mit passenden Werten für Best- und für Worst-Case Ausführungszeiten aufgerufen werden. Sind diese Werte nicht vorab bekannt, müssen diese zunächst durch eine WCET Analyse der Altkomponente beispielsweise mittels Bound-T³ ermittelt werden.

Quellcode-Analysewerkzeuge BLAST (Berkeley Lazy Abstraction Software Verification Tool⁴) ist ein Werkzeug, welches eine Gegenbeispiel-getriebene Verifikation auf Basis eines vorliegenden Programmcodes in C durchführt [HJMS03].

³<http://www.tidorum.fi/bound-t/>

⁴<http://mtc.epfl.ch/software-tools/blast/>

Als interne Repräsentation des Quellcodes wird zunächst ein Kontrollflussautomat (CFA) konstruiert. Zu dieser ersten Abstraktion wird ein mit Prädikaten und den entsprechenden Knotennamen des CFA annotierter Erreichbarkeitsbaum konstruiert. Durch eine Erreichbarkeitsanalyse wird in diesem ein Fehlerpfad gesucht. Wird auf diese Weise ein Fehler gefunden, so gibt eine symbolische Ausführung des entsprechenden Codefragments Aufschluss darüber, ob dieser dem Quellcode nach möglich ist, oder ob es sich um eine Fehlererkennung aufgrund einer zu ungenauen Abstraktion handelt. Während bei einem echten Fehler die Analyse abgebrochen werden kann, muss bei einem falsch-positiven Fehler eine Verfeinerung der Abstraktion durchgeführt werden.

Um die hierzu erforderlichen zusätzlichen Prädikate zu ermitteln, wird ein Theorembeweiser eingesetzt. Anschließend wird erneut ein Fehlerpfad gesucht. Das durch Blast umgesetzte Prinzip wird als „Counterexample Guided Abstraction Refinement“ (CEGAR), also gegenbeispielgetriebene Abstraktionsverfeinerung bezeichnet.

Bei CBMC und SATABS handelt es sich um zwei White-Box-Verifikationswerkzeuge zur Analyse von C- und C++-Programmen, die an der Carnegie Mellon Universität (teilweise in Zusammenarbeit mit der ETH Zürich und IBM) entwickelt wurden. Beide werden auch unter der Bezeichnung CPROVER zusammengefasst und sind in der Verwendung ähnlich.

Beide Werkzeuge wurden als Alternative zu Blast in Betracht gezogen, weil sie eine bessere Unterstützung einiger C-Programmkonstrukte (insbesondere von Arrays) bieten und sie zusätzlich zu C auch C++ unterstützen.

CBMC (C Bounded Model Checker⁵) ist ein Werkzeug für Bounded Model Checking, einer Variante des Model Checking, bei der die Schleifen eines Programms nur endlich oft durchlaufen werden [CKL04]. Für die Anzahl an durchzuführenden Iterationen kann eine obere Grenze spezifiziert werden. Programme, die Schleifen enthalten, welche öfter als erlaubt durchlaufen werden, werden daher von CBMC nur unvollständig analysiert. Es ist also im Vergleich zu anderen Model Checking Varianten zu beachten, dass die Abwesenheit von Fehlern nur bei Vorhandensein und Kenntnis besagter oberer Grenze gezeigt werden kann.

SATABS⁶ führt ähnlich wie Blast eine gegenbeispielgetriebene Verifikation mit iterativer Abstraktionsverfeinerung durch [CKSY05]. Dieses Werkzeug kann also vollständiges Model Checking durchführen. Wie bei Blast müssen dazu aber geeignete Prädikate gefunden werden, um eine Verfeinerung auf Basis eines Gegenbeispiels durchführen zu können. Werden keine Prädikate gefunden, so kann die Verifikation fehlschlagen.

Bei der Anwendung von Blast auf praxisorientierte Beispiele wurde festgestellt, dass das Werkzeug Arrays im untersuchten Programm nicht korrekt behandelt. Da Arrays in vielen Programmen vorkommen, beispielsweise in Form von Sende- oder Empfangspuffern, ist dieses Problem gravierend. Noch problematischer ist der ergebnislose Abbruch des Werkzeugs SATABS bei Berücksichtigung von Zeitbedingungen durch unser Framework.

⁵<http://www.cprover.org/cbmc/>

⁶<http://www.cprover.org/satabs/>

Beide Werkzeuge weisen zudem eine teilweise mangelhafte Stabilität auf und haben für die untersuchten Szenarien eine im Vergleich zu CBMC oft deutlich längere Laufzeit selbst bei kleinen Beispielen. Da alle zwei Werkzeuge in der Verwendung recht ähnlich sind, werden neben CBMC, SATABS und Blast in unserer Implementierung zumindest teilweise ebenfalls unterstützt. Sie ist allerdings für CBMC optimiert und müsste für eine vollständige Anwendbarkeit der übrigen Model Checker teilweise modifiziert werden.

Die für uns interessanten Funktionalitäten, die CBMC und SATABS anbieten, sind praktisch identisch. Auch die Syntax der Kommandozeilenparameter und die Formatierung der Programmausgabe ist sehr ähnlich, da beide Werkzeuge an derselben Universität (und teilweise von denselben Personen) entwickelt wurden. Mit minimalem Zusatzaufwand können daher beide Werkzeuge, statt nur einem, verwendet werden.

CBMC und SATABS bieten jeweils die Möglichkeit an, Assertions zu spezifizieren, also Annahmen über die Gültigkeit eines booleschen Ausdrucks über Programmvariablen an einer bestimmten Stelle im Programmablauf. Bei Ungültigkeit der Formel wird die jeweilige Stelle im Quellcode als ein Fehlerzustand behandelt, nach dem der Model Checker sucht.

Nichtdeterministisches Verhalten im Programm kann jeweils über spezielle Funktionen definiert werden: Für alle in C verfügbaren Datentypen für primitive Variablen existiert eine Prozedur, die nichtdeterministisch einen beliebigen Wert im jeweiligen Wertebereich zurückgeben kann. Zusätzlich kann dieser durch Annahmen eingeschränkt werden; dies sind Garantien an den Model Checker, dass für einzelne Variablen bestimmte Ausdrücke gelten.

Blast kann Spezifikationen von Korrektheitsbedingungen auf verschiedenen Ebenen nutzen. Letztendlich werden diese jedoch immer auf eine Erreichbarkeitsanalyse zurückgeführt, bei der nach Fehlerzuständen gesucht wird.

Die einfachste Möglichkeit, diese Zustände zu spezifizieren ist, sie direkt durch Definition einer bestimmten Markierung im C-Quelltext des untersuchten Programmes anzugeben. Ebenfalls im Quelltext lassen sich, ähnlich wie bei CBMC und SATABS, Assertions angeben. Dies entspricht damit einer zu UPPAAL (siehe Abschnitt 2.4) sehr ähnlichen Vorgehensweise, da die Analysen ebenfalls auf reine Erreichbarkeitsprobleme eines Fehlerzustands reduziert werden [JLS00].

Eine abstraktere Variante zur Definition fehlerhaften Verhaltens bietet Blast's Spezifikationssprache [BCH⁺04]. Diese besteht aus zwei Ebenen: Zum einen können so genannte Beobachterautomaten spezifiziert werden, durch die temporale Abhängigkeiten zwischen Programmereignissen überprüft werden können. Zum anderen können in einer speziellen Scriptsprache relationale Anfragen („relational queries“) formuliert werden, die sich auf diese Automaten beziehen.

Eine in dieser Sprache definierte Spezifikation kann mit einem zu Blast gehörenden Werkzeug automatisch in eine entsprechende Instrumentierung des untersuchten Programms umgewandelt werden.

Die einzigen Varianten, die in unserer Implementierung verwendet werden, sind Fehlermarkierungen und Assertions.

Wie CBMC und SATABS unterstützt auch Blast die Formulierung nichtdeterministischen Verhaltens im Quellcode durch Definition einer bestimmten Variablen. Die Möglichkeiten, die Blast hier anbietet sind allerdings gegenüber denen der beiden CPROVER Werkzeuge eingeschränkt: Die spezielle Variable BLAST NONDET kann nur zur Modellierung von binären Entscheidungen eingesetzt werden. Über den Umweg einer Schleife lässt sich allerdings auch eine nichtdeterministische Auswahl aus einem größeren Wertebereich realisieren.

Simulationsumgebung und Codegenerierung Analog zum Gray Box- und Black Box Checking sind wir besonders daran interessiert, das Kommunikationsverhalten der Altkomponente mit dem entwickelten Kontext auf Fehler zu untersuchen. Dabei liegt das Altsystem als C-Code vor, der mit dem C-Model-Checker direkt analysiert werden kann. Der Kontext andererseits ist jedoch als REAL-TIME STATECHART gegeben, also in einer Form, die diesem Werkzeug nicht zugänglich ist.

Somit ist es erforderlich, das durch das Modell definierte Verhalten in eine Form zu transformieren, die dem C-Model-Checker als Eingabe dienen kann. Dies wird durch die Erzeugung von C-Quellcode aus dem Modell realisiert. Für die Verifikation genügt, zumindest für eine Überprüfung des Kommunikationsprotokolls ohne Berücksichtigung von Zeit, eine einfache Abbildung von Zuständen und Transitionen des Modells auf Variablen und Kontrollstrukturen eines entsprechenden C-Programms.

Schnittstelle zwischen Kontext und Altkomponente sind dabei die beiden als bekannt angenommene Funktionen „sendMsg(Message m)“ und „receiveMsg():Message“, die das Senden und Empfangen von Nachrichten ermöglichen. Es wird also eine komplett nachrichtengekoppelte Kommunikation ohne Zugriff auf einen gemeinsamen Speicher vorausgesetzt. Durch geeignete Implementierung dieser Funktionen kann das Kommunikationsverhalten des Kontexts dem C-Model-Checker gegenüber sichtbar gemacht werden.

Kommunikation Die Möglichkeiten zur Erkennung von Fehlern im Kommunikationsverhalten sind abhängig vom verwendeten Kommunikationsmodell. Daher werden im Folgenden der Fall der synchronen und der der asynchronen Kommunikation getrennt behandelt.

Wie bei dem Gray Box und Black Box Checking kann bei synchroner Kommunikation jedes Empfangen einer Nachricht, für die kein Verhalten definiert ist, als Fehler betrachtet werden. Das kann bereits auf Modellebene durch Transitionen für nicht definierte Nachrichten von jedem anderen Zustand aus zu einem speziellen Fehlerzustand F ausgedrückt werden. Für jeden Zustand Z , $Z! = F$, mit ausgehenden Transitionen für die Eingangsnachrichten m_1, m_2, \dots werden also Transitionen für die Nachrichten $* \setminus \{m_1, m_2, \dots\}$ von Z zu F angelegt.

Durch eine ähnliche Konstruktion lässt sich das Komplement des erlaubten Verhaltens einer Rolle bilden (siehe Paragraph Anforderungen und Voraussetzungen auf Seite 98). Durch Verifikation der Kommunikation mit einem Altsystem kann dann überprüft werden, ob letzteres eine gültige Verfeinerung dieser Rolle ist.

Dieser Fehlerzustand muss dem C-Model-Checker gegenüber als solcher bekannt gemacht werden. Da der C-Model-Checker jedoch nur Zustände im Code (bzw. dem entsprechenden CFA) kennt, muss bei der Codegenerierung der Code für jede Transition in den Fehlerzustand als fehlerhaft markiert werden. Dies kann bei allen zwei Werkzeugen durch Spezifikation von Assertions geschehen. Bei Blast kann auch ein Label „ERROR“ einen Fehler markieren.

Alternativ kann in Blast mittels der Blast-Spezifikationsprache eine derartige Transition durch ein passendes Muster erkannt werden. Voraussetzung ist allerdings, dass der entsprechende Quellcode mittels dieses Musters von dem zu anderen Transitionen gehörigen unterschieden werden kann.

Anstatt einen Fehlerzustand bereits auf Modellebene einzuführen, kann auch ein zum Kontextmodell passender Blast-Beobachterautomat erzeugt werden, der prüft, dass Nachrichten nur durch die in diesem Modell definierten Verhalten eintreffen können. Konkret muss dazu (automatisch) für jede Nachricht ein Ereignis mit zum generierten Code passendem Muster definiert werden, das diesen Code erkennt und den Zustand des Beobachterautomaten passend umschaltet. Weiterhin muss für dieses Ereignis ein Guard definiert werden, der prüft, dass der aktuelle Zustand in der Menge von Zuständen enthalten ist, in denen diese Nachricht empfangen werden darf.

Eine solche Spezifikation muss nicht notwendigerweise aus demselben Modell erzeugt werden, wie der C-Quellcode. Beides lässt sich prinzipiell entkoppeln, solange die Muster im Modell zum erzeugten Code passen. Vorteil dieser Generierung von Spezifikationen aus Automaten ist möglicherweise eine höhere Flexibilität (bei Entkopplung von der Codesynthese); Nachteil sind, zumindest bei der hier beschriebenen Art der Verifikation, Redundanzen zur Codesynthese.

Aufgrund der Bevorzugung von CBMC vor Blast wird in unserer Implementierung von den Blast-Beobachterautomaten kein Gebrauch gemacht. Auch die Spezifikation von Fehlern auf Modellebene wurde nicht weiter betrachtet.

Bei asynchroner Kommunikation können Deadlocks nicht direkt festgestellt werden, wie bei synchroner Kommunikation. Sie manifestieren sich hier über Invarianten oder Deadlines. Da diese nicht für die Altkomponenten festgelegt werden können, können Deadlocks in diesem nur indirekt über den Kontext festgestellt werden.

Zeitbedingungen Die Berücksichtigung von Zeitbedingungen in der Verifikation ist einerseits für die von uns betrachteten echtzeitkritischen Systeme sehr wichtig, wird andererseits aber von keinem C-Model-Checker unterstützt. Um das Werkzeug selbst nicht verändern zu müssen, stellen wir daher ein Framework sowie eine Codegenerierung zur Verfügung, die diese Anforderungen erfüllen.

Um Zeit in die Verifikation einbeziehen zu können, ist es erforderlich, diese zu simulieren. Dazu kann die aktuelle Zeit in einer Variablen erfasst werden, die bei Ausführung von Funktionsaufrufen oder primitiven Anweisungen um die entsprechende Ausführungszeit erhöht wird. Diese ist nicht nur von der Art der Anweisung abhängig, sondern auch von Parametern, globalen Variablen oder anderen Einflussgrößen.

Die maximale Ausführungszeit von Funktionen lässt sich für eine gegebene Plattform durch eine WCET-Analyse (z.B. Bound-T) ermitteln. Der addierte Wert muss also zwischen eins (bzw. einer ermittelten minimalen Ausführungszeit) und der WCET liegen. Diese kann bei allen drei betrachteten Werkzeugen durch Konstrukte im C-Quellcode realisiert werden.

Die Verwendung dieser Möglichkeit macht allerdings eine Instrumentierung des untersuchten Programms erforderlich: In jeder Funktion muss (z.B. am Anfang oder am Ende) eine eigene spezielle Prozedur aufgerufen werden, die die erwähnte Erhöhung der simulierten Zeit durchführt. Da der Zeitbedarf der durch die Instrumentierung eingefügten Operationen selbst nicht berücksichtigt wird, kann dadurch kein „Probe Effect“ auftreten; das analysierte Systemverhalten kann lediglich durch die explizit simulierte Zeit beeinflusst werden.

Durch diese simulierte Zeit kann zeitabhängiges Verhalten des Kontextes, das durch Guards und Invarianten im Modell spezifiziert wird, umgesetzt werden. Zeitabhängiges Verhalten innerhalb der Altkomponente andererseits ist nicht derart explizit formuliert und lässt sich nur unter bestimmten Voraussetzungen berücksichtigen. Dazu gehören ein diskretes Zeitmodell und die Abwesenheit von Verhalten, welches von der absoluten Systemzeit abhängt. Auch für die Kommunikation kann durch entsprechende Anpassung der receive- und send-Funktionen einfach Zeit simuliert werden.

Unter der Voraussetzung, dass die Altkomponente nur durch eine einzelne, bekannte Funktion die Zeit aktualisiert, kann auch diese Zeit simuliert werden. Dazu kann diese Funktion durch eine eigene Variante ersetzt werden, die die virtuelle Zeit zurückliefert.

Diese Simulation von Zeit erhöht zunächst generell die Genauigkeit der Verifikation, da die Abhängigkeit des Verhaltens von Kontext und Altkomponente von Echtzeitbedingungen berücksichtigt wird. Die Verifikation von zeitabhängigen Systemen wäre ohne diese Simulation zwangsläufig fehlerhaft bzw. ungenau (falsche Positive und/oder falsche Negative). Es ist zu beachten, dass durch die Kommunikationsbeziehung zwischen Kontext und Altkomponente das Verhalten beider Systeme abweichen kann, wenn auch nur eines davon zeitabhängig ist und dies nicht berücksichtigt wird (oder werden kann).

Durch eine solche Simulation von Zeit lassen sich aber auch Korrektheitsbedingungen mit direktem Bezug zur Zeit überprüfen, beispielsweise die Forderung, dass Zustände (des Kontextes) nach einer bestimmten Zeit verlassen werden müssen oder dass nach Empfangen einer Nachricht nur ein maximaler Zeitraum bis zum Senden der Antwort vergehen darf. Die Überprüfung solcher Bedingungen ist durch Vergleich der virtuellen Zeit mit den entsprechenden Einschränkungen möglich. Die Bedingungen müssen, wie oben diskutiert, in Code bzw. Assertions übersetzt werden.

Die Genauigkeit bei der Überprüfung von Zeitschranken hängt von der Genauigkeit der ermittelten WCET Zeiten und davon ab, ob das zeitabhängige Verhalten sowohl von Kontext als auch von der Altkomponente korrekt simuliert wird. Selbst wenn sich die Echtzeitbedingung nur auf eines der Systeme bezieht und das unberücksichtigte Zeitverhalten im anderen liegt, kann die Verifikation der Bedingung dadurch fehlerhaft werden, aufgrund indirekter Verhaltensabhängigkeiten über die Kommunikation.

Parameter Das durch das Framework beschriebene Szenario kann teilweise durch Parameter definiert werden. Diese können den Umfang des verifizierten Verhaltens maßgeblich beeinflussen. Dadurch können die Laufzeit des Verfahrens einerseits und die Einschränkungen, unter denen das System korrekt ist, andererseits gesteuert werden.

Das Zeitverhalten beider Komponenten kann jeweils an zwei Stellen beeinflusst werden (siehe Abbildung 4.9): Zum einen kann die Ausführung des Systems um einen nichtdeterministisch ausgewählten Wert in einem wählbaren Bereich verzögert werden. Zum anderen kann auch für die Verzögerung der Ausführung der Hauptprozedur innerhalb einer Periode ein solcher Bereich angegeben werden. Die nichtdeterministische Auswahl kann dann in jeder Periode eine andere sein. Je größer diese Bereiche gewählt werden, umso mehr mögliche Abläufe muss der Model Checker berücksichtigen, was zu einer inakzeptabel langen Laufzeit des Verfahrens führen kann.

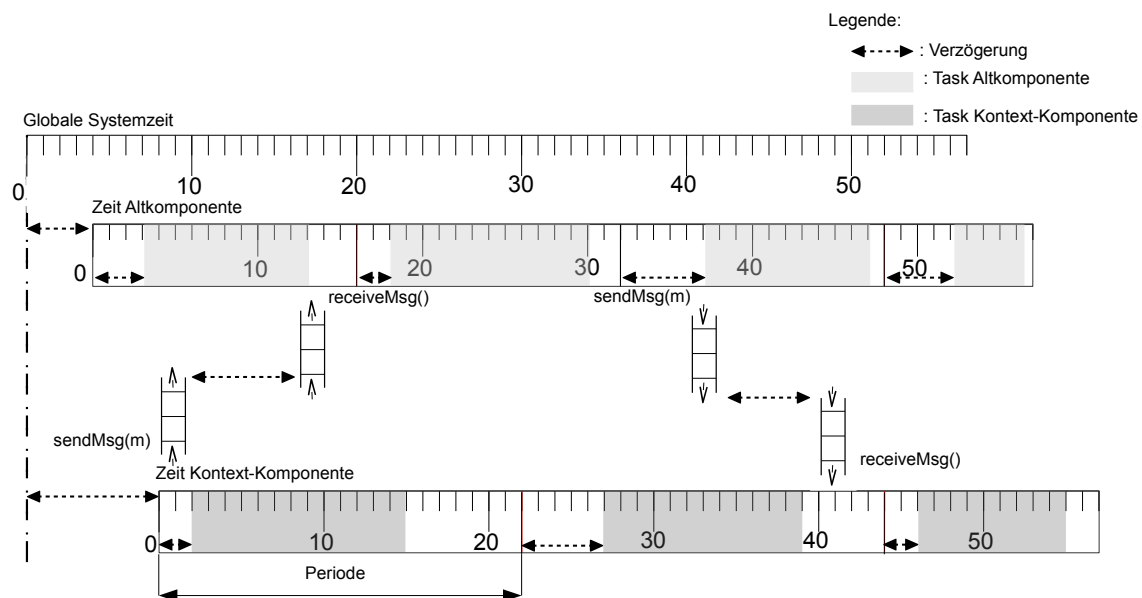


Abbildung 4.9: Parameter White Box Checking

Für den Kontext lässt sich neben der FIFO-Queue eine weitere Queue hinzuschalten, aus der nicht zwangsläufig die erste, sondern die erste im aktuellen Zustand behandelbare Nachricht entnommen wird. Dadurch wird ein Blockieren des Komponentenverhaltens aufgrund nicht behandelbarer Nachrichten verhindert. Letzteres wird bei Verwendung reiner FIFO-Queues als Deadlock erkannt. Bei Verwendung der nicht-FIFO-Variante ist dieses Kriterium aber nicht gültig. Deadlocks können hier jedoch indirekt über verletzte Invarianten festgestellt werden.

Neben diesen beschriebenen Voraussetzungen muss eine Quelle-Codegenerierung zur Verfügung gestellt werden, die die Ausführungssemantik nicht verletzt. Da unsere Modelle auf REAL-TIME

STATECHARTS basieren können wir auf den grundlegenden Ansatz von [Bur06] zurückgreifen, der eine Abbildung der wesentlichen Modellelemente diskutiert hat.

4.4 Identifikation von Reglerverhalten

Neben dem Kommunikationsverhalten sind auch die verschiedenen Reglerverhalten (-modi) für ein mechatronisches System von hoher Wichtigkeit. Das Verhalten des Systems / der Altcomponente ist maßgeblich von dem Reglerverhalten abhängig. Es ist daher wichtig zu identifizieren, welches Reglerverhalten in welchem Zustand aktiv ist. Erfüllen die Protokollverhalten die geforderten Sicherheitseigenschaften, lässt sich daher noch nicht darauf schließen, ob das Reglerverhalten tatsächlich den Anforderungen entspricht. Es kann z.B. sein, dass sich ein hinterher fahrendes RailCab im Convoy befindet, jedoch der Distanz-Regler nicht aktiv ist.

Systemidentifikation [Gra72, Ise92, FPW98, Lju98] ist das Verfahren, mit dem es möglich ist, kontinuierliche Systeme zu identifizieren. Als Systemidentifikation wird der Ansatz bezeichnet, welcher es ermöglicht aus beobachteten Ein- und Ausgangswerten eines Reglers, bzw. Systems, Rückschlüsse auf das Verhalten zu schließen. Das kontinuierliche Verhalten kann als mathematisches Modell durch eine Übergangsfunktion beschrieben werden. Das Modell stellt eine Abstraktion des realen Systems dar und beschreibt für die Problemstellung hinreichend die Prozesse oder das Verhalten des Systems. Durch Techniken der Systemidentifikation kann ein Modell erstellt oder die Parameter eines Modells eingestellt werden. Das Modell des Systems ermöglicht es, zukünftige Ausgangssignale vorherzusagen und das dynamische Verhalten des Systems durch Simulationen zu untersuchen.

Das (diskrete) Protokollverhalten und das (kontinuierliche) Reglerverhalten führen zu einem hybriden System. Hybride Systemidentifikationsansätze sind in der Regelungstheorie bekannt; aufgrund ihrer schlechten Laufzeiten und der schlechten Erkennungsgenauigkeiten des Verhaltens sind sie allerdings im Allgemeinen nicht gut anwendbar [Lju10].

Unser Ansatz adressiert dieses Problem, indem das hybride Systemidentifikationsproblem reduziert wird auf ein lineares Systemidentifikationsproblem. Hierzu teilen wir die Identifikation in zwei Schritte auf. Der erste Schritt ist die Identifikation des Protokollverhaltens basierend auf den beschriebenen Ansätzen. Der zweite Schritt ist die Identifikation des Reglerverhaltens innerhalb der erkannten Zustände des Protokollverhaltens.

Eine solche Aufteilung ist nur für Komponenten möglich, die durch Zustandsverhalten gesteuert werden. Dies ist z.B. für viele Anwendungen im Bereich der Automobil-Bordelektronik der Fall. Handelt es sich allerdings um eine Komponente, die primär eine regelungstechnische Aufgabe erfüllt, unabhängig von einem Zustandsverhalten, lässt sich dieses Verfahren nicht anwenden. Beispiele hierfür können Motorregelungen sein.

Der AUTOSAR-Standard zeigt allerdings, dass z.B. für die Automobilindustrie diskret gesteuerte Systeme von hoher Relevanz sind, da die Spezifikation der Schnittstelle von Komponenten sogenannte Modi unterstützen, die den aktuellen Zustand der Komponente repräsentieren. Im

Fall der Motorregelungen ist zudem auch anzumerken, dass mittlerweile moderne Automatik-Motorregelungen unterschiedliche Zustände je nach Benutzerprofil besitzen und damit der hier vorgestellte Ansatz ebenfalls Anwendung finden kann. Im Rahmen dieser Arbeit wurde allerdings keine Evaluierung durchgeführt, wie weitreichend der hier skizzierte Ansatz tatsächlich Anwendung findet.

Die Systemidentifikation erfolgt durch Simulation des Systems. Wir führen dabei das System in die einzelnen Zustände, indem wir jeden Pfad, der den Zustand erreicht, ausführen, da unterschiedliche Pfade zu unterschiedlichen Reglerverhalten führen können. Die konkrete Eingabe für die Systemidentifikation ist eine spezifizierte Testtrajektorie oder eine realistische Ausführung des Systems in seiner Umgebung.

Als Testtrajektorien können z.B. Sprungantworten oder Impulsantworten genutzt werden. Eine Sprungantwort beschreibt die Reaktion eines Systems (einer Altkomponente) am Ausgang auf eine angelegte Sprungfunktion am Eingang. Eine Impulsantwort eines Systems beschreibt die Reaktion des Systems (der Altkomponente) am Ausgang auf einen angelegten Impuls.

Basierend auf dem ein- und ausgehenden Verhalten wird dann z.B. eine Übergangsfunktion für lineare Systeme ermittelt. Wenn alle Übergangsfunktionen bekannt sind, können Rekonfigurationen durch unterschiedliche Übergangsfunktionen in aufeinander folgenden Zuständen erkannt werden.

Umsetzung Zur Erstellung der Modelle, wird das System zunächst einer Diagnose unterzogen. Die Aufzeichnung der kontinuierlichen Werte wird manuell vom Benutzer durchgeführt. Dies geschieht in einem externen Werkzeug, in dem die Altkomponente ausgeführt und überwacht werden kann.

Dieser Prozess ist je nach Anwendungsfall und Domäne sehr unterschiedlich und muss von einem Domänenxperten in einem ihm bekannten Werkzeug durchgeführt werden. Der hier vorgestellte Ansatz unterstützt den Import einer MATLAB Datei (*.mat). Diese muss die kontinuierlichen Eingangs-, Ausgangs- und Zeitdaten des zu identifizierenden Verhaltens enthalten.

Um aus den zeitlich gemessenen Ein- und Ausgangssignalen zu einem Modell zu gelangen muss eine gute Modellstruktur gefunden werden, die das beobachtete Verhalten möglichst gut widergibt. Es gibt eine Vielzahl von Modellstrukturen für die Identifikation linearer Modelle. Beispielsweise gibt es das lineare Autoregressionsmodell (AR) oder das Moving-Average-Modell (MA) [Lju98].

Eine konkrete Evaluierung des Ansatzes ist durch Integration der Matlab System Identification Toolbox erfolgt⁷. Die Systemidentifikation versucht dabei mittels der Methode "Minimierung des Vorhersagefehlers" die Modellparameter für unbekannte Gesetzmäßigkeiten zu bestimmen. Es werden nur Daten im Zeitbereich unterstützt. Der Grad des Modells wird automatisch aus dem Bereich 1-10 gewählt. Die Modellstrukturen AR sowie ARMA (eine Addition aus AR und MA) werden ebenfalls unterstützt. Mit Hilfe dieser Werkzeugunterstützung war es uns möglich

⁷<http://www.mathworks.com/products/sysid/>

unterschiedliche Regler in unserem Convoy Beispiel mit einer Altkomponente zu identifizieren, womit wir auch Rekonfigurationen erkennen konnten [BGH⁺08, BBB⁺09, HBB⁺09, BBB⁺09].

Der Systemidentifikationsansatz hat, trotz der Werkzeugintegration, immer noch eine sehr hohe Abhängigkeit zu Entwicklern aus der Regelungstechnik, da die Systemidentifikation an sich trotz vieler Forschungsaktivitäten immer noch ein hochgradiger manueller Akt ist, der entsprechendes Expertenwissen benötigt.

4.5 Diskussion

In diesem Kapitel haben wir eine Synthese des Kommunikationsverhaltens für mechatronische Altkomponenten durch Kombination von einer kompositionellen Verifikation, modellbasierten Testen und Lernansätzen vorgestellt. Diese Ansätze ermöglichen eine kontextspezifische Konflikterkennung in frühen Lernschritten. Weiterhin ermöglichen wir dem Ingenieur das regelungstechnische Verhalten und Rekonfigurationen zu identifizieren. Im Vergleich zu klassischen Ansätzen, die erst in der Integrationsphase eine Integration von Altkomponenten ermöglichen, sind wir damit insgesamt in der Lage früh Konflikte zu erkennen und Kosten zu sparen.

Die präsentierten X-Box-Checking-Ansätze decken einen großen Bereich an Techniken ab, um Altkomponenten mit unterschiedlichen zur Verfügung stehenden Informationen in ein Komponentenmodell zu integrieren. Unsere Evaluierungen in Kooperation mit der Industrie und dem RailCab-Projekt bestätigen diese Aussage [HMSN10a, HMSN10b, HMS⁺10].

Nach Möglichkeit versuchen wir für die Verhaltenserkennung den Gray-Box-Checking-Ansatz zu verwenden, da dieser die besten Laufzeiteigenschaften aufweist. Dies ist allerdings nur möglich, wenn der Zustand der Altkomponente sicher erkannt werden kann. Ist dies nicht der Fall, wird je nach Informationsstand der White-Box-Checking- oder Black-Box-Checking-Ansatz angewandt.

Kapitel 5

Synthese von Komponentenverhalten

Die in den Kapiteln 3 und 4 vorgestellten Ansätze unterstützen die Komposition und Wiederverwendung von Komponenten in dem modellgetriebenen Entwicklungsansatz MECHATRONIC UML durch eine Definition und Überprüfung einer Verfeinerung in hierarchischen Komponentensystemen und durch Integration von Altkomponenten. Bei der Komposition kann zu dem die Anforderung entstehen, dass Abhängigkeiten zwischen den verschiedenen Kompositionen berücksichtigt werden müssen. In diesem Abschnitt stellen wir einen Ansatz vor, der konstruktiv durch eine formale Abhängigkeitsbeschreibung das Synchronisationsverhalten (gesamte Komponentenverhalten) synthetisieren kann. Hiermit betrachten wir den letzten Anwendungsfall einer Konkretisierung nach Abschnitt 2.1 auf Seite 13.

Um die Komplexität bei der Entwicklung eines Systems zu beherrschen ist eine Dekomposition des Systems in separate Einheiten eines der weitverbreitetsten Paradigmen in der Softwaretechnik [Dij76]. Hierdurch werden Software-Entwicklungsziele wie Wartbarkeit, Adaptierbarkeit, Erweiterbarkeit und Wiederverwendung gefördert. In der MECHATRONIC UML wird dies durch eine Dekomposition des Systems in Komponenten und Kommunikationen zwischen Komponenten, die das Protokollverhalten beschreiben, erreicht (siehe z.B. Abschnitt 2.4.1). Die Kommunikationen werden dabei durch Muster (REAL-TIME COORDINATION PATTERNS) spezifiziert, die die Kommunikation strukturell in Rollen und einem Konnektor zwischen diesen aufteilen. Die Rollen, an denen eine Komponente beteiligt ist, beschreiben das Verhalten einer Komponente.

Eine der wesentlichen Aufgaben, um eine solche getrennte Entwicklung zu unterstützen, ist die komponentenspezifische Komposition separater, womöglich abhängiger Rollen (Protokolle) [TOHS99]. Für die hier zu betrachtende Verhaltenskomposition werden Ansätze in [Mil89, GV06] vorgestellt. Die Komplexität von mechatronischen Systemen fordert über diese Ansätze hinaus die Betrachtung von Echtzeitbedingungen sowie von Sicherheits- und begrenzten Lebendigkeitseigenschaften für die Komposition (siehe Abschnitt 2.4.1 und 2.4.6.1).

Aktuelle komponentenbasierte Ansätze, wie die MECHATRONIC UML (siehe Abschnitt 2.4.2) oder der Ansatz von Gössler und Sifakis [GS03], unterstützen diesen Entwicklungsschritt, indem manuell Abhängigkeiten durch einen zusätzlichen Beobachterautomaten oder Synchronisationsautomaten spezifiziert werden (siehe Abbildung 2.9). Dies ist allerdings schon allein für eine einfache Abhängigkeit schwierig umzusetzen. Ein Entwickler muss, um protokollübergreifende Anforderungen einer Komponente zu implementieren, folgendes berücksichtigen:

- das Gesamtverhalten der beteiligten Rollen (Produktautomat von Registree und Rear),
- die gestellten Anforderungen (unregistered-Zustand der Rolle Registree und convoy-Zustand der Rolle Rear dürfen nicht gleichzeitig aktiv sein),
- das eingeschränkte Verhalten muss eine Verfeinerung der jeweiligen Rollen sein.

Eine Anforderung selbstoptimierender Systeme ist zudem, dass das Verhalten möglichst flexibel auf unterschiedliche Szenarien zur Laufzeit reagieren soll, damit zwischen Alternativen (z.B. im Konvoi fahren oder nicht) optimiert werden kann. Das bedeutet wiederum, dass das Verhalten der Rollen nur so wenig wie möglich eingeschränkt werden sollte.

Um nur die beschriebene einfache Anforderung an die Komposition der Rollen Registree und Rear umzusetzen, muss der Entwickler an den richtigen Stellen in den jeweiligen Protokollen Synchronisationen (z.B. die `notInConvoy`-Synchronisation aus Abbildung 2.9) mit einem ebenfalls zusätzlich zu spezifizierenden Synchronisationsautomaten einführen. Damit werden bestimmte Pfade oder Zustände eingeschränkt. Da der Entwickler nicht weiß, ob damit die Anforderung an die Komposition erfolgreich umgesetzt wurde, muss er zudem eine Überprüfung durchführen. Die Überprüfung kann vorzugsweise durch einen Model Checker (in unserem Fall Uppaal) durchgeführt werden. Es muss also zudem die gestellte Anforderung in Form einer Sicherheits- oder Lebendigkeitseigenschaft für den Model Checker spezifiziert werden. Werden Fehler festgestellt, so muss manuell, mit Hilfe der Gegenbeispiele, das Verhalten angepasst werden. Der Entwickler weiß allerdings zu keinem Zeitpunkt während der Entwicklung, ob die Anforderung überhaupt realisierbar ist. Vielleicht findet der Entwickler ein Verhalten, welches die Anforderung umsetzt, jedoch keine gültige Verfeinerung darstellt, die auch überprüft werden muss. Wurden all diese Schritte erfolgreich durchgeführt, so ist immer noch unbekannt, ob mehr Verhalten eingeschränkt wurde als notwendig. Das eigentliche Ziel der Dekomposition, nämlich eine Wiederverwendung von Protokollen zu ermöglichen, ist durch eine manuelle Umsetzung damit fraglich.

Wir stellen einen Syntheseansatz vor (siehe Abschnitt 5.2), der eine wohldefinierte automatische Komposition von Protokollverhalten unter Berücksichtigung von Kompositionsregeln (siehe Abschnitt 5.1), die die Abhängigkeiten zwischen den Protokollen beschreiben, unterstützt. Die definierten Kompositionsregeln erhalten Sicherheitseigenschaften. Dies ist eine wesentliche Voraussetzung für sicherheitskritische Systeme. Die Komposition berücksichtigt zudem eine Verfeinerungsbeziehung, die wir *Rollen-Konformität* nennen (siehe Abschnitt 5.3). Auf diese Weise bleiben Lebendigkeitseigenschaften erhalten. Mit unserem Ansatz kann der Entwickler explizit eine verbotene Situation spezifizieren, ohne zusätzliches Verhalten für eine Beobachtung zu beschreiben und ohne eine Instrumentierung der Protokollverhalten vorzunehmen. Zudem kann der Entwickler die einzelnen Abhängigkeiten getrennt voneinander spezifizieren, da die Abhängigkeiten automatisch durch den Synthesearchivus aufgelöst werden. Der Entwickler muss folglich manuell nur noch eine Spezifikation der Abhängigkeiten durchführen.

Nach dem wir den Ansatz in den Abschnitten 5.1 bis 5.3 vorgestellt haben, werden wir weitere Anwendungsfälle des Ansatzes in Abschnitt 5.4 betrachten und Abschließend in Abschnitt 5.5 den Beitrag diskutieren. Bevor wir mit den Details beginnen, werden wir die Anforderun-

gen und Voraussetzungen unseres Ansatzes im Folgenden Anhand des RailCab Anwendungsbeispiels beschreiben. Hierzu werden wir uns dem Konvoi-Szenario bedienen und eine Einbettung des Beitrags in die MECHATRONIC UML skizzieren.

Anforderungen und Voraussetzungen Im Folgenden verdeutlichen wir unseren Ansatz, in dem wir das Konvoi-Beispiel aus Abbildung 2.17 um eine Basisstation (siehe Abbildung 5.1) erweitern. Die Basisstation ist für die Energieversorgung sowie für das Management der RailCabs auf einem dedizierten Streckenabschnitt zuständig. RailCabs nutzen diese Information, um Unfälle zu vermeiden und um Konvois zu bilden.

Wir spezifizieren das System mit zwei REAL-TIME COORDINATION PATTERNS (siehe Abschnitt 2.4.1) Registration und DistanceCoordination sowie zwei Komponenten BaseStation und RailCab (siehe Abbildung 5.2). Für die Kommunikation und die Komponenten gelten die in Abbildung 2.1 gezeigten Eigenschaften bzgl. der Verfeinerungs- und Synthesebeziehung sowie die Bestimmung des Gesamtverhaltens.

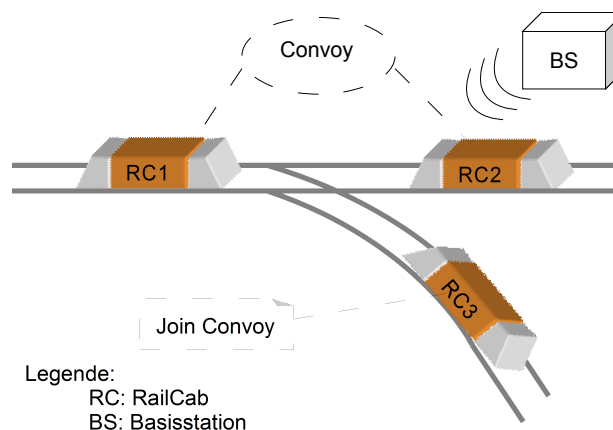


Abbildung 5.1: Beispiel Konvoirestrukturierung mit Basisstation

In Abbildung 5.2 sind die beteiligten Rollen des REAL-TIME COORDINATION PATTERNS Registration dargestellt (registrar und registree). Die Rollen des DistanceCoordination-REAL-TIME COORDINATION PATTERNS sind front und rear. Das Verhalten der Rollen wird durch REAL-TIME STATECHARTS spezifiziert (siehe Abschnitt 2.4.2).

Abbildung 5.3 und 5.4 zeigen stark vereinfachte Protokollverhalten der rear-Rolle und der registree-Rolle. Die Vereinfachung wird lediglich eingeführt, um die Synthese anschaulich an einem Beispiel zu illustrieren. Wir gehen zudem auch nicht näher auf die Gegenstücke der Rollen, dies sind die front-Rolle und die registrar-Rolle, ein.

Initial befindet sich die rear-Rolle in dem Zustand noConvoy und sendet eine startConvoy-Nachricht. Die Uhr cr wird auf null zurückgesetzt, bevor der convoy-Zustand betreten wird. Im Intervall zwischen 200 und 1000 Zeiteinheiten kann die breakConvoy-Nachricht empfangen werden, da die Invariante des Zustands $convoy$ $cr \leq 1000$ und der Time Guard der ausgehenden

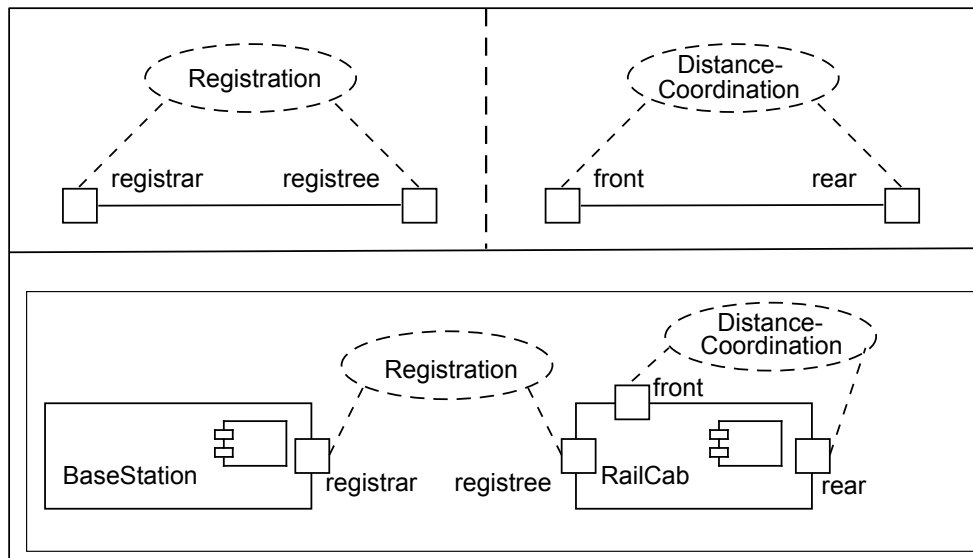


Abbildung 5.2: Kombination von separaten Protokollen in der MECHATRONIC UML

Transition $cr \geq 200$ ist. Andernfalls kann im Zeitintervall 400 bis 1000 periodisch eine update-Nachricht verschickt werden. Falls beide Transitionen gleichzeitig schaltbar sind, erfolgt die Entscheidung nichtdeterministisch.

Die Rolle registree ist Initial im Zustand unregistered, sendet eine register-Nachricht und setzt die Uhr zurück. Im Intervall zwischen 800 und 2000 Zeiteinheiten wird periodisch eine lifetick-Nachricht verschickt oder im Intervall von 500 bis 2000 Zeiteinheiten eine unregistered-Nachricht verschickt. Um beide Entscheidungen in der abstrakten Rolle zu ermöglichen, ist die Entscheidung nichtdeterministisch.

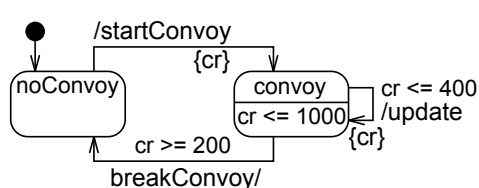


Abbildung 5.3: Vereinfachte rear-Rolle

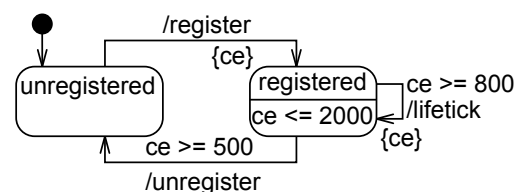


Abbildung 5.4: Vereinfachte Registree-Rolle

Um das Verhalten einer Komponente zu bestimmen, werden später im Entwicklungsprozess, die separat entwickelten REAL-TIME COORDINATION PATTERNS (bzw. ihre Rollen) angewandt (siehe Abbildung 5.2). Die Dekomposition hat auf der einen Seite dazu geführt, das System beherrschbar zu machen. Auf der anderen Seite führt dies zu dem Problem, dass das Verhalten der unabhängig entwickelten Protokolle Abhängigkeiten aufweisen kann, die während der Entwicklung von REAL-TIME COORDINATION PATTERNS (bzw. ihren Rollen) nicht betrachtet werden. Während des Prozesses der Verhaltensbeschreibung einer Komponente, muss daher zusätzlich

zu den separat entwickelten Rollen, die durch eine Komponente angewandt werden sollen, Abhängigkeiten zwischen den Rollen betrachtet werden.

In unserem RailCab Beispiel (Figure 5.2) wendet die RailCab-Komponente die REAL-TIME COORDINATION PATTERNS Registration und DistanceCoordination an. Während die beiden Muster unabhängig voneinander entwickelt wurden, muss für die gleichzeitige Anwendung beider Muster gelten:

Im Konvoi-Modus muss jeder Teilnehmer des Konvois an einer Basisstation registriert sein.

Dies ist eine (typische) Anforderung, die möglicherweise gewollt während der Entwicklung der Rollen nicht berücksichtigt wurde, um die Muster möglichst abstrakt und damit weitreichend einsetzen zu können. Hiermit werden eine Erhöhung der Wiederverwendung der einzelnen Rollen und gleichzeitig eine Verringerung der Komplexität erreicht. Wie in [TOHS99] beschrieben kann es aber auch vorkommen, dass Abhängigkeiten zwischen separaten Teilen im System erst durch Anforderungen entstehen, die später im Entwicklungsprozess bekannt werden.

Entsprechend dieser Anforderung, besteht eine Abhängigkeit zwischen den Rollen rear und registree, wenn sie durch die RailCab-Komponente angewandt werden. Um diese Abhängigkeit zu beschreiben, muss das Verhalten der registree-Rolle und das Verhalten der rear-Rolle verfeinert und miteinander synchronisiert werden.

Allgemein können wir zwischen mehreren unterschiedlichen Synchronisations-Anwendungsfällen unterscheiden (siehe Abbildung 5.5). Der 1. Fall betrachtet eine Synchronisation auf gleicher Hierarchieebene. Das Gesamtverhalten abhängiger Protokollverhalten ($M_{1,1}^C$, $M_{1,2}^C$ und $M_{j,k}^C$) ergibt sich dabei durch ein gemeinsames Synchronisationsverhalten (M_1^S). Im Fall einer Synchronisation auf unterschiedlichen Hierarchieebenen ergibt sich das Gesamtverhalten zudem aus dem Verhalten eingebetteter Komponenten ($M_{1,1}^r$ und $M_{1,2}^r$). Der dritte Fall ergänzt die ersten beiden um einen Multi-Port.

Diese drei Anwendungsfälle betrachten eine lokale Synchronisation einer (hierarchischen) Komponente. Darüber hinaus kann eine Synchronisation auch verteilt auf Musterebene stattfinden. Da der 1. Fall grundlegend die Synchronisation zwischen abhängigen Protokollverhalten betrachtet, werden wir im Folgenden vertiefend diesen Fall betrachten. Anschließend werden wir diskutieren, wie die anderen Fälle ebenfalls abgedeckt werden können.

Eine wesentliche Aufgabe der Synthese ist es, dass neben dem Einhalten der Kompositionsregeln, die Eigenschaften der Rollenverhalten¹ nicht verletzt werden. Entsprechend muss für den Ansatz eine geeignete Verfeinerungsdefinition für zeitkontinuierliche Echtzeitsysteme beschrieben werden. Zusammen mit den Kompositionsregeln, einer Verfeinerungsdefinition sowie den Rollenverhalten ermöglichen wir eine automatische Synthese des Komponentenverhaltens.

Zur Beschreibung der Verfeinerungsdefinition werden wir auf die in Abschnitt 3.1 diskutierte Verfeinerung zurückgreifen. Im Rahmen dieser Arbeit haben wir uns für eine implementierungsnahen (operationale) Definition einer Verfeinerung entschieden. Hiermit wird es uns im Vergleich

¹Ohne Einschränkung des Ansatzes bezeichnen wir die separaten Protokolle mit Rollenverhalten.

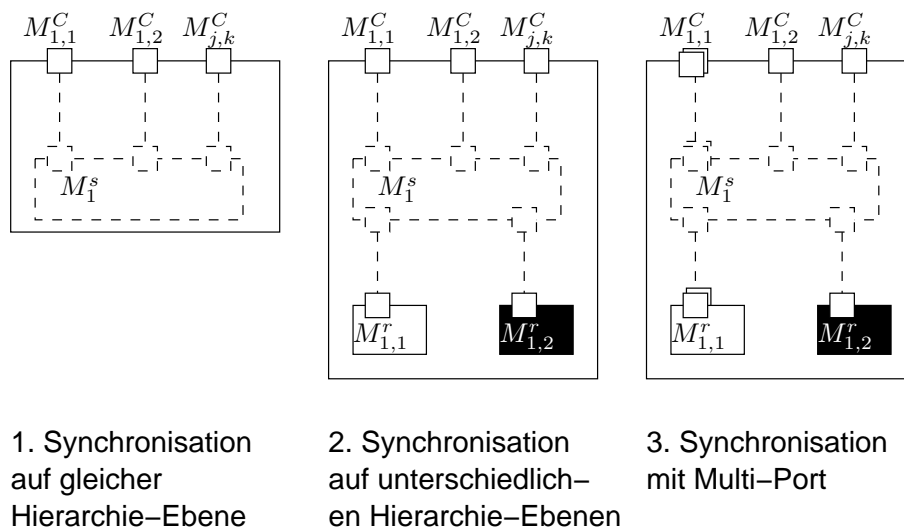


Abbildung 5.5: Synchronisationsverhalten Komponente: Anwendungsfälle

zu deklarativen Ansätzen (siehe z.B. [AB11]) ermöglicht, die nur über Mengen und nicht über die Ordnung von Nachrichten sprechen, einen Algorithmus (einfach) umzusetzen. Dies führt allerdings dazu, dass wir die beschriebene Verfeinerung aus Abschnitt 3.1 nicht direkt eins zu eins übernehmen können, sondern diese um die hier betrachteten Modelle anpassen müssen. Das bedeutet im Speziellen, dass wir hier mehrere abstrakte Modelle betrachten müssen (potentiell mehrere Rollenverhalten) und als Konkretisierung ein parallel Produkt dieser Rollenverhalten. Die Eigenschaften der Verfeinerung verändern sich dadurch nicht.

Eingabe in den Algorithmus sind Kompositionsregeln und die separaten Rollenverhalten (siehe Abbildung 5.6). Wenn die Synthese möglich ist, ohne das extern sichtbare Verhalten zu verletzen, dann ist die Ausgabe ein parallel komponiertes Komponentenverhalten, welches die Eingabeverhalten sowie die Kompositionsregeln kombiniert. Wenn die Synthese nicht möglich ist, wird eine Konfliktbeschreibung zurückgegeben. Grundlegende Arbeiten des Ansatzes wurden in [HSG08, HGH⁺09, Eck09, EH09, EH10a] vorgestellt.

5.1 Kompositionsregeln

Mit Kompositionsregeln können Abhängigkeiten zwischen Rollenverhalten spezifiziert werden. Aufgrund des unterliegenden Timed Automata Formalismus ermöglichen wir die Beschreibung von Abhängigkeiten zwischen Zuständen und Nachrichten bzw. Sequenzen von Nachrichten, die jeweils auch zeitlich über die definierten Uhren ausgeprägt sein können. Damit kann über alle Elemente des Formalismus eine Abhängigkeit beschrieben werden.

Wir teilen Kompositionsregeln in *Zustands-Kompositionsregeln* und *Nachrichten-Kompositionsregeln* ein. Zustands-Kompositionsregeln ermöglichen entsprechend die Syn-

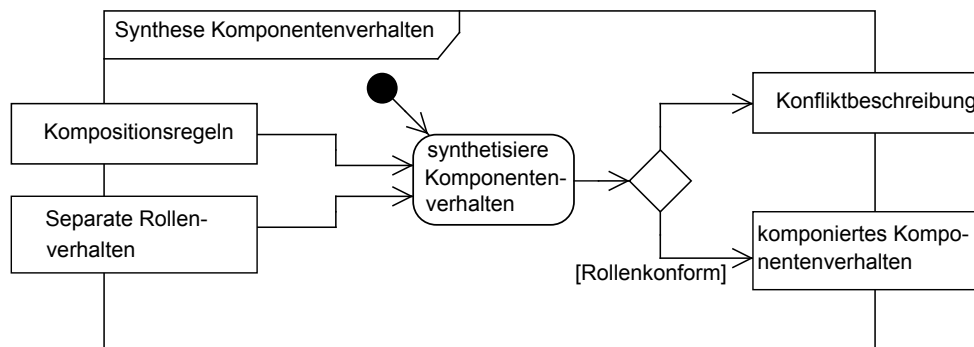


Abbildung 5.6: Ansatz Komponentenverhaltenssynthese

chronisation zwischen abhängigen Zustandskombinationen zu beschreiben. Nachrichten-Kompositionsregeln beschreiben mögliche Synchronisation zwischen Ereignissen und Sequenzen von Ereignissen. Beide Formalismen beinhalten zudem die Spezifikation von Zeitinformationen für die Synchronisation.

Um den Anforderungen der betrachteten Systeme gerecht zu werden, müssen zudem Kompositionsregeln allgemein Sicherheits- und begrenzte Lebendigkeitseigenschaften ausdrücken können (siehe z.B. Abschnitt 2.4 oder [Lam77, Hen92]).

Sicherheitseigenschaften können durch Zustands-Kompositionsregeln spezifiziert werden, indem verbotene Zustandskombinationen der parallel ausgeführten Rollenverhalten definiert werden. Nachrichten-Kompositionsregeln können Sicherheitseigenschaften ausdrücken, indem zusätzliche Zeitbedingungen bestimmten Transitionen hinzugefügt werden.

Lebendigkeitseigenschaften können durch Zustands-Kompositionsregeln und Nachrichten-Kompositionsregeln ausgedrückt werden, indem weitere Zeitbedingungen Zustandsinvarianten von Zustands-Kombinationen der parallelen Ausführung der Rollen hinzugefügt werden.

Im Folgenden werden wir diese beiden Formalismen genauer vorstellen.

5.1.1 Zustands-Kompositionsregeln

Wenn Rollen anwendungsspezifisch durch Komponenten angewandt werden, kann es sein, dass einige Zustandskombinationen der parallel geschalteten Rollen aufgrund von Systemanforderungen nicht erlaubt sind. Entsprechend wird ein Formalismus benötigt, der durch Synchronisation der beteiligten Rollenverhalten bestimmte Zustandskombinationen einschränkt. Darüber hinaus stellen wir die Anforderung an den Formalismus, dass er für den Systementwickler einfach zu benutzen ist, ohne einen neuen Formalismus zu erlernen.

Um diese Anforderungen zu erfüllen, werden die Zustands-Kompositionsregeln direkt über die Zustände der separaten Rollenverhalten definiert. Die gestellte Anforderung aus Para-

graph Anforderungen und Voraussetzungen auf Seite 141 kann z.B. wie folgt mit Zustands-Kompositionsregeln ausgedrückt werden:

$$r_1 = \neg((unregistered, true) \wedge (convoy, true)).$$

Syntaktisch besteht eine Zustands-Kompositionsregel aus einer Menge von Zustands-Prädikaten, die auf einen Booleschen-Wert (wahr oder falsch) abgebildet werden, die alle zusammen von einer Negation umgeben sind. Die Prädikate sind zudem über eine Verundung oder Veroderung miteinander verbunden. Ein Zustands-Prädikat spezifiziert einen Zustand in Kombination mit einer Menge an Uhr-Bedingungen. Die Uhr-Bedingungen werden dabei nur über bereits bekannte Uhren der Rollenverhalten definiert. In der Zustands-Kompositionsregel r_1 sind die Zustands-Prädikate $(unregistered, true)$ und $(convoy, true)$ jeweils mit einem $true$ verbunden. Das bedeutet, dass für alle Uhr-Bewertungen, die Zustandskombination von $unregistered$ und $convoy$ nicht erlaubt ist.

Im Folgenden wollen wir noch eine entschärfte Anforderung betrachten, die aussagt, dass die Zustandskombination von $unregistered$ und $convoy$ erlaubt ist, allerdings nicht länger als 50 Zeiteinheiten. Diese Anforderung beschreibt damit eine begrenzte Lebendigkeitseigenschaft, da dieser Zustand eventuell betreten werden kann, aber innerhalb von 50 Zeiteinheiten wieder verlassen werden muss. Die folgende Zustands-Kompositionsregel beschreibt diese Eigenschaft:

$$r_2 = \neg((unregistered, true) \wedge (convoy, cr > 50)).$$

Damit können Zustandskombinationen auch nur für bestimmte Zeitintervalle verboten werden.

Im Folgenden definieren wir erst die erlaubten Uhr-Bedingungen für Zustands-Prädikate. Hiermit wird es möglich sein, Zustandsinvarianten einzuschränken. Eine Anforderung ist, dass die Eigenschaften der unterliegenden Timed Automata nicht verletzt werden dürfen, da ansonsten ein Automat synthetisiert werden könnte, der die Semantik der Timed Automata verletzen könnte. Damit ist es nicht erlaubt, die Untergrenze der Invarianten zu verändern, da Invarianten nur die obere Grenze der Verweildauer in einem Zustand begrenzen (siehe Abschnitt 2.4.2 und [Alu92]). Dies wird über *nach oben geschlossenen Uhr-Bedingungen* erreicht.

Definition 51 (Nach oben geschlossene Uhr-Bedingungen)

Sei eine Menge C von Uhren gegeben, die Menge $\Phi_{uc}(C) \subset \Phi(C)$ von nach oben geschlossenen Uhr-Bedingungen ist induktiv definiert durch

$$\varphi ::= x \sim n \mid x - y \sim n \mid \varphi \wedge \varphi \mid true,$$

mit $x, y \in C$, $\sim \in \{\geq, >\}$, $n \in \mathbb{N}$.

Nach oben geschlossene Uhr-Bedingungen erlauben nur Uhr-Bedingungen oder Vereinigungen von Uhr-Bedingungen zu beschreiben, die eine Untergrenze für eine Uhr oder die Differenz zwischen zwei Uhren betrachten. Durch die einbezogene Negation einer Zustands-Kompositionsregel wird damit das Uhr-Intervall nach oben eingeschränkt. In Kompositionsregel

r_2 bedeutet das Zustands-Prädikat (*convoy*, $cr > 50$) durch die umgreifende Negation, dass die Zustandsinvariante von $cr \leq 1000$ auf $cr \leq 50$ eingeschränkt wird.

Damit können wir Zustands-Prädikate wie folgt definieren.

Definition 52 (Zustands-Prädikate)

Für einen Timed-Automata $A = (L, l^0, \Sigma, C, I, T)$, einen Zustand (Location) $l \in L$ und eine nach oben geschlossene Uhr-Bedingung $\varphi \in \Phi_{uc}(C)$, ist die Menge $\Gamma(A)$ von Zustands-Prädikaten $\gamma = (l, \varphi)$ definiert durch:

$$\Gamma(A) = L \times \Phi_{uc}(C).$$

Wie schon in den Beispielen r_1 und r_2 gezeigt, besteht ein Zustands-Prädikat aus einer Kombination von einem Zustand und nach oben geschlossenen Uhr-Bedingungen für einen Timed Automata A . Damit werden verbotene Uhr-Intervalle φ für einen Timed Automata Zustand l beschrieben. Wenn alle Uhr-Intervalle verboten werden sollen, wird dies durch $(l, true)$ für einen Zustand l ausgedrückt, da $true \in \Phi_{uc}(C)$.

Auf Basis dieser Definitionen können wir Zustands-Kompositionsregeln im Folgenden definieren.

Definition 53 (Zustands-Kompositionsregel)

Für zwei Timed Automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$, ist die Menge $R^S(A_1, A_2)$ von möglichen Zustands-Kompositionsregeln ρ definiert durch die folgende Grammatik:

$$\begin{aligned} \rho & ::= \neg(\rho_\gamma) \\ \rho_\gamma & ::= (\rho_\gamma) \wedge (\rho_\gamma) \mid (\rho_\gamma) \vee (\rho_\gamma) \mid \gamma \end{aligned}$$

mit $\gamma \in \Gamma(A_1) \cup \Gamma(A_2)$ (siehe Definition 52).

Zustands-Kompositionsregeln kombinieren mehrere Zustands-Prädikate für zwei verschiedene Timed Automata. Die Prädikate werden explizit als verboten gekennzeichnet durch die Negation. Beispiele hierfür sind die Kompositionsregeln r_1 und r_2 .

In Abschnitt 5.2 werden wir zeigen, wie wir Zustands-Kompositionsregeln automatisch auf separate Rollenverhalten anwenden können. Im Folgenden werden wir Nachrichten-Kompositionsregeln definieren.

5.1.2 Nachrichten-Kompositionsregeln

Um auf Basis von Nachrichten und Sequenzen von Nachrichten die separaten Rollenverhalten zu synchronisieren, beschreiben wir im Folgenden Nachrichten-Kompositionsregeln. Für die

Nachrichten-Kompositionsregeln gilt wie auch für die Zustands-Kompositionsregeln die Anforderung an den Formalismus, dass er für den Systementwickler einfach zu benutzen ist, ohne einen neuen Formalismus zu erlernen.

Daher beschreiben wir die Nachrichten-Sequenzen ebenfalls in der Syntax von Timed Automata. Um das extern sichtbare Echtzeitverhalten der Rollen nicht zu verletzen, fügen Nachrichten-Kompositionsregeln keine weiteren Nachrichten dem Verhalten hinzu. Nachrichten-Kompositionsregeln können allerdings den beobachteten Nachrichten-Sequenzen weitere Zeit-Bedingungen hinzufügen. Dies wird sowohl in Form von Time Guards wie auch Zustands-Invarianten ermöglicht, womit sowohl Sicherheits-, wie auch begrenzte Lebendigkeitseigenschaften spezifiziert werden können.

Für unsere Rollen rear und registree (siehe Abbildung 5.3 und 5.4) nehmen wir noch eine weitere Anforderung an. Ein RailCab muss an einer Basisstation mindestens 2500 Zeiteinheiten registriert sein, bevor ein Konvoi gestartet werden kann. Da diese Anforderung mehr als ein Rollenverhalten betrifft, muss die Anforderung als ein Synchronisationsverhalten beschrieben werden, um das Verifikationsergebnisse bzw. das extern sichtbare Verhalten nicht zu verletzen. Offensichtlich kann diese Anforderung auch nicht durch eine Zustands-Kompositionsregel umgesetzt werden, da als Voraussetzung die startConvoy-Nachricht empfangen werden muss. In Abbildung 5.7 ist entsprechend die Nachrichten-Kompositionsregel eca_1 spezifiziert.

Der Nachrichten-Kompositionsregel-Automat eca_1 schaltet von Zustand `ec_initial` nach Zustand `ec_registered`, wenn die Nachricht `register` von der `registree`-Rolle verschickt wird. Beim Schalten der Transition wird ebenfalls die Uhr `ec_c1` zurückgesetzt. Aus dem Zustand `ec_registered` wird entweder zurück in den Zustand `ec_initial` gewechselt, wenn die Rolle `registree` die Nachricht `unregister` verschickt oder es wird in Zustand `ec_registeredConvoy` geschaltet, wenn die Nachricht `startConvoy` von der `rear`-Rolle verschickt wird. Die Transition von Zustand `ec_registered` nach `ec_registeredConvoy` ist zudem mit dem Time Guard `ec_c1 >= 2500` annotiert. Damit wird erreicht, dass der korrespondierende Zustand des Rollenverhaltens erst 2500 Zeiteinheiten in dem entsprechenden Zustand verweilen muss, bevor ein Konvoi gebildet werden kann. Der Zustand `ec_registeredConvoy` wird verlassen, wenn die `registree`-Rolle eine `unregister`-Nachricht verschickt. Damit wird insgesamt die Anforderung spezifiziert.

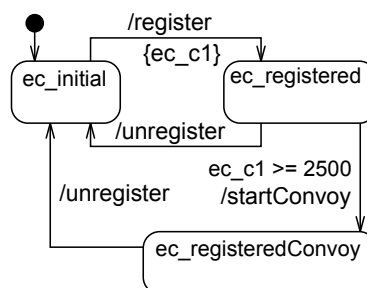


Abbildung 5.7: Nachrichten-Kompositionsregel eca_1

Nachdem nun informal Nachrichten-Kompositionsregeln eingeführt wurden, definieren wir diese im Folgenden formal.

Definition 54 (Nachrichten-Kompositionsregeln)

Seien $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ zwei Timed Automata. Eine Nachrichten-Kompositionsregel $A_E \in R^A(A_1, A_2)$ ist ein Timed Automaton $(L_E, l_E^0, \Sigma_E, C_E, I_E, T_E)$, mit

- L_E ist eine endliche, nicht-leere Menge an Zuständen,
- $l_E^0 \subseteq L$ ist der initiale Zustand,
- $\Sigma_E \subseteq \Sigma_1 \cup \Sigma_2$ ist eine endliche Menge an beobachtbaren Nachrichten,
- $I : L \rightarrow \Phi_{dc}(C_E)$ weist jedem Zustand eine nach unten geschlossene Zeitbedingung zu,
- C_E ist eine endliche Menge an Uhren, mit $C_E \cap (C_1 \cup C_2) = \emptyset$,
- $T_E \subseteq L_E \times \Sigma_E \times \Phi(C_E) \times 2^{C_E} \times L_E$ ist eine endliche Menge von Transitionen $t = (l, e, g, r, l')$ in T_E , mit
 - $l \in L_E$ ist der Quellzustand,
 - $e \in \Sigma_E$ ist die beobachtete Nachricht,
 - $g \in \Phi(C_E)$ ist der Time Guard,
 - $r \subseteq C_E$ ist eine Menge von Uhren, die zurückgesetzt werden sollen, und
 - $l' \in L_E$ ist der Zielzustand.

Nachrichten-Kompositionsregeln sind damit über Nachrichten der beteiligten Rollenverhalten A_1 und A_2 definiert, womit kein weiteres externes Echtzeitverhalten hinzugefügt werden kann. Damit beobachten Nachrichten-Kompositionsregeln lediglich die beteiligten Rollenverhalten.

Eine weitere Einschränkung ist, dass die Menge der Uhren disjunkt mit den beteiligten Rollenverhalten ist. Auf diese Weise wird garantiert, dass die Nachrichten-Kompositionsregeln nicht die Zeitintervalle der Nachrichten-Sequenzen der Rollenverhalten verletzen. Andernfalls wäre es möglich, dass vorher verifizierte Deadlines der Rollenverhalten nicht mehr eingehalten werden können.

Zusammenfassend können damit Nachrichten-Kompositionsregeln eine Sequenz von Nachrichten beobachten und möglicherweise weitere Zeitbedingungen den korrespondierenden Transitionen und Zuständen hinzufügen.

In dem nächsten Abschnitt werden wir betrachten, wie Nachrichten-Kompositionsregeln automatisch durch unsere Synthese angewandt werden.

5.2 Synthese

In diesem Abschnitt stellen wir eine Synthese von Statecharts für das Komponentenverhalten vor. Eingaben in die Synthese sind Kompositionsregeln sowie Rollenverhalten, die über die Kompositionsregeln eingeschränkt werden sollen (siehe Abbildung 5.6). Aus Vereinfachungsgründen werden wir im Folgenden nur zwei separate Rollenverhalten betrachten. Der Ansatz kann allerdings einfach auf beliebige Anzahlen an separaten Automaten erweitert werden, in dem die Definitionen, die zwei Eingabe-Automaten betrachten einfach auf eine Menge von Eingabe-Automaten erweitert werden.

Die Syntheseaktivität „synthetisiere Komponentenverhalten“ teilen wir in vier verschiedene Aktivitäten auf, welche wir im Folgenden näher betrachten. Als erstes wird die parallele Komposition berechnet (siehe Abschnitt 5.2.1). Auf diesen parallel komponierten Rollenverhalten werden die Kompositionsregeln angewandt. Die verbotenen Zustands-Kombinationen der Zustands-Kompositionsregeln werden dabei entfernt (siehe Abschnitt 5.2.2) und die spezifizierten Nachrichten-Kompositionsregeln werden in das parallel komponierte Verhalten integriert (siehe Abschnitt 5.2.3). Da die Anwendung der Kompositionsregeln zu einer Verletzung der Eigenschaften der Rollenverhalten führen kann, wird im letzten Schritt überprüft, ob die Eigenschaften nicht verletzt wurden (siehe Abschnitt 5.3).

5.2.1 Parallele Komposition

Wie wir schon in Definition 2 für Timed Automata gezeigt haben, führt die parallele Komposition zu einem expliziten Modell der parallelen Ausführung der separaten Verhalten. Diese parallele Komposition ist Voraussetzung, um die Kompositionsregeln anzuwenden, da die Kompositionsregeln für die parallele Ausführung der separaten Rollenverhalten ausgelegt sind. Die parallele Komposition der vereinfachten Rollenverhalten `rear` und `registree` ist in Abbildung 5.8 dargestellt. An dem Beispiel ist zu sehen, dass die nebenläufige Umsetzung der Nachrichten immer noch zu einer parallelen Ausführung von unterschiedlichen Nachrichten führt, da die Transitionen eines Timed Automata in Nullzeit schalten und Zeit nicht unbedingt in Zuständen verbraucht werden muss.

Ein möglicher Pfad in diesem Automaten ist z.B. von `register` von `(noConvoy,unregistered)` nach `(noConvoy,registered)`, gefolgt durch `startConvoy` von `(noConvoy,registered)` nach `(convoy,registered)`, während alle Uhrenwerte Null bleiben. Damit treten `register` und `startConvoy` parallel auf.

5.2.2 Anwendung von Zustands-Kompositionsregeln

Zustands-Kompositionsregeln, wie in Abschnitt 5.1.1 definiert, beschreiben Sicherheits- und Lebendigkeitseigenschaften, die durch die parallele Anwendung der separaten Rollenverhalten nicht verletzt werden dürfen. Ziel ist es, im Vergleich zum reinen Model Checking, nicht nur zu

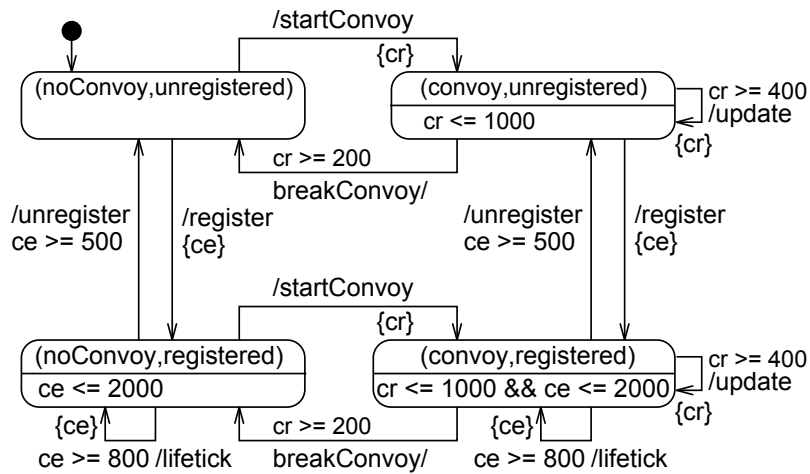


Abbildung 5.8: Beispiel eines parallelen Kompositionsautomaten (Rollen rear und registree)

überprüfen, dass diese Eigenschaften erfüllt sind, sondern automatisch das Modell so anzupassen, dass keine Verletzung dieser Eigenschaften eintritt. Entsprechend muss der im vorherigen Abschnitt definierte parallele Kompositionsautomat von separaten Rollenverhalten, um diese Eigenschaften angepasst werden, so dass z.B. bestimmte Zustandskombinationen verboten werden.

Verdeutlichen können wir dies durch Anwendung der Kompositionsregeln r_1 und r_2 (siehe Abschnitt 5.1.1) auf den im vorherigen Abschnitt gezeigten parallelen Automaten (siehe Abbildung 5.8).

Als erstes wenden wir $r_1 = \neg((unregistered, true) \wedge (convoy, true))$ an. Hiermit wird spezifiziert, dass für jeden Zeitbereich die Zustandskombination `unregistered` und `convoy` nicht erlaubt sind. In unserem Beispiel trifft dies nur für den kombinierten Zustand `(convoy,unregistered)` zu. Der resultierende Automat ist in Abbildung 5.9 gezeigt.

Wenden wir stattdessen die relaxierte Kompositionsregel r_2 an, so ist der kombinierte Zustand `(convoy,unregistered)` für das Zeitintervall $cr \leq 50$ gültig. Abbildung 5.10 beinhaltet entsprechend die Zustandskombination `(convoy,unregistered)`. Das bisherige Intervall $cr \leq 1000$ wurde entsprechend durch das kleinere $cr \leq 50$ ersetzt, da dies aus der Auswertung von $cr \leq 1000 \wedge cr \leq 50$ folgt. Diese Einschränkung führt dazu, dass die Nachrichten `update` sowie `breakConvoy` nicht in diesem Zustand verarbeitet werden können da beide einen Time Guard besitzen, dessen untere Schranke größer 50 ist. Die extern sichtbaren Echtzeiteigenschaften bleiben allerdings erhalten, da im Folgenden kombinierten Zustand `(convoy,registered)` diese Nachrichten verarbeitet werden können.

Verallgemeinert gilt damit als erster Schritt für einen gegebenen parallel komponierten Zustand l und einer gegebenen Zustandsregel r , zu überprüfen, ob die Invarianten von l durch r beeinflusst wird.

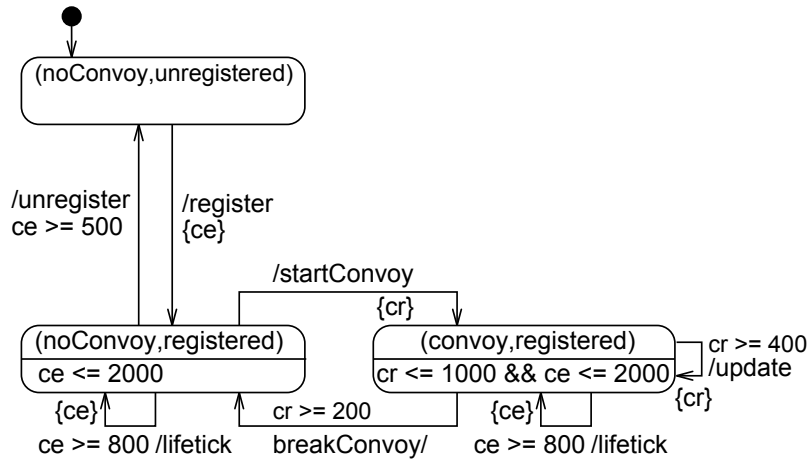


Abbildung 5.9: Anwendung von Zustands-Kompositionsregel r_1

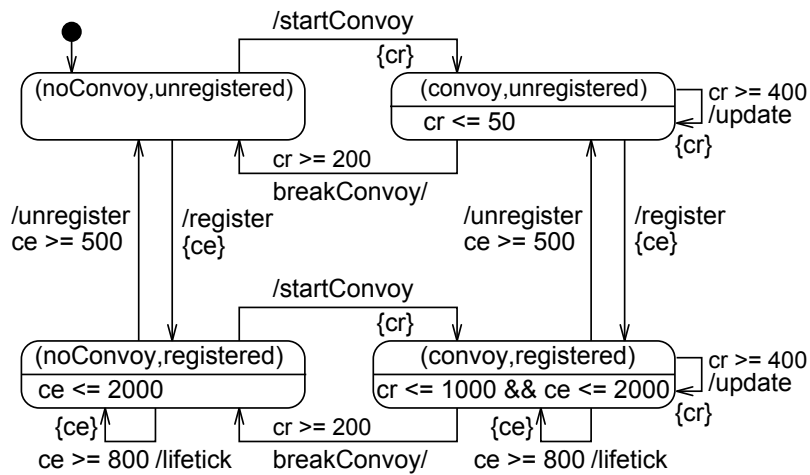


Abbildung 5.10: Anwendung von Zustands-Kompositionsregel r_2

Definition 55 (Zustands-Prädikat-Evaluierung)

Gegeben seien zwei Automaten $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ sowie ihre parallele Komposition $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$, ein korrespondierender parallel komponierter Zustand (Location) $l_p = (l_1, l_2)$, ein Zustands-Prädikat $\gamma = (l, \varphi)$ mit $l \in L_1 \cup L_2$ und $\varphi \in \Phi_{uc}(C_1) \cup \Phi_{uc}(C_2)$. Die Zustands-Prädikat-Evaluierung ist eine Funktion $\gamma : L_P \rightarrow \Phi_{uc}(C_P) \cup \{false\}$, definiert durch:

$$\gamma(l_p) = \begin{cases} \varphi, & \text{iff } (l = l_1) \vee (l = l_2), \\ false, & \text{else.} \end{cases}$$

Die Zustands-Prädikat-Evaluierung gibt die verbotenen Zeitbereiche in Form von Zeitbedingungen für einen parallel komponierten Zustand (l_1, l_2) und ein gegebenes Zustands-Prädikat (l, φ) zurück. Wenn einer der Zustände referenziert in dem komponierten Zustand gleich l ist, dann ist die verbotene Menge an Zeitbereichen genau durch φ beschrieben. Wenn keiner dieser Zustand gleich l ist, so ist φ nicht anwendbar und wird damit durch *false* beschrieben. Dies bedeutet, dass für den Zustand (l_1, l_2) kein Zeitbereich durch das Zustands-Prädikat (l, φ) restriktiert wird.

Beispielhaft wird dies durch das Zustands-Prädikat $\gamma_1 = (\text{convoy}, cr > 50)$ und die Zustände $(\text{noConvoy}, \text{unregistered})$ und $(\text{convoy}, \text{registered})$ in der folgenden Gleichung gezeigt:

$$\begin{aligned} \gamma_1((\text{noConvoy}, \text{unregistered})) &= false, \\ \gamma_1((\text{convoy}, \text{registered})) &= cr > 50. \end{aligned}$$

Jede Zustands-Kompositionsregel besteht aus einer Negation der Zustands-Prädikate oder einer Konjunktion und Disjunktion der Zustands-Prädikate. Die Evaluierung einer Zustands-Kompositionsregel muss entsprechend diese Fälle auch betrachten.

Die Negation einer Zustands-Kompositionsregel wird einfach durch die Negation des Ergebnisses der nicht-negierten Regel evaluiert. Dies ist einfach durch Anwendung Boolescher-Algebra möglich. Die Negation einer nach oben geschlossenen Zeitbedingung wird durch Invertierung des Relationalen-Operators $>$ nach \leq und \geq nach $<$ erreicht. Die Konjunktion und Disjunktion wird evaluiert durch die Anwendung der korrespondierenden Booleschen Operatoren auf die Evaluierung der Regeln. Atomare Zustands-Prädikate werden wie oben beschrieben evaluiert (siehe Definition 55).

Beispielhaft zeigen wir durch folgende Gleichung die Evaluierung der Zustands-Kompositionsregel r_1 für die komponierten Zustände $(\text{convoy}, \text{unregistered})$:

$$\begin{aligned} r_1((\text{convoy}, \text{unregistered})) &= \neg((\text{unregistered}, true) \wedge (\text{convoy}, true)) \\ &= \neg(true \wedge true) \\ &= \neg true \\ &= false. \end{aligned}$$

Da die Evaluierung von $r_1((convoy, unregistered))$ *false* ergibt und die Invariante $I((convoy, unregistered))$ ebenfalls *false* ergibt, wird der Zustand $(convoy, unregistered)$ aus dem parallel komponierten Automaten entfernt.

Auf Basis dieser Definitionen beschreiben wir im Folgenden die Definition der *Zustands-Kompositions-Konformität* eines Timed Automata.

Definition 56 (Zustands-Kompositions-Konformität)

Lasse $A_P = A_1 \parallel A_2 = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$ die parallele Komposition von Timed Automata A_1 und A_2 sein. Weiterhin sei $R_1^S \subseteq R^S(A_1, A_2)$ eine Menge von Zustands-Kompositionsregeln spezifiziert über A_1 und A_2 . Der zustands-kompositions-konforme, parallel komponierte Timed Automaton $A_{SC} = (L_{SC}, l_{SC}^0, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ ist definiert durch:

- $L_{SC} = L_P \setminus L_R$, mit $L_R = \{l_p \mid l_p \in L_P \text{ and } \forall \rho_1, \dots, \rho_n \in R_1^S : I(l_p) \wedge \rho_1(l_p) \wedge \dots \wedge \rho_n(l_p) = false\}$,
- $l_{SC}^0 = l_P^0 \Leftrightarrow l_P^0 \in L_{SC}$,
- $\Sigma_{SC} = \Sigma_P$,
- $I_{SC} : L_{SC} \rightarrow \Phi(C_{SC})$ mit $I_{SC}(l_p) = I_P(l_p) \wedge \rho_1(l_p) \wedge \dots \wedge \rho_n(l_p), \forall \rho_1, \dots, \rho_n \in R_1^S$,
- $C_{SC} = C_P$,
- $T_{SC} \subseteq L_{SC} \times \Sigma_{SC} \times \Phi(C_{SC}) \times 2^{C_{SC}} \times L_{SC}$, mit $(l_p, e, g, r, l_p') \in T_{SC} \Leftrightarrow (l_p, e, g, r, l_p') \in T_P \wedge l_p, l_p' \in L_{SC}$.

Für eine gegebene Menge an Zustands-Kompositionsregeln R_1^S , wird der zustands-kompositions-konforme Timed Automaton definiert als ein Timed Automaton, für den gilt, dass jede der Kompositionsregeln $\rho \in R_1^S$ auf jeden Zustand angewandt wurde $l_p \in L_P$.

Die Menge an zustands-kompositions-konformen Zuständen L_{SC} , ist die Menge von Zuständen L_P , ohne den Zuständen in L_R , welche durch die Zustands-Kompositionsregeln eingeschränkt wurden. Die eingeschränkten Zustände L_R sind entsprechend die Zustände, wo die Invariante $I_P(l_p)$ in Konjunktion mit jeder der Zustands-Kompositionsregeln mit $\rho_n(l_p)$ als *false* evaluiert wurden.

Der initiale Zustand l_{SC}^0 des zustands-kompositions-konformen Automaten ist der gleiche als der des parallel komponierten Automaten, so lange wie dieser nicht durch Anwendung von Zustands-Kompositionsregeln entfernt wurde. Die Menge an Ereignissen und die Menge an Uhren ist die gleiche.

Die Invariante jedes Zustands $I_{SC}(l_p)$ ist definiert durch die Konjunktion der ursprünglichen Invariante $I_P(l_p)$ mit jeder Evaluierung der Zustands-Kompositionsregel $\rho_n(l_p)$.

Die Menge an Transitionen T_{SC} eines zustands-kompositions-konformen Timed Automaton, ist die Menge an Transitionen T_P des parallel komponierten Timed Automaton, ohne die Transitionen, welche eingehende oder ausgehende Transition einer verbotenen Zustandskombination sind.

Nachdem wir in diesem Abschnitt beschrieben haben, wie wir Zustands-Kompositionsregeln anwenden, werden wir im nächsten Abschnitt die Anwendung von Nachrichten-Kompositionsregeln betrachten.

5.2.3 Anwendung von Nachrichten-Kompositionsregeln

Im letzten Abschnitt haben wir beschrieben, wie wir Zustands-Kompositionsregeln anwenden. In diesem Abschnitt betrachten wir die Anwendung von Nachrichten-Kompositionsregeln. Nachrichten-Kompositionsregeln spezifizieren zusätzliches Synchronisationsverhalten für die parallel komponierten Rollenverhalten (siehe Abschnitt 5.1.2). Entsprechend beschreiben wir in diesem Abschnitt wie Nachrichten-Kompositionsregeln auf einen parallel komponierten, zustands-kompositions-konformen Automaten angewandt werden.

Ähnlich wie die parallele Komposition definiert für die parallele Ausführung von Rollenverhalten (siehe Abschnitt 5.2.1), ist die Anwendung von Nachrichten-Kompositionsregeln vergleichbar mit der Komposition in der Prozessalgebra [Mil89] oder dem vernetzten Timed Automata-Formalismus [YPD94]. Der wesentliche Unterschied ist, dass die Nachrichten-Kompositions-Automaten nur Synchronisationsnachrichten betrachten und keine weiteren externen Nachrichten Hinzufügen. Entsprechend ist das Ergebnis ein Produktautomat, wobei die Zustände der parallelen Komposition mit den Zuständen des Nachrichten-Kompositions-Automaten multipliziert werden. Weiterhin werden die Transitionen des Nachrichten-Kompositions-Automaten mit den Transitionen des Automaten der parallelen Komposition synchronisiert. Diese Synchronisation verändert nicht das externe Verhalten, da der Nachrichten-Kompositions-Automat nur eine Beobachtung durchführt. Dies stellt ebenfalls einen Unterschied zu der Komposition der Prozessalgebra und der vernetzten Timed Automata dar. Diese Einschränkung ist notwendig, wie in Abschnitt 5.1.2 definiert, um gerade das externe Rollenverhalten nicht zu verletzen, da das für die betrachteten Systeme nicht erlaubt ist.

Durch einen Nachrichten-Kompositions-Automaten kann die Menge an Uhren-Zurücksetzungen der synchronisierten Transitionen verändert werden, sowie die Time Guards der synchronisierten Transitionen und die Zustands-Invarianten der korrespondierenden parallel komponierten Zustände. Die hinzugefügten Uhren sind dabei disjunkt von denen des parallel komponierten Automaten. Auf diese Weise wird erreicht, dass die zusätzlichen Zeitbedingungen nicht die Zeitintervalle des parallel komponierten Automaten verletzen. Damit bleiben die Verifikationsergebnisse, die Deadlines der Rollenverhalten betrachten, immer noch erhalten.

Im Folgenden wenden wir die Nachrichten-Kompositionsregel eca_1 (siehe Abbildung 5.7) auf die parallele Komposition der vereinfachten Rollenverhalten an (siehe Abbildung 5.8). Die Zustandskompositionsregel r_1 wurde bereits angewandt (siehe Abbildung 5.9).

Dies resultiert in den in Abbildung 5.11 gezeigten Timed Automaten. Jeder Zustand des Automaten bezieht sich auf die Zustände der Rollenautomaten sowie die Zustände der Nachrichten-Kompositionsregel. Weiterhin sind nur die Zustände in dem Automaten enthalten, die von dem Startzustand des Kompositionsautomaten ($noConvoy, unregistered, ec_{initial}$) durch synchro-

nisierte Transitionen erreichbar sind. Der Zustand $(noConvoy, unregistered, ec, egistered)$ kann beispielsweise nicht von dem Startzustand aus erreicht werden und ist daher nicht in dem Automaten enthalten.

Informell ist der Aufbau wie folgt. Ausgehend von dem Startzustand $(noConvoy, unregistered, ec_{initial})$, gilt für jede ausgehende Transition des parallelen Zustands, dass der Nachrichten-Kompositionsautomat ebenfalls seinen Zustand wechselt, falls der Nachrichten-Kompositionsautomat genau die gleichen Nachrichten referenziert. In diesem Fall sind die Time Guards und Clock Resets des Nachrichten-Kompositionsautomat in den Synchronisations-Transitionen der parallelen Komposition integriert. Dies ist der Fall für die $/register$ -Transition von $(noConvoy, unregistered)$ nach $(noConvoy, registered)$ in dem parallel komponierten Timed Automaton und von $ec_{initial}$ nach $ec_{registered}$ in dem Nachrichten-Kompositionsautomaten. Das Clock Reset ec_{c1} wurde der Transition hinzugefügt. Der Time Guard $ec_{c1} \geq 2500$, der Transition $/startConvoy$ von $(noConvoy, registered, ec_{registered})$ nach $(convoy, registered, ec_{registeredConvoy})$, wurde ebenfalls hinzugefügt. Zudem können Zustandsinvarianten durch Konjunktion hinzugefügt werden, wie für die Zustands-Kompositionsregeln beschrieben.

Weiterhin ist anzumerken, dass ein Zustand eines Nachrichten-Kompositionsautomaten nur wechselt, wenn der Nachrichten-Kompositionsautomat tatsächlich die gleichen Nachrichten referenziert, wie der entsprechende Zustand des Kompositionsautomaten. Ein Beispiel hierfür ist die $breakConvoy/-$ Transition von Zustand $(convoy, registered, ec_{registeredConvoy})$ nach $(noConvoy, registered, ec_{registeredConvoy})$. In diesem Fall wechselt der Nachrichten-Kompositionsautomat nicht den Zustand, da $ec_{registeredConvoy}$ keine ausgehende Transition für die Nachricht $breakConvoy/$ referenziert.

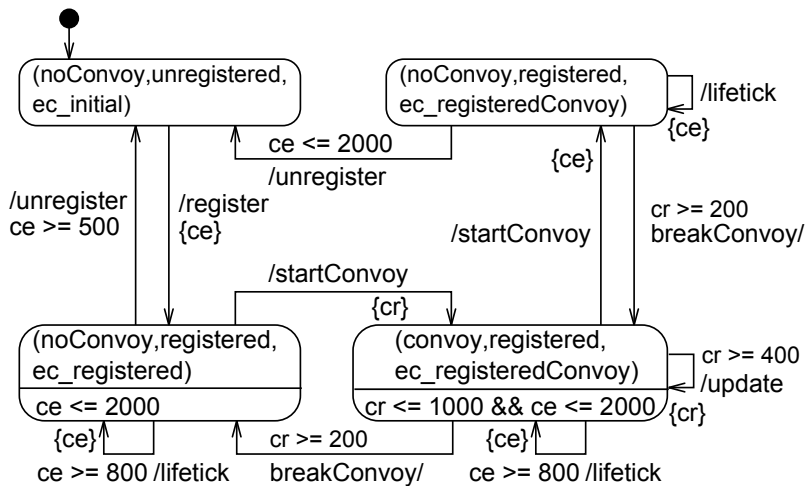


Abbildung 5.11: Anwendung von Nachrichten-Kompositionsregel eca_1

Die beschriebene Anwendung von Nachrichten-Kompositionsregeln zeigt wie Zeitbedingungen an Transitionen und Zustandsinvarianten hinzugefügt werden. Dies wird erreicht durch eine Syn-

chronisation zwischen dem parallelen komponierten Timed Automaten und dem Nachrichten-Kompositionsautomaten, ohne Instrumentierung des parallel komponierten Timed Automaten. Dies hat damit den Vorteil gegenüber klassischen Beobachterautomaten, dass der zu beobachtende Automat (in unserem Fall der parallel komponierte Timed Automaten) nicht durch zusätzliche Synchronisations-Nachrichten verändert wird.

Im Folgenden stellen wir die formale Definition des *nachrichten-kompositions-konformen* Timed Automaten vor, welcher ein zustands-kompositions-konformer, parallel komponierter Timed Automaten ist, auf den ein Nachrichten-Kompositions-Automaten angewandt wurde.

Definition 57 (Nachrichten-Kompositions-Konformität)

Lasse $A_{SC} = (L_{SC}, l_{SC}^0, \Sigma_{SC}, C_{SC}, I_{SC}, T_{SC})$ ein zustands-kompositions-konformer, parallel komponierter Timed Automaten sein, welcher aus den Timed Automaten $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ mit $C_1 \cap C_2 = \emptyset$ und $\Sigma_1 \cap \Sigma_2 = \emptyset$ entstanden ist. Weiterhin lasse $A_E = (L_E, l_E^0, \Sigma_E, C_E, I_E, T_E) \in R^A(A_1, A_2)$ ein Nachrichten-Kompositions-Automaten für A_1 und A_2 sein. Wir definieren den nachrichten-kompositions-konformen und zustands-kompositions-konformen, parallel komponierten Timed Automaten $A_{EC} = (L_{EC}, l_{EC}^0, \Sigma_{EC}, C_{EC}, I_{EC}, T_{EC})$ mit

- $L_{EC} \subseteq L_1 \times L_2 \times L_E$, mit $(l_1, l_2, l_e) \in L_{EC}$ iff $(l_1, l_2) \in L_{SC}$ und $I_{SC}((l_1, l_2)) \wedge I_E(l_e) \neq \text{false}$ and (l_1, l_2, l_e) ist erreichbar durch T_{EC} ,
- $l_{EC}^0 = (l_1^0, l_2^0, l_e^0)$, iff $(l_1^0, l_2^0, l_e^0) \in L_{EC}$,
- $\Sigma_{EC} = \Sigma_1 \cup \Sigma_2$,
- $I_{EC} : L_{EC} \rightarrow \Phi(C_1) \cup \Phi(C_2) \cup \Phi(C_E)$ mit $I_{EC}((l_1, l_2, l_e)) = I_{SC}((l_1, l_2)) \wedge I_E(l_e)$,
- $C_{EC} = C_1 \cup C_2 \cup C_E$,
- $T_{EC} \subseteq L_{EC} \times \Sigma_{EC} \times \Phi(C_{EC}) \times 2^{C_{EC}} \times L_{EC}$, mit
 - $((l_1, l_2, l_e), e_1, g_1, r_1, (l_1', l_2, l_e)) \in T_{EC} \Leftrightarrow$
 $((l_1, l_2), e_1, g_1, r_1, (l_1', l_2)) \in T_{SC} \wedge$
 $\forall l_e' \in L_E : (l_e, e_1, g_e, r_e, l_e') \notin T_E,$
 - $((l_1, l_2, l_e), e_2, g_2, r_2, (l_1, l_2', l_e)) \in T_{EC} \Leftrightarrow$
 $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_{SC} \wedge$
 $\forall l_e' \in L_E : (l_e, e_2, g_e, r_e, l_e') \notin T_E,$
 - $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1', l_2, l_e')) \in T_{EC} \Leftrightarrow$
 $((l_1, l_2), e_1, g_1, r_1, (l_1', l_2)) \in T_{SC} \wedge (l_e, e_1, g_e, r_e, l_e') \in T_E,$
 - $((l_1, l_2, l_e), e_1, g_1 \wedge g_e, r_1 \cup r_e, (l_1, l_2', l_e')) \in T_{EC} \Leftrightarrow$
 $((l_1, l_2), e_2, g_2, r_2, (l_1, l_2')) \in T_{SC} \wedge (l_e, e_2, g_e, r_e, l_e') \in T_E.$

Der nachrichten-kompositions-konforme Automaten bildet das explizite Modell für die parallele Ausführung der Rollenautomaten. Dabei berücksichtigt der Automaten die spezifizierten Zustands-Kompositionsregeln und die Nachrichten-Kompositionsregeln.

Die Zustände des nachrichten-kompositions-konformen Automaten L_{EC} sind eine Untermenge des Kreuzprodukts von $L_1 \times L_2 \times L_E$. Es werden nur solche Zustände berücksichtigt, die zustands-kompositions-konform sind, deren Invariante $I_{EC}((l_1, l_2, l_e)) = I_{SC}((l_1, l_2)) \wedge I_E(l_e)$ nicht gleich *false* ist und welche erreichbar durch die Transition T_{EC} dieses Automaten sind.

Die Invarianten eines nachrichten-kompositions-konformen Zustands (l_1, l_2, l_e) werden durch Konjunktion der Invariante des zustands-kompositions-konformen Zustands $I_{SC}(l_1, l_2)$ und der Invariante des nachrichten-kompositions-konformen Zustands $I_E(l_e)$ erstellt.

Die Definition der Transition T_{EC} von A_{EC} können wir in vier Fälle unterscheiden. Die ersten beiden Fälle beschreiben die Transitionen, welche nicht zwischen A_{SC} und A_E synchronisiert werden. Dabei wird unterschieden, ob A_1 oder A_2 den Zustand wechselt. In beiden Fällen gibt es keine korrespondierende Transition in T_E , welche synchronisiert werden könnte. Die anderen beiden Fälle beschreiben, dass eine Synchronisation zwischen A_{SC} und A_E stattfindet. Dabei wird ebenfalls unterschieden, ob A_1 oder A_2 seinen Zustand wechselt.

Zusammengesetzte Transitionen, welche eine Transition $t_e \in T_E$ des Nachrichten-Kompositions-Automaten referenzieren, aber nicht eine Transition $t_{sc} \in T_{SC}$ des zustands-konformen Automaten, werden nicht berücksichtigt. Dies liegt daran, dass die Transitionen $t_e \in T_E$ nicht mit den Transitionen des zustands-konformen Automaten synchronisiert werden können. Diese Transitionen können also immer dann existieren, wenn eine Anwendung einer Zustands-Kompositions-Regel die Nachrichten aus A_1 oder A_2 entfernt, die die Nachrichten-Kompositions-Regel beobachtet. Da die Regeln unabhängig von der Anwendung der Regeln spezifiziert werden, kann ein Entwickler der Regeln dies nicht berücksichtigen.

Im Folgenden bezeichnen wir einen zustands-kompositions-konformen und nachrichten-kompositions-konformen, parallel komponierten Timed Automaten mit *kompositions-konform*.

In diesem und im vorherigen Abschnitt haben wir vorgestellt, wie wir Kompositionsregeln spezifizieren und automatisch anwenden können. Durch das Hinzufügen von Zeitbedingungen und das Entfernen von Zustandskombinationen schränkt die Anwendung von Kompositionsregeln das Zeitverhalten der parallelen Rollenautomaten ein. Es kann daher passieren, dass relevantes Verhalten entfernt wird, wenn ein Zeitintervall gleich Null ist oder wenn ein einzelner Zustand einer Rolle vollständig entfernt wird. Damit kann eine Verletzung der Eigenschaften der Rollenautomaten, welche vor der Anwendung der Kompositionsregeln verifiziert wurden, nicht verhindert werden. Im folgenden Abschnitt werden wir daher einen Ansatz vorstellen, der solche Verletzungen der Eigenschaften entdeckt.

5.3 Erhalt von Rollenverhalten

Die in den beiden vorherigen Abschnitten vorgestellten Kompositionsregeln sind so eingeschränkt, dass eine Verletzung von (zeitlichen) Sicherheitsanforderungen der Rollenverhalten verhindert werden kann. Es kann jedoch nicht vermieden werden, dass die Anwendung von einer Kompositionsregel zu einer Verletzung des sichtbaren Echtzeitverhaltens führt.

Die Anwendung der Kompositionsregel $r_4 = \neg((registered, true) \wedge (convoy, cr > 100))$ auf den komponierten Timed Automaton aus Abbildung 5.8 resultiert in einer neuen Zustands-Invariante ($cr \leq 100 \ \&\& \ ce \leq 2000$) für den Zustand $(convoy, registered, ec_registeredConvoy)$. Eine Konsequenz aus dieser Anwendung ist, dass die ausgehende Nachricht `breakConvoy/` der Transition nicht mehr aktiviert werden kann, da der Time Guard $cr \geq 200$ nicht mehr mit `true` ausgewertet werden kann. Das resultierende Verhalten ist entsprechend nicht mehr in dem kompositions-konformen Automaten enthalten.

Gerade wenn mehrere Kompositionsregeln spezifiziert werden, sind die Verletzungen der Rolleigenschaften nicht einfach durch einen Entwickler zu erkennen. Die alleinige Anwendung von Regel r_4 auf den ursprünglichen parallel komponierten Automaten (siehe Abbildung 5.8) würde zum Beispiel nicht die Ausführbarkeit der `breakConvoy/`-Transition verhindern, da der Automat in den Zustand $(unregistered, convoy)$ wechseln kann und dort die Transition `breakConvoy/` ausführen kann.

Um dieses Problem zu adressieren, definieren wir in diesem Abschnitt auf Basis der Verfeinerungsdefinition in Abschnitt 3.1 die Rollenkonformität (siehe Abschnitt 5.3.1)). Hiermit wird überprüft, ob die Verfeinerungsbeziehung zwischen dem ursprünglichen parallel komponierten Timed Automaton und den kompositions-konformen Timed Automaten eingehalten wird. Da wir hier zusätzlich explizit das parallele Verhalten aller beteiligter Rollen an der Synthese berücksichtigen müssen, können wir nicht die Verfeinerungsüberprüfung aus Abschnitt 3.2 anwenden. Die zu erhaltenen Eigenschaften der Verfeinerungsdefinition ändern sich hierdurch jedoch nicht. In Abschnitt 5.3.2 beschreiben wir zudem, wie ein kompositions-konformer Automat angepasst werden kann, falls ein Deadlock bei der Überprüfung der Rollenkonformität identifiziert wurde.

5.3.1 Rollenkonformität

Aufgrund der kontinuierlichen Zeitsemantik implizieren Timed Automata einen unendlichen Zustandsraum. Eine geeignete diskrete Abstraktion der Timed Automata wird daher benötigt, um eine Analyse zu ermöglichen. Der *Zone Graph* (siehe Definition 6) ist ein weitverbreitetes Verfahren, um den unendlichen Zustandsraum der Timed Automata in endlich viele Zustände zu abstrahieren.

Der Ansatz zur Überprüfung der Rollenkonformität ist daher wie folgt aufgebaut: (1) der Zone Graph für den zustands- und nachrichten-kompositions-konformen Produktautomaten wird erstellt. (2) Es wird überprüft, ob der aus (1) erstellte Zone Graph eine Verfeinerung der einzelnen Rollen-Automaten ist.

Der erste Schritt des Ansatzes ist die Konstruktion des Zone Graphen, um das Modell aller erreichbaren Transitionen, unter Berücksichtigung des zeitlichen Verhaltens, zu erhalten. Ein Ausschnitt² des Zone Graphen konstruiert aus dem kompositions-konformen Automaten der rear-

²Wir zeigen hier nur einen Ausschnitt des Zone Graphen, da der vollständige Graph 186 Zone Zustände und 396 Transitionen besitzt.

Rolle, der registree-Rolle und der Kompositionsregeln $r1$ und eca_2 (siehe Abbildung 5.11) zeigt Abbildung 5.12.

Der Ausschnitt beinhaltet den initialen Zone Zustand $((noConvoy,unregistered,ec_initial),cr==ce \ \& \ ce==ec_c1 \ \& \ ec_c1==0)$ sowie einen direkten und neun indirekte Nachfolger. Der Pfad $/register, /lifetick, /lifetick, /startConvoy, /lifetick, breakConvoy/, /unregister$ zeigt zum Beispiel einen Ablauf, in dem jeder Zustand mindestens einmal besucht wurde.

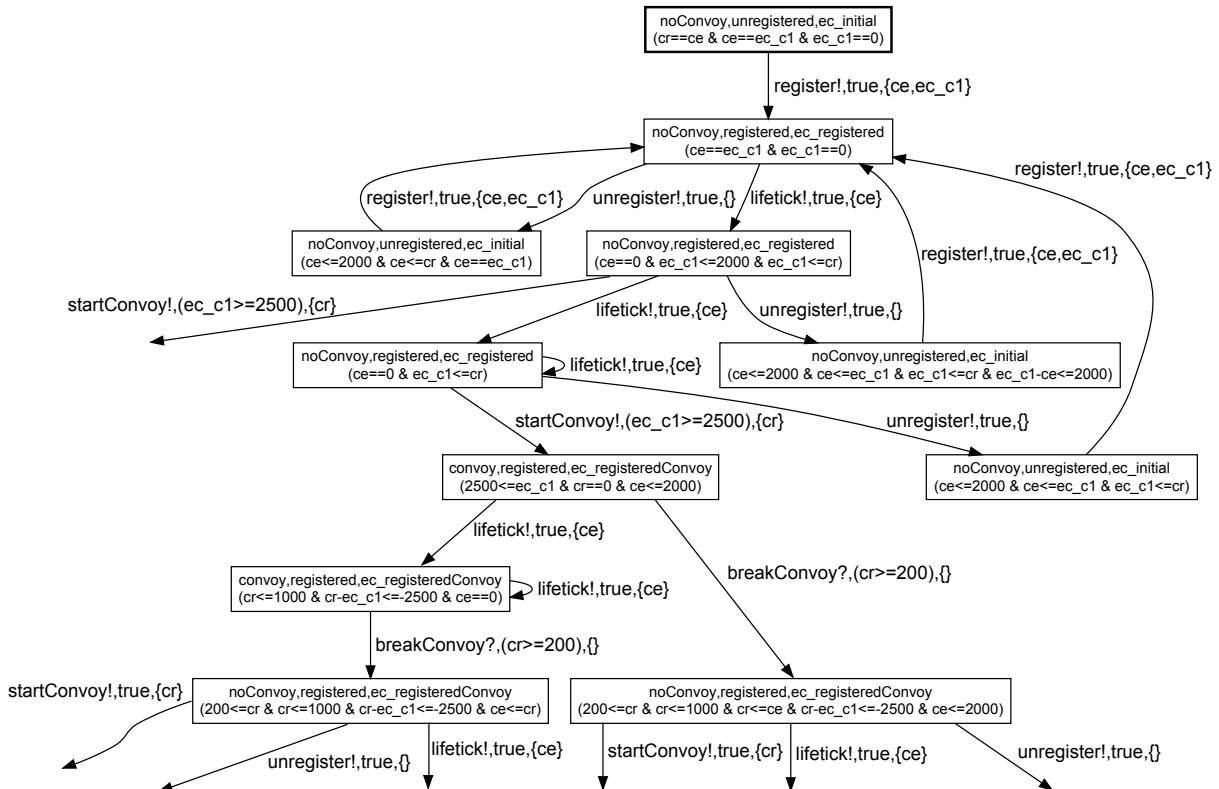


Abbildung 5.12: Ausschnitt eines Zone Graphen des Konvoi-Beispiels (siehe Abbildung 5.11)

In einem rollen-konformen, kompositions-konformen Timed Automaton ist jeder Pfad der ursprünglichen parallelen Komposition des Rollen-Automaten ebenfalls unter Berücksichtigung von Zeitverzögerungen und internen Verhalten enthalten. Um dies in einem gegebenen Zone Graphen herauszufinden, müssen wir überprüfen, ob jeder Zone Zustand immer noch alle Nachrichten des korrespondierenden Timed Automaton-Zustands *anbietet*. Für den Zone Zustand $((noConvoy,unregistered,ec_initial),cr==ce \ \& \ ce==ec_c1 \ \& \ ec_c1==0)$ müssen wir zum Beispiel überprüfen, ob die Nachrichten $/register$ und $/startConvoy$ angeboten werden, da dies die Nachrichten der ausgehenden Transition des Zustands $(noConvoy,unregistered)$ des ursprünglichen parallel komponierten Automaten sind (siehe Abbildung 5.11).

Die Überprüfung des *angebotenen Verhaltens*, muss explizit berücksichtigen, dass beliebig viele Nachrichten-Transitionen der anderen parallelen Rolle dazwischen sein können. Diese sogenannten transitiven Delay-Transitionen werden inhärent durch Zone Graphen berücksichtigt. Problem ist allerdings, dass dabei keine transitiven Transitionen der anderen Rollen betrachtet werden. Damit müssen diese explizit in der Überprüfung berücksichtigt werden.

Um diesen Ansatz beispielhaft zu zeigen, inspizieren wir den initialen Zone Zustand ((noConvoy,unregistered,ec_initial),cr==ce & ce==ec_c1 & ec_c1==0), ob dieser die Nachrichten /register und /startConvoy anbietet. /register wird direkt durch den Zone Zustand angeboten. Die /startConvoy-Nachricht müssen wir suchen, indem wir nur Transitionen mit Nachrichten der registree-Rolle verfolgen, da /startConvoy eine Nachricht der rear-Rolle ist. Diese Transition kann auf dem Pfad /register, /lifetick, /startConvoy gefunden werden sowie auf dem Pfad /register, /lifetick, /lifetick, ..., /startConvoy. Entsprechend bietet der initiale Zone Zustand die gleichen Nachrichten wie der korrespondierende Timed Automaton-Zustand an. Wir bezeichnen einen solchen Zustand mit *konsistent*.

Für zu sendende Nachrichten ist dieser Ansatz ausreichend. Für zu empfangende Nachrichten muss zusätzlich überprüft werden, ob der letzte Zeitpunkt der ursprünglichen Transition durch diesen Zustand eingehalten wird. Wir zeigen dies am Beispiel der ausgehenden breakConvoy/-Transition des Zone Zustands ((convoy,registered,ec_registeredConvoy), 2500<=ec_c1 & cr==0 & ce<=2000). Das Zeitintervall, in dem die breakConvoy/-Transition aktiviert wird, wird wie folgt berechnet (siehe Definition 6): (1) lasse Zeit auf der Zone des Start-Zustands vergehen, (2) schneide die resultierende Zone aus Schritt (1) mit der Invariante des korrespondierten, kompositions-konformen Timed Automaton-Zustands und (3) schneide die resultierende Zone aus Schritt (2) mit den Time Guards der korrespondierten, kompositions-konformen Timed Automaton Transition.

Dieses Zeitintervall muss mit dem Zeitintervall des ursprünglichen parallel komponierten Timed Automaton verglichen werden. Das ursprüngliche Zeitintervall wird auf die gleiche Weise mit den Zustands-Invarianten und Time Guards des ursprünglichen Timed Automaton erstellt. Bei dem Vergleich beider Zeitintervalle, muss berücksichtigt werden, dass das kompositions-konforme Zeitintervall auch später starten kann. Es müssen also nur die oberen Grenzen miteinander verglichen werden.

Um zu überprüfen, ob die Clock Zones gleich sind, subtrahieren wird die kompositions-konforme Zone von der ursprünglichen Zone. Wenn das Ergebnis eine leere Zone ist, sind beide Intervalle gleich und die betrachtete Transition des Zone Graphen bietet das Verhalten der breakConvoy/-Nachricht an. Wenn das Ergebnis eine nicht-leere Zone ist, muss nach einer anderen Transition gesucht werden (wobei, wie oben beschrieben, wieder nur Transitionen der registree-Rolle verfolgt werden). Wird keine Transition gefunden, wird der Zone Zustand als inkonsistent markiert.

Die gesamte Prozedur überprüft jeden Zone Zustand des Zone Graphen, ob er konsistent ist. Die inkonsistenten Zustände und korrespondierende eingehende und ausgehende Nachrichten werden entfernt. Nachdem all diese Zone Zustände entfernt wurden, müssen nochmal alle anderen Zone Zustände überprüft werden, da sich das angebotene Verhalten durch die Entfernung der

Transitionen verändert haben kann. Wenn zu jeder Zeit alle Zone Zustände konsistent sind und der initiale Zone Zustand nicht entfernt wurde, ist der resultierende Timed Automaton rollenkonform.

Durch das Entfernen von Zone Zuständen entspricht der rollenkonforme Timed-Automaton nicht mehr dem korrespondierenden Zone Graphen, da einige Pfade, die in einen Deadlock führen, entfernt wurden. Dies kann allerdings durch Hinzufügen von Time Guards wieder behoben werden, wie in Abschnitt 5.3.2 beschrieben.

Wir fahren fort mit der Formalisierung des Ansatzes. Wie oben beschrieben, müssen wir zwischen zu sendenden und zu empfangenden Nachrichten unterscheiden, da die zu empfangenden Nachrichten die oberen Grenzen des ursprünglichen Zeitintervalls berücksichtigen müssen. Weiterhin müssen wir zwischen dem angebotenen Verhalten eines Timed Automaton-Zustands, der parallelen Komposition der Rollen-Automaten und dem angebotenen Verhalten eines Zone Zustands, des kompositions-konformen Timed Automaton, unterscheiden. Wir fangen an mit der Definition des angebotenen Verhaltens eines Timed Automaton-Zustands. Hiermit wird das angebotene Verhalten eines Timed Automaton-Zustands berechnet.

Definition 58 (Angebotenes Sendeverhalten (Timed Automaton Zustand))

Für einen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$ ist das angebotene Verhalten eines Zustands (einer Location) $l \in L$ und einer Clock Zone $\vartheta \in \Psi(C)$ definiert durch die Funktion *offers* : $L \times \Psi(C) \rightarrow 2^\Sigma$, mit

$$offers_l(l, \vartheta) = \{ /e | \exists (l, /e, g, r, l') \in T : (\vartheta^\uparrow \wedge I(l) \wedge g) \neq false \}.$$

Das *angebotene Sendeverhalten* eines gegebenen Timed Automaton-Zustands l und einer Clock Zone ϑ ist gegeben durch alle zu sendenden Nachrichten, die von dem Zustand l ausgehend von der Zone ϑ erreichbar sind. Das sind im Wesentlichen all die Nachrichten, die als ausgehende Nachrichten des Zustands l markiert sind. Zudem muss allerdings überprüft werden, ob die Transition ausgehend von der gegebenen Zone aktiviert werden kann. Entsprechend müssen wir (1) auf der Eingabe Zone ϑ Zeit vergehen lassen, (2) diese mit der Invariante des Zustands l schneiden und (3) die Zone mit dem Time Guard der Transition schneiden. Auf diese Weise finden wir heraus, ob die korrespondierende Nachricht für eine ausgehende Transition in dem gegebenen Intervall ϑ angeboten wird oder nicht.

Beispielhaft zeigen wir dies an dem Zustand $((convoy,unregistered),cr \leq 50)$ des zustandskonformen Automaten (siehe Abbildung 5.10). Wir nehmen zudem an, dass die ausgehende *breakConvoy*-Transition mit einer eingehenden Nachricht */breakConvoy* markiert wurde. Die Eingangs Zone ist gegeben mit $(cr = 0 \wedge ce \geq 0)$. Die folgende Berechnung zeigt, ob die */breakConvoy*-Nachricht angeboten wird:

$$\begin{aligned} & (cr = 0 \wedge ce \geq 0)^\uparrow \wedge (cr \leq 50) \wedge (cr \geq 200) \\ &= (cr \leq ce) \wedge (cr \leq 50) \wedge (cr \geq 200) \\ &= (cr \leq ce \wedge cr \leq 50) \wedge (cr \geq 200) \\ &= false. \end{aligned}$$

Da das Ergebnis *false* ist, ist die einzige angebotene Nachricht des Zustands $((\text{convoy}, \text{unregistered}), \text{cr} \leq 50) / \text{register}$, da:

$$\text{offers}_1((\text{convoy}, \text{unregistered}), \text{cr} \leq 50), (\text{cr} = 0 \wedge \text{ce} \geq 0) = \{ / \text{register} \}.$$

Wir fahren fort mit der Definition des angebotenen Empfangsverhaltens eines Timed Automaton-Zustands. Das angebotene Empfangsverhalten muss die obere Grenze des ursprünglichen Intervalls berücksichtigen und kann von dem unterem Intervall abstrahieren. Entsprechend muss die untere Grenze aus den Clock Zones entfernt werden. Dies nennen wir *schwächste Verzögerungs-Vorbedingung* (siehe auch schwächste Vorbedingung in [BY03, p. 106]) und ist wie folgt definiert.

Definition 59 (Schwächste Verzögerungs-Vorbedingung auf Clock Zones)

Für eine Clock Zone $\vartheta \in \Theta(C)$ ist die schwächste Verzögerungs-Vorbedingung, beschrieben mit ϑ^\Downarrow , definiert durch:

$$\begin{aligned} \vartheta^\Downarrow = \{ \nu \mid & \forall \nu' \in \vartheta, \forall c \in C, \forall \delta \in \mathbb{R}_+ : \\ & (\forall c \in C : \delta \leq \nu'(c)) \Rightarrow \nu(c) = \nu'(c) - \delta \}. \end{aligned}$$

Die *schwächste Verzögerungs-Vorbedingung* auf einer Clock Zone entfernt alle unteren Schranken in der Zone. Dabei müssen Bedingungen über Clock-Differenzen berücksichtigt werden. Entsprechend müssen alle möglichen Werte δ von jeder Clock Zone subtrahiert werden, so lange wie die Bewertung nicht negativ wird. Für eine gegebene Clock-Bewertung ν' müssen alle Clock-Werte größer oder gleich Null nach der Subtraktion von *delta* sein. Die schwächste Verzögerungs-Vorbedingung wird während der Berechnung des *angebotenen Empfangsverhaltens* angewandt.

Definition 60 (Angebotenes Empfangsverhalten (Timed Automaton Zustand))

Für einen gegebenen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$ ist das angebotene Verhalten eines Zustands (einer Location) $l \in L$ und einer Clock Zone $\vartheta \in \Psi(C)$ definiert durch die Funktion $\text{offers}_\tau : L \times \Psi(C) \rightarrow 2^\Sigma$, mit

$$\begin{aligned} \text{offers}_\tau(l, \vartheta) = \{ (e/, \vartheta_e) \mid & \exists (l, e/, g, r, l') \in T, \vartheta_e = (\vartheta^\uparrow \wedge I(l) \wedge g)^\Downarrow : \\ & \vartheta_e \neq \text{false} \}. \end{aligned}$$

Im Unterschied zu dem angebotenen Sendeverhalten ist das *angebotene Empfangsverhalten* eine Menge von Nachrichtenpaaren $e/$ und einer Clock Zone ϑ_e . Die Berechnung, ob eine korrespondierende Transition der Nachrichten aktiviert ist, ist die gleiche wie für die zu empfangenden Nachrichten. Allerdings wird hier das resultierende Zeitintervall genutzt, um die Zone zu repräsentieren, wo die Nachricht $e/$ aktiv ist. Die unteren Grenzen werden wie oben beschrieben entfernt.

Beispielhaft zeigen wir dies an dem Zustand $((\text{convoy}, \text{registered}), \text{cr} \leq 1000 \ \&\& \ \text{ce} \leq 2000)$ und der ausgehenden *breakConvoy/-Transition* (siehe Abbildung 5.10). Die Berechnung der

korrespondierenden Clock Zone ϑ_e wird im Folgenden vorgestellt, wobei die Eingabe Zone $(cr \leq 50 \wedge ce = 0)$ gegeben ist durch:

$$\begin{aligned}
 \vartheta_{breakConvoy/} &= ((cr \leq 50 \wedge ce = 0)^\uparrow \wedge (cr \leq 1000 \wedge ce \leq 2000) \wedge (cr \geq 200))^\downarrow \\
 &= ((cr - ce \leq 50 \wedge ce \leq cr) \wedge (cr \leq 1000 \wedge ce \leq 2000) \wedge (cr \geq 200))^\downarrow \\
 &= ((cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000) \wedge (cr \geq 200))^\downarrow \\
 &= (cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000 \wedge cr \geq 200)^\downarrow \\
 &= cr - ce \leq 50 \wedge ce \leq cr \wedge cr \leq 1000.
 \end{aligned}$$

Bis hier her haben wir die Berechnung des angebotenen Verhaltens eines Timed Automaton-Zustands definiert. Im Folgenden definieren wir das angebotene Verhalten von Zone Zuständen von kompositions-konformen, parallel komponierten Timed Automaton. Wir müssen dabei berücksichtigen, dass für eine gegebene Nachricht, Nachrichten von anderen Rollen als internes Verhalten der korrespondierenden Komponente aufgefasst werden können. Wir definieren daher eine *transitive Transitionsbeziehung* für den Zone Graphen, um das angebotene Verhalten eines Zone Zustands zu berechnen.

Definition 61 (Transitive Transitionsbeziehung (Zone Graph))

Für einen Zone Graphen $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$, erstellt aus einem kompositions-konformen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$, welcher wiederum aus dem Timed Automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ zusammengesetzt ist, definieren wird die Menge von transitiven Transitionen T_τ durch:

$$T_\tau = \{((s, e, s'), s_{pre}) \mid ((s, e, s'), s_{pre}) \in T_{\tau_1} \vee ((s, e, s'), s_{pre}) \in T_{\tau_2}\}$$

mit

$$\begin{aligned}
 T_{\tau_1} &= \{((s, e_1, s'), s_{pre}) \mid \exists e_1 \in \Sigma_1 : (s, e_1, s') \in T_\Theta, s_{pre} = s \vee \\
 &\quad (\exists e_2 \in \Sigma_2 : (s, e_2, s'') \in T_\Theta, s_{pre} = s'' \wedge \\
 &\quad \exists e_1 \in \Sigma_1 : ((s'', e_1, s'), s_{pre}) \in T_{\tau_1})\} \\
 T_{\tau_2} &= \{((s, e_2, s'), s_{pre}) \mid \exists e_2 \in \Sigma_2 : (s, e_2, s') \in T_\Theta, s_{pre} = s \vee \\
 &\quad (\exists e_1 \in \Sigma_1 : (s, e_1, s'') \in T_\Theta, s_{pre} = s'' \wedge \\
 &\quad \exists e_2 \in \Sigma_2 : ((s'', e_2, s'), s_{pre}) \in T_{\tau_2})\}.
 \end{aligned}$$

Eine transitive Verzögerung wird damit nicht explizit berücksichtigt, wie dies typischerweise der Fall ist für transitive Transitionsbeziehungen für zeitbehafte Transitionssysteme [WL97, TY01], sondern implizit durch die Konstruktion des Zone Graphen. Zudem berücksichtigen wir explizit den Zone Zustand s_{pre} durch Annotation an jeder transitiven Transition. Hierdurch wird direkt eine Transition mit der korrespondierenden Nachricht angeboten, womit ein Vergleich der Zeitintervalle der empfangenden Nachrichten ermöglicht wird. Der Zone Zustand s_{pre} wird *letzter Vorgänger* genannt.

Beispielhaft zeigen wir die transitive Transitionsbeziehung an dem Zone Zustand $((noConvoy, unregistered, ec_initial), cr == ce \ \& \ ce == ec_c1 \ \& \ ec_c1 == 0)$ des Zone Graphen aus Abbildung 5.12. Nehmen wir an das $T_{\tau-s_1-registree}$ die Menge der transitiven Transitionen für die registree-Rolle repräsentiert und das $T_{\tau-s_1-rear}$ die für die rear-Rolle repräsentiert. Es werden nur die ausgehenden transitiven Transitionen des initialen Zone Zustands betrachtet. Da die erste und einzige ausgehende Transition des initialen Zone Zustands mit register annotiert ist, ist die einzige transitive Transition in Bezug auf die Nachrichten der registree-Rolle die /register-Transition:

$$T_{\tau-s_1-registree} = \{((s_1, /register, s_2), s_1)\},$$

mit

$$\begin{aligned} s_1 &= ((noConvoy, unregistered, ec_initial), cr = ce = ec_c1 = 0) \text{ und} \\ s_2 &= ((noConvoy, registered, ec_registered), ce = ec_c1 = 0). \end{aligned}$$

Für die rear-Rolle müssen wir die /startConvoy-Transition untersuchen, indem wir den Pfad /register, /lifetick, /startConvoy and /register, /lifetick, /lifetick, /startConvoy betrachten. Die Menge $T_{\tau-rear}$ enthält entsprechend zwei transitive Transitionen:

$$T_{\tau-s_1-rear} = \{((s_1, /startConvoy, s_5), s_3), ((s_1, /startConvoy, s_6), s_4)\}$$

mit

$$\begin{aligned} s_3 &= ((noConvoy, registered, ec_registered), \\ &\quad ce = 0 \wedge ec_c1 \leq 2000 \wedge ec_c1 \leq cr), \\ s_4 &= ((noConvoy, registered, ec_registered), ce = 0 \wedge ec_c1 \leq cr) \text{ und} \\ s_6 &= ((convoy, registered, ec_registeredConvoy), \\ &\quad ec_c1 \geq 2500 \wedge cr = 0 \wedge ce \leq 2000). \end{aligned}$$

s_5 wurde nicht dargestellt. Die Menge aller ausgehender transitiven Transitionen $T_{\tau-s_1}$ von s_1 , ist die Vereinigung der Menge $T_{\tau-s_1-rear}$ und $T_{\tau-s_1-registree}$: $T_{\tau-s_1} = T_{\tau-s_1-registree} \cup T_{\tau-s_1-rear}$.

Die transitive Transitionsbeziehung wird in der Berechnung des angebotenen Verhaltens eines Zone Zustands angewandt. Im Folgenden werden wir das angebotene Verhalten definieren. Wir beginnen mit der Definition des *angebotenen Senderverhalten*.

Definition 62 (Angebotenes Senderverhalten (Zone Zustand))

Für einen Zone Graphen $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ eines kompositions-konformen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$ (erstellt aus den Timed Automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$), ist das angebotene Verhalten eines Zone Zustands (l, ϑ) definiert durch die Funktion: $offers : L \times \Psi(C) \rightarrow 2^\Sigma$ mit

$$offers_l(s) = \{/e \mid ((s, /e, s'), s_{pre}) \in T_\tau\}.$$

Das *angebotene Sendeverhalten* eines Zone Zustands bestimmt die Menge der sendenden Nachrichten eines bestimmten Zone Zustands s . Diese Nachrichten sind all die Nachrichten welche erreichbar sind durch transitive Transitionsbeziehung ausgehend von s . Da für das Senden von Nachrichten das Zeitintervall in welchem die Nachricht aktiviert wird nicht relevant ist, wird der Zone Zustand s_{pre} nicht betrachtet. Die Menge der angebotenen zu versendenden Nachrichten für den initialen Zone Zustand $((noConvoy,unregistered,ec_initial),cr==ce \ \& \ ce==ec_c1 \ \& \ ec_c1==0)$ des Zone Graphen aus Abbildung 5.12 wurde im vorherigen Beispiel berechnet und ist entsprechend $\{/register, /startConvoy\}$. Wir fahren mit der Definition des *angebotenen Empfangsverhalten* fort.

Definition 63 (Angebotenes Empfangsverhalten (Zone Zustand))

Für einen Zone Graphen $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ eines kompositions-konformen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$, welcher aus den Timed Automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ zusammengesetzt wurde, ist das angebotene Verhalten eines Zone Zustands (l, ϑ) definiert durch die Funktion $offers : L \times \Psi(C) \rightarrow 2^\Sigma$ mit

$$offers?(s) = \{(e/, \vartheta_e) \mid \exists ((s, e/, s'), (l, \vartheta)) \in T_\tau, \exists (l, e/, g, r, l') \in T, \vartheta_e = (\vartheta^\uparrow \wedge I(l) \wedge g)^\downarrow : \vartheta_e \neq false\}.$$

Ähnlich zu dem angebotenen Empfangsverhalten eines Timed Automaton-Zustands (siehe Definition 60), besteht das angebotene Empfangsverhalten eines Zone Zustands aus Tupeln von empfangenden Nachrichten $e/$ und korrespondierenden Clock Zones ϑ_e . Die Clock Zones repräsentieren das Zeitintervall in dem die Nachricht angeboten wird. Für dieses Zeitintervall wird der letzte Vorgänger der transitiven Transitionsbeziehung berücksichtigt. Dies wird benötigt, da sich die Clock Zone möglicherweise entlang der Transitionen und den Zuständen dazwischen verändern kann. Daher wird die Clock Zone direkt vor der Transition, die die Nachricht $e/$ anbietet, benötigt, um das Zeitintervall für das Ereignis zu bestimmen. Das Zeitintervall wird genauso berechnet, wie zu dem angebotenen Empfangsverhalten eines Timed Automaton-Zustands beschrieben (siehe Definition 60). Ein Beispiel für die Berechnung des angebotenen Verhaltens wurde bereits in diesem Abschnitt zur Beschreibung der transitiven Transitionsbeziehung gezeigt.

Um zu verifizieren, ob ein Zone Zustand, welcher aus dem kompositions-konformen Automaten abgeleitet wurde, konsistent ist, muss überprüft werden, ob die Menge von angebotenen Nachrichten äquivalent zur Menge der angebotenen Nachrichten der ursprünglichen parallelen Komposition der Rollenautomaten ist. Für die zu sendenden Nachrichten ist es ausreichend zu überprüfen, ob die Menge der erreichbaren Nachrichten äquivalent ist. Für die zu empfangenden Nachrichten müssen zudem die Zeitintervalle berücksichtigt werden, in welchen die korrespondierenden Transitionen aktiviert wurden. Da die untere Grenze von beiden Zeitintervallen entfernt wurde, muss für den Vergleich nur eine Clock Zone von der anderen subtrahiert werden (siehe folgende Definition).

Definition 64 (Subtraktion auf Clock Zones)

Für zwei Clock Zones $\vartheta_1, \vartheta_2 \in \Theta(C)$ ist die Subtraktion $\vartheta_2 - \vartheta_1$ definiert durch

$$\vartheta_2 - \vartheta_1 = \{\nu \mid \nu \in \vartheta_2 \wedge \nu \notin \vartheta_1\}.$$

Die Subtraktion auf Clock Zones ist ebenso definiert wie die Subtraktion auf Mengen, da eine Clock Zone aus einer *Menge* von Clock-Bewertungen besteht (siehe Definition 6). Wenn also eine Clock Zone ϑ_1 von einer Clock Zone ϑ_2 subtrahiert wird, dann entfernen wir einfach alle Clock-Bewertungen, die in ϑ_1 und ϑ_2 enthalten sind von ϑ_2 . Das Ergebnis ist die modifizierte Menge ϑ_2 . Wenn die resultierende Menge leer ist, dann sind ϑ_1 und ϑ_2 äquivalent. Diese Eigenschaft wird ausgenutzt, um die Gleichheit von Clock Zones für die Überprüfung der Konsistenz zwischen dem kompositions-konformen Automaten und der ursprünglichen parallelen Komposition zu bestimmen.

Auf Basis dieser Definition können wir im Folgenden die Rollen-Konformität definieren.

Definition 65 (Rollen-Konformität)

Lasse $A = (L, l^0, \Sigma, C, I, T)$ ein kompositions-konformer, parallel komponierter Timed Automaton sein, welcher aus den Timed Automata $A_1 = (L_1, l_1^0, \Sigma_1, C_1, I_1, T_1)$ und $A_2 = (L_2, l_2^0, \Sigma_2, C_2, I_2, T_2)$ und den nachrichten-kompositions-konformen Automaten $A_{EC} = (L_{EC}, l_{EC}^0, \Sigma_{EC}, C_{EC}, I_{EC}, T_{EC})$ erstellt wurde. Weiterhin lasse $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ den korrespondierenden Zone Graphen sein und lasse $A_P = (L_P, l_P^0, \Sigma_P, C_P, I_P, T_P)$ die parallele Komposition $A_1 \parallel A_2$ sein. Wir definieren A als rollen-konform wenn,

$$\exists Z'_A = (S'_\Theta \subseteq S_\Theta, s^0, \Sigma, C, T'_\Theta \subseteq T_\Theta)$$

und

$$\begin{aligned} \forall ((l_1, l_2, l_e), \vartheta) \in S'_\Theta : \\ offers_1(((l_1, l_2, l_e), \vartheta)) = offers_1((l_1, l_2), \vartheta) \wedge \\ offers_?(((l_1, l_2, l_e), \vartheta)) \supseteq offers_?((l_1, l_2), \vartheta), \end{aligned}$$

mit $(l_1, l_2) \in L_P$ und

$$(e/, \vartheta_e) \in offers_?(((l_1, l_2, l_e), \vartheta)) = (e_p/, \vartheta_{e_p}) \in offers_?((l_1, l_2), \vartheta) \Leftrightarrow e/ = e_p/ \wedge \vartheta_{e_p} - \vartheta_e = false.$$

Ein kompositions-konformer Timed Automaton A ist *rollen-konform* zu der parallelen Komposition $A_P = A_1 \parallel A_2$ der ursprünglichen Rollenautomaten A_1 und A_2 , wenn ein korrespondierender Zone Graph Z'_A existiert, wobei jeder Zone Zustand konsistent ist. Der Zone Graph kann dabei möglicherweise weniger Zone Zustände und korrespondierende Transitionen als die ursprünglichen Automaten aufweisen. Es gilt daher, dass jeder Zone Zustand $((l_1, l_2, l_e), \vartheta) \in Z'_A$ ausgehend von der Zone ϑ das gleiche Verhalten anbietet wie der korrespondierende Zone Zustand $(l_1, l_2) \in A_P$. Weiterhin gilt, dass ein Zone Zustand das gleiche Verhalten anbietet, wenn die Menge der zu sendenden Nachrichten gleich ist und wenn die Menge der zu empfangenden Nachrichten des Zone Zustands alle $(e/, \vartheta_e)$ Tupel des ursprünglichen Automaten enthält. Es können also auch mehr Tupel enthalten sein, da empfangende Nachrichten angeboten sein können, deren Zeitintervall kleiner als das ursprüngliche ist.

Die Analyse des gesamten Beispiels ergibt, dass der kompositions-konforme Timed Automata rollen-konform ist. Um zudem zu zeigen, dass dieser auch eine korrekte Verfeinerung der

Rollenautomaten ist, müssen wir zudem z.B. mit UPPAAL überprüfen, dass der kompositions-konforme Timed Automata keine Time Stopping Deadlocks enthält. Dies ist für unser Beispiel der Fall, womit der Automat eine korrekte Verfeinerung der Rollenautomaten ist.

Da die Definition der Rollenkonformität nicht verlangt, dass der ursprüngliche Zone Graph Z_A des synthetisierten kompositions-konformen Automaten äquivalent zu Z'_A ist, kann es sein, dass die Time Stopping Deadlock Analyse fehlschlägt, obwohl der kompositions-konforme Automat rollen-konform ist. Um dieses Problem zu beheben müssen die Zone Zustände des ursprünglichen Zone Graphen Z_A , welche nicht konsistent sind, entfernt werden, um die Situationen zu vermeiden in denen der Timed Automaton Transitionen ausführt, die in einen Deadlock führen. Im Folgenden Abschnitt stellen wir vor, wie dies erreicht werden kann.

5.3.2 Erhalt von Deadlock Freiheit

Für einen rollen-konformen, kompositions-konformen Timed Automaton A ist nicht zugesichert, dass es einige Pfade gibt, die nicht korrekt das Verhalten der korrespondierenden Rollenautomaten verfeinern. Dies liegt daran, dass der Zone Graph Z'_A des rollen-konformen Automaten weniger Zone Zustände haben kann als der Zone Graph Z_A des kompositions-konformen Automaten. Hierdurch kann es Zone Zustände in Z'_A geben, die in einen Time Stopping Deadlock führen. Im Folgenden stellen wir einen Ansatz vor, der automatisch durch Anpassung des Zone Graphen an diesen kritischen Stellen die Deadlock Freiheit erhält. Hierdurch kann ein manuelles eingreifen des Entwicklers zum Teil verhindert werden.

Um den Ansatz beispielhaft darzustellen, passen wir die Rollen rear und registree an, wie in Abbildung 5.13 und Abbildung 5.14 dargestellt. Beide Automaten können nun höchstens eine Zeiteinheit in jedem Zustand verweilen. Weiterhin müssen sie wenigstens eine Zeiteinheit in den Zuständen convoy und registered verweilen. Wenn die Automaten zurück in den initialen Zustand wechseln, wird die korrespondierende Uhr zurückgesetzt.

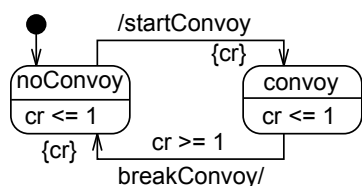


Abbildung 5.13: Modifizierte einfache rear-Rolle

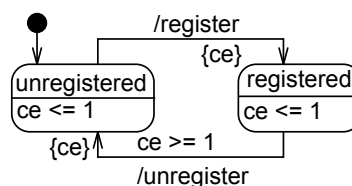


Abbildung 5.14: Modifizierte einfache registree-Rolle

Wir wenden nun auf der parallelen Komposition dieser Automaten die Zustands-Kompositionsregel $r_1 = \neg((unregistered, true) \wedge (convoy, true))$ an. Der resultierende komponierte Automat ist kompositions-konform (siehe Abbildung 5.15).

Um nun festzustellen, ob der Automat rollen-konform ist, erstellen wir den korrespondierenden Zone Graphen (siehe Abbildung 5.16). Bis auf der Zustand $((convoy, registered), cr-ce==1$

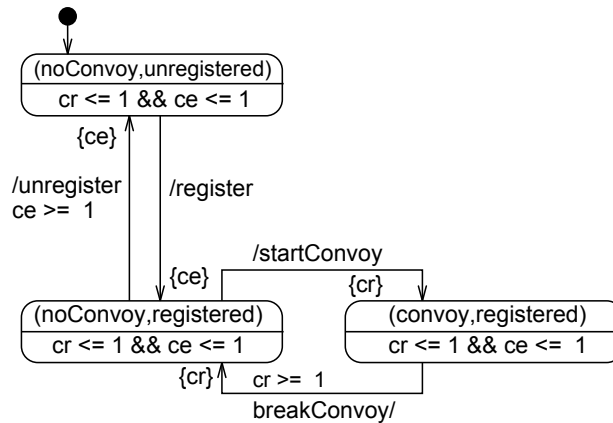


Abbildung 5.15: Kompositions-konformer Automat der vereinfachten Rollenautomaten

& ce==1), welcher keine ausgehenden Transitionen besitzt und damit in einen Deadlock führt, ist jeder Zone Zustand dieses Zone Graphen konsistent. Wenn wir diesen Zustand entfernen, erhalten wir einen Zone Graphen, indem jeder Zone Zustand die benötigten Ereignisse des ursprünglichen Rollenautomaten anbietet. Dann ist der kompositions-konforme Timed Automaton auch rollen-konform.

Der Zone Graph enthält damit Pfade, welche nicht die benötigten Nachrichten der einzelnen Rollenautomaten korrekt anbieten. Dies sind all die Pfade, welche in den Zustand ((convoy,registered),cr-ce==1 & ce==1) über die startConvoy-Transition führen. Daher müssen wir diese Transition entfernen, um einen Timed Automaton zu erhalten, indem keine Ausführung einer Transition in einen Deadlock führt. Im Folgenden definieren wir das Entfernen solcher Transitionen.

Definition 66 (Entfernen einer Transition eines Zone Graphen)

Gegeben sei ein Zone Graph $Z_A = (S_\Theta, s^0, \Sigma, C, T_\Theta)$ eines kompositions-konformen Timed Automaton $A = (L, l^0, \Sigma, C, I, T)$, eine Timed Automaton Transition $t = (l, e, g, r, l')$ $\in T$, ein initialer Zone Zustand $s = (l, \vartheta) \in S_\Theta$ und eine Zone Graph Transition $t_\Theta = (s, e, s')$. Die Transition t_Θ wird von dem Timed Automaton A entfernt, und daher auch von dem Zone Graphen Z_A , durch Ersetzen des Guards g der Transition t mit dem Guard g_r , definiert durch:

$$g_r = g \wedge (\text{true} - (\vartheta^\uparrow \wedge I(l))).$$

Um also diese Zone Graphen Transitionen von dem korrespondierenden Timed Automaton zu entfernen, ersetzen wir den Time Guard der Transition t durch den modifizierte Time Guard g_r . Dieser Time Guard teilt den ursprünglichen Time Guard g mit dem Intervall, welches nicht beschränkt ist. Dieses Intervall wird durch Subtraktion des beschränkten Intervalls des initialen Zone Zustands von der universellen Menge von Clock-Bewertungen mit *true* berechnet. Das Ergebnis ist ein Guard g_r , welcher die Clock-Bewertungen des ursprünglichen Guards g beinhaltet und die Clock-Bewertungen des Guards entfernt, welche die Transition in dem betroffenen Zustand l aktiviert.

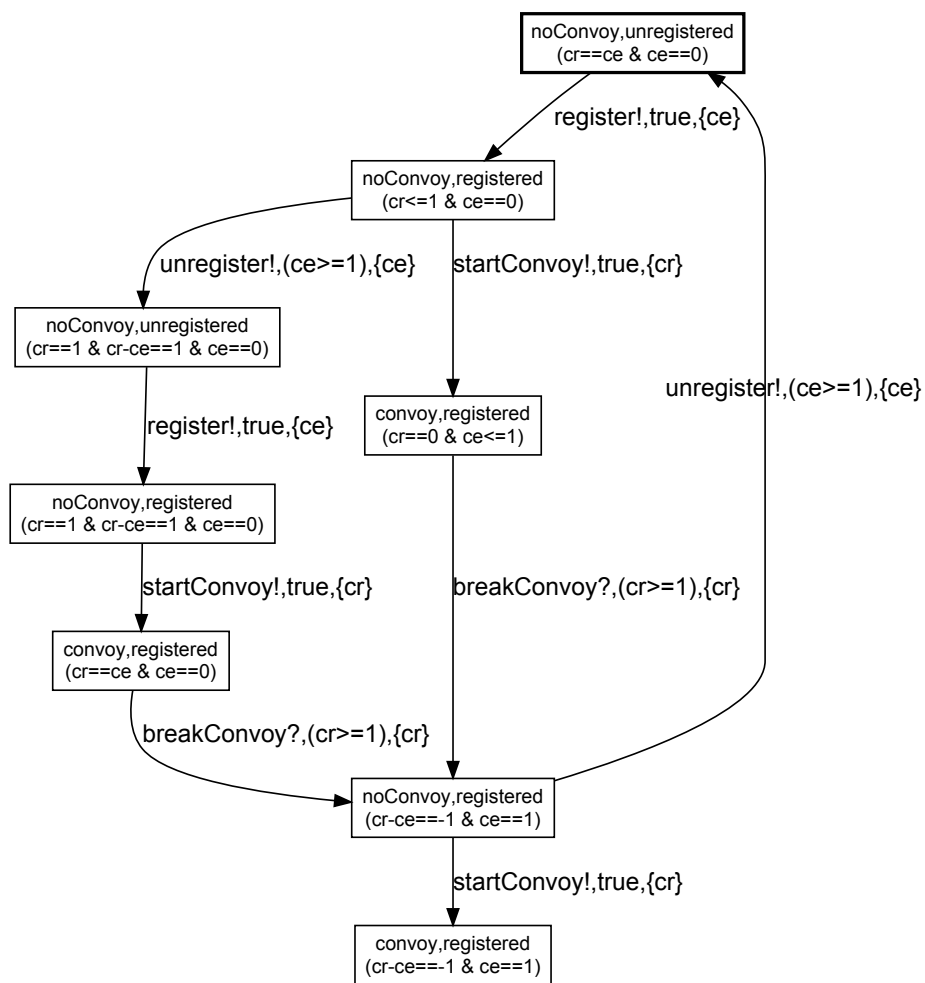


Abbildung 5.16: Zone Graph des vereinfachten kompositions-konformen Timed Automaton (siehe Abbildung 5.14)

Für den Guard der `/startConvoy`-Transition des kompositions-konformen Timed Automaton, ergibt die Berechnung den in Abbildung 5.17 dargestellten Guard. Beachte, dass der Guard eine Disjunktion beinhaltet, die nach Definition der Clock Constraints nicht erlaubt ist (siehe Definition 3). Dies kann durch Aufteilung der Transition in zwei Transitionen mit den gleichen Nachrichten und Clock Resets aufgelöst werden. Die Time Guards werden von den unterschiedlichen Bedingungen des ursprünglichen disjunktiven Time Guards bestimmt. Aus Vereinfachungsgründen haben wir das nicht in dem Beispiel gezeigt.

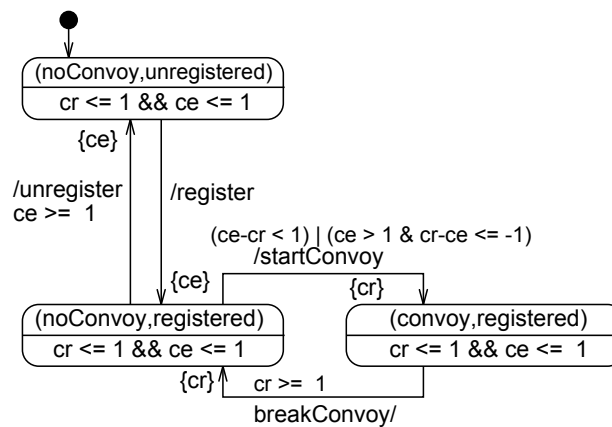


Abbildung 5.17: Modifizierter rollen-konformer Automat aus Abbildung 5.15

Der Zone Graph für die modifizierten rollen-konformen Automaten zeigt Abbildung 5.18. Jeder Pfad dieses Zone Graphen verfeinert das Verhalten der beteiligten Rollenautomaten korrekt. Entsprechend haben wir erfolgreich die Transition entfernt, die in einen Deadlock führt.

Es ist allerdings auch möglich, dass Time Stopping Deadlocks existieren, die nicht nur durch Analyse der ausgehenden Transitionen der Zone Zustände gefunden werden können. Trotz unserer Analyse muss entsprechend immer noch mittels eines Model Checkers nach Time Stopping Deadlocks gesucht werden.

In unserem Beispiel hat UPPAAL einen Deadlock über folgenden Pfad entdeckt: $(/register, ce = cr = 0)$, $(/startConvoy, ce = cr > 0)$, $(deadlock, ce > 0 \wedge cr = 0)$. Dies ist ein Deadlock, da der Zeit Guard der einzigen ausgehenden Transition des Zustands $(convoy,registered)$ wenigstens für eine Zeiteinheit in dem Zustand verweilen muss. Aufgrund der Zustandsinvariante und dem Wert von ce , der größer als Null ist, ist dies allerdings nicht möglich.

Eine Konsequenz hieraus ist, dass der Entwickler diese Deadlocks manuell entfernen muss. Bei der Anpassung dürfen keine Zeitintervalle verletzt werden, da hierdurch die Simulationsbeziehung zwischen der ursprünglichen parallelen Komposition und dem kompositions-konformen Automat verletzt würde. Nachdem die Deadlocks entfernt wurden muss der Timed Automaton noch einmal auf Rollenkonformität überprüft werden. Ist dieser Test erfolgreich, ist der resultierende Timed Automaton eine korrekte Verfeinerung der beteiligten Rollen.

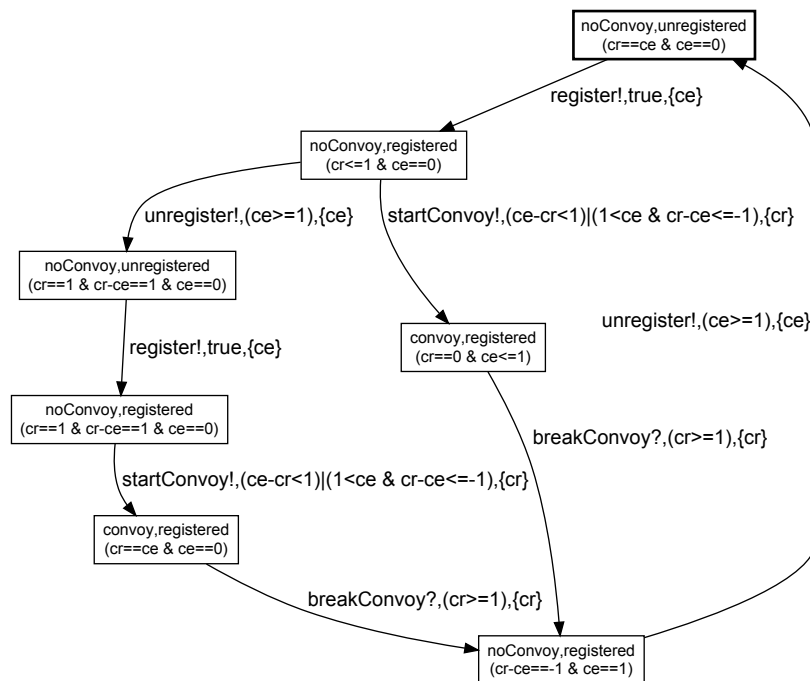


Abbildung 5.18: Deadlock-freier Zone Graph des modifizierten Automaten aus Abbildung 5.17

Offensichtlich können auch diese komplizierteren Deadlocks automatisch entfernt werden, in dem ganze Pfade angepasst werden, die in einen Deadlock führen können. Dies führt dazu, dass die in UPPAAL entwickelten Algorithmen zur Deadlock-Erkennung größtenteils nachimplementiert werden müssten, um eine automatische Anpassung zu ermöglichen. Im Rahmen dieser Arbeit wurde hierauf verzichtet.

5.4 Weitere Anwendungsfälle

Wie zu Abbildung 5.5 beschrieben, können wir weitere Anwendungsfälle von abhängigen Rollenverhalten betrachten. Es ist möglich, dass eine Synchronisation auf unterschiedlichen Hierarchieebenen stattfindet. Damit ist gemeint, dass die abhängigen Verhalten zum Teil durch eine eingebettete Komponente umgesetzt sind, die in der Synchronisation berücksichtigt werden müssen. Ein weiterer Fall sind Multi-Ports, deren Verhalten erst zur Laufzeit durch Strukturanpassungen ausgeprägt wird. Neben diesen lokalen Abhängigkeiten, können auch Abhängigkeiten direkt auf Muster-Ebene entstehen, wenn diese allgemein, komponentenunabhängig, aufgelöst werden sollen. Im Folgenden diskutieren wir, wie der vorgestellte Ansatz mit diesen Anwendungsfällen umgehen kann.

Synchronisation auf unterschiedlichen Hierarchieebenen Wenn wir gezeigt haben, dass die eingebettete(n) Komponente(n) eine korrekte Verfeinerung der übergeordnete(n) Komponente(n) ist, bzw. sind, müssen wir in der Synthese lediglich das Portverhalten der eingebetteten Komponente(n) berücksichtigen. Die Delegation zum übergeordneten Port kann einfach über das synthetisierte Verhalten erfolgen, wie in Abbildung 5.5 dargestellt. Es muss dabei, wie für den schnittstellen-beschränkten Automaten definiert (siehe Definition 20), eine Filterung der Ereignisse durchgeführt werden, damit nur die für einen Port relevanten Ereignisse weitergeleitet werden. Durch eine eindeutige Bezeichnung der Nachrichten durch die Schnittstelle des übergeordneten Ports ist dies einfach realisierbar.

Synchronisation von Multi-Ports Als grundlegender Formalismus wurde für die Synthese Timed Automata verwendet. Daher können grundsätzlich keine Strukturanpassungen in der Synthese berücksichtigt werden, wie dies durch ein Adaptionsverhalten ermöglicht wird.

Für den Fall, dass das Adaptionsverhalten alle Abhängigkeiten berücksichtigt, kann eine Synchronisation mit dem Adaptionsverhalten durchgeführt werden. Nach der grundlegenden Idee der Adaptionsverhalten wie in [HHG08] präsentiert, gilt dies für PARAMETERIZED REAL-TIME COORDINATION PATTERN.

Wurde ein PARAMETERIZED REAL-TIME STATECHART allerdings nicht nach diesen Kriterien umgesetzt, so kann eine Synthese nur durchgeführt werden, wenn die Obergrenze der Anzahl der Instanzen bekannt ist. So kann für die gesamte Anzahl der Portinstanzen ein gemeinsames Verhalten synthetisiert werden.

Verteilung von Verhalten Wenn Anforderungen über mehrere Rollen unterschiedlicher Muster definiert werden, so ergibt die Synthese ein Verhalten, welches potentiell verteilt von mehreren Komponenten ausgeführt werden kann. Da die Synthese ein Gesamtverhalten synthetisiert, muss, um ein verteiltes ausführen zu ermöglichen, das Verhalten wieder zurück in die einzelnen Rollen verteilt werden. Ausgehend von den bekannten Rollenverhalten müssen die einzelnen Zustände der jeweiligen Rollen identifiziert werden und die Konkretisierungen, wie z.B. Synchronisation zwischen den Rollen, berücksichtigt werden.

Im Allgemeinen ist dies das gleiche Problem wie Standard Syntheseansätze beschreiben, die lediglich einen globalen „Controller“ synthetisieren (z.B. [HKP05]). Eine einfache Möglichkeit dieses Problem generell zu lösen ist, dass jeder der Rollenverhalten das Gesamtverhalten anwendet. Die Synchronisationsnachrichten werden dann verteilt verschickt. Es muss dabei berücksichtigt werden, dass die Synchronisation zusätzlich Zeit durch die vernetzte unterliegende Struktur benötigt. Um das Verhalten lokal möglichst klein zu halten, kann zudem das nicht benötigte Verhalten der anderen Rolle aus dem Gesamtverhalten geschnitten werden.

5.5 Diskussion

Mit dem hier vorgestellten Ansatz bieten wir eine Unterstützung zur Beschreibung von Abhängigkeiten zwischen Rollen, die eine Komponente anwendet, an und ermöglichen unter Berücksichtigung der abhängigen Rollen eine automatische Synthese des Komponentenverhaltens. Dieser Ansatz ist grundsätzlich auf beliebige abhängige Timed Automata anwendbar, womit die skizzierten Szenarien aus Abbildung 2.1 umgesetzt werden können.

Gibt es Anforderungen, die beliebig strukturelle Abhängigkeiten zwischen Multi-Ports erfordern, so ist eine effiziente Umsetzung allerdings nicht gegeben. Dies liegt daran, dass explizit die Automaten aller Portinstanzen, deren obere Anzahl bekannt sein muss, berücksichtigt werden müssen. Ein Ausblick für weiterführende Arbeiten ist daher, TIMED STORY CHARTS direkt als unterliegenden Formalismus für die Synthese zu betrachten.

Wie in der Einleitung gefordert, unterstützt der vorgestellte Syntheseansatz den Entwickler bei der Berücksichtigung von Abhängigkeiten. Bis auf die Formalisierung der Abhängigkeit in Form von Kompositionsregeln, automatisiert die Synthese alle weiteren Schritte.

Nachdem die Synthese erfolgreich war, kann prinzipiell aus dem synthetisierten Modell Code generiert werden. Da es sich hier immer noch um einen Timed Automaton handelt, kann die Codegenerierung der MECHATRONIC UML nur genutzt werden, wenn das Modell zurück in ein REAL-TIME STATECHART oder HYBRID RECONFIGURATION CHART geführt wird. Im Rahmen dieser Arbeit wurde dies allerdings nicht weiter betrachtet.

Muss der synthetisierte Automat noch weiter um spezifische Seiteneffekte oder plattformspezifische Informationen, wie eine WCET (siehe Abschnitt 2.4.2), erweitert werden, so ist dies auf Grund der Größe des parallel komponierten Automaten ungünstig. Im Idealfall werden daher diese anwendungsspezifischen Informationen vor der Synthese den Rollenautomaten hinzugefügt. Die Zurückführung des synthetisierten Verhalten zu separaten Rollenverhalten, wie in Abschnitt 5.4 diskutiert, stellt eine alternative hierzu dar, falls die Informationen zu Beginn der Synthese noch nicht vorhanden sind. Allerdings beinhaltet dieses Modell immer noch den notwendigen Synchronisationsanteil.

Kapitel 6

Werkzeugunterstützung

Diese Arbeit stellt nicht nur die Konzepte zur Entwicklung hierarchischer Komponentensysteme vor, in deren Mittelpunkt die Wiederverwendung von Komponenten und Kommunikationen zwischen Komponenten steht, sondern auch eine Werkzeugunterstützung. Wir werden dabei im Besonderen in Abschnitt 6.1 die Unterstützung für eine Codegenerierung und Laufzeitumgebung erläutern, um tatsächlich auch werkzeugtechnisch einen modellgetriebenen Ansatz zu unterstützen. In Abschnitt 6.2 wird die Umsetzung des Werkzeugs vorgestellt und dabei deren grobe Architektur erläutert. Abschließend werden wir eine Validierung des Ansatzes in Abschnitt 6.3 vorstellen.

6.1 Ausführung

In Abschnitt 2.1 haben wir unseren Ansatz zur Modellierung und Analyse hierarchischer Komponentensysteme vorgestellt. Um selbstoptimierende, mechatronische Systeme zu adressieren, unterstützt der Ansatz kompositionelle Strukturanpassungen. Hiermit wird es ermöglicht, komplexe Systeme durch einen modularen Aufbau und notwendige Erzeugungen zu beschreiben.

Ein offensichtliches Problem ist allerdings, dass wir für eine konkrete Umsetzung die Anforderungen eingebetteter Echtzeit-Systeme berücksichtigen müssen. Insbesondere müssen wir daher die Ressourcen-Beschränkungen von Mikrocontrollern betrachten. Damit dürfen die beschriebenen Erzeugungen nur in den gegebenen Ressourcen-Beschränkungen des Speichers und der CPU-Kapazität ausgeführt werden.

Wir verfeinern daher unsere Modelle plattformspezifisch durch die Berücksichtigung von Ressourcen-Beschränkungen. Unser Ansatz ermöglicht es, ebenso wie bisherige Ansätze, vorhersagbar die Ressourcen einzuhalten. Zudem erlauben wir flexible Ressourcen-Anpassungen zur Laufzeit, die prinzipiell wieder beliebige Erzeugungen ermöglichen.

Für HYBRID RECONFIGURATION CHARTS kann die Codegenerierung unverändert zu der von Burmester [Bur06] genutzt werden, deren Umsetzung wir in [BGH⁺07] gezeigt haben. Speziell müssen wir hier die Codegenerierung und Laufzeitanalyse der Seiteneffekte betrachten, die die Erzeugungen mit Story Diagrammen beschreiben (siehe Abschnitt 6.1.2). Darüber hinaus stellen wir im Folgenden vor, wie wir eine geeignete Laufzeitumgebung auf Basis der in [Bur06] vor-

gestellten anbieten können, um auch als Verifikations- und Simulations-Umgebung zu dienen, wie dies durch die beschriebenen Integrations-Ansätze für Altkomponenten benötigt wird (siehe Abschnitt 6.1.1).

6.1.1 Laufzeitumgebung

Die Laufzeitumgebung erweitert die Implementierungssprache um Konzepte der Modellierungssprache MECHATRONIC UML. Grundlegend ist die Laufzeitumgebung wie eine offene Architektur aufgebaut, die architektonische und mechanistische Entwurfsmuster unterstützt, wie Komponentenmanagement, Komponenteninteraktion und Nachrichtenverteilung [Dou99, Gom00, Dou02].

Zusätzlich unterstützt diese Laufzeitumgebung eine Integration des IPANEMA-Frameworks [Hon98], um eine Codegenerierung für hybride Systeme zu unterstützen [Bur06, BGH⁺07]. Für Details des IPANEMA-Frameworks sei auf [Bur06] verwiesen.

Im Folgenden werden wir auf die notwendigen Konzepte der Laufzeitumgebung eingehen, um eine Simulationsumgebung für die Integrationsverfahren (siehe Abschnitt 4) beschreiben zu können. Die grundlegenden Konzepte der Laufzeitumgebung wurden in [GH06a] auf Basis von [Hen05] vorgestellt.

Die Laufzeitumgebung (siehe Abbildung 6.1) kapselt den Anwendungsentwickler vor betriebs-systemspezifischer Programmierung. Sie dient als eine Abstraktionsschicht, um die Modellsemantik im Kontext einer spezifischen Programmiersprache zu unterstützen. Integriert in die modellbasierte Entwicklung mit der Fujaba Real-Time Tool Suite wird damit eine Schnittstelle für die automatische Codegenerierung angeboten.

Durch die vorgegebene offene Architektur wird dem Entwickler durch Spezialisierung ermöglicht, benutzerspezifische Anforderungen zu realisieren. Konkret werden abstrakte Komponenten und Ports angeboten, die durch Spezialisierung benutzerspezifisch erweitert werden können. Hierdurch wird eine Infrastruktur zur Ausführung der Komponenten bereitgestellt.

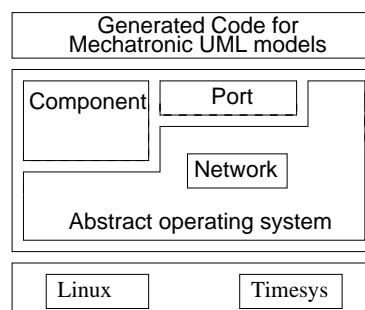


Abbildung 6.1: Schichtenarchitektur der Laufzeitumgebung

Die Kommunikation findet via Nachrichten statt. Durch das Proxy- und Nachrichtenschlangen- sowie Broker-Entwurfsmuster wird eine Entkopplung zwischen den Kommunikationspartnern erreicht. Um den Einsatz im Echtzeitbereich zu gewährleisten, wird das Pool-Allokations- Entwurfsmuster angewandt. Für die Details zu den Mustern verweisen wir auf [Dou02].

Für den Echtzeitbetrieb teilt sich die Initialisierungsphase von Komponenten in `init`, `lookup` und `register` auf. Während der Initialisierungsphase wird der benötigte Speicher allokiert. Während der Registerphase werden angebotene Rollen bekannt gemacht (`LookupService`) und anschließend benötigte Rollen während der Lookupphase durch Anfrage an den `LookupService` aufgesucht. Falls der Ort des Kommunikationspartner a priori bekannt ist, kann die Registrierung und der Lookup via dem `LookupService` umgangen werden, um statisch eine Kommunikation aufzubauen.

Komponentenschicht Die Basis-Klassen der Komponentenschicht sind in Abbildung 6.2 gezeigt. Die Klasse `Komponente` basiert auf der `RealtimeThread`-Klasse und die Klasse `RealtimeThread` erbt von der `Thread`-Klasse.

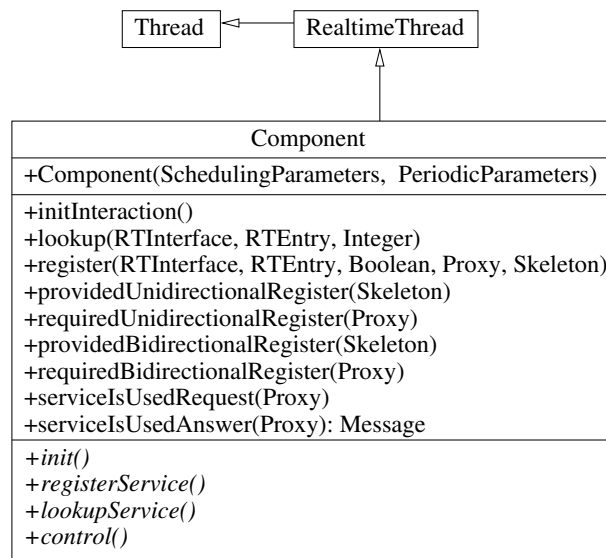


Abbildung 6.2: Basis-Klassen der Komponentenschicht

Wir bieten zudem eine einfache `Schedulable`-Klasse an, die es ermöglicht spezifische Scheduler zu implementieren. Die `Component`-Klasse erbt entsprechend von dieser Klasse.

Eine Komponente wird immer periodisch ausgeführt (siehe Abbildung 6.3). Der `RealtimeThread` führt die `Component` aus, welche periodisch `ConcreteComponent` durch die Methode `control` aufruft. Die periodische Synchronisation mit dem Scheduler wird durch den Aufruf der Methode `waitForNextPeriod` ermöglicht.

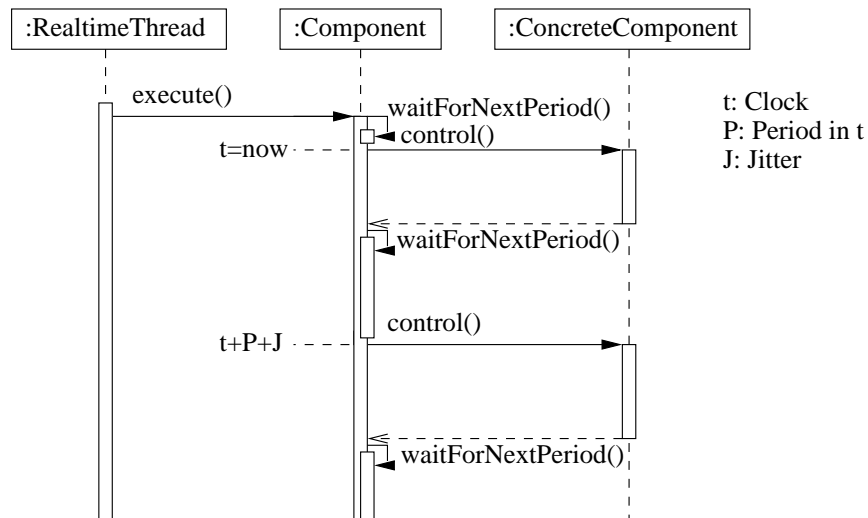


Abbildung 6.3: Ausführungssequenz einer Komponente

Portschnitt Die Interaktion zwischen Komponenten basiert auf einer nicht-blockierenden Kommunikation via Nachrichten. Die Komponenteninteraktion wird durch das Proxy-Muster kombiniert mit einem Nachrichten-Schlangen-Muster [Dou02] umgesetzt. Um die Echtzeiteigenschaften der betrachteten Domäne zu adressieren, wird zusätzlich das Pool-Allokations-Muster angewandt, um ein Speicher-Management zu ermöglichen.

Die Schnittstelle der Portschnitt ist in Abbildung 6.4 gezeigt. Ein Skeleton ist ein Rollenanbieter und ein Proxy ist ein Rollennutzer. Ein Port erbt von der Proxy-Klasse, wenn der Port eine Bedarfsschnittstelle implementiert (required-Schnittstelle, siehe Abschnitt 2.3), andernfalls wird von der Skeleton-Klasse geerbt. Es ist natürlich auch möglich von beiden Klassen zu erben. Die Schnittstelle des Proxy und Skeleton bietet einfach ein senden und empfangen von Nachrichten an.

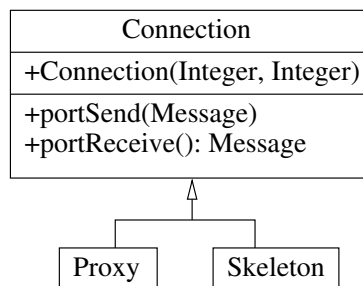


Abbildung 6.4: Basis-Klassen der Portschnitt

Eine Sequenz von interagierenden Komponenten wird in Abbildung 6.5 gezeigt. In der Abbildung wird gezeigt, wie eine Komponente mit einem Port (Proxy oder Skeleton) verbunden ist

und wie die Netzwerkschicht mit der Portsicht verbunden ist. Das Senden einer Nachricht (`send(Message)`) wird ausgelöst durch die `ConcreteComponent`. Die Nachricht wird dann durch den Framework-Port zur `SendQueue` weitergeleitet. Die Netzwerkschicht überprüft periodisch ob sich eine Nachricht in der `SendQueue` befindet. Ist dies der Fall, wird die Nachricht über das Netzwerk der `ReceivingQueue` des Kommunikationspartners zugeschickt. Im Falle einer lokalen Kommunikation wird die Nachricht direkt in die Empfangsschlange gelegt.

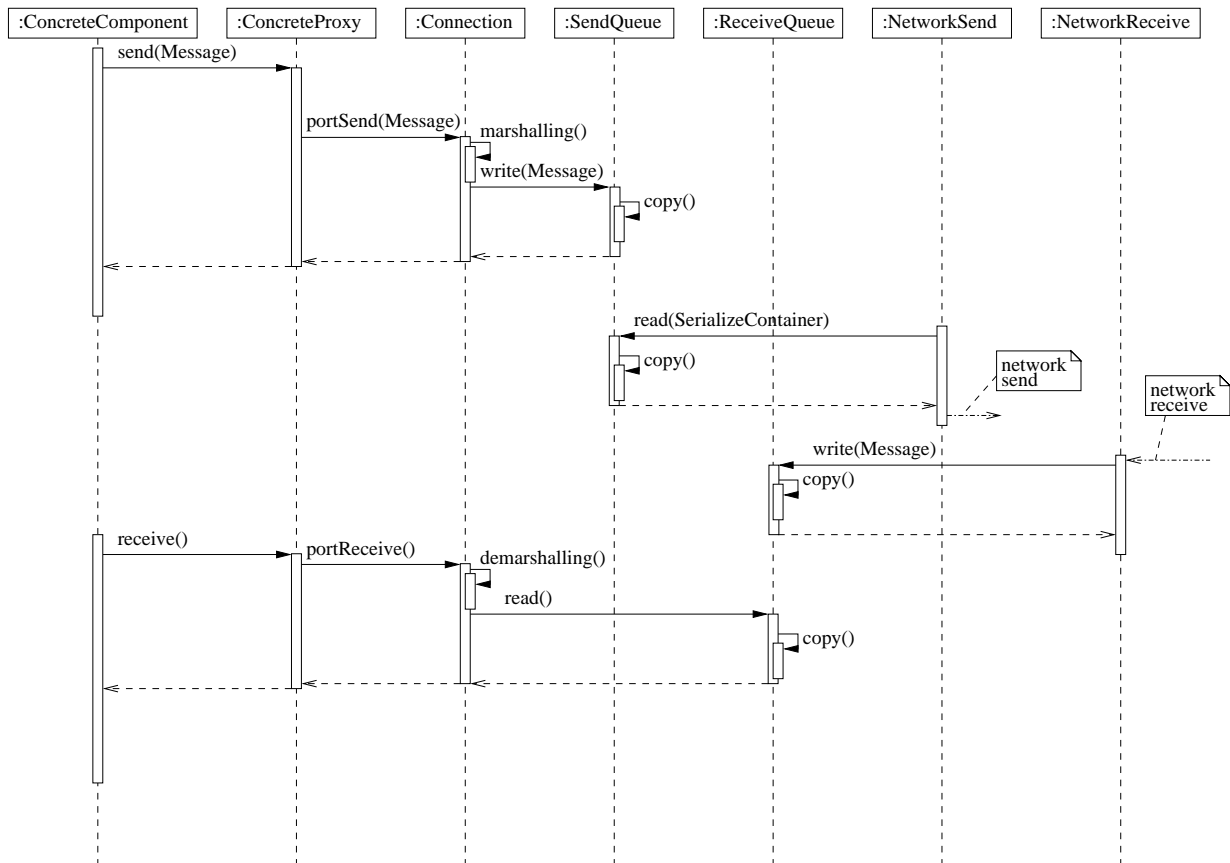


Abbildung 6.5: Eine Nachrichtensequenz

Simulations- und Verifikationsumgebung Die Anforderungen an die Simulations- und Verifikationsumgebung ergeben sich im Wesentlichen aus den in Abschnitt 4 vorgestellten Integrationsverfahren. Die Umgebung muss daher in der Lage sein, eine deterministische Wiederholung zu unterstützen, um die benötigten Mehrfachausführungen der Integrationsverfahren zu ermöglichen. Hierzu gehört auch, dass ein Scheduling sowie Zeit simulativ unterstützt werden muss.

Um eine deterministische Wiederholung für MECHATRONIC UML Modelle zu ermöglichen, müssen wir in der Lage sein 1) alle relevanten Ereignisse für eine deterministische Wiederholung

zu beobachten und 2) die Ausführung der Komponenten und Ports während der Wiederholung zu kontrollieren (siehe Abbildung 6.6).

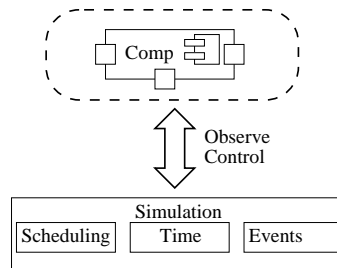


Abbildung 6.6: Beobachtung und Kontrolle der Ausführung einer MECHATRONIC UML Komponentenarchitektur

Um dies zu ermöglichen, führen wir eine Diagnoseschicht ein, welche für die Anwendung transparent ist (siehe Abbildung 6.7). Über diese Diagnoseschicht können alle relevanten Ereignisse beobachtet werden, ohne den Quellcode der Anwendung anzupassen. Die Schicht ermöglicht eine Kontrolle und Beobachtung aller Interaktionen der Komponente mit der Umgebung. Eine Interaktion besteht aus allen externen Nachrichten, Scheduling-Ereignisse (Aktivierung, Verdrängung, ...) und allen Zeitereignissen der Komponente.

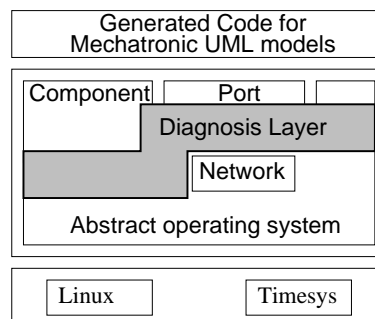


Abbildung 6.7: Laufzeitumgebung mit Simulationsschicht

Im Folgenden betrachten wir zuerst, welche Ereignisse für eine deterministische Wiederholung relevant sind. Anschließend betrachten wir den Wiederholungsansatz unter Berücksichtigung von Einflüssen durch eine zusätzliche Instrumentierung des Systems.

Relevante Ereignisse Eine elementare Erkenntnis für die deterministische Wiederholung von MECHATRONIC UML Komponenten ist, dass es ausreicht, die Komponenten mit exakt den gleichen eingehenden Nachrichten periodenspezifisch zu wiederholen, um das gleiche funktionale Verhalten wiederzugeben.

Im allgemeinen muss auch das Scheduling identisch wiederholt werden. Damit die implementierten Komponenten allerdings eine korrekte Verfeinerung der Modelle sind, müssen diese für ein eingehendes Echtzeitverhalten auch das gleiche ausgehende Verhalten wiedergeben (siehe Abschnitt 3.1).

Grundlage für die korrekte Abbildung der Modelle ist, dass aus den möglicherweise nichtdeterministischen plattformunabhängigen Modellen deterministische Modelle generiert werden können, wie in [AMPS98] für Timed Automata gezeigt wurde. Ein einfaches Beispiel ist, dass „non-urgent“ Transitionen, die irgendwann schalten können, durch die Codegenerierung als „urgent“ Transitionen umgesetzt werden, die sofort schalten, wenn alle notwendigen Bedingungen erfüllt sind. In [Bur06] wurde eine entsprechende Abbildung aller relevanter Verhaltenselemente gezeigt.

Aufgrund dieser Eigenschaften müssen für eine deterministische Wiederholung der Komponenten nur die eingehenden Nachrichten sowie der Zeitpunkt der Nachrichten korrekt in der gleichen Periode wiedergeben werden¹ (siehe Abbildung 6.8).

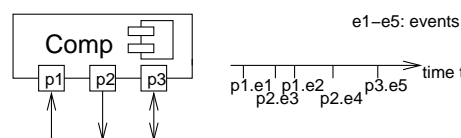


Abbildung 6.8: Externe Ereignisse einer Komponente

Da das Verhalten einer Komponente nicht nur von den externen Ereignissen, sondern auch von internen Zeitabfragen abhängig ist, müssen wir zudem generell alle Zeitanfragen während der Wiederholung identisch wiedergeben. Durch die Diagnoseschicht ist dies einfach möglich.

Durch Beobachtung der eingehenden Nachrichten, Zeitereignisse sowie der Periode können wir damit eine Komponente unabhängig von dem unterliegenden Scheduling deterministisch wiederholen. Dies sind auch die relevanten Informationen, die wir für die Integration von Altkomponenten benötigen (siehe Abschnitt 4).

Wiederholung Abbildung 6.9 zeigt neben der Beobachtung einer Komponente, auch die kontrollierte Ausführung der Komponente während der deterministischen Wiederholung. Die Simulation, bzw. deterministische Wiederholung, ist insofern einfach, da wir lediglich die aufgezeichneten relevanten Ereignisse wiederholen müssen, unabhängig von der Plattform. Die korrekte Ordnung der Ereignisse wird durch die Diagnoseschicht erreicht. In der simulierten Interaktion wird dies durch das Connection-Objekt implementiert (siehe Abbildung 6.9)

¹In [GH06a] haben wir zudem beschrieben, wie wir mit parallel ausgeführten Verhalten umgehen können, wenn die ausgehenden Ereignisse nicht eindeutig voneinander unterschieden werden können sowie nichtdeterministisches Verhalten. Für die hier betrachteten Systeme sind diese Fälle allerdings nicht notwendig.

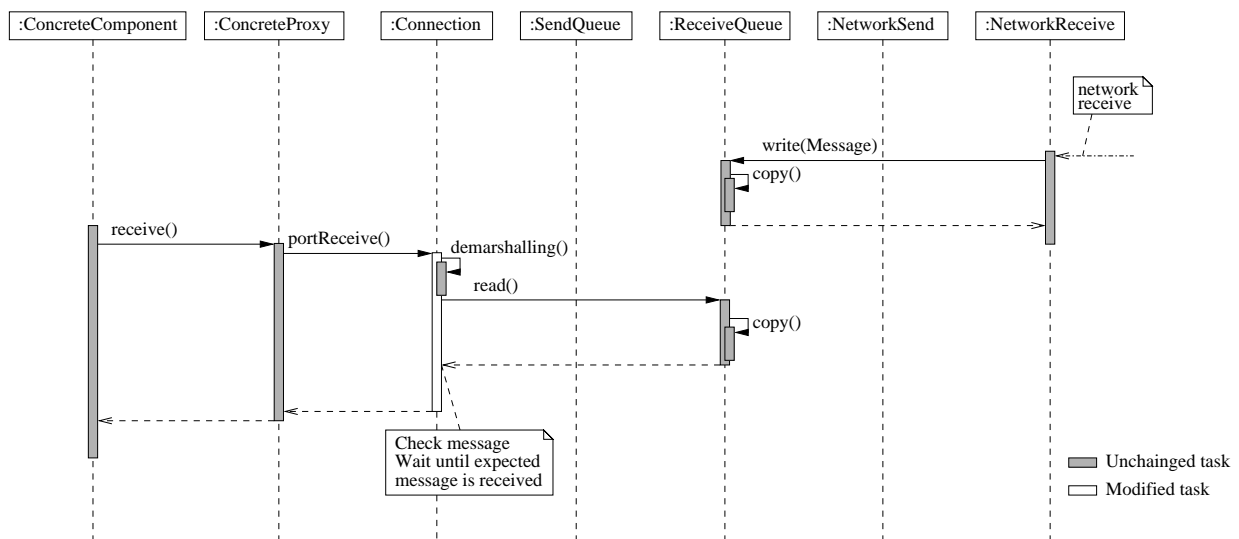


Abbildung 6.9: Nachrichtensequenz einer deterministischen Wiederholung

Minimierung der Ereignisaufnahme Da die Aufnahme gerade der Nachrichten einen hohen Zeitaufwand und Speicherverbrauch bedeuten kann, ist es wichtig die Aufnahme zu minimieren.

Die oben beschriebenen relevanten Ereignisse können allerdings nicht minimiert werden, da diese das Verhalten der Komponente beeinflussen.

Eine Möglichkeit den Aufwand zu reduzieren, ist das Eliminieren des Nachrichteninhalts. Während der Wiederholung kann dieser dann wieder reproduziert werden, indem die vollständige Interaktion mit Nachrichteninhalt aufgezeichnet wird.

Aufnahme Eine einfache Möglichkeit die Daten aufzunehmen ist die Verwendung eines extra Threads, mit niedrigerer Priorität als die der implementierten Komponente (z.B. [TFCB90]). Hierüber ist es dann auch möglich, die notwendigen Daten auf eine zusätzliche externe Festplatte zu speichern, ohne das System zu beeinflussen.

Dies führt allerdings zu dem Problem, dass eventuell Daten verloren gehen, da das Schreiben der Daten im Vergleich zur realen Ausführung lange dauern kann [Zam99]. Dieses Problem kann grundsätzlich nicht umgangen werden, jedoch können die auftretenden Aufnahmefehler ebenfalls notiert werden. Eine deterministische Wiederholung ist dann nicht mehr vollständig möglich.

Vermeidung des Probe Effects Durch eine software-basierte Beobachtung wird zusätzlicher Quellcode für die notwendige Instrumentierung des Systems benötigt. Typischerweise ändert sich die Instrumentierung während der Entwicklung und dem finalen System. Durch diese

unterschiedliche Instrumentierung kann das System unterschiedlich beeinflusst werden, wodurch auch unterschiedliches Verhalten auftreten kann.

In unserem Fall ist dies insofern problematisch, da die Aufnahmen für die deterministische Wiederholung typischerweise nicht während des normalen Betriebs ständig durchgeführt werden. Kann eine gleichbleibende Aufnahme nicht gewährleistet werden, so kann folglich auch ein Probe Effect auftreten [Fid96].

Dieses Problem können wir nur umgehen, indem wir eine minimale notwendige Instrumentierung auch während des normalen Betriebs durchführen. Dies muss nicht zwangsläufig dazu führen, dass diese Daten auch auf einen externen Speicher geschrieben werden. Die Speicherung der Daten, durch einen zusätzlichen Thread, muss nicht zu einer Beeinflussung führen, da diese unabhängig von der eigentlichen Instrumentierung durchgeführt werden kann. Damit kann für den Fall, dass die Komponente deterministisch wiederholt werden soll, ein zusätzlicher Thread aktiviert werden, der die Daten auf eine externe Festplatte schreibt. Unter der Annahme, dass ein Echtzeitscheduling durch Hinzunahme des weiteren Threads mit niedriger Priorität, die höher priorisierten Threads weiterhin vorhersagbar ausführt, tritt kein zusätzlicher Probe Effect auf.

Handelt es sich um eine Altkomponente kann dies nur erreicht werden, indem die Ausführung im beschriebenen Anwendungsszenario stets mit dem hier beschriebenen Framework umgesetzt wird.

Eine Evaluierung der deterministischen Wiederholung in einer Eclipse Debugging Umgebung haben wir in [GH06a] vorgestellt.

6.1.2 Codegenerierung und Laufzeitanalyse

Aufgrund der engen Verzahnung zwischen der Codegenerierung und der Laufzeitanalyse betrachten wir die Verfahren in diesem Abschnitt zusammen. Ohne vorherige Codegenerierung kann keine Laufzeitanalyse durchgeführt werden. Wegen den in diesem Kapitel einleitend diskutierten Anforderungen, dass eine Laufzeitanalyse auch zur Laufzeit durchgeführt werden muss, da die Strukturanpassungen a priori nicht alle vorausgedacht werden können, muss die Codegenerierung ebenfalls den Anforderungen der Laufzeitanalyse genügen.

Die Codegenerierung basiert grundlegend auf der von Burmester [Bur06], deren Umsetzung wir in [BGH⁺07] gezeigt haben. Grundlegende Arbeiten zu den Erweiterungen der Codegenerierung und zur Laufzeitanalyse wurden in [GHH06b, Ric08, GHH08b, HBB⁺09, GHH11, HOGS10, HOGS12] vorgestellt.

Abbildung 6.10 veranschaulicht die unterliegende OCM-Mikroarchitektur für die hier betrachteten mechatronischen Systeme. Mit den hier vorgestellten Techniken können Software-Komponenten für den reflektorischen Operator entwickelt werden. Aus der Abbildung wird zudem die grundlegende Idee der Umsetzung einer Codegenerierung und Laufzeitanalyse ersichtlich.

Um die plattformspezifischen Informationen geeignet zu berücksichtigen, wird eine Schnittstelle zwischen dem reflektorischen Operator und der Laufzeitumgebung zur Verfügung gestellt. Durch Profil-Informationen, mit denen Ressourcenattribute wie eine WCET oder der Speicherbedarf beschrieben werden können, wird der Austausch zwischen dem Betriebssystem, welches die Ressourcen verwaltet, und der Anwendung ermöglicht.

Durch eine Profilbeschreibung kann eine Anwendung zur Laufzeit dem Betriebssystem neue Anforderungen mitteilen. Um dies zu ermöglichen wurde in [BGG04a] ein Ansatz beschrieben, der Profilbeschreibungen in HYBRID RECONFIGURATION CHARTS berücksichtigt. In [HBB⁺09, BBB⁺09] haben wir diesen Ansatz erweitert, um parametrisierte Profile zu beschreiben, die zur Laufzeit erweitert werden können. Dies ist die Basis, um Strukturanpassungen zur Laufzeit flexibel den Umweltbedingungen optimal anzupassen.

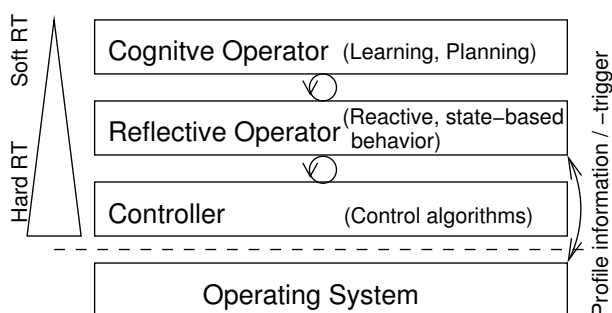


Abbildung 6.10: Integration plattformspezifischer Informationen

Im Folgenden werden wir relevante Ausschnitte des Konvoibeispiels einführen. Hieran werden wir die notwendigen Erweiterungen an der Codegenerierung sowie Laufzeitanalyse verdeutlichen. In Abschnitt 6.1.2.1 werden wir dann die flexible Profilverwaltung einführen und formalisieren. Hierauf aufbauend werden wir in Abschnitt 6.1.2.2 unsere WCET-Analyse für dynamische Strukturen einführen. In Abschnitt 6.1.2.3 werden die Erweiterungen an der Codegenerierung vorgestellt und in Abschnitt 6.1.2.4 werden die für die Codegenerierung notwendigen Evaluierungsreihenfolgen hybrider Systeme mit Strukturanpassungen diskutiert. Abschließend diskutieren wir den Ansatz in Abschnitt 6.1.2.5.

Beispiel Wir betrachten hier wieder das Beispiel der Konvoirestrukturierung aus Abbildung 2.17. Im Folgenden gehen wir davon aus, dass für eine Menge von RailCabs innerhalb eines Streckenabschnitts überprüft werden soll, ob diese bereits an einem Konvoi teilnehmen. Ist dies nicht der Fall, soll ein `ConvoyCoordinationPattern` zwischen den RailCabs angelegt werden.

Abbildung 6.11 zeigt ein einfaches Adaptionsverhalten, welches die Koordination zwischen den RailCabs initiiert, immer dann wenn ein neues RailCab den Streckenabschnitt betritt. Wird eine `newParticipant`-Nachricht empfangen, dann wird als Seiteneffekt `initiateCoordination` ausgeführt. Eine solche Nachricht kann z.B. durch eine Streckenabschnittskontrolle verschickt werden (siehe Abschnitt 5). Für das betrachtete Szenario ist das allerdings irrelevant.

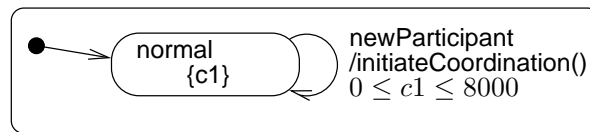


Abbildung 6.11: Einfaches Adaptionverhalten zur Erzeugung einer Musterbeziehung

Abbildung 6.12 zeigt das Story Diagramm, welches durch den Seiteneffekt aufgerufen wird. Das Story Diagramm besteht aus drei Story Pattern. Der coordinator (siehe Abbildung 1.2) erzeugt zwischen allen RailCabs in dem Abschnitt ein ConvoyCoordinationPattern. Der Vollständigkeit halber müssten anschließend auch entsprechende Portinstanzen erzeugt (wie in Abschnitt 3 gezeigt) und gelöscht werden. Hierauf wird im Folgenden verzichtet.

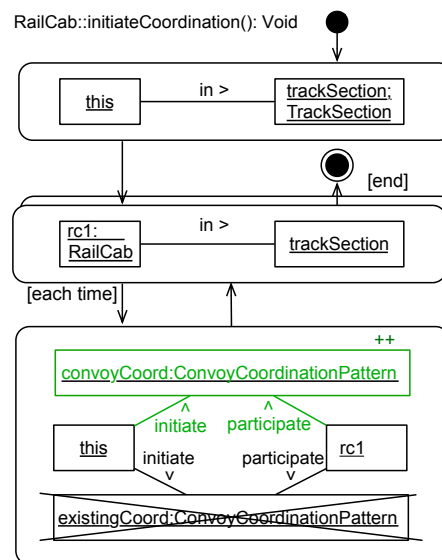


Abbildung 6.12: Story Diagram zur Beschreibung des initiateCoordination Seiteneffekts

Das unterliegende Klassendiagramm ist in Abbildung 6.13 gezeigt. Das Klassendiagramm besteht aus den Klassen TrackSection, RailCab, ConvoyCoordinationPattern und Track. Die Klasse Track wird in dem Story Diagramm nicht direkt verwendet. Da eine TrackSection allerdings aus einer Menge von Tracks besteht, werden entsprechend diese Objekte ebenfalls berücksichtigt.

Das Klassendiagramm verdeutlicht die beschriebene Problematik für eingebettete Echtzeitsysteme. Prinzipiell können auf Modellierungsebene einer TrackSection beliebig viele Tracks und RailCabs assoziiert werden. Ein RailCab kann wiederum mit beliebig vielen RailCabs einen Konvoi eingehen. Auf der Modellierungsebene ist diese Beschreibung legitim und sinnvoll, da plattform-spezifische Einschränkungen bzgl. der Obergrenze der Kardinalität der Assoziation dazu führen, dass das Modell und die Analysen auch nur für diese Plattformen gültig sind. Eine Festlegung der Kardinalität führt zudem dazu, dass es zur Laufzeit auch nicht möglich ist, eine bzgl. der Umgebung optimale Auslastung zu finden, da die Ressourcen a priori eingeschränkt wurden.

Wir werden daher im Folgenden einen Ansatz vorstellen, der zum einen die Vorhersagbarkeit des umgesetzten Modells für eine bestimmte Plattform garantiert und zum anderen aber flexibel die Ressourcen verteilen kann, so dass zur Laufzeit unterschiedliche Ressourcenbelegungen ermöglicht werden, ohne vorherige Einschränkung auf eine fixe obere Kardinalität. Voraussetzung für den Ansatz ist, dass die Anwendung mit der MECHATRONIC UML umgesetzt wird.

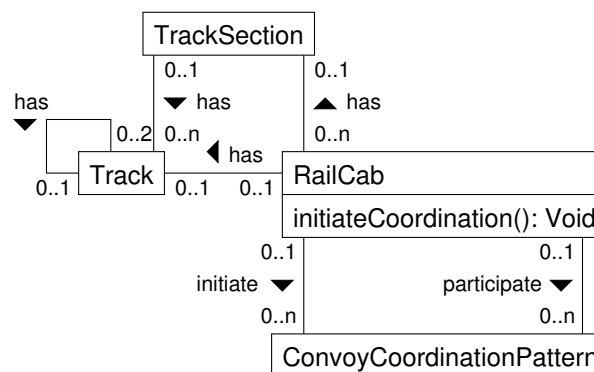


Abbildung 6.13: Unterliegendes Klassendiagramm des initiateCoordination Seiteneffekts

6.1.2.1 Flexible Ressourcenverwaltung

Der Standard-Ansatz, um ein vorhersagbares Verhalten für Multi-Prozessoren eines Echtzeitsystems zu garantieren, ist die Allokation der maximal benötigten Ressourcen im Voraus [But05]. Dieser Ansatz ermöglicht die zeitliche Ausführung der Prozesse, führt allerdings zu einer schlechten Ressourcennutzung und ist kaum anwendbar für dynamische Strukturen.

Die flexible Ressourcenverwaltung (FRM) [LO08] wurde entwickelt, um die Ressourcennutzung zu verbessern. Jede Anwendung (z.B. Konvoi oder Feder/Neige-Modul) wird zusätzlich mit einer Menge von Profilen und Transitionen zwischen diesen beschrieben. Jedes Profil beinhaltet Informationen über die maximale und minimale Ressourcenanforderung, Umschaltbedingungen und der Profilqualität. Die Profilqualität beschreibt, welche Anwendung bevorzugt behandelt werden soll. Dies kann z.B. übergeordnet durch einen Lernalgorithmus im kognitiven Operator zur Laufzeit berechnet werden und/oder a priori von dem Entwickler vorgegeben werden. Für eine fixe Anzahl an Profilen wurde in [BGGO04a] ein semi-automatischer Algorithmus beschrieben.

Der größte Nutzen entsteht durch das FRM, wenn mehrere Profile je Anwendung spezifiziert werden. Für mechatronische Systeme bietet es sich häufig an, die einzelnen Regler-Konfigurationen als ein Profil zu beschreiben oder Verhalten, deren Struktur angepasst wird in mehrere Profile zu unterteilen. Für unser Konvoi-Beispiel kann z.B. ein Profil festgelegt werden mit einer kleinen Anzahl an Konvoiteilnehmern und einer größeren Anzahl. Das FRM kann zur Laufzeit die Ressourcennutzung unter Berücksichtigung der Qualitäten der Anwendungen optimieren.

Dynamische WCET Eine Veränderung der WCET führt zu einer Veränderung der Prozessorauslastung $U = \frac{WCET}{T}$ des betroffenen Profils (T gibt die Periode der Hauptfunktion des Profils an). Im Fall der CPU verwaltet das FRM die Prozessorauslastung. Daher wird eine Anpassung der Profilgrenzen der WCET als eine Ressourcenanfrage behandelt (wie z.B. eine Speicheranfrage).

Der FRM-Ansatz erlaubt eine Ressourcenallokation nur in den spezifizierten Grenzen des aktiven Profils der Anwendung. Eine WCET-Anfrage über die maximale Ressourcenverwendung hinaus, ist entsprechend nicht erlaubt. Durch die in Abschnitt 6.1.2.2 vorgestellte WCET-Analyse und Codegenerierung wird dies verhindert.

Das FRM berechnet für die aktuellen Profile einen einfachen Plan, der angibt, wie die Profile in eine gültige Konfiguration rekonfiguriert werden können, so dass alle Ressourcenanfragen unter Berücksichtigung der Qualität aufgelöst werden können.

Der Schedulability-Test des FRM stellt sicher, dass so ein Plan ohne Verletzung einer Deadline ausgeführt werden kann [LO08].

Eine Veränderung der WCET kann allerdings zu einer Verletzung der vorliegenden Pläne führen. Für jede Veränderung der WCET muss der vorliegende Plan überprüft werden. Falls aufgrund der WCET-Veränderung kein gültiger Plan erstellt werden kann, wird die Veränderung abgelehnt. Die Anwendung darf also nicht mit mehr Ressourcen als den aktuellen im Profil ausgeführt werden.

Die Berechnung, ob ein Schedule für einen neuen Plan gefunden werden kann, wird im Hintergrund ausgeführt, ohne zeitliche Begrenzungen. Dies führt zu keinen Einschränkungen der Sicherheit, da eine Profilanpassung lediglich zu einer Optimierung führt.

Eine Konsequenz aus der strikten Ressourcenvergabe innerhalb der Profilgrenzen ist, dass die Änderung einer WCET dem FRM mitgeteilt werden muss, bevor die Anwendung tatsächlich diese Ressourcen verwendet. In den folgenden Abschnitten werden wir beschreiben, wie eine Anwendung zur Laufzeit dynamisch Profile anpassen kann. Hierfür werden wir parametrisierte Profile einführen.

Formalisierungen In diesem Abschnitt stellen wir eine Formalisierung des Profilkonzepts vor. Hierunter fällt auch die dynamische Erzeugung und Zerstörung von Profilen, die benötigt wird, um eine Anpassung des Ressourcenbedarfs zur Laufzeit zu ermöglichen. Eine Anpassung des Ressourcenbedarfs ist Voraussetzung, um strukturelle Anpassungen realistisch umzusetzen. Die hier gezeigte Formalisierung ist notwendig, um präzise das Zusammenspiel zwischen dem FRM und einer Anwendung zu beschreiben.

Tasks und Konfigurationen Bevor wir das Profilkonzept vorstellen, definieren wir einige relevante Begriffe. Ein System besteht aus einer Menge von periodischen Tasks Γ , wie in Definition 67 gezeigt. Diese Tasks werden angemessen durch den Scheduler des FRM ausgeführt, so dass keine Deadline-Verletzungen auftreten.

Definition 67 (Periodischer Task)

Ein periodischer Task $\tau_i \in \Gamma$ ist ein 3-Tupel $\tau_i = (\mathcal{B}_i, \mathcal{S}_i, \mathcal{P}_i)$ mit: einer Menge von möglichen Verhalten (Seiteneffekten) $\mathcal{B}_i = \{b_i^1, \dots, b_i^n\}$, einer Menge von Zuständen $\mathcal{S}_i = \{s_i^1, \dots, s_i^n\}$ und eine Menge von möglichen Profilen $\mathcal{P}_i = \{p_i^1, \dots, p_i^n\}$, in denen der Task ausgeführt werden kann. $\mathcal{S}(\tau_i) : \Gamma \rightarrow \mathcal{S}$ ist der aktuelle Zustand des Tasks τ_i zur Laufzeit. $\mathcal{B}(\tau_i, s_i^j) : (\Gamma \times \mathcal{S}) \rightarrow \mathcal{B}$ ist eine Menge von möglichen Verhalten durch einen gegebenen Task τ_i und einen Zustand $s_i^j \in \mathcal{S}_i$. $\mathcal{C}(\tau_i) : \Gamma \rightarrow \mathcal{C}$ ist die aktuelle Konfiguration eines Tasks τ_i . $\mathcal{P}(\tau_i) : \Gamma \rightarrow \mathcal{P}$ ist das aktuelle Profil eines Tasks.

Jeder Task τ_i hat eine Identität zur Laufzeit, die sich mit der Ausführung eines Seiteneffekts $b_i^j \in \mathcal{B}_i$ ändert. Die Identität eines Tasks kann daher durch die Funktion $\mathcal{S}(\tau_i)$, $\mathcal{B}(\tau_i, s_i^j)$, $\mathcal{C}(\tau_i)$ und $\mathcal{P}(\tau_i)$ beschrieben werden. Die aktuelle Konfiguration $\mathcal{C}(\tau_i)$ bestimmt die aktuelle Situation der Ressourcen eines Tasks.

Eine Konfiguration beschreibt die aktuelle Instanzsituation eines Tasks (siehe Definition 68). Grundsätzlich kann ein anderer Task direkt die Instanzsituation beeinflussen. Dies führt allerdings zu nicht ganz unerheblichen Nebeneffekten.

Die gegenseitige Beeinflussung kann zu einer inkonsistenten Instanzsituation führen, in dem die Konfiguration über eine mögliche maximale Ressourcenbelegung eines Tasks steigt. Um dieses Problem aufzulösen muss jeder Task zusätzlich die möglichen Konfigurationen der abhängigen Tasks betrachten.

Die bisher untersuchten Anwendungen haben allerdings keinen Bedarf an dieser engen Verzahnung zwischen Tasks gezeigt. Aus der Sicht eines sicherheitskritischen Systems ist dies zudem fraglich, da hierdurch ein modularer Entwurf und damit auch eine (effiziente) Analyse verhindert wird. Im schlimmsten Fall müssten daher alle Seiteneffekte zusammen hinsichtlich der geforderten Eigenschaften analysiert werden. Daher treffen wir im Folgenden die Annahme, dass jeder Task eine eigene Instanzrepräsentation der Umgebung besitzt, die nicht durch andere Tasks direkt verändert werden kann.

Definition 68 (Konfiguration)

Eine Konfiguration $C_i \in \mathcal{C}$ ist definiert durch $C_i = (c_i^1, \dots, c_i^n)$ mit $c_i^j : T(c_i^j) \rightarrow \mathbb{N}_0$ ist die Anzahl der aktuellen Instanzen des Typs $T(c_i^j)$. \mathcal{C} beschreibt alle möglichen Konfigurationen des Systems.

Profile Zur Laufzeit wird ein Task immer in einem Profil ausgeführt. Die Hauptcharakteristiken eines Profils sind die maximale Konfiguration, welche die maximalen Ressourcengrenzen definiert und die Qualität, welche das FRM nutzt, um das System angemessen zu schedulen. Ein Profil ist damit wie folgt definiert.

Definition 69 (Profil)

Ein Profil $p_i \in \mathcal{P}$ ist ein 4-Tupel $p_i = (\omega_i, m_i, \hat{C}_i, q_i)$ mit $\omega_i \in \mathbb{N}_0$ ist die WCET, die ein Task in dem Profil nicht überschreiten soll, $m_i \in \mathbb{N}_0$ ist der maximal benötigte Speicherbedarf, die ein Task in dem Profil nutzen darf, $\hat{C}_i \in \mathcal{C}$ ist die maximale Konfiguration des Profils und q_i ist die Qualität des Profils.

Für einen gegebenen Task $\tau_i = (\mathcal{B}_i, \mathcal{S}_i, \mathcal{P}_i)$ definieren wir weiterhin für die Profile $p_k \in \mathcal{P}_i$ und $p_l \in \mathcal{P}_i$, dass das Profil p_k sicher durch das Profil p_l entfernt werden kann, wenn $\hat{C}_k \leq \hat{C}_l$ gilt.

Definition 70 (Konfigurationsordnung)

Eine Konfiguration $C_i \in \mathcal{C}$ dominiert eine Konfiguration $C_j \in \mathcal{C}$, ausgedrückt durch $C_j \leq C_i$, wenn $\forall c_i^k : c_i^k \leq c_j^k$.

Das FRM verwaltet die verschiedenen Tasks durch die Betrachtung der verschiedenen Profile. Die Beziehung zwischen den Profilen wird durch einen Profilgraph ausgedrückt.

Definition 71 (Profilgraph)

Ein Profilgraph ist ein 3-Tupel $G_p = (V, E, l)$ mit: einer Menge von Knoten $V = \{v_1, \dots, v_n\}$, einer Kante $(v_i, v_j) \in E$ und einer Markierungsfunktion $l(v_i) : V \rightarrow (\mathcal{P}_1 \times \dots \times \mathcal{P}_m)$, die einen Knoten $v_i \in V$ mit einem m -Tupel von Profilen verbindet. \mathcal{P}_i ist die Menge der Profile des Tasks τ_i .

Um das System sicher zu initialisieren, wird ein Worst-Case-Profil p_i^{max} für jeden Task τ_i angelegt. Diese Profile werden entsprechend Offline definiert und garantieren, dass das System Initial in einem sicheren Zustand startet. Da wir a priori die oberen Grenzen für eine Konfiguration kennen, können wir das Worst-Case-Profil bestimmen.

Die Strategie für die Erzeugung von neuen Profilen kann sehr mannigfaltig sein. Da diese Strategie allerdings für die WCET-Analyse, der hier betrachteten Anwendungen, die sich im reflektorischen Operator wieder finden, nicht relevant ist, verweisen wir auf [OZKV08] für geeignete Strategien. Die Strategien zur Profilverbesserung werden typischerweise im kognitiven Operator umgesetzt (siehe Abbildung 6.10).

Wenn ein neues Profil erzeugt wird, müssen wir die maximale Konfiguration des Profils p_i bestimmen, um die Ressourcengrenzen festzulegen. Um eine maximale Konfiguration für ein neues Profil zu bestimmen, nehmen wir an, dass wir eine Menge von möglichen Verhalten innerhalb einer einzelnen Periode betrachten müssen. Die maximale Konfiguration für ein neues Profil lässt sich damit wie folgt definieren.

Definition 72 (Maximal resultierende Konfiguration)

Für einen gegebenen Task $\tau_i = (\mathcal{B}, \mathcal{S}, \mathcal{P})$ und $\mathcal{B}' = \mathcal{B}(\tau_i, \mathcal{S}(\tau_i))$ ist die Menge des möglichen Verhaltens in einen Zustand $\mathcal{S}(\tau_i)$, die maximal resultierende Konfiguration eines Verhaltens $b_j \in \mathcal{B}'$, definiert durch $\vec{C}_{b_j}^{max} : \mathcal{C} \rightarrow \mathcal{C}$ mit:

$$\forall k = \{1, \dots, n\} : C = (I_k \cdot c_1^{\Delta, k} + c_1, \dots, I_k \cdot c_m^{\Delta, k} + c_m).$$

$C^{\Delta, k} = (c_1^{\Delta, k}, \dots, c_m^{\Delta, k})$ ist der Konfigurationsunterschied, der bestimmt, wie sich eine individuelle Konfiguration durch Anwendung von Seiteneffekten, die durch Story Diagramme umgesetzt werden, verändert. $I_k \in \mathbb{N}$ ist die WCNI (Worst Case Number of Iterations) des Story Diagramms, welche abhängig ist von der aktuellen Konfiguration C . I_k muss für jedes Story Diagramm berechnet werden (siehe Abschnitt 6.1.2.2). Die maximal resultierende Konfiguration $\vec{C}_{\mathcal{B}'}^{max} = \{c_1^{max}, \dots, c_m^{max}\}$ mit $\forall c_l^{max} : c_l^{max} = \max\{c_{b_1}^l, \dots, c_{b_n}^l\}$.

Da ein Task immer in einem Profil läuft, können wir die maximal resultierende Konfiguration eines neuen Profils mittels der Story Diagramm Laufzeitanalyse bestimmen (siehe Abschnitt 6.1.2.2).

Das Hinzufügen eines neuen Profils führt zu einer Aktualisierung des Profilgraphen. Wie in Abschnitt 6.1.2.1 beschrieben, gilt für zwei beliebige Profile, dass eine Transition von p_j nach p_k angelegt ist, wenn entweder gilt, dass $\hat{C}_j \leq \hat{C}_k$ (für jedes Element) oder es existiert ein i , so dass $mrcf_{b_i}(\hat{C}_j) \leq \hat{C}_k$, mit $mrcf_{b_i}$ berechnet die maximal resultierende Konfiguration für b_i .

Dieser Test muss für alle Transitionen durchgeführt werden, an denen das neue Profil beteiligt ist. Wenn zudem Zustände S_j für ein Profil p_j und S_k für ein Profil p_k relevant sind, muss zudem für jeden Zustand $s \in S_j$ die obige Bedingung gelten. Entweder muss im ersten Fall $s \in S_k$ gelten oder es muss im zweiten Fall ein angemessenes b_i existieren, welches es ermöglicht von s nach s' mit $s' \in S_k$ zu gehen.

Um ein Profil zu entfernen, können wir die gleichen Techniken anwenden, wie für das Hinzufügen. Eine Voraussetzung für das Entfernen ist, dass sich die Anwendung nicht in dem Profil befindet. Der resultierende Profilgraph muss wieder, wie oben beschrieben, stark zusammenhängend sein.

Der Aufwand, um einen Profilgraphen zu aktualisieren ist $\mathcal{O}(1)$, wenn die Veränderung nicht die untere oder obere WCET-Grenze beeinträchtigt. Andernfalls muss ein erneutes Scheduling durchgeführt werden, wie in Abschnitt Dynamische WCET auf Seite 186 beschrieben.

6.1.2.2 WCET-Analyse für dynamische Strukturen

Klassische WCET-Analyse-Werkzeuge unterstützen nur sehr eingeschränkt dynamische Objektstrukturen mit Schleifen. Für unsere Modelle wird durch solche Ansätze keine Analyse unterstützt. Um dieses Problem zu umgehen und trotzdem die bisherigen Werkzeuge für plattformspezifische Analysen auszunutzen, stellen wir einen Ansatz vor, der auf der Modellebene bereits Analysen durchführt. Konkret berechnen wir auf Basis der wohldefinierten Modelle maximale Schleifendurchläufe. Diese Berechnungen geben wir dann WCET-Analyse-Werkzeugen mit, so dass für die betrachteten Systeme eine WCET-Berechnung ermöglicht wird.

Durch geeignete Parametrisierung der Modelle ermöglichen wir zudem eine Anpassung der Instanzsituation zur Laufzeit unter Berücksichtigung von Ressourcenschranken. Dabei unterstützen wir für deterministische Plattformen eine Laufzeit-WCET-Analyse, ohne extra ein WCET-Analyse-Werkzeug anzustoßen, wodurch eine effiziente WCET-Berechnung ermöglicht wird. Verhält sich die Plattform unregelmäßig (nicht vorhersehbar), so muss eine vollständige WCET-Analyse durchgeführt werden.

WCET-Analyse Um eine (dynamische) WCET-Analyse von Story Diagrammen (siehe Abschnitt 2.4.5.4) zu unterstützen, beschreiben wir im Folgenden deren möglichen WCET, bzw. WCNI, Ausführungspfade. Wir definieren ein Story Diagramm durch ein n -Tupel $d = (d^1, \dots, d^n)$

mit $d^i = (((l_1^1, \dots, l_n^1), p_1), \dots, ((l_1^m, \dots, l_n^m), p_m))$, definiert einen Pfad von einer Startaktivität zu einer Stopaktivität.

Zwischen diesen Aktivitäten sind Story Pattern p_i , welche innerhalb eines Pfades ausgeführt werden. Jeder mögliche Pfad eines Story Diagramms d^i wird Offline, während der Systementwicklung berechnet.

Ein Pfad d^i eines Story Diagramms beinhaltet Schleifeninformationen für jedes individuelle Story Pattern, ausgedrückt durch ein n -Tupel von Story Pattern-Indizes (pl_1^i, \dots, pl_n^i) . Jeder Index des Tupels ist eine Referenz auf ein Story Pattern. Die Ordnung der Indizes gibt die Ordnung der Story Pattern an, wie sie in einem Pfad ausgeführt werden. Wenn also ein Story Pattern das erste Pattern in einer Schleife ist, hat es den Index, welcher der Initiator der Schleife ist.

Die WCET eines Story Diagramms d kann dynamisch durch $\omega(d) = \max(\omega(d^1), \dots, \omega(d^n))$ berechnet werden. Wir nehmen also die WCET eines Story Diagramm-Pfades, welche die Maximale ist. Die WCET eines Story Diagramm-Pfades d^i wird durch die WCNI (Worst Case Number of Iterations) und der WCET ω wie folgt berechnet:

$$\omega(d^i) = \sum_{i=1}^n \prod_{j=1}^m \psi(pl_j^i) \cdot \omega(p_i),$$

mit

$$\psi(pl_j^i) := \prod_{k=1}^i c_k^{max},$$

für eine verschachtelte (For-Each) Schleife.

Ein Story Pattern p ist definiert als ein n -Tupel $p = ((o_1, (ol_1^1, \dots, ol_n^1), \dots, (o_n, (ol_1^n, \dots, ol_n^n)))$, mit 4-Tupel $(o_i, (ol_1^i, \dots, ol_n^i), \eta, \zeta)$. Hierbei sind o_i Operationen, (ol_1^i, \dots, ol_n^i) ist ein n -Tupel von Operationen-Indizes, die Schleifeninformationen für jede Assoziation beinhalten, η ist eine Menge von ausgelassenen Operationen, welche durch negative Anwendungsbedingungen (NACs) benötigt werden und ζ bestimmt, ob die Operation zur rechten Hand-Seite (RHS) (ζ ist gleich dem Index der Operation) oder zur linken Hand-Seite (LHS) (ζ ist 1) gehört.

Wie für einen Story Diagramm-Pfad d^i definieren wir einen Story Pattern-Pfad p^i , welcher ebenfalls Offline berechnet wird. Jeder Index im Tupel der Operationen-Indizes ist eine Referenz zu einer Operation. Die Ordnung der Indizes gibt die Reihenfolge wider, in der die Operation ausgeführt werden.

Die WCNI einer Operation o_i definieren wir durch:

$$\psi(o_i) := \prod_{j=1}^i c_j^{max}.$$

c_j^{max} ist die maximale Konfiguration einer Operation o_i (siehe Abschnitt 6.1.2.1). Die Gesamt-WCET eines Story Pattern p ist definiert durch:

$$\omega(p) := \sum_{i=1}^n \prod_{j=\zeta_i, \eta_i \setminus j=\eta_i}^i WCNI(o_j) \cdot \omega(o_i).$$

Die WCET der Operationen o_i kann Offline durch ein Standard-WCET-Analysewerkzeug (z.B. Bound-T) bestimmt werden. Die bestimmte Ausführungszeit ist zu jeder Operation eindeutig referenziert und wird zur Laufzeit benötigt, wenn die aktuelle WCET eines Story Diagramms bestimmt wird.

Komplexität Die Berechnung $\omega_{sp_i}(\hat{C})$ und $\psi_{sp_i}(\hat{C})$ für alle Story Pattern kann zur Laufzeit für eine bestimmte Konfiguration sehr teuer sein, da alle möglichen Pfade mit ihrer Konfiguration betrachtet werden müssen. Allerdings müssen wir nur maximale Konfigurationen \hat{C}_j betrachten. Damit ist die Komplexität der WCET und WCNI Analyse $O(|V|)$ und $O(|V| + |E|)$. Zur Laufzeit wird diese Analyse im Hintergrund ausgeführt, da eine verzögerte Antwort keine Auswirkungen auf die Sicherheit des Systems hat.

Beispiel Beispiel 6.12 zeigt ein Story Diagramm $d = (d^1)$ mit drei Story Pattern RTSP p_1, p_2 und p_3 (durchnummeriert von oben nach unten). Das Story Diagramm besteht aus einem Story Diagramm-Pfad $d^1 = ((((), p_1), ((), p_2), ((2), p_3))$, wobei der Index 2 auf das zweite Story Pattern p_2 verweist.

Für dieses Beispiel nehmen wir eine initiale Situation an mit 50 TrackSection-Instanzen, 20 RailCab-Instanzen und einer einzelnen ConvoyCoordinationPattern-Instanz (siehe Beispiel 6.14).

Story Pattern $p_1 = ((3, 1, \emptyset, 1))$ zeigt ein Match einer trackSection-Instanz durch den this-Zeiger. Die Ausführungszeit der Operation ist 3 und die WCNI ist 1. Die Ausführungszeit haben wir mit Bound-T ermittelt.

Story Pattern $p_2 = ((17, 20, \emptyset, 1))$ ist das Matching der gebundenen trackSection und einer RailCab-Instanz. Da es möglicherweise 20 RailCab-Instanzen geben kann, ist die WCNI dieses Story Pattern 20.

Story Pattern $p_3 = ((40, 1, \emptyset, 1), (6, 1, \emptyset, 1), (21, 1, \{1, 2\}, 3), (30, 1, \{1, 2\}, 4), (30, 1, \{1, 2\}, 5))$ besteht aus einer Sequenz an Operationen. Die erste Operation ist das Matching einer existingCoord-Instanz mit einer initiate-Assoziation zum this-Zeiger. Die zweite Operation ist die Überprüfung, ob eine RailCab-Instanz an diesem existingCoord-Muster teilnimmt. Die WCNI dieser Operation ist 1, da die existingCoord-Instanz höchstens eine RailCab-Instanz als Teilnehmer haben kann. Operation drei ist die Erzeugung einer convoyCoord-Instanz. In diesem Fall ist ζ definiert als der Index der Sequenz der Operationen, da diese Operation teil der rechten Regelseite ist. η dieser Operation ist $\{1, 2\}$, da diese Operationen teil einer negativen Anwendung ist. Die vierte Operation besteht aus dem Hinzufügen eines Links von dem this-Zeiger zu der erzeugten

convoyCoord-Instanz und die fünfte Operation fügt ebenfalls einen Link zwischen der RailCab-Instanz und der convoyCoord-Instanz hinzu. Die WCNI dieser beiden Operationen ist 1, da in diesem Fall die existingCoord-Instanz die erste Instanz im System ist.

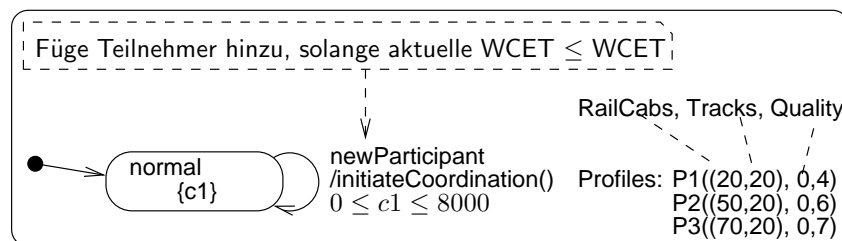


Abbildung 6.14: Parametrisiertes Profil

Auf Basis der WCNI können wir die WCET für jedes Story Pattern berechnen und anschließend die des Story Diagramms durch die oben beschriebenen Berechnungen. $\omega(p_1) = 3$, $\omega(p_2) = 340$, $\omega(p_3) = 127$ und $\omega(d^1) = 3 + 340 + (20 \cdot 127) = 2883$ Zeiteinheiten, welches die WCET des Story Diagramms d ist.

Für unser Beispiel haben wir Initial zwei Profile umgesetzt. Eins mit 20 RailCabs und das andere mit 50 RailCabs. Während der Laufzeit haben wir unterschiedliche Konvoisituationen simuliert, mit unterschiedlichen Anzahlen an möglichen Konvoiteilnehmern (siehe Abbildung 6.14). Diese Anpassungen führen zu Anpassungen des Profilgraphen. Wie aber in diesem Abschnitt beschrieben, müssen wir nur für jeden Story Diagramm-Pfad einmal die WCET neu berechnen. In unserem Fall müssen wir entsprechend für eine neues Profil mit z.B. 70 Konvoi-Teilnehmern einfach in der obigen Formel die Anzahl der RailCab Instanzen anpassen: $\omega(d^1) = 3 + 340 + (70 \cdot 127) = 9233$.

Wie in Abschnitt 6.1.2.1 kann ein Schalten in ein Profil mit mehr Ressourcen ermöglicht werden, indem eine andere Anwendung in ein Profil mit weniger Ressourcen durch das FRM geschaltet wird [OZKV08]. Wie in [BGG04a] beschrieben wäre hier ein Zusammenspiel mit dem Feder-/Neigemodul des RailCabs möglich, da dieses ebenfalls über drei unterschiedliche Ressourcen verfügt. Für ausführliche Evaluierungen diesbezüglich sei an dieser Stelle auf die Arbeiten von Oberthür verwiesen (z.B. [OZL10]).

Anzumerken ist, dass diese Form von Laufzeitanpassung eine Voraussetzung für die betrachteten selbstoptimierenden, mechatronischen Systeme ist. Klassische WCET und Scheduling-Ansätze müssen im Vergleich die Berechnungen vor der Inbetriebnahme durchführen. Für Situationen, wo das System nicht mit allen maximalen Ressourcen ausgelegt werden kann, müssen die klassischen Ansätze einen Kompromiss finden, der allerdings eine zur Laufzeit optimale Lösung verhindert.

6.1.2.3 Codegenerierung und Profilaktualisierung

Die Codegenerierung basiert auf der Codegenerierung für die sogenannten eingebetteten Story Diagramme [Sei05] (für Details sei auf diese Arbeit verwiesen). Es werden dabei Story Diagramme analysiert, um eine optimierte Graphmatching-Sequenz (siehe Abschnitt 2.4.5.2) für eine bestimmte Instanzsituation für jedes Story Pattern zu berechnen. Die Graphmatching-Sequenzen werden anschließend durch Schleifen und Container im Quellcode abgebildet.

Um die verwendeten Ressourcen einer Komponente verwalten zu können, wurde die Codegenerierung um das Pool-Allokations-Muster (Factories) erweitert (siehe Abschnitt 6.1.1). Die Factories unterstützen die Möglichkeit, die Anzahl der von Ihnen erzeugten Objekte anzupassen. Diese Funktion wird genutzt, um sicher zu stellen, dass eine Komponente nicht mehr Ressourcen beansprucht als ihr bereitgestellt wird.

Zusätzlich erhält jede Factory eine Methode, die als gemeinsame Schnittstelle zum Setzen von Instanzgrenzen verwendet werden kann. Das Setzen von Instanzgrenzen einer Factory bietet damit die Möglichkeit spezifizierte Profile und die damit verbundenen Ressourceneinschränkungen für eine Komponente auf Implementierungs-Ebene umzusetzen.

Ein Story Diagramm wird nur innerhalb der Instanzgrenzen des Profils ausgeführt. Wenn eine Anwendung nun durch eine bestimmte Strategie lernt (siehe Abschnitt 6.1.2.1), dass mehr Ressourcen benötigt werden, da z.B. kein weiteres RailCab im Konvoi aufgenommen werden kann, kann die Anwendung die Qualität der Ressourcen erhöhen oder ein Profil (siehe Abschnitt 6.1.2.1) mit den benötigten Ressourcen anlegen (siehe Abschnitt 6.1.2.1).

Eine Profilaktualisierung oder die Instanziierung eines neuen Profils ist nur erlaubt, wenn die aktualisierte WCET keine Deadline der übergeordneten Statecharts verletzt. Die relevante Deadline eines Story Diagramms ist die der Transition, die das Story Diagramm als Seiteneffekt aufruft. Hierdurch bleiben die formalen Analysen auf Modellebene weiterhin erhalten.

6.1.2.4 Evaluierungsreihenfolge hybrider Systeme mit Strukturanpassungen

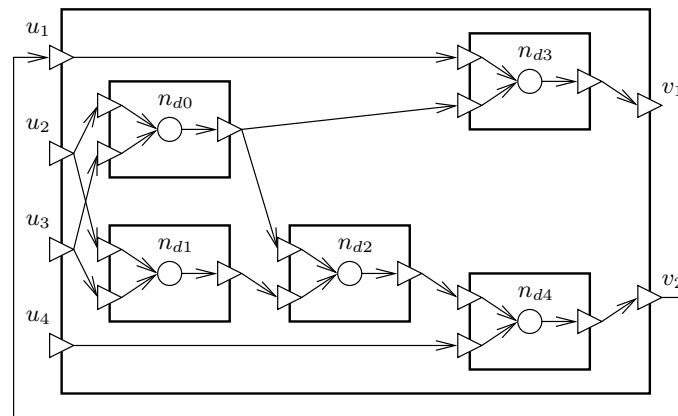
Die parallele (verteilte) Ausführung eines hybriden Systems impliziert, dass einzelne hybride Komponenten oder Teile von Komponenten getrennt voneinander ausgeführt werden und Nachrichten sowie Signale austauschen, um ihre jeweiligen Funktionen bearbeiten zu können.

Problematisch ist dies gerade aus regelungstechnischer Sicht, unter Berücksichtigung von Rekonfigurationen. Hierdurch wird gefordert, dass einzelne Blöcke oder Zusammenschlüsse von Blöcken ausgetauscht werden können. Durch eine grobgranulare Aufteilung der einzelnen regelungstechnischen Blöcke, kann allerdings eine Verklebung nicht vermieden werden (siehe Abbildung 6.15).

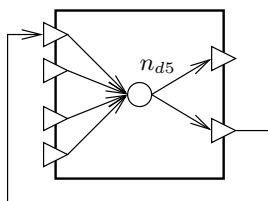
Abbildung 6.15 a) zeigt ein Beispiel für eine abstrakte Konfiguration. Wird diese zu grob granular, wie Abbildung 6.15 b) zeigt, partitioniert, dann tritt eine Verklebung auf, da die oberste Eingabe des Blocks n_{d5} eine Abhängigkeit zur Ausgabe des Blocks hat. Abbildung 6.15 c) zeigt

eine gültige Partitionierung. Eine Verklemmung tritt nicht auf, da die Schleife durch sequentielle Ausführung von n_{d8} und n_{d7} aufgelöst werden kann.

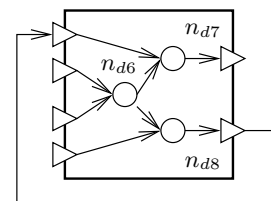
Für die verteilten Ausführungen werden daher die regelungstechnischen Modelle partitioniert. Aufgrund dieser Aufteilung kann eine verklemmungsfreie Ausführungsreihenfolge bestimmt werden.



a) Abstrakte Konfiguration



b) Unpassende Partitionierung



c) Passende Partitionierung

Abbildung 6.15: Abstraktes Partitionierungsbeispiel [Bur06]

Grundsätzlich gibt es zwei unterschiedliche Verfahren, um mögliche Ausführungsreihenfolgen (Evaluierungsreihenfolgen) zu bestimmen. Dies ist die sogenannte White-Box-Integration und die Black-Box-Integration [Hon98].

Die *White-Box-Integration* ermöglicht die Berücksichtigung der inneren Teilmodelle einer Blockstruktur zur Bestimmung der Evaluierungsreihenfolge. Durch diese enge Verzahnung der blockinternen Teilmodelle können einfach Verklemmungsfreie Evaluierungsreihenfolgen bestimmt werden, da die Kommunikationen genau dann intern stattfinden, wenn dies benötigt wird. Nachteil hierbei ist, dass ein Block oder Zusammensetzungen von Blöcken nicht einfach ersetzt oder rekonfiguriert werden können, da die internen Abhängigkeiten wieder neu aufgelöst werden müssen.

Die *Black-Box-Integration* berücksichtigt nicht die internen Teilmodelle eines Blockes. Abhängigkeiten werden entsprechend gebündelt zu bestimmten Zeitpunkten im Programmablauf

aufgelöst (durch fest definierte Kommunikationszeitpunkte). Dieser Ansatz erlaubt es damit Blockstrukturen zu rekonfigurieren, da lediglich die vorgegebene Taktung der Kommunikation eingehalten werden muss. Problem ist allerdings, dass Verklemmungen nicht vermieden werden können, wie in Abbildung 6.15 b) dargestellt.

Aufgrund der diskutierten Vor- und Nachteile beider Integrationsverfahren, werden in der Praxis häufig beide Verfahren in einer Anwendung angewandt [Hon98]. Daher wurde in [OGBG04, BGGO04b, GHH11] eine *Gray-Box-Integration* vorgestellt, die die Vorteile der White-Box- und Black-Box-Integration ausnutzt, um eine möglichst optimale Evaluierungsreihenfolge für hybride rekonfigurierende Systeme zu ermitteln.

Der Gray-Box-Integrationsansatz ist aufgeteilt in zwei Schritte. Zuerst werden die internen Abhängigkeiten der Blöcke, bzw. Blockstrukturen unabhängig von den externen Abhängigkeiten aufgelöst, so dass für diese unabhängig von anderen Modulen eine verklemmungsfreie Evaluierungsreihenfolge bestimmt werden kann. Im zweiten Schritt wird dann auf Basis der Evaluierungsreihenfolgen der Module eine Gesamtevaluierungsreihenfolge bestimmt. Wie auch ausführlich in [Bur06] diskutiert, unterstützt dieser Ansatz die Anforderungen rekonfigurierender Systeme, da die zu rekonfigurierenden Module getrennt voneinander betrachtet werden können.

Ein Problem ist allerdings, dass durch kompositionelle Strukturanpassungen a priori keine Evaluierungsreihenfolgen bestimmt werden können, da die konkrete Ausprägung nicht bekannt ist. Das in Abschnitt 3 beschriebene Konvoibeispiel (siehe Abbildungen 3.1, 3.2 und 3.3) verdeutlicht die notwendige kompositionelle Strukturanpassung, um die Konvoiparameter durch den PosCalc-Regler berechnen zu können.

Da die Struktur nicht, wie bisher durch die HYBRID RECONFIGURATION CHARTS vorgegeben, a priori für alle abhängigen möglichen Module festgelegt wird, kann auch keine Evaluierungsreihenfolge, wie für den zweiten Schritt gefordert, für das System festgelegt werden.

In [GHH11] haben wir verschiedene Ansatz vorgeschlagen, die den bisherigen Gray-Box-Ansatz erweitern, um eine Evaluierungsreihenfolge aller abhängiger Module berechnen zu können (Schritt 2), die einer kompositionellen Strukturanpassung ausgesetzt sind.

Eine erste Möglichkeit ist die datenflussgetriebene Integration. Die *Datenfluss-Integration* berechnet keine Evaluierungsreihenfolge offline. Zur Laufzeit wird eine Evaluierungsreihenfolge auf Basis des Datenflusses (der sich durch die lokalen Berechnungen sowie den Verbindungen zwischen den Modulen ausdrückt) bestimmt. Da die Abhängigkeiten nur bedingt vorhergesehen werden können, kann eine Verklemmung auftreten. Dies ist damit keine geeignete Lösung, da die Verklemmungen auch nur aufgelöst werden können, wenn es Rückfallpunkte gibt, an denen die Evaluierung erneut ausgeführt werden kann. Aufgrund der harten zeitlichen Restriktionen ist dies allerdings nur sehr bedingt möglich.

Alternativ ist es möglich für die Module, die eine kompositionelle Strukturanpassung unterstützen, an den jeweiligen Schnittstellen der Module vorher fest die Evaluierungsreihenfolgen zu definieren. Dieser Ansatz ist zwar weniger flexibel, ermöglicht allerdings eine einfache Integration in den im vorherigen Abschnitt vorgestellten WCET-Analyse und Codegenerierungsansatz.

Da die möglichen Module a priori bekannt sind (z.B. der PosCalc-Controller), kann a priori die lokale Evaluierungsreihenfolge durch die Gray-Box-Integration für jedes Modul bestimmt werden. Dies bedeutet allerdings auch, dass das eingebettete Modul nur innerhalb der vorgegebenen Grenzen kompositionelle Strukturanpassungen durchführen darf.

Zur Laufzeit kann dann, wie in den Abschnitten 6.1.2.1 und 6.1.2.2 beschrieben, die Grenze angepasst werden, wenn denn eine neue gültige Evaluierungsreihenfolge für das übergeordnete Modul bestimmt wurde. Die Flexibilität des Ansatzes ist damit sehr stark abhängig von den möglichen Ressourcen-Freiräumen zur Laufzeit, um alternative, möglicherweise bessere, Evaluierungsreihenfolgen zu bestimmen. Dieses Vorgehen ist allerdings notwendig, um die Vorhersagbarkeit des Systems zu gewährleisten. Durch die Integration mit dem in Abschnitt 6.1.2.2 vorgestellten Ansatz wird dies zugesichert.

6.1.2.5 Diskussion

In diesem Abschnitt haben wir eine neuartige WCET-Analyse in Kombination mit einem flexiblen Ressourcenverwalter vorgestellt. Dieser Ansatz garantiert Vorhersagbarkeit trotz der verwendeten komplexen Objektstrukturen mit prinzipiell unbekannter oberer Schleifengrenze. Ermöglicht wird dies durch einen modellgetriebenen Entwicklungsansatz mit entsprechender Codegenerierung. Hiermit sind wir in der Lage komplexe Funktionalitäten umzusetzen, wie sie in selbstoptimierenden, mechatronischen Systemen benötigt werden, um auf Umgebungsänderungen zur Laufzeit optimal reagieren zu können.

6.2 Umsetzung

Das Werkzeug wurde als eine Erweiterung in Form von Plugins der *Fujaba Real-Time Tool Suite* (Fujaba RT) umgesetzt. Fujaba RT ist eine Tool Suite basierend auf der Fujaba4Eclipse Tool Suite, die 2002 durch einen Neuentwurf des Open Source UML Case Tools Fujaba initiiert wurde [PTH⁺09].

Abbildung 6.16 zeigt eine Übersicht über die Architektur des Werkzeugs in Form von Plugins. Intern enthalten diese weitere Plugins, die wir im Folgenden genauer erläutern werden.

Die Fujaba4Eclipse Plugins wurden im Wesentlichen unverändert genutzt. Diese Plugins enthalten den Kern von Fujaba, die Unterstützung für Story Driven Modeling (mit Klassendiagrammen und Storydiagrammen) und (Java-) Codegenerierung.

Fujaba RT setzt die MECHATRONIC UML um. Dies beinhaltet die Strukturmodellierung mit Komponentendiagrammen, die Verhaltensmodellierung mit REAL-TIME STATECHARTS und HYBRID RECONFIGURATION CHARTS sowie eine (C++) Codegenerierung.

Um die Konzepte dieser Arbeit umzusetzen wurde zum einen, wie in Abschnitt 2.6.1 beschriebenen eine Veränderung des Metamodells vorgenommen. Die Umstellung vollständig durchzuführen ist allerdings sehr aufwendig, da z.B. auch die Codegenerierung umfangreich angepasst

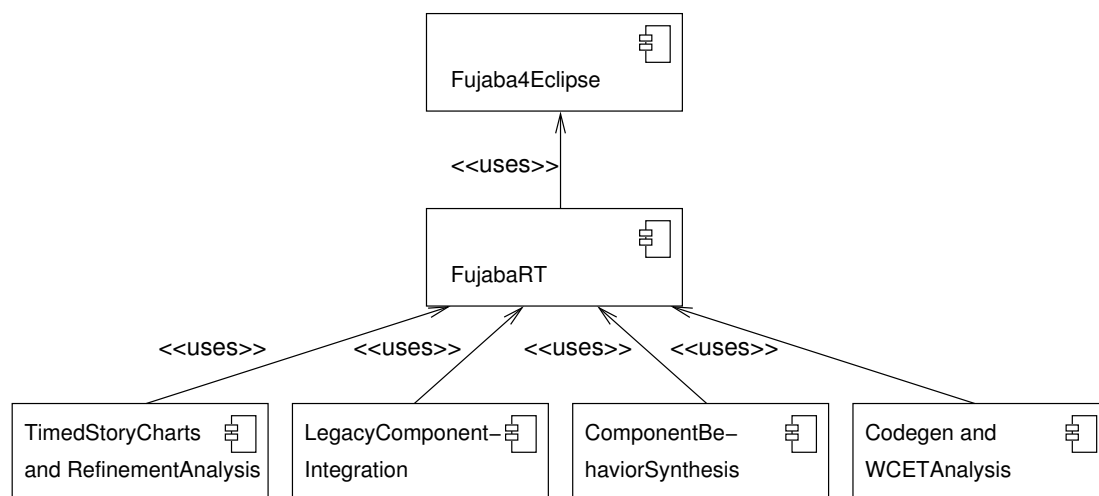


Abbildung 6.16: Übersicht Werkzeugarchitektur

werden muss. Dies liegt daran, dass bisher nur für eine Komponente ein Statechartverhalten generiert wird. Um allerdings das Löschen und Erzeugen von z.B. Ports zu unterstützen, muss auch ein Port eigenes Verhalten unabhängig von der Komponente besitzen. Im Rahmen dieser Arbeit waren diese Umstellungen nicht mehr möglich, so dass gerade die Codegenerierung zwar für die neuen Konzepte gezeigt werden kann, jedoch nicht komplett integriert mit der derzeitigen Codegenerierung für HYBRID RECONFIGURATION CHARTS. Die vollständige Umstellung aller Plugins auf das erweiterte Metamodell wird aktuell von dem Fujaba RT Team² vorangetrieben.

Weitere Erweiterungen sind unter dem Motto Konsistenzhaltung zwischen Fujaba RT Diagrammen erfolgt. Z.B. wurde eine MessageInterface-Plugin in der Projektgruppe Mauritius [ACE⁺08] eingeführt, um Konsistenz zwischen Nachrichten sicherzustellen, die in der vorherigen Version nur über Strings umgesetzt wurden oder auch eine automatische Konsistenzhaltung zwischen Rollen- und Portverhalten (umgesetzt in der PG ReCab [BBB⁺09]). Speziell der Aspekt der Konsistenzhaltung zwischen mehreren Diagrammen von Fujaba RT wurde auf den Fujaba Days vorgestellt [ACE⁺08]. Eine Erweiterung des Umschaltkonzepts für Regler, die eine vorhersagbare Umschaltung ermöglichen, wurde in [Poh08] realisiert.

Das Plugin TimedStoryCharts and RefinementAnalysis beinhaltet die Implementierung der TIMED STORY CHARTS (siehe Kapitel 2.6) und der Verfeinerungsüberprüfung für die Wiederverwendung von modellierten Komponenten (siehe Kapitel 3). Um Berechnungen auf dem Zone Graphen (siehe Abschnitt 3.2.1) vorzunehmen, nutzen wir die UPPAAL UDBM Bibliothek³ aus. Die Erreichbarkeitsanalyse innerhalb der Verfeinerungsüberprüfung nutzt eine Erreichbarkeitsanalyse auf Storydiagrammen aus [Zün09] und erweitert diese um eine zeitbehaftete Erreichbarkeitsanalyse. Implementiert wurden diese Plugins im Rahmen der Masterarbeit von Christian

²Das Fujaba RT Team besteht im Wesentlichen aus den Mitarbeitern des Teilprojekts B1 des Sonderforschungsbereichs 614 (<http://www.sfb614.de/sfb614/projektbereiche/projektbereich-b/teilprojekt-b1/>)

³<http://www.cs.aau.dk/adavid/UDBM/>

Heinzemann [Hei09]. Die Ergebnisse wurden im Rahmen einer Fujaba Days Demo in Kooperation mit Albert Zündorf vorgestellt [HHZ09]. Aktuell werden die bisherigen Plugins in der Form erweitert, dass auch eine Verifikation von Eigenschaften möglich ist. Um dabei nicht den Charakter einer reinen Erreichbarkeitsanalyse zu verlieren wurde hier das Konzept von Testautomaten verfolgt, die eine bestimmte Klasse von Eigenschaften in eine reine Erreichbarkeitsanalyse transformieren [Bre10]. Christian Heinzemann, der sich bereits in seiner Masterarbeit [Hei09] mit Timed Story Charts und deren Verfeinerung auseinandergesetzt hat, forciert dieses Thema im Rahmen seiner Dissertation, so dass auch schon die Verifikationsumgebung auf einem Tool Contest vorgestellt wurde [HSJZ10].

Das LegacyComponentIntegration-Plugin beinhaltet die Umsetzung der Konzepte zur Integration von Altkomponenten (siehe Kapitel 4). Intern wird die Umsetzung auf vier Plugins verteilt: 1) LegacyComponent-Editor, 2) BlackBoxChecking, 3) WhiteBoxChecking und 4) GrayBoxChecking. Die Plugins 1) - 3) wurden im Rahmen der Projektgruppe ReCab [BBB⁺09] umgesetzt und das Plugin 4) in der Bachelorarbeit von Christian Brenner [Bre08]. Die Plugins 1) - 3) sowie auch 4) wurden jeweils auf den Fujaba Days vorgestellt [HBB⁺09, BGH⁺08]. Der Gesamtansatz der Integration von Altkomponenten, die FRiTS^{Cab} Tool Suite, wurde auf dem Research Demonstration Track der International Conference on Software Engineering in Kooperation mit der Hella KGaA Hueck & Co.⁴ vorgestellt [HMS⁺10].

Die Konzepte der Komponentenverhaltenssynthese werden in dem ComponentBehaviorSynthesis-Plugin implementiert. Um Berechnungen auf dem Zone Graphen vorzunehmen (siehe Abschnitt 5.3.1) nutzen wir, wie bereits oben für die Verfeinerungsüberprüfung beschrieben, eine Anbindung der UDBM Bibliothek aus. Umgesetzt wurde die Synthese in der Diplomarbeit von Tobias Eckardt [Eck09], die auch als Demo auf den Fujaba Days vorgetragen wurde [EH09]. Eine Integration mit der Synthese von Rollenverhalten (siehe Abschnitt 2.4.1) wurde auf dem Research Demonstration Track der International Conference on Software Engineering vorgestellt [HGH⁺09].

Das Plugin Codegen and WCETAnalysis lässt sich in die drei Bereiche Codegenerierung, WCET-Analyse und Ausführungsumgebung aufteilen, die die Konzepte zur Codegenerierung und Ausführung (siehe Kapitel 6.1) umsetzen. Die Umsetzung der Codegenerierung erweitert die (C++) Codgenerierung für Story Diagramme um ein Factory-Konzept zum Erzeugen und Löschen von den Elementen einer Komponentenarchitektur. Die Erzeugung wird dabei in Abhängigkeit von den möglichen Ressourcen, die durch ein Ressourcenprofil festgelegt werden (siehe Abschnitt 6.1.2.1), kontrolliert. Wie bereits oben erläutert konnte hier keine vollständige Integration mit der Codegenerierung für HYBRID RECONFIGURATION CHARTS [BGH⁺07, HH08b] realisiert werden. Die WCET-Analyse nutzt intern eine Anbindung an das WCET Analyse-Werkzeug BoundT aus, um für einzelne Codefragmente eine WCET zu berechnen (siehe Abschnitt 6.1.2.2). Eine Assoziation zur Codegenerierung ermöglicht dabei die vorherige Codegenerierung und Übersetzung, die als Eingabe für das WCET Analyse-Werkzeug benötigt wird.

⁴<http://www.hella.com/hella-de-de/index.html>

Die Laufzeitumgebung baut auf der Laufzeitumgebung für HYBRID RECONFIGURATION CHARTS [GH06a] auf, und erweitert diese um die Möglichkeit der Simulation von Zeit und einer deterministischen Wiederholung, wie dies für die Integration von Altkomponenten benötigt wird (siehe Kapitel 4). Umgesetzt wurden diese Plugins im Wesentlichen im Rahmen der Projektgruppe ReCab [BBB⁺09]. Die Plugins wurden ebenfalls auf den Fujaba Days vorgestellt [HBB⁺09].

6.3 Validierung

Das Ziel des in dieser Arbeit vorgestellten Ansatzes (siehe Kapitel 2.6 bis 5) ist es, einen modellgetriebenen Entwicklungsansatz für selbstoptimierende, mechatronische Systeme, in dessen Mittelpunkt die Komposition und Wiederverwendung von Softwarekomponenten und deren Protokollverhalten zu komplexen hierarchischen Komponentensystemen stehen, bereitzustellen (siehe auch Abbildung 2.1).

Diese Ergebnisse wurden mit Hilfe des im vorherigen Abschnitt 6.2 vorgestellten Werkzeugs validiert. Nach [EFR08, Bec08] können die Ergebnisse, die entstandenen Methoden, auf drei unterschiedliche Arten validiert werden: 1) mit Typ I Validierung kann gezeigt werden, dass die Absicht (Vorhersage) der Methoden der beobachteten Realität entspricht, wenn die Methode und das Werkzeug richtig angewendet werden. 2) Die Typ II Validierung zeigt, dass die Methoden erfolgreich von geschulten Benutzern angewandt werden können. Einen Vorteil des entwickelten Gesamtansatzes gegenüber verwandten Ansätzen zeigt die Typ III Validierung.

Der Schwerpunkt dieser Arbeit liegt stärker im Bereich der Entwicklung von notwendigen Methoden für die Entwicklung von selbstoptimierenden, mechatronischen Systemen (die durch bisherige Ansätze noch nicht unterstützt werden) und weniger im Bereich der experimentellen Validierung des Gesamtansatzes der MECHATRONIC UML und den hier vorgestellten Erweiterungen. Daher werden Typ II und Typ III Validierung nicht näher betrachtet. Ein Vergleich mit anderen Ansätzen, in Form einer Validierung, ist zudem nicht ohne weiteres möglich, da die MECHATRONIC UML und die hier vorgestellten Erweiterungen notwendige Konzepte für die betrachteten Systeme anbieten, die durch verwandte Ansätze nicht unterstützt werden [GH06b] (siehe auch Kapitel 7).

Um eine Typ I Validierung für den Ansatz dieser Arbeit durchzuführen, müssen wir zum einen zeigen, dass wir die Klasse der betrachteten selbstoptimierenden, mechatronischen Systeme modellieren können. Zum anderen müssen wir die entwickelten Analysen und Synthesen dahingehend validieren, dass diese den gestellten Anforderungen standhalten.

Das in Abschnitt 1.2 eingeführte RailCab-Projekt adressiert alle gestellten Anforderungen an den Ansatz und ist daher eine geeignete Anwendung, um die Validierung zu zeigen. Im Speziellen betrachten wir das Konvoi-Szenario. Die wesentlichen Argumente für eine erfolgreiche Validierung wurden bereits in den einzelnen Abschnitten anhand des durchgängig betrachteten Konvoi-Szenarios gezeigt. Im folgenden Abschnitt 6.3.1 werden wir die Spezifikation und Analyse der

RailCab-Anwendung mit Hilfe der Werkzeugumgebung darstellen und Evaluierungsergebnisse in Form von Laufzeit und Speicherverbrauch diskutieren. Weitere Anwendungsszenarien betrachten wir in Abschnitt 6.3.2. Die Validierung ist im Wesentlichen im Rahmen der in Abschnitt 6.2 beschriebenen Master- und Bachelorarbeiten sowie durch Projektgruppen und Tool-Demos erfolgt.

6.3.1 Konvoi-Anwendungsszenario

In den Einleitungen zu den Hauptkapiteln dieser Arbeit (Kapitel 3 bis 5) wurde jeweils zugeschnitten für die Kapitel ein Ausschnitt des Konvoi-Szenarios gezeigt, an dem die Notwendigkeit der dort erläuterten Methode diskutiert wurde. Wir werden in diesem Abschnitt das entwickelte Werkzeug in den Vordergrund stellen und die Spezifikation und Analyse des Konvoi-Szenarios hieran demonstrieren. Der Ablauf orientiert sich dabei an Abbildung 2.1. Den ersten Schritt, Szenarien modellieren und Rollenverhalten synthetisieren, werden wir allerdings überspringen. Für Details hierzu sei auf [BGK05, HGH⁺09] verwiesen.

Wir beginnen mit einem Ausschnitt der Modellierung des RailCab Beispiels mit der Fujaba Real-Time Tool Suite in Abschnitt 6.3.1.1. In den Abschnitten 6.3.1.2 bis 6.3.1.4 werden wir zeigen, wie die umgesetzte Werkzeugunterstützung die Wiederverwendung von Komponenten, Alt Komponenten und Protokollverhalten adressiert. Abschließend in Abschnitt 6.3.1.5 werden wir die Werkzeugunterstützung für die Codegenerierung sowie die WCET-Analyse vorstellen, die wir für die Integration von Alt Komponenten benötigen und die zudem den modellgetriebenen Entwicklungsansatz vervollständigt.

6.3.1.1 Modellierung

Die Modellierungsumgebung erweitert die bisherige Werkzeugunterstützung der MECHATRONIC UML. Basisarbeiten der Werkzeugunterstützung der MECHATRONIC UML wurden in [BGH⁺05b] und [BGH⁺07] vorgestellt. Besonderer Schwerpunkt der Erweiterungen der Modellierungsumgebung liegen dabei zum einen auf der Unterstützung von Alt Komponenten sowie den bereits in Abschnitt 6.2 angesprochenen Erweiterungen des unterliegenden Metamodells, um eine durchgängige Entwicklung zu unterstützen. Die Durchgängigkeit bezieht sich dabei auf die in Abbildung 2.1 dargestellten Hauptaktivitäten der MECHATRONIC UML. So wurde in der Projektgruppe Mauritius [ACE⁺08] besonders die Durchgängigkeit zwischen einer Anforderungsspezifikation mit sogenannten Goals (z.B. [Lam09]), über eine Szenario Spezifikation und der Synthese von Zustandsverhalten hieraus, bis hin zur Komponentenspezifikation erarbeitet. Der Übergang zur hierarchischen Komponentenspezifikation mit Strukturanpassung, Constraint-Definition und Analyse bis hin zur Codegenerierung wurde in der Projektgruppe Re-Cab [BBB⁺09] erarbeitet. Im Folgenden werden wir einen Ausschnitt des RailCab Szenarios anhand dieser erweiterten Werkzeugumgebung vorstellen.

Die im Folgenden mit der Werkzeugumgebung illustrierten Modellelemente wurden ausgewählt, um zu zeigen, dass mit der Werkzeugumgebung ein Entwickler auch die notwendigen Modelle für die betrachtete Anwendungsdomäne beschreiben kann. Der Vollständigkeit halber wird an den entsprechenden Stellen auf die umfangreichere Beschreibung der Beispielanwendung in den Konzeptkapiteln verwiesen.

Wir beginnen mit der Modellierung der Kommunikation. Abbildung 6.17 zeigt die Struktur des DistanceCoordination-Musters. Hierfür haben wir ein REAL-TIME COORDINATION PATTERN mit den Rollen front und rear angelegt. Zusätzlich haben wir die einzuhaltenden Eigenschaften definiert. Dies ist zum einen $A[] \text{ rear.Convoy imply front.Convoy}$ sowie die Deadlock-Freiheit. Diese Eigenschaften beziehen sich auf das Verhalten der Rollen, welche wir mit REAL-TIME STATECHARTS spezifizieren.

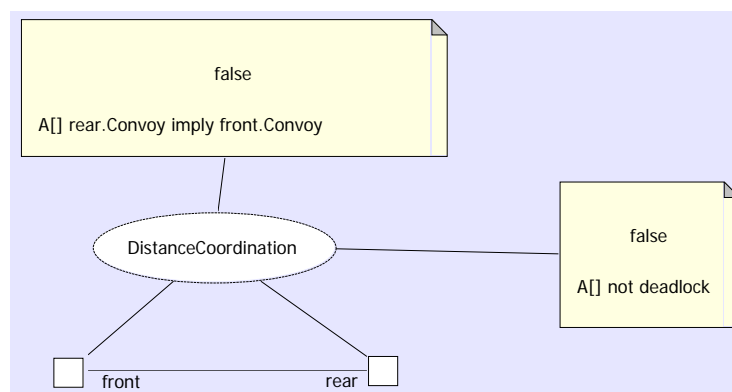


Abbildung 6.17: REAL-TIME COORDINATION PATTERN DistanceCoordination

Abbildung 6.18 zeigt das REAL-TIME STATECHART der front-Rolle. Um festzulegen welche Nachrichten die Rollen austauschen können, werden im Message Interface Editor die Nachrichten definiert. Explizit zeigen wird die Schnittstelle nicht. Die Nachrichtenbezeichnung ist jedoch im Statechart ersichtlich. `msgIFace_context` und `msg_IFace_legacy` sind die beiden Schnittstellen. `msgIFace_context` definiert die Nachrichten `LEAVE_CONVOY`, welche die Transition `leave` nach `noConvoy` schaltet, `CONVOY_REQUEST`, die die Transition `noConvoy` nach `waitConvoy` schaltet und die Nachricht `LEAVE_CONVOY_REQUEST`, die die Transition von `convoy` nach `waitNoConvoy` schaltet. Die restlichen Nachrichten der front-Rolle werden durch die `msg_IFace_legacy`-Schnittstelle definiert.

Wie bereits in den vorherigen Verhaltensbeschreibungen zur front-Rolle erläutert (siehe Abschnitt 2.4.2) besteht die Koordination zum einen aus dem Teil der Anfrage, ob ein Konvoi realisiert werden soll (Schleife zwischen den Zuständen `noConvoy` und `waitConvoy`) und zum anderen aus der periodischen Anfrage (im Intervall $0 \leq c1 \leq 100$) ob der Konvoi aufgelöst werden soll. Im Intervall $150 \leq c1 \leq 200$ kann der Konvoi aufgelöst werden. Fehler, wie z.B. Netzwerkfehler, werden durch das Verhalten nicht berücksichtigt, um die Anschaulichkeit des Beispiels nicht zu verlieren. Netzwerkfehler werden z.B. in [HHG08] betrachtet. Das Verhal-

ten der rear-Rolle werden wir im Folgenden nicht explizit modellieren, sondern durch unsere Altkomponenten-Integration erlernen (siehe Abschnitt 6.3.1.3).

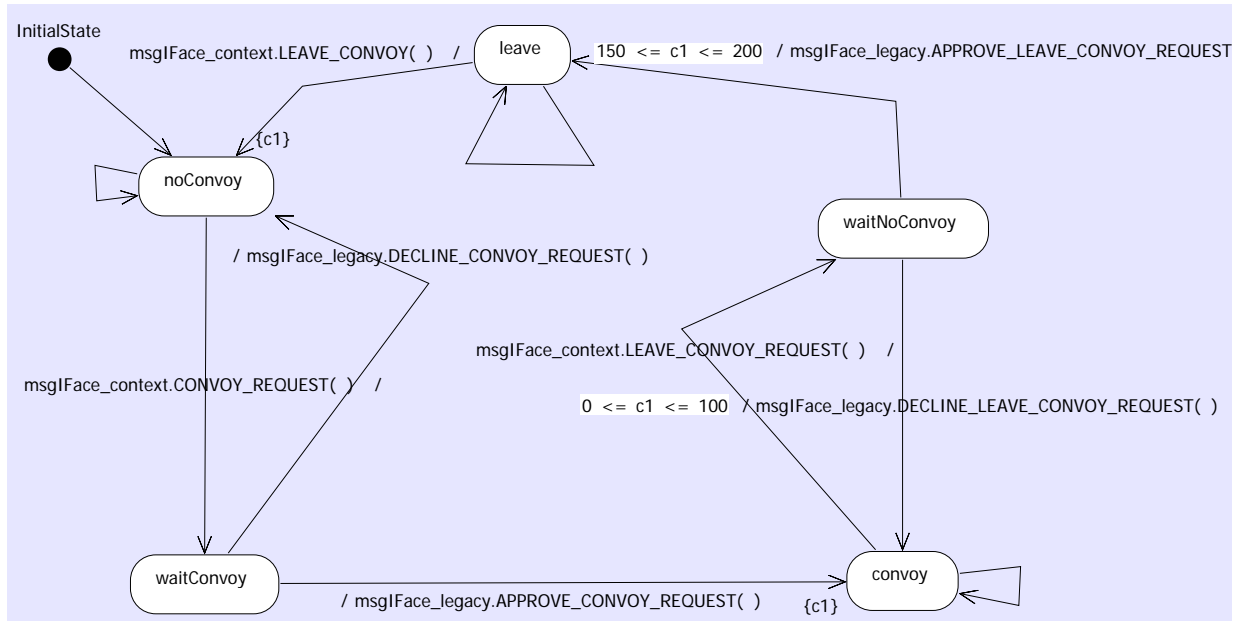


Abbildung 6.18: REAL-TIME STATECHART front-Rolle

Neben dem DistanceCoordination-REAL-TIME COORDINATION PATTERN haben wir in dieser Arbeit noch das Registration-REAL-TIME COORDINATION PATTERN betrachtet sowie das ConvoyCoordination-PARAMETERIZED REAL-TIME COORDINATION PATTERN. Die Möglichkeit der Spezifikation eines Musters ohne Multi-Rolle haben wir oben gezeigt. Wir werden daher in 6.3.1.4 nur noch auf notwendige Anpassungen des Registration-Musters eingehen, um die Synthese von Komponentenverhalten illustrieren zu können. Der Wesentliche Unterschied eines PARAMETERIZED REAL-TIME COORDINATION PATTERN zu einem REAL-TIME COORDINATION PATTERN ist die Spezifikation des Verhaltens mit Parametern. Wir werden daher diesen Aspekt der Werkzeugunterstützung beleuchten.

Abbildung 6.19 zeigt die Rolle coordinator. Das Statechart der Coordinator-Komponente ist in Abbildung 6.19 dargestellt. Dieses enthält einen Zustand mit zwei AND-States, die parallel ausgeführt werden. Der untere AND-State beschreibt das Adaptionverhalten für den Multi-Port, der obere AND-State das parametrisierte Rollenstatechart für die Coordinator Rolle. Wie in Kapitel 3 auf Seite 65 beschrieben, wird mit dem Synchronisationkanal *next* über den Parameter *k* des Rollenstatecharts jede Rolleninstanz geordnet nacheinander angestoßen, um die einzelnen Konvoiteilnehmer (*n* speichert die aktuelle Anzahl der Konvoiteilnehmer) nacheinander zu aktualisieren (update-Nachricht). Angestoßen werden die Rolleninstanzen durch das Adaptionstatechart (unterer AND-State), indem die Transition mit dem Synchronisationskanal *next[!]* von Zustand Convoy nach SendUpdates geschaltet wird. In dem vorderen Teil des Adaptionstatecharts wird der initiale Port (Übergang von noConvoy nach CreatePort) angelegt. Die Seiteneffekte und Syn-

chronisationskanäle an den Transitionen von Zustand CreatePort nach Convoy erzeugen weitere Portinstanzen.

Nachdem alle Rollen spezifiziert sind, definieren wir den Komponententyp RailCab, der alle Rollen der Muster anwendet (siehe Abbildung 6.20). Weiterhin wird der Komponententyp PosCalc wiederverwendet, indem er als Part in dem Komponententyp RailCab eingebettet wird, die damit hierarchisch aufgebaut ist. Zudem können auf der Typebene Kompositionsregeln definiert werden, um Abhängigkeiten zwischen Rollenverhalten zu bestimmen (siehe Kapitel 5). In dem Beispiel wird gefordert, dass ein Konvoi nur durchgeführt werden darf, wenn das RailCab auch bei einer Streckenabschnittskontrolle registriert ist (`!(registree.unregistered AND rear.convoy)`).

Wie schon zu Abbildung 3.1 beschrieben, ist das spezifizierte Verhalten des Multi-Parts PosCalc (siehe Abbildung 6.21) verschieden zu dem Verhalten des coordinator-Multi-Ports. Zum einen zeigt das Statechart eine andere Struktur (unterschiedliche Anzahl an Zuständen und Transitionen) auf sowie ein anderes zeitliches Verhalten (im Zustand AwaitAck kann länger verweilt werden). Die verarbeiteten Nachrichten sind allerdings identisch, so dass keine Schnittstellenbeschränkung vorgenommen werden muss (siehe Definition 20). Das Adaptionverhalten des PosCalc-Multi-Parts unterscheidet sich zum einen von dem Adaptionverhalten der Coordinator-Multi-Rolle durch die aufgerufenen Seiteneffekte. Zum anderen unterscheiden sich die Synchronisationen. Das Adaptionverhalten der Coordinator-Multi-Rolle startet durch die createPort-Synchronisation das Erzeugen der Delegations- und Part-Instanzen.

Die Synchronisation zur Erzeugung der internen Elemente wird nicht direkt von der Multi-Rolle zum Multi-Part angesteuert, sondern über das Adaptionverhalten der Delegation (siehe Abbildung 6.22). Der Multi-Port führt bei der Erstellung einer neuen Portinstanz eine Synchronisation über den Kanal createAbsPort durch. Danach kann über den Synchronisationkanal createRefPart eine Instanz des Multi-Parts angelegt werden. Nachdem die Rekonfiguration ausgeführt wurde, werden der neu erstellte Port und der neu erstellte Part über die Funktion createDelegation mit einer Delegation verbunden. Die Funktion ist in Abbildung 6.23 als Story Diagramm angegeben. Es werden der Port und der Part verbunden, die keine Assoziation zu einer Delegation besitzen. Für eine umfangreiche Beschreibung der Strukturanpassung sei auf Abschnitt 2.6 Seite 47 verwiesen.

Um Altkomponenten zu integrieren, unterstützt die Werkzeugumgebung zusätzlich die Spezifikation von Legacy-Komponenten, die im Editor schwarz dargestellt werden. Zuerst wird dabei ein Typ der Altkomponente angelegt und alle bekannten Informationen beschrieben. Abbildung 6.24 beschreibt die Eigenschaften einer RailCab Altkomponente. Je nachdem, welche Informationen der Altkomponente vorliegen, können, wie in Kapitel 4 beschrieben, verschiedene Integrationsverfahren durchgeführt werden. In dem dargestellten Beispiel liegen alle relevanten Informationen für alle drei Integrationsverfahren vor. Für das Black Box Checking ist das der Pfad zu der binären Datei der Altkomponente sowie die Obergrenze der Zustände der Altkomponente. Für das White Box Checking wird der Pfad zur Quelldatei benötigt, die Send-/Empfangsmethode, Zeitmethode (nicht notwendig, ohne kann allerdings keine Zeit berücksichtigt werden), die periodisch ausgeführte Methode sowie die Initialisierungsdatei. In den Präferenzen gibt es zudem noch die Möglichkeit für das Black Box Checking zwischen den Äquivalenzalgorithmus nach

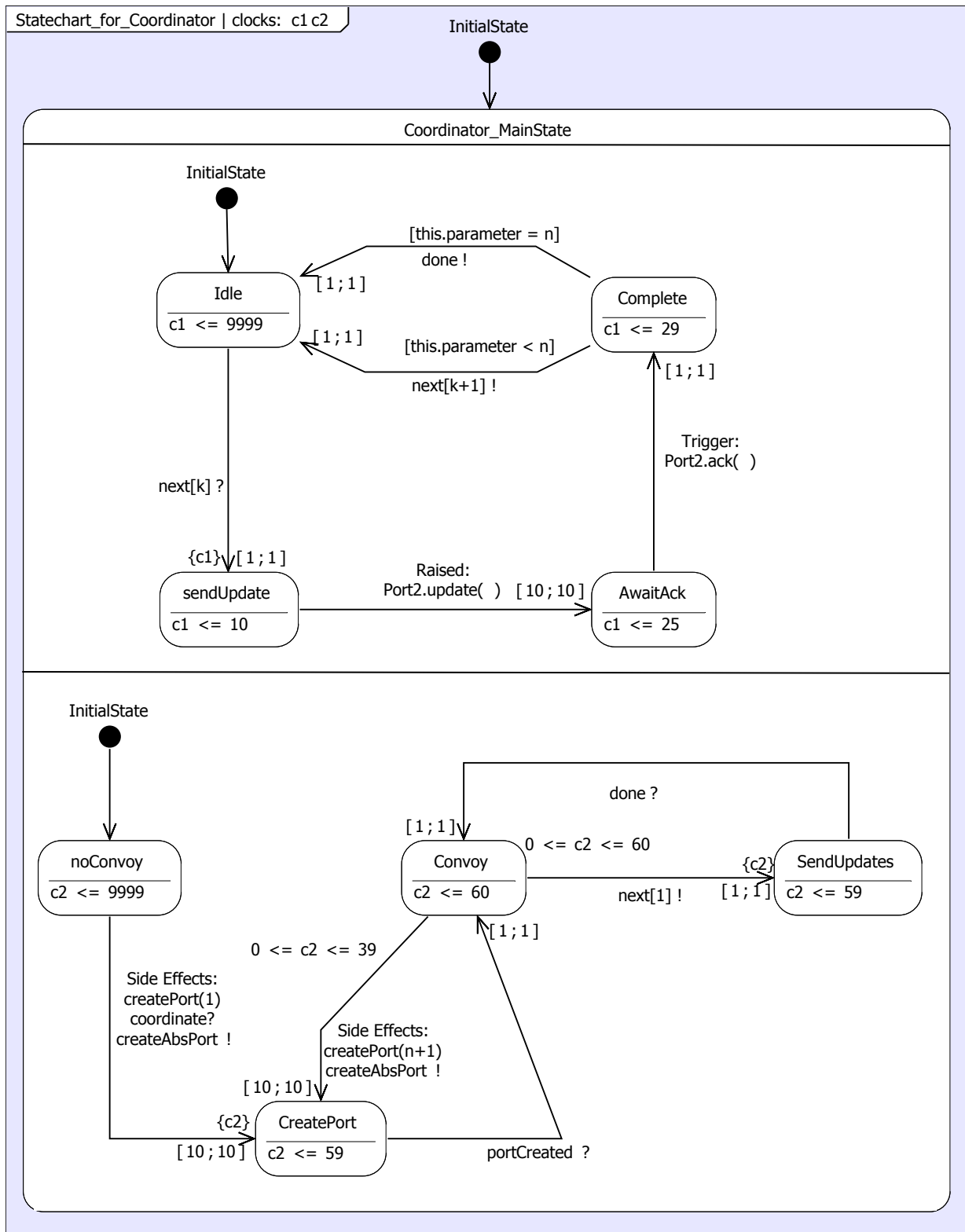


Abbildung 6.19: PARAMETERIZED REAL-TIME STATECHART coordinator-Rolle

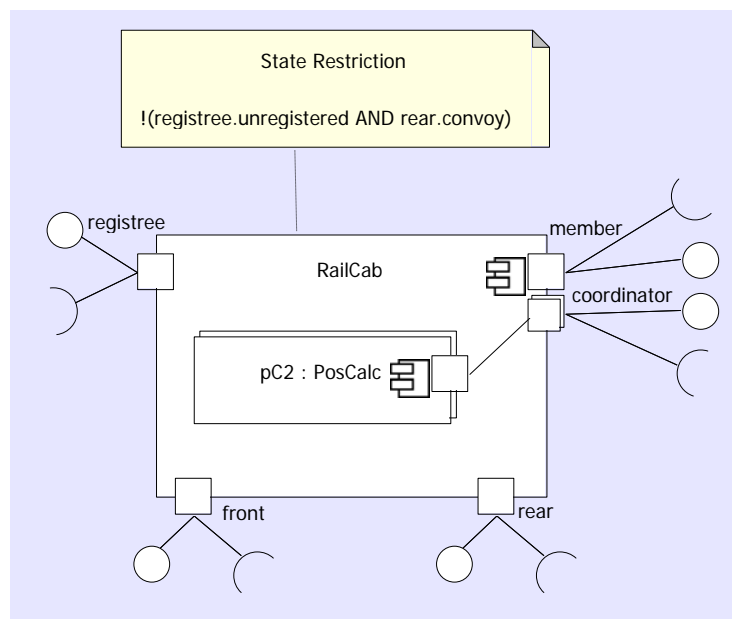


Abbildung 6.20: RailCab Komponententyp

Vasileskii und Chow sowie einen einfachen Algorithmus zu wählen, der ohne Optimierung alle Möglichkeiten überprüft. Für das Gray Box Checking kann in den Präferenzen der Pfad für die Constraint-Datei oder auch eine andere Binärdatei, als die für das Black Box Checking, angegeben werden.

Eine Instanzsicht der Beispielanwendung zeigt Abbildung 6.25. Die Verbindung der Komponenten ist dabei nur möglich, wenn die Schnittstellen zueinander passen (die ein-/ausgehenden Nachrichten müssen den gleichen Typ haben). Hieraus wird zudem die für die Integration einer Altkomponente relevante Schnittstelle mit dem Modell spezifiziert, indem die entsprechenden Schnittstellen miteinander verbunden werden.

6.3.1.2 Verfeinerungsüberprüfung

Als erstes wollen wir im Folgenden die Unterstützung zur Wiederverwendung von Komponenten betrachten, die durch eine Hierarchisierung ausgedrückt wird, indem eine Komponente in eine andere eingebettet wird (siehe hierzu Konzeptkapitel 3). Für unser Beispiel soll eine Verfeinerung zwischen der Multi-Rolle coordinator (hier also das abstrakte Verhalten) und dem Multi-Part PosCalc überprüft werden (konkretes Verhalten). Die Benutzerführung ist bisher nicht vollständig umgesetzt, so dass die Überprüfung noch nicht automatisch durch Selektion der Port-/Part-Elemente erfolgen kann, da eine automatische Überführung des parametrisierten Verhaltens in TIMED STORY CHARTS zum Teil fehlt. Um daher eine Verfeinerung zu überprüfen, muss dieser

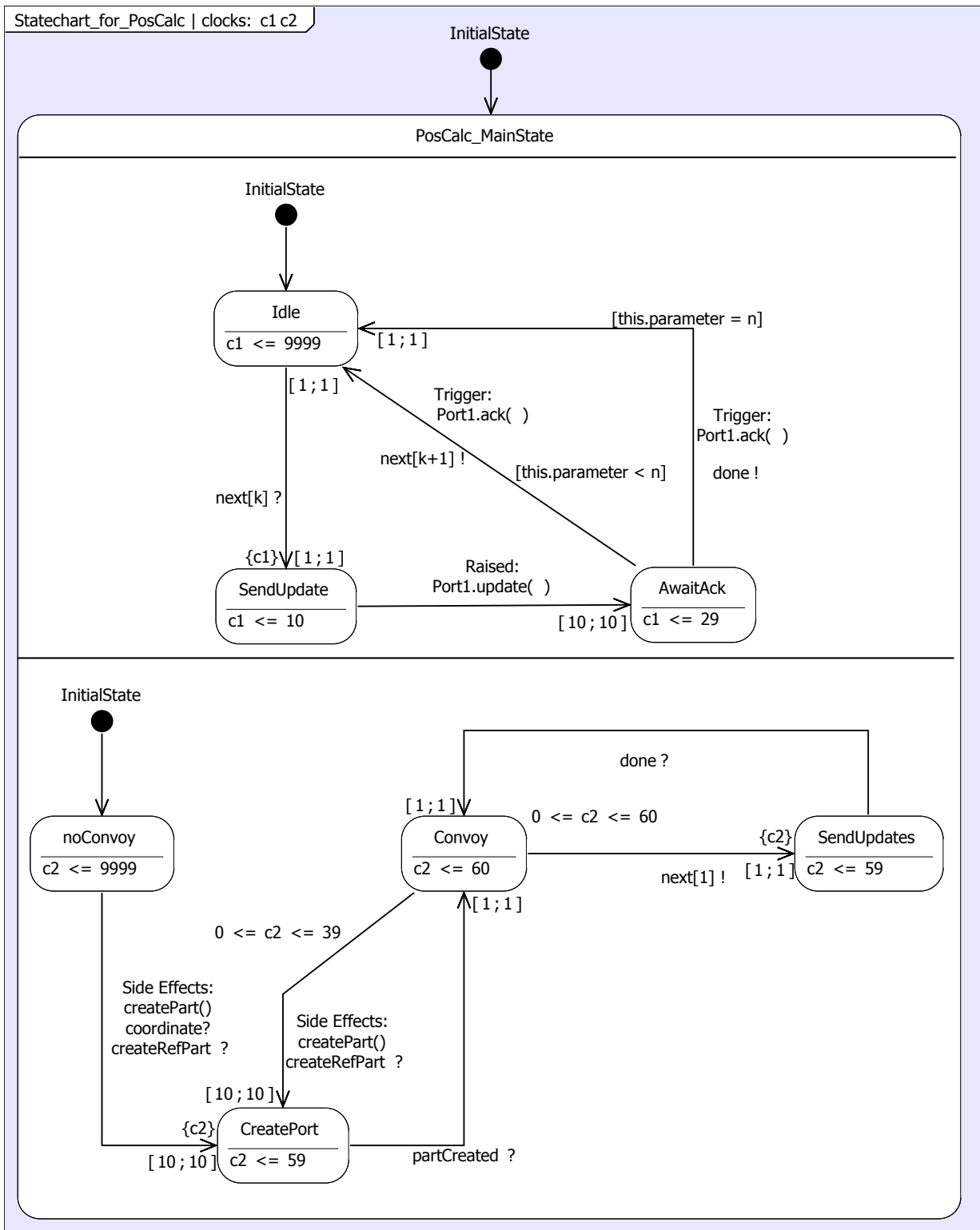


Abbildung 6.21: PARAMETERIZED REAL-TIME STATECHART PosCalc-Port

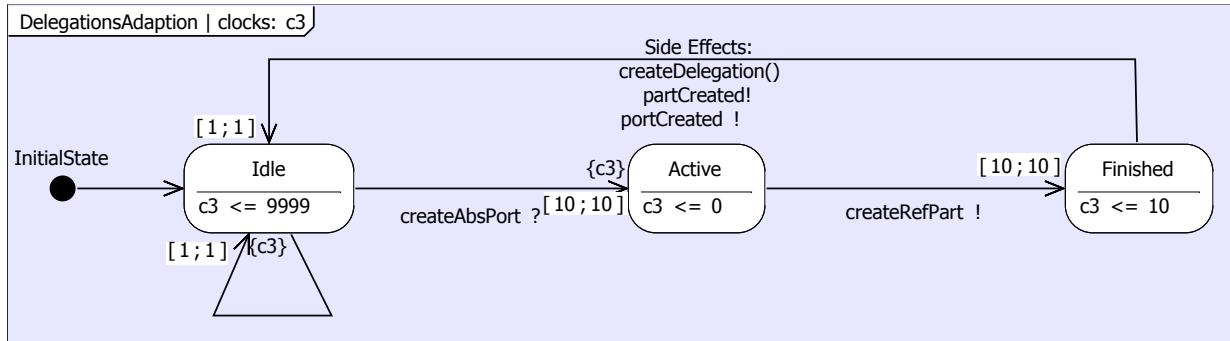


Abbildung 6.22: REAL-TIME STATECHART Delegation

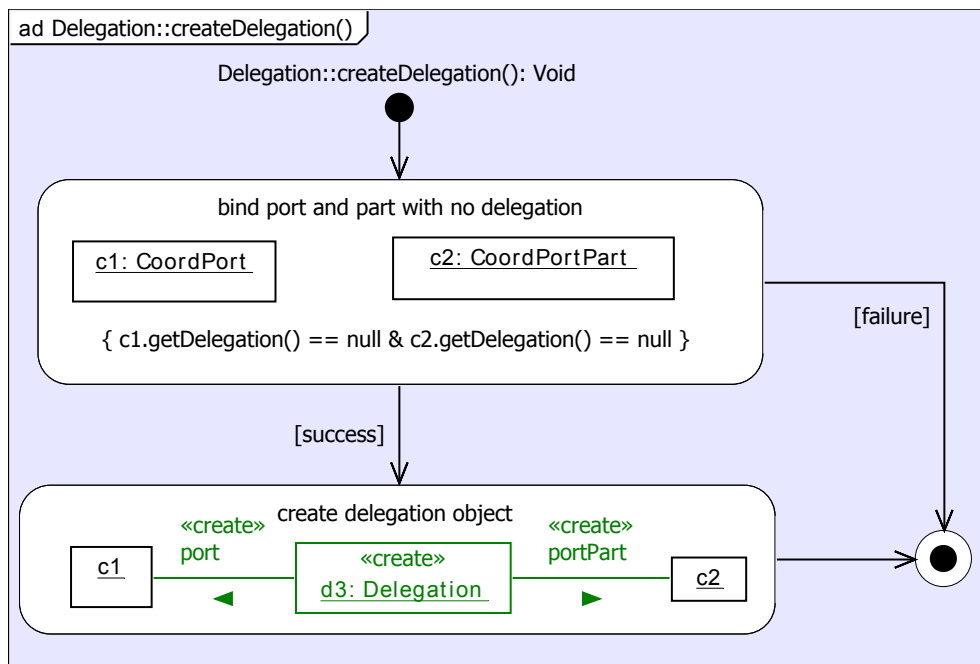


Abbildung 6.23: Erzeugen einer Delegation

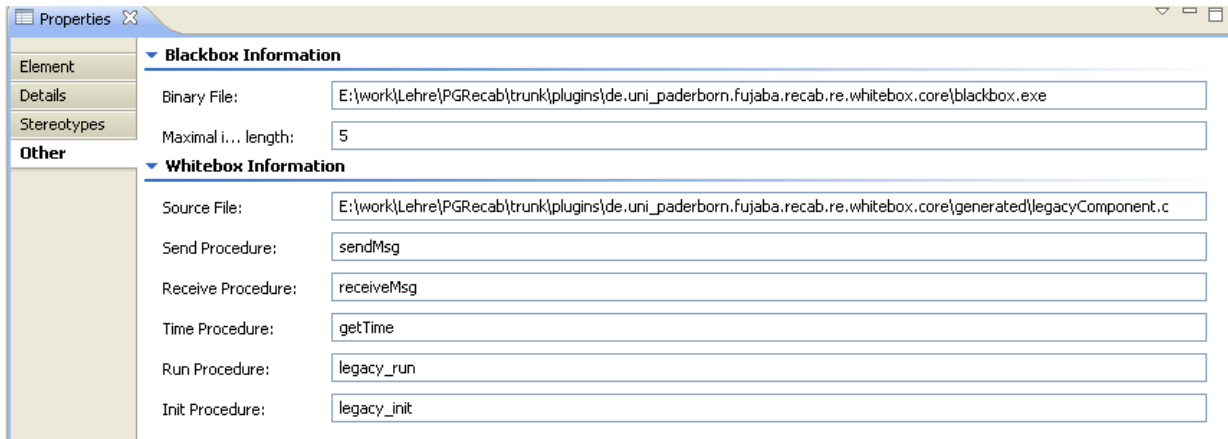


Abbildung 6.24: Eigenschaften Altkomponente

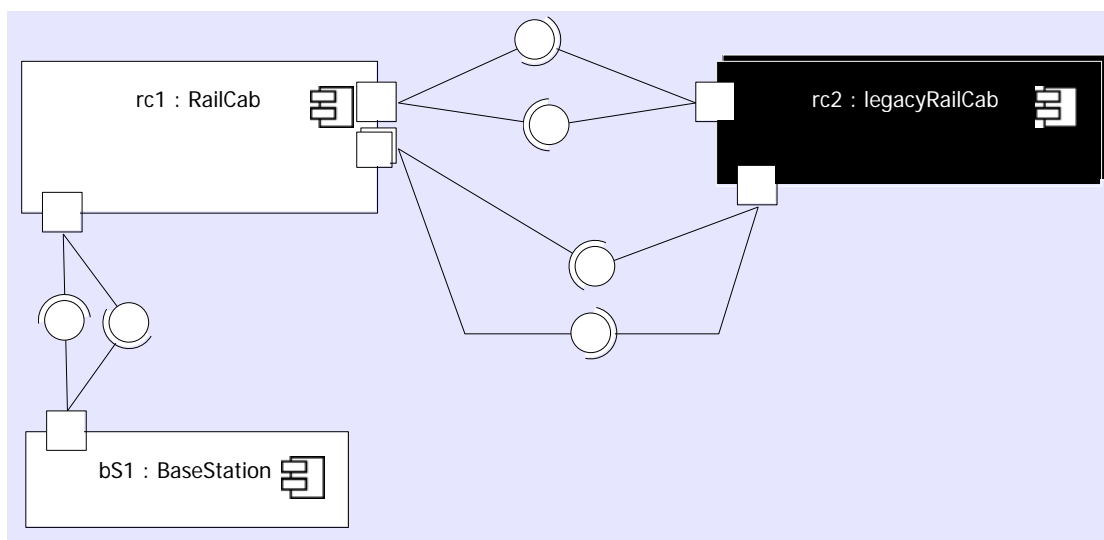


Abbildung 6.25: RailCab-Konvoi mit Altkomponente

Schritt bisher zum Teil manuell durchgeführt werden (in [Hei09] wurde dieser Schritt dargestellt).

Die Verifikation der Beispielanwendung ergibt, dass der Multi-Part PosCalc eine korrekte Verfeinerung nach der relaxierten Timed Bisimulation aus Abschnitt 3.1.2 der Multi-Rolle coordinator ist, obwohl die zeitlichen Intervalle verschieden sind. Die Verschiebung des Echtzeitverhaltens führt allerdings dazu, dass dies nur eine gültige Verfeinerung nach der relaxierten Form ist. Die bisherige Verfeinerungsdefinition der MECHATRONIC UML, die eine strikte Bisimulation fordert, oder auch die aus dem UPPAAL-Umfeld verbreitete Timed Ready Simulation (siehe Abschnitt 2.4.7.3) würden einen Konflikt aufgrund der Zeitverschiebung erkennen. Außer Betracht steht bei diesem Vergleich, dass die verwandten Verfeinerungen keine Strukturanpassungen betrachten, also auch keine Verfeinerungsüberprüfung für das Beispiel durchführen können. Nicht erfüllende Verfeinerungen wurden ebenfalls betrachtet, indem z.B. die Invariante des Zustands SendUpdate auf 22 gesetzt wurde. Die Nachricht update kann dann in der Verfeinerung zu spät verschickt werden. Diese Fehler wurden korrekterweise erkannt.

Im Rahmen der Masterarbeit von Christian Heinzemann [Hei09] wurden zudem noch einige Messungen durchgeführt. Diese sind auf einem PC mit Intel Core2Duo Prozessor mit 3 Ghz und 3GB Arbeitsspeicher durchgeführt worden. Als Betriebssystem wurden Microsoft Windows XP eingesetzt. Eclipse wurde in der Version 3.4 mit 1,5 GB initialisiert. Während des Testdurchlaufs wurden die Invarianten und Time Guards des Adaptionstatecharts in der Anzahl der instanzifizierbaren Ports parametrisiert. Die Invarianten der Zustände Convoy, CreatePort und SendUpdates werden auf Werte $30x$ bzw. $30x - 1$ gesetzt, wobei x die Anzahl der Ports bezeichnet. Der Wert 30 ergibt sich aus der Länge eines Durchlaufs durch das Statechart. Die Erreichbarkeitsanalyse wurde mit und ohne Zeit durchgeführt. Die Ergebnisse werden in Abbildung 6.26 dargestellt.

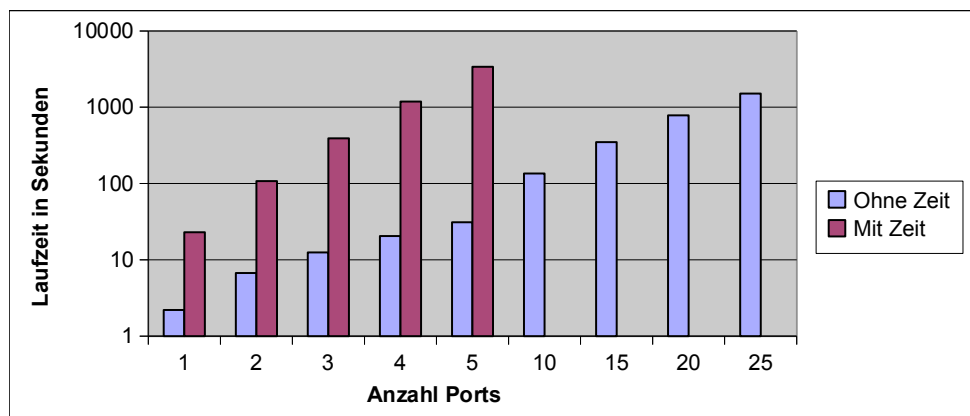


Abbildung 6.26: Laufzeit der Erreichbarkeitsanalyse

Erwartungskonform zeigen die Ergebnisse der Messung einen starken Anstieg durch Betrachtung von Zeit. Für drei Zustände werden unter Berücksichtigung von Zeit über eine Stunde benötigt. Ohne Zeit lassen sich problemlos 25 Ports expandieren. Bei der Verwendung von Zeit erwies sich das UDBM-Binding als ausschlaggebend für die lange Laufzeit. Mit zunehmender Laufzeit

dauerten die einzelnen Berechnungsschritte zunehmend länger und ca. 95% der Rechenzeit entfiel auf das UDBM-Binding. Aktuelle weiterführende Arbeiten der Erreichbarkeitsanalyse zur Nutzung für ein Model Checking zeigen einen erheblichen Performancegewinn (von 20 Min. für vier Ports auf unter eine Sekunde) durch Auslagerung des UDBM-Bindings in Java-Quellcode statt der Skriptsprache Ruby aus [HSJZ10].

Weiterhin ergaben die Messungen, dass bei der Erreichbarkeitsanalyse ohne Zeit ca. 95% der Laufzeit allein auf das Kopieren der Graphen entfiel. Auf das Finden der Matchings, die Anwendung der Regeln und die Überprüfung von isomorphen Zuständen entfielen die restlichen 5%. In der Erreichbarkeitsanalyse mit Zeit sank der Anteil für das Kopieren der Graphen auf ca. 15% bei 2 Ports und auf 5% bei 4 Ports. Dies ist zu begründen mit einem stärkeren Einfluss des UDBM-Bindings auf die Laufzeit.

Abbildung 6.28 zeigt die Entwicklung der Anzahl der Zustände im erreichten Transitionssystem. Der Anstieg bei Berücksichtigung von Zeit fällt dabei wesentlich stärker aus als ohne Zeit. Dies liegt daran, dass mit Zeit auch die Delay-Kanten expandiert werden müssen, die ohne Berücksichtigung von Zeit nicht weiter betrachtet werden. Die Anzahl der Objekte in jedem Zustand stieg jedoch nur langsam an. Dies ist auf die gewählte Abbildung der Statecharts auf Graphen zurückzuführen, die für eine neue Instanz eines Statecharts nur ein neues ActiveState-Objekt anlegt.

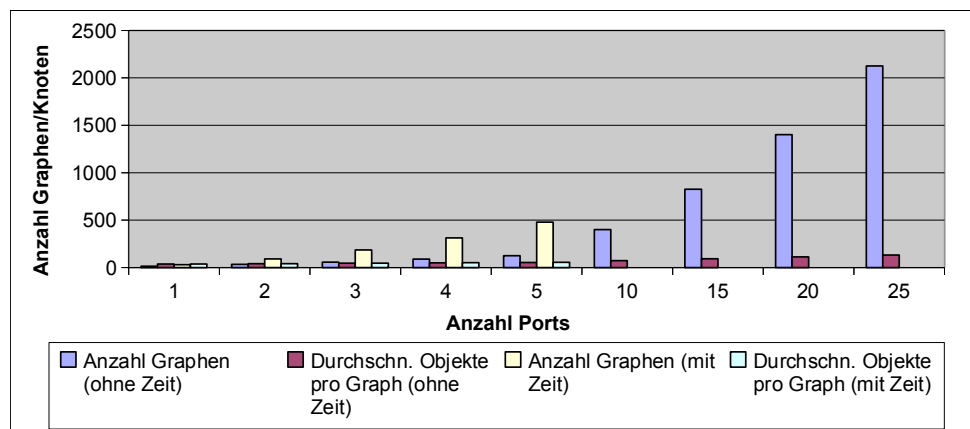


Abbildung 6.27: Anzahl expandierter Graphen und maximale Anzahl der Knoten

Abbildung 6.28 zeigt schließlich die Laufzeit des Verfeinerungsalgorithmus mit Zeit. Das Ergebnis zeigt, dass die Laufzeit trotz relativ kleiner Anzahl und Größe der Graphen relativ hoch ist. Da die Überprüfung sehr viele Zugriffe auf die UDBM-Bibliothek ausführt, um die oberen Schranken der Clocks zu erhalten, ist diese hohe Laufzeit, wie bei der Erreichbarkeitsanalyse, auf die hohe Laufzeit der UDBM-Bibliothek zurückzuführen.

Insgesamt lässt sich folgern, dass die Implementierung zum Zeitpunkt der Erstellung dieser Arbeit lediglich die prinzipielle Machbarkeit der Verfeinerung gezeigt hat. Eine Optimierung der Implementierung ist notwendig, um größere Anzahlen an Portinstanzen betrachten zu können.

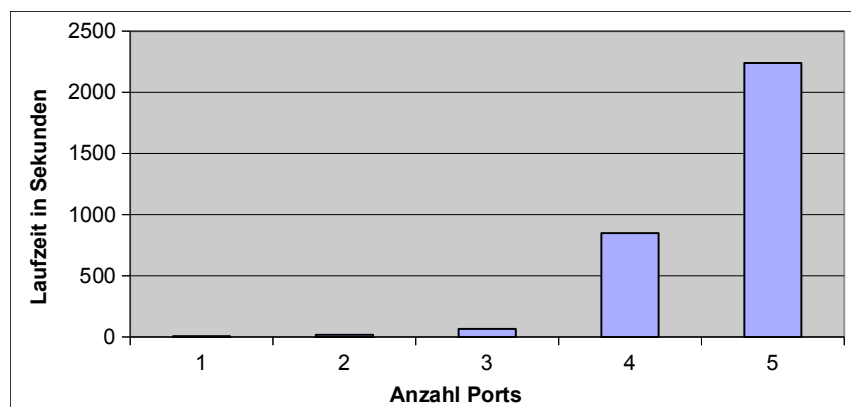


Abbildung 6.28: Laufzeit Verfeinerung

Die möglichen Stellen die Optimierungspotential aufweisen, wurden aufgezeigt. In aktuellen, weiterführenden Arbeiten wurde bereits eine erhebliche Verbesserung der Laufzeit (Reduzierung um ca. 80%) erreicht [HSE10].

6.3.1.3 Integration Altkomponenten

Die Integration von Altkomponenten wurde in der *FRiTS^{Cab}* Tool Suite vollständig umgesetzt. Die Validierung wurde, soweit möglich, auf Basis von Altkomponenten aus dem RailCab Projekt durchgeführt, die zum Teil angepasst wurden, um die verschiedenen Szenarien der Integration durchspielen zu können. Dies sollte allerdings die Qualität der Aussage, dass die Integrationsverfahren anwendbar sind, nicht einschränken (siehe hierzu auch Abschnitt 6.3.2). Wir werden im Folgenden zuerst die Schnittstelle der drei verschiedenen Verfahren erläutern und anschließend auf Evaluierungsergebnisse eingehen.

Abbildung 6.29 zeigt den Aufruf der Analyseverfahren. Die Auswahl der Analyseverfahren stehen dem Benutzer nach Selektion der Port(s) der Altkomponente und Kontext zur Verfügung.

Wird das White Box Checking ausgewählt, so kann der Benutzer anschließend eine Reihe von Parametern für die Verifikation einstellen (siehe Abbildung 6.30). Zum einen ist es möglich unterschiedliche Model Checker zu verwenden, wobei, wie in Abschnitt 4.3 beschrieben, der CBMC Model Checker die meisten benötigten Konstrukte unterstützt und damit auch die erste Wahl für eine Verifikation ist. Zudem können die Parameter des Kommunikationskanals und der Ausführungsumgebung der Komponenten eingestellt werden. Z.B. ist die Pufferkapazität oder die Länge der Ausführungsperiode einstellbar.

Wird das White Box Checking auf dem gezeigten Anwendungsbeispiel durchgeführt, so wird ein Fehler erkannt (siehe Gegenbeispiel aus Abbildung 6.31). Gegenbeispiele können im Kontextverhalten durchgespielt werden, um den Benutzer direkt zu der möglichen Fehlerursache zu führen. In unserem Fall wird ein Deadlock erkannt, da die Altkomponente eine LEAVE_CONVOY-

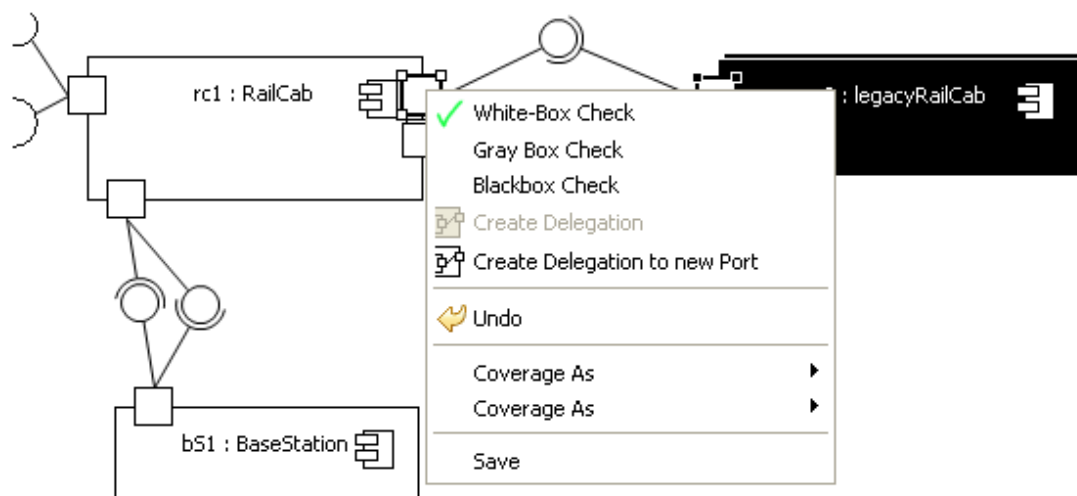


Abbildung 6.29: Legacy Checking

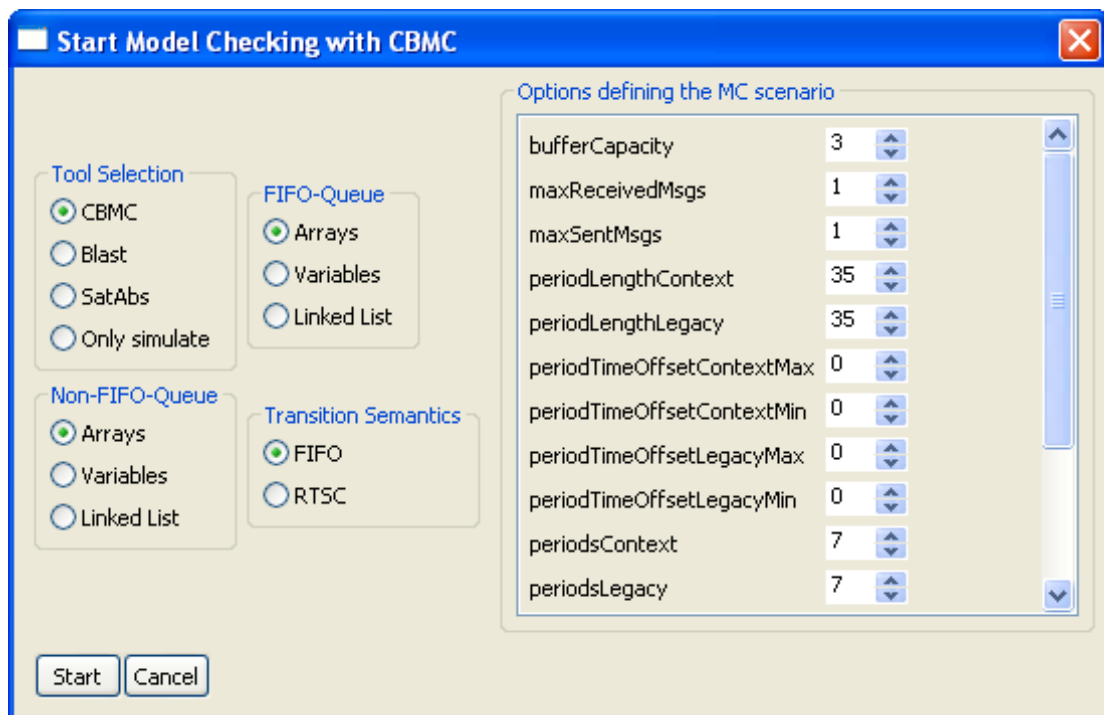


Abbildung 6.30: Parameter White Box Checking

Nachricht verschickt während der aktive Zustand des Kontexts `waitNoConvoy` ist, der diese Nachricht verarbeiten kann.

Das Black Box Checking und Gray Box Checking haben ebenfalls diesen Fehler erkannt. Im Fall des Black Box Checking kann zudem noch das gesamte Verhalten der Altkomponente erlernt werden (siehe Abbildung 6.32). Hieraus wird ersichtlich, dass die Altkomponente unabhängig von der Entscheidung des Kontextverhaltens den Konvoi auflöst. Dies ist durchaus eine mögliche Interpretation für die rear-Rolle, die allerdings in unserem Fall durch die front-Rolle nicht berücksichtigt wurde. Eine Möglichkeit, um die Integration erfolgreich zu gestalten, ist eine Anpassung des Kontextverhaltens in der Form, dass ein Auflösen des Konvois zu jeder Zeit durch die rear-Rolle ohne zusätzliche Bestätigung ermöglicht wird. Dies wurde ebenfalls durch alle drei Integrationsverfahren bestätigt.

Ein weiterer Aspekt der Integration von Altkomponenten ist die Betrachtung von Reglerverhalten durch die Anbindung von Systemidentifikationsverfahren (siehe Abschnitt 4.4). Die Werkzeugunterstützung ermöglicht dabei das Hinzufügen von Experimenten zur Erkennung von Reglerverhalten zu einer Altkomponente (siehe Abbildung 6.33). Das heißt, diese Experimente werden nicht aus dem Werkzeug selbst heraus gestartet, sondern aus Spezialwerkzeugen zur Identifikation von kontinuierlichen Verhalten, wie z.B. Matlab mit entsprechenden Erweiterungen. Es wird dabei davon ausgegangen, dass die Experimentdaten in einer Datei gespeichert werden, so dass zu jeder Datei ein kontinuierlicher Port angelegt wird. Auf diesem kann eine Identifikation des Reglerverhaltens durchgeführt werden (siehe Abbildung 6.34). Das erkannte Verhalten wird dann dem kontinuierlichen Port hinterlegt und kann dann im Weiteren durch einen Experten einem speziellen Regler zugewiesen werden. In unserem Fall ist dies ein Drehzahlregler und ein Drehmomentregler (siehe Abbildung 6.35).

Das erlernte Zustandsverhalten sowie das für einen Zustand spezifische Reglerverhalten kann anschließend manuell zu einem Gesamtverhalten zusammengeführt werden. Abbildung 6.36 zeigt den erlernten Automaten der Altkomponente angereichert mit den Reglerkonfigurationen.

Evaluierungsergebnisse Im Folgenden betrachten wir zuerst die Evaluierungsergebnisse der Systemidentifikation, dann die des Black Box , White Box und Gray Box Checking.

Die Systemidentifikation benötigt für das Erkennen des Drehmoment-Reglers ca. 20 Sekunden und ca. 150 MB Speicher und für den Drehzahlregler ca. 8 Sekunden und ebenfalls ca. 150 MB Speicher. Für ausführlichere Evaluierungsergebnisse zur Systemidentifikation sei auf die referenzierte Literatur aus Abschnitt 4.4 verwiesen.

Ohne Berücksichtigung der Periode benötigt das Black Box Checking für das betrachtete Evaluierungsbeispiel weniger als eine halbe Minute, dabei werden über 1000 Zugehörigkeitsanfragen gestellt. Unsere Optimierung der Anzahl der Anfragen führt dabei zu über 900 Cache Hits und über 30.000 Präfix Cache Hits (siehe Abschnitt 4.2.2). Die Laufzeit erhöht sich unter Berücksichtigung der Periode um den Faktor Anzahl der Zugehörigkeitsanfragen mal Größe der Periode. Bei einer Periode von 1 Sekunde würde sich entsprechend die Laufzeit um 1000 Sekunden erhöhen. Dies gilt gleichermaßen für die anderen Analyseverfahren. Zusätzlich ist die Laufzeit

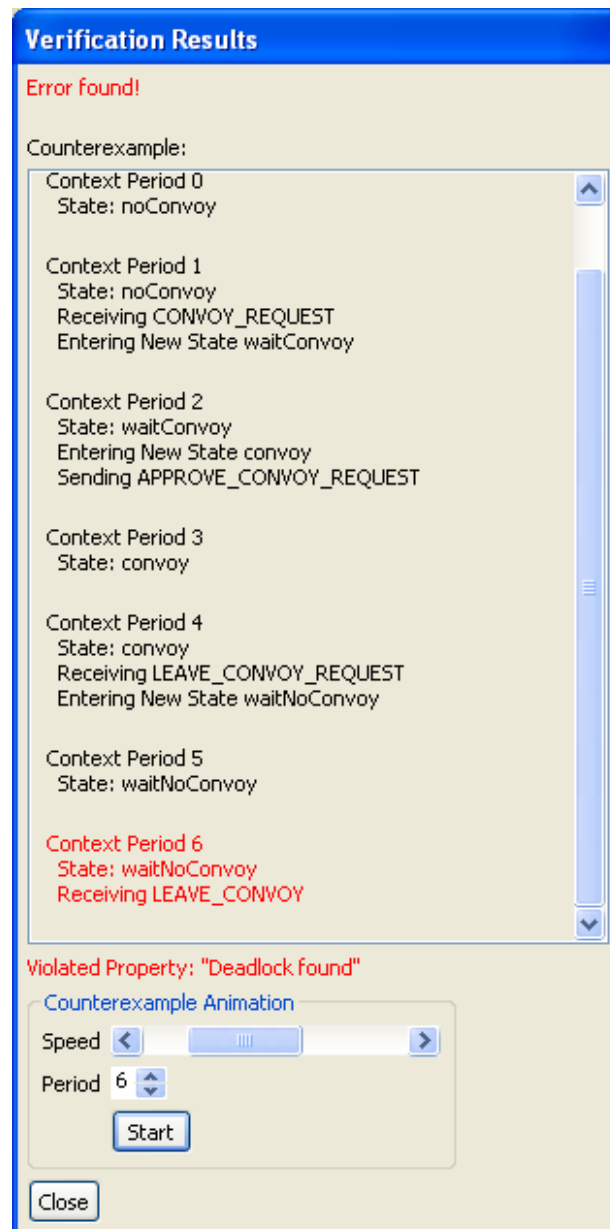


Abbildung 6.31: Gegenbeispiel White Box Checking

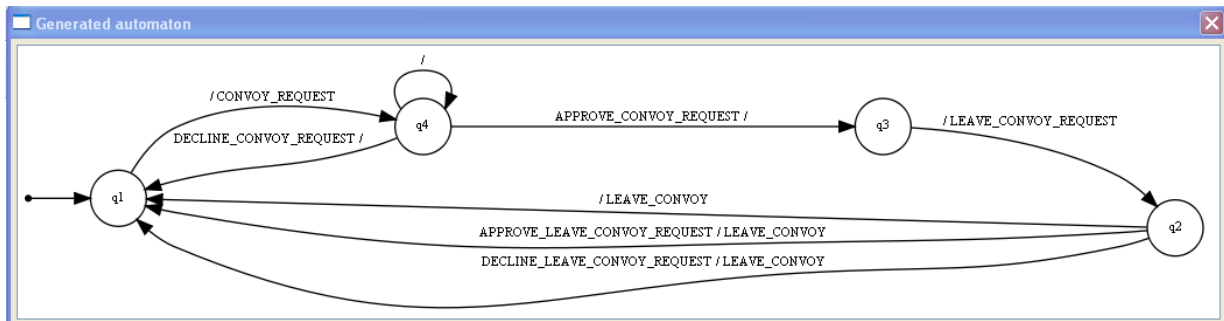


Abbildung 6.32: Erlerner Automat der Altkomponente

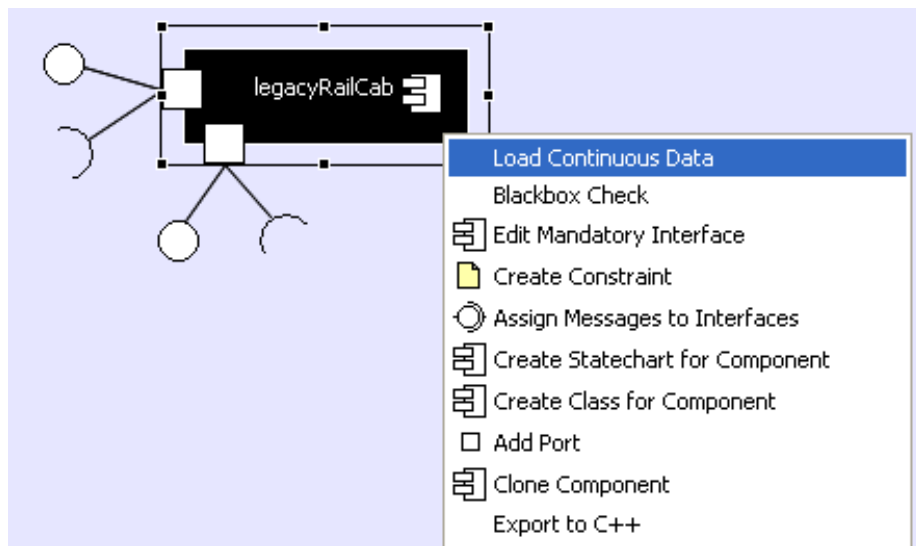


Abbildung 6.33: Altkomponente: laden der (kontinuierlichen) Daten

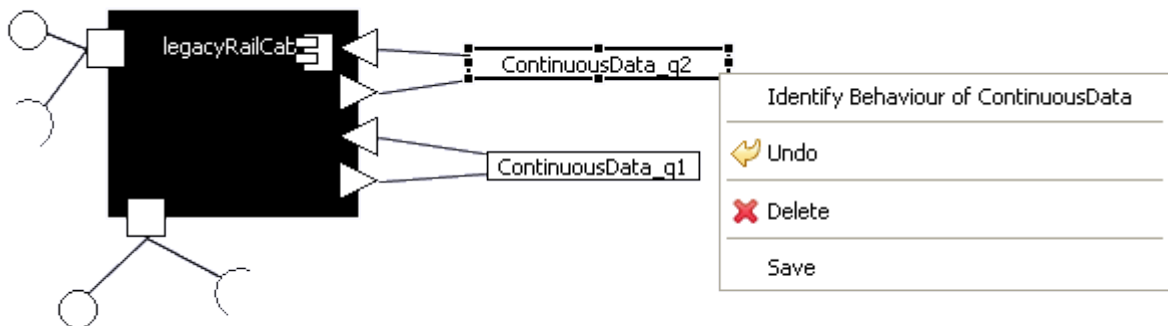


Abbildung 6.34: Starte Systemidentifikation

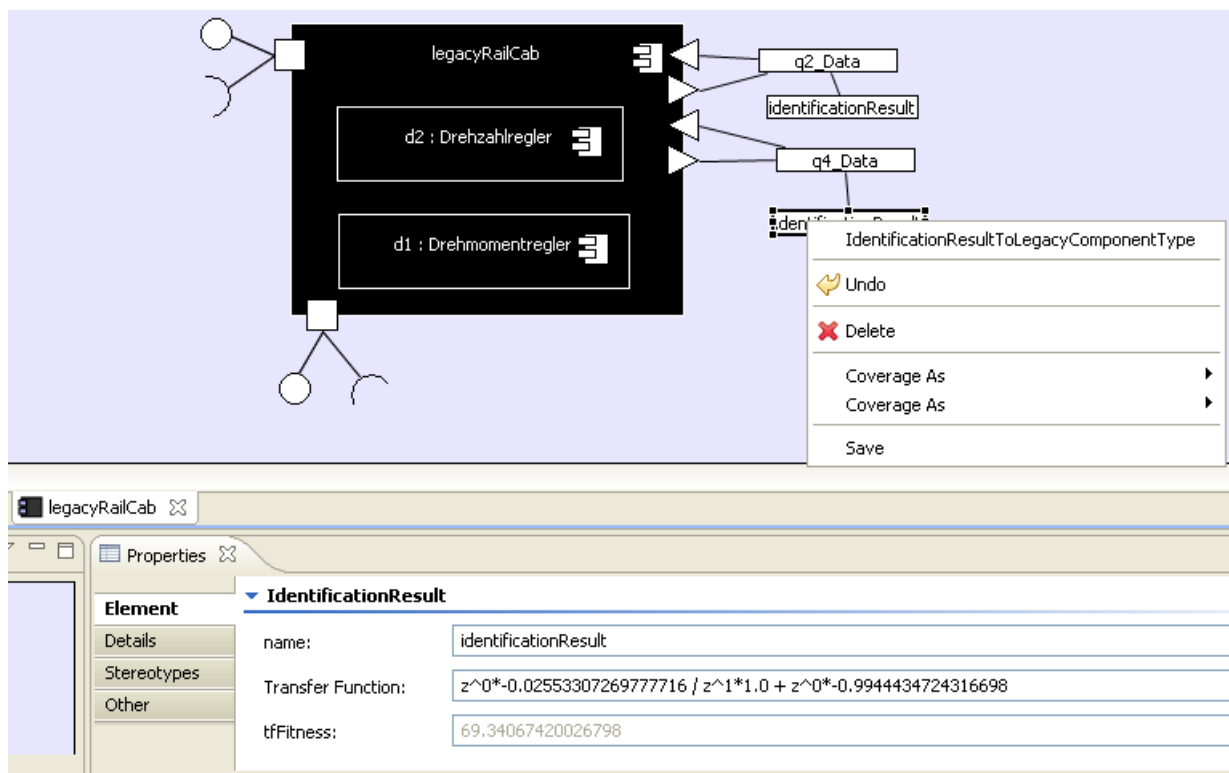


Abbildung 6.35: Erkannte Regler / Transferfunktion

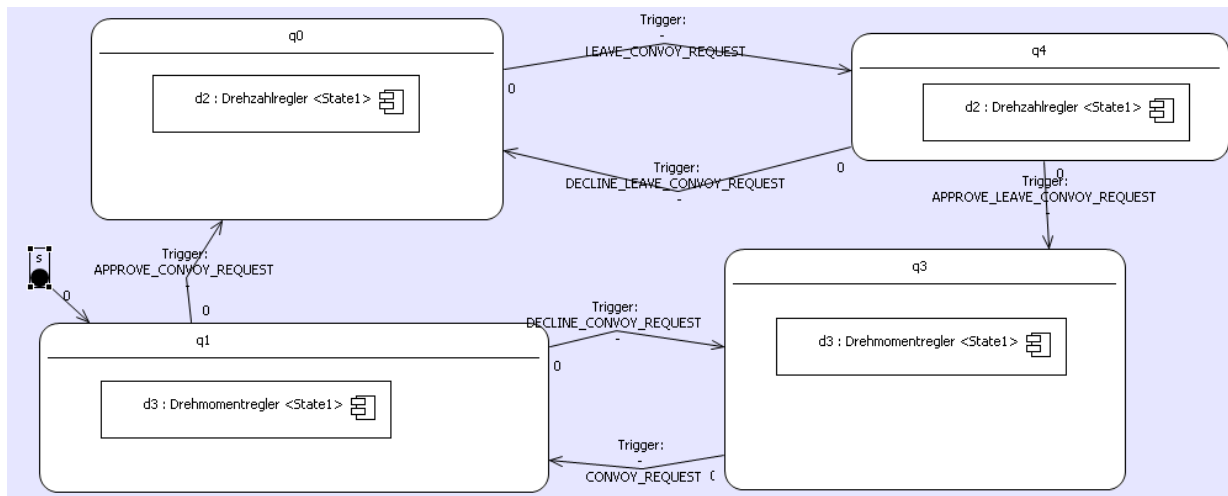


Abbildung 6.36: Erlernter Automat der Altkomponente mit Reglerkonfigurationen

im Fall des Black Box Checking maßgeblich von der Anzahl der zu erkennenden Zustände der Altkomponente abhängig. In unserem Beispiel wird bei genauer Angabe der zu erlernenden Zustände die oben genannte Laufzeit benötigt. Sollen stattdessen zehn Zustände erkannt werden beträgt die Laufzeit ca. 90 Sekunden. Interessanterweise wurden bei künstlichen Beispielen problemlos 15 Zustände innerhalb von 25 Minuten erkannt, wenn die obere Grenze der Zustände nur geringfügig von der tatsächlichen Anzahl abweicht. Zum Teil liegt dies an der quadratischen Abhängigkeit der Anzahl der zu erlernenden Zustände auf die Gesamtlaufzeit des Vasileskii und Chow Algorithmus, der für die Äquivalenzanfrage implementiert wurde (siehe Abschnitt 4.2.2). Wie die gezeigten Zahlen aber schon andeuten ist die Anzahl der benötigten Äquivalenzanfragen sehr groß, gleich wohl die Optimierungen schon viele unnötige ausschließen. Dies lässt folgern, dass eine optimistische Abschätzung der Obergrenze der Zustände verfolgt werden sollte, gleich wohl dann das Verfahren mehrfach angestoßen werden muss, bis der erlernte Automat sich nicht mehr verändert, gegenüber dem vorherigen Durchlauf. Eine wesentliche Laufzeitverbesserung kann durch den Einsatz eines Model Checkers erreicht werden, da hierdurch auf die Äquivalenzanfrage verzichtet werden kann.

Die folgende Abbildung 6.37 zeigt zusammenfassend die Evaluierungsergebnisse. Es wurden 5, 7 und 10 Zustände mit und ohne Model Checking betrachtet. Durch das Einbeziehen von Model Checking kann die Laufzeit um ca. 50 % reduziert werden. Der Speicherverbrauch konnte ebenfalls reduziert werden. Da durch das Model Checking die Anzahl der Anfragen reduziert wird, sinkt gleichermaßen die Anzahl an (Prefix) Cache Hits. Der extreme Anstieg der Prefix Cache Hits lässt sich mit der höher angesetzten Anzahl an zu erlernenden Zuständen erklären.

Im Fall des White Box Checking ist die Laufzeit neben der Periodenlänge stark von der Anzahl und Größe der verwendeten Puffer und Zeitvariablen abhängig. Wie in Abschnitt 4.3 diskutiert, haben wir das CBMC-Werkzeug verwendet, da dieses im Vergleich zu den anderen tatsächlich Puffer in der Umsetzung unterstützt. Während der Speicherverbrauch für einen Ein- und Aus-

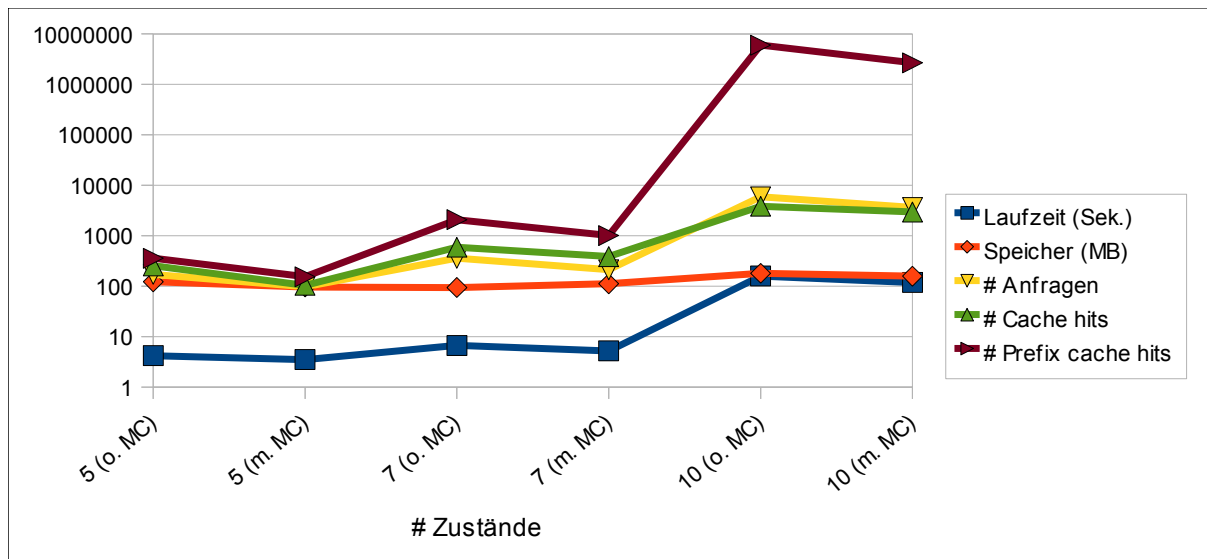


Abbildung 6.37: Evaluierungsergebnisse Black Box Checking

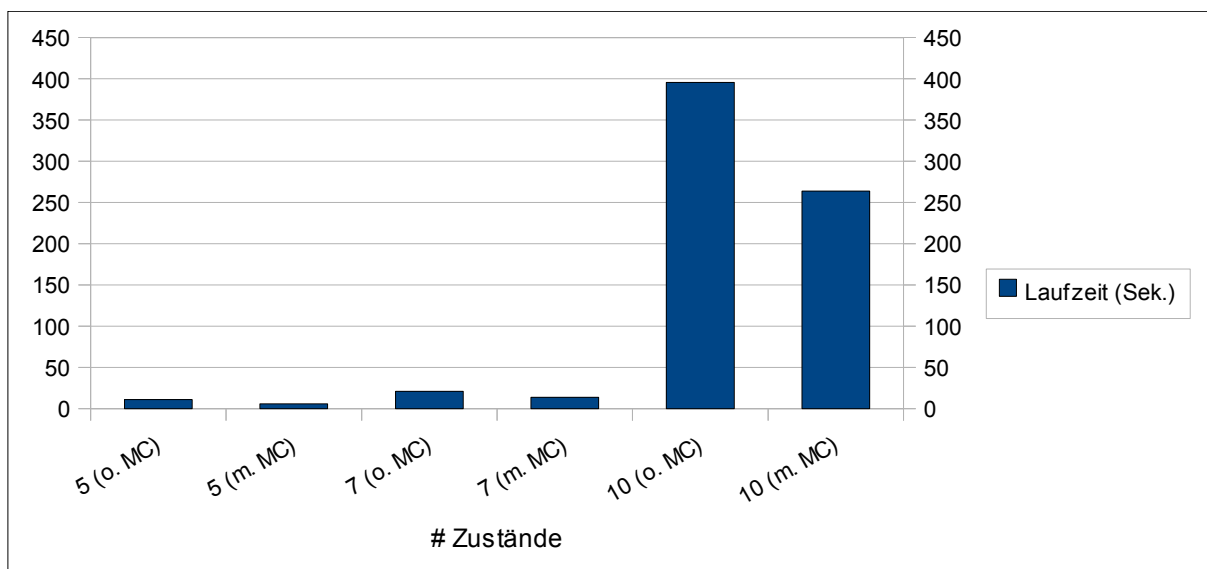


Abbildung 6.38: Laufzeiten Black Box Checking mit einer Periode von 400 ms

gangspuffer der Größe 2, 3 und 6 in etwa bei 300 MB beträgt, unterscheiden sich die Laufzeiten stark. Für 2 sind es 75 Sekunden, für 3 90 Sekunden und für 6 180 Sekunden. Noch stärker sind die Auswirkungen, wenn eine variable Periodenlänge und eine variable Länge des Send- und Empfangsintervalls berücksichtigt wird. Die Laufzeit erhöht sich für den Fall mit Pufferlänge 3 auf ca. 25 Minuten. Insgesamt lassen sich allerdings schwer Evaluierungsergebnisse erheben oder reproduzieren, da das (die) Werkzeug(e) sehr unzuverlässig die betrachteten Anwendungen analysieren. Ein Grund hierfür könnte die eher untypische Anwendungs-kategorie sein, die Parallelität und Zeit benötigt, und daher Konstrukte verwendet die eher selten in den gezeigten Evaluierungen der Werkzeuge zum Einsatz kommen.

Eine vielversprechende Stellschraube zur Verbesserung der Laufzeit der Integrationsverfahren ist die Optimierung der erkannten Zustände pro Iteration und damit eine Minimierung der Anzahl der Iterationen. Grund für diese Annahme ist, dass der größte Ressourcenaufwand in der Überprüfung der Äquivalenz liegt. Eine weitere Beobachtung ist, dass die potentiell erkannten Zustände maßgeblich von der Länge des Gegenbeispiels abhängig sind. Je länger ein Gegenbeispiel, desto mehr Zustände können erkannt werden. Die meisten Model Checker, wie auch UPPAAL, verfügen über eine Einstellungsmöglichkeit, um lange Gegenbeispiele zu bevorzugen. Für das Gray Box Checking können wir darüber hinaus noch weitere Optimierungen an dem zu überprüfenden Modell vornehmen, die wir im Folgenden diskutieren. Anschließend stellen wir die Evaluierungsergebnisse des Gray Box Checking unter Betrachtung der diskutierten Optimierungen vor.

1.) Festlegung eines minimalen Anteils an unbekanntem Verhalten für Gegenbeispiele Eine Möglichkeit der Minimierung des unbekanntem Verhalten für Gegenbeispiele besteht in der Verwendung von Transitionen der chaotischen Hülle (also bei bisher unbekanntem Verhalten), indem das Modell und die UPPAAL-Query so modifiziert werden, dass möglichst viele dieser Transitionen verwendet werden. Im Prinzip genügt es auch, die Zählvariable bei Transitionen zu s_V (einschließlich Selbsttransitionen) zu erhöhen⁵.

Lediglich eine minimale Anzahl zu fordern, reicht jedoch alleine nicht aus, da so möglicherweise Deadlocks unerkannt bleiben könnten, die nur unter Verwendung von weniger Transitionen zu erreichen sind. Durch Benutzen der UPPAAL Kommandozeilenoption `-t2` lässt sich allerdings erreichen, dass möglichst das Gegenbeispiel mit der geringsten verwendeten Zeit ermittelt wird. Ersetzt man nun die bisherige Query-Formel durch eine Variante, die dem Model Checker eine „Zeitstrafe“ gibt, wenn zu wenige Transitionen zu s_V genutzt werden, dann kann ein Minimum für neues Verhalten festgelegt werden, das in Gegenbeispielen enthalten sein soll. UPPAAL versucht dann, die Anzahl dieser Transitionen der chaotischen Hülle bis zur angegebenen Schranke zu maximieren. Kann der festgelegte Wert nicht erreicht werden, dann kann UPPAAL dennoch ein Gegenbeispiel ermitteln, sofern eines existiert. Für eine Clock t , eine Integer-Variable x und ein gewünschtes Minimum M muss dazu unsere Query

⁵Dies reduziert die notwendigen Änderungen am Modell deutlich, hat allerdings kaum Auswirkungen auf die Länge des Gegenbeispiels.

der Form $A[] \text{ not } p$ mit $p = (\text{ilegacyComponent3.sDelta or deadlock})$ durch $A[] \text{ not } (t > M - x \text{ and } p)$ ersetzt werden.

Problematisch bei diesem Ansatz sind allerdings Kreise im Verhaltensmodell der Altkomponente, sofern diese im Zusammenspiel mit dem Kontext beliebig oft durchlaufen werden können. UPPAAL tendiert dazu, diese dazu zu verwenden, die Zählvariable zu erhöhen, ohne das Modell weiter durchsuchen zu müssen (was wünschenswert wäre). Sämtliche Schleifeniterationen nach der ersten sind für die Verhaltenssynthese uninteressant, kosten jedoch beim Testen Zeit. Daher sollte die Minimalanzahl für neue Interaktionen nicht zu groß gewählt werden, zumal Schleifen in reaktiven Systemen zwangsläufig vorkommen.

2.) Maximierung der Anzahl der unterschiedlichen Transitionen zu s_V im Gegenbeispiel Um nutzlose Schleifendurchläufe zu vermeiden, kann auch die Anzahl der unterschiedlichen Transitionen zu s_V im Gegenbeispiel maximiert werden, statt für die Gesamtdurchläufe durch diese Transitionen einen Mindestwert festzulegen. Dies kann erreicht werden, indem an Stelle einer einzelnen Booleschen-Variablen je eine für jede dieser Transitionen definiert wird. Diese werden mit 0 initialisiert und bei Verwendung der jeweiligen Transition auf 1 gesetzt. Analog zum letzten Ansatz kann als Query $A[] \text{ not } (t > M - x_1 - x_2 \dots - x_n \text{ and } p)$ verwendet werden. Die Variablen x_1 bis x_n sind die erwähnten Markierungsvariablen für die n Transitionen zu s_V . Als M sollte hier mindestens n gewählt werden.

Ein Nachteil dieses Ansatzes ist allerdings, dass eine einzelne der s_V -Transitionen mehreren tatsächlichen Transitionen der Altkomponente entsprechen kann. Damit würden die Gegenbeispiele bei kleinem IO-Alphabet, abhängig von Altkomponente und Kontext, möglicherweise kleiner ausfallen als bei dem anderen Ansatz. Bei dem hier vorgestellten Beispiel ist das allerdings nicht der Fall, da jede Nachricht nur einmal im Statechart der Altkomponente vorkommt.

3.) Maximierung der Überdeckung von Transitionen des Kontextes Eine weitere Alternative ist die Maximierung der Anzahl der im Gegenbeispiel verwendeten Transitionen des Kontextes. Im Prinzip kann dafür analog zum vorhergehenden Ansatz vorgegangen werden, mit dem Unterschied, dass die Markierungsvariablen für andere Transitionen verwendet werden. Vorteil hierbei ist, dass nicht lediglich jede Nachricht einmal verwendet wird. Allerdings wird hier Verhalten der Altkomponente, das erst durch mehrere Schleifendurchläufe innerhalb des Kontextes erreicht wird, nicht sofort erreicht. Zudem wird das bereits synthetisierte Verhalten nicht beachtet.

Anmerkungen All diese Optimierungen können das Erkennen von Gegenbeispielen im bereits synthetisierten Teil des Modells verzögern, da der Model Checker zunächst in der chaotischen Hülle die „Zeitstrafe“ abbauen wird. Ein Lösungsansatz ist, statt die Query zu verändern, die entsprechende Zeitbedingung für alle Transitionen, die zum Deadlock-Zustand führen, zu verwenden. Allerdings kann dadurch das Modell deutlich größer werden.

Alternativ kann auch die Bedingung, dass kein echter Deadlock erreicht wird, aus p herausgezogen werden und auf Ebene der Konjunktion gestellt werden, sodass die Zeitbedingung für echte Deadlocks nicht gilt. Allerdings müsste dazu zusätzlich der explizite Deadlock-Zustand so angepasst werden, dass er alle Eingaben akzeptiert, damit kein „falscher“ Deadlock des Kontextes entsteht⁶.

Die einzelnen Lösungsvorschläge sind jeweils nicht für jede Situation optimal und sollten, möglicherweise in Kombination, auf das jeweilige Einsatzszenario (also die Altkomponente und den Kontext) abgestimmt verwendet werden.

Generell ist das Optimierungspotential eingeschränkt, wenn ϵ -Transitionen (die ein Zeitvergehen simulieren) zugelassen werden: Problematisch ist, dass diese zunächst für jeden Zustand angenommen werden müssen und nicht durch den Kontext eingeschränkt werden. Selbst nach einer ansonsten vollständigen Synthese müssen alle Zustände, für die weder bereits eine Transition mit ϵ -Eingabe vorliegt noch eine ausgeschlossen wurde (durch Beenden des Testlaufs in diesem Zustand), erneut besucht werden, nur um abzuwarten, ob innerhalb des maximalen Zeitrahmens in einen anderen Zustand geschaltet wird.

Evaluierung zu den Verbesserungsvorschlägen Die vorgeschlagenen Ansätze zur Gewinnung längerer Gegenbeispiele wurden jeweils evaluiert, um ihren Einfluss auf das Verfahren untersuchen und vergleichen zu können. Da die Laufzeit der Simulation hier wenig über den Testaufwand im realen System aussagt, wurde als Maßstab zum Vergleich der Varianten die Anzahl an Einzelschritten und Iterationen bis zum Ende des Syntheseverfahrens erhoben. Der Speicherverbrauch wurde hier ebenfalls vernachlässigt, da sich dieser im Wesentlichen aus der Größe des Zustandsraums ergibt, der sich durch die Optimierung in den Beispielen kaum unterscheidet (verbrauchter Speicher liegt bei ca. 60 MB).

Die Evaluierung wurde sowohl mit dem intakten als auch dem fehlerhaften Modell durchgeführt. Da, wie oben beschrieben, die Periodenzeit (ϵ) einen hohen Einfluss auf den Testaufwand hat, wurde das Verfahren jeweils einmal mit und ohne ϵ durchgeführt.

Tabelle 6.1 zeigt die Ergebnisse der Simulationsdurchläufe für eine erfolgreiche Integration der Altkomponente für die verschiedenen Verfahren der Gray-Box-Integration, Tabelle 6.2 zeigt die entsprechenden Ergebnisse für eine fehlerhafte Integration. Die Resultate können wie folgt interpretiert werden.

Die Maximierung von Transitionen zu s_V im Gegenbeispiel bis zu 5 („1. Vorschlag“) liefert bei Verwendung mit ϵ -Transitionen in der chaotischen Hülle eine Verschlechterung. Dies kann in einer ungünstigen Wahl der Grenze, bis zu der maximiert werden soll, begründet sein. Die entsprechenden Gegenbeispiele zeigten jedoch, dass Kreise im Automaten unnötig oft durchlaufen wurden. Dennoch ergibt sich ohne Zulassen von ϵ -Transitionen eine Verbesserung gegenüber der normalen Anfrage an den UPPAAL Model Checker.

⁶Der Deadlock-Zustand würde dadurch zweckentfremdet und zu einer Art zweitem s_V -Zustand, bis die Zeitbedingung in der Query erfüllt ist.

Die Ergebnisse bei Maximierung der Anzahl verschiedener Transitionen zu s_V im Gegenbeispiel („2. Vorschlag“) und die bei Maximierung der Transitionsüberdeckung im Kontext („3. Vorschlag“) ähneln einander. Diese Ansätze liefern für die erfolgreiche Integration auch mit ϵ -Transitionen bessere Resultate, ohne können sie das komplette Verhalten sogar in einem einzigen Durchlauf synthetisieren. Bei Anwendung auf die fehlerhafte Integration sind die Ergebnisse ohne Verwendung von ϵ -Transitionen deutlich besser als bei den anderen beiden Varianten. Werden diese zugelassen, dann sind die Laufzeiten schlechter als bei normaler Anfrage an den Model Checker. Hier sind eventuell Verbesserungen möglich, wenn die Vorschläge aus den vorherigen Anmerkungen umgesetzt werden, mit denen zunächst echte Deadlocks erkannt werden könnten.

Tabelle 6.1: Ergebnisse der Evaluierung der Verbesserungsvorschläge für die korrekte Integration (Iterationen / Einzelschritte).

	normale Anfrage	1. Vorschlag	2. Vorschlag	3. Vorschlag
mit ϵ-Transitionen	10 / 62	9 / 81	6 / 46	6 / 54
ohne ϵ-Transitionen	10 / 60	4 / 40	1 / 17	1 / 16

Tabelle 6.2: Ergebnisse der Evaluierung der Verbesserungsvorschläge für die fehlerhafte Integration (Iterationen / Einzelschritte).

	normale Anfrage	1. Vorschlag	2. Vorschlag	3. Vorschlag
mit ϵ-Transitionen	7 / 31	7 / 41	7 / 39	7 / 49
ohne ϵ-Transitionen	7 / 29	4 / 24	3 / 18	3 / 22

6.3.1.4 Synthese Komponentenverhalten

Das Ziel der Syntheseumsetzung ist es die in Kapitel 5 vorgestellten Konzepte zu implementieren, so dass eine Synthese des Komponentenverhaltens direkt aus Fujaba RT angestoßen werden kann. Dieses Ziel wurde nur zum Teil erreicht. Grund hierfür ist, dass die initiale Implementierung auf einem diskreten Zeitmodell nach [Bey02] aufbaut. Diese Implementierung ist zwar aus Fujaba RT aufrufbar, skaliert allerdings nicht und wurde auch nicht vollständig auf das in Kapitel 5 vorgestellte kontinuierliche Zeitmodell umgestellt. Wir können daher die Rollenverhalten `rear` und `registree` des Registration- und DistanceCoordination-Musters, über die eine Kompositionsregel gilt (siehe Abbildung 6.20), nicht in dem vorgestellten Umfang betrachten, da diese nach der diskreten Zeitsemantik zu keinem Ergebnis kommt. Dies liegt an der zu hohen Speicherlast und Laufzeit des Ansatzes mit diskreter Zeitsemantik. Wir werden daher im Folgenden zuerst zwei angepasste Verhalten, den synthetisierten Automaten sowie ein Laufzeitvergleich der diskreten und kontinuierlichen Zeitsemantik betrachten.

Die vereinfachte `rear`- und `registree`-Rolle spezifizieren im Vergleich zu den in Kapitel 5 auf Seite 141 betrachteten Verhalten Uhren mit minimalen Werten (siehe Abbildung 6.39 und 6.40), um den Zustandsraum durch die diskrete Zeitsemantik möglichst klein zu halten. Die Reduzierung

des Zustandsraums ist hierdurch extrem, da nach der diskreten Zeitsemantik jeder Integerwert ein zusätzlicher Zustand ist. Dies macht sich besonders während der notwendigen Produktautomatenbildung für die Synthese bemerkbar. Um den Unterschied zu den bisherigen Automaten der gezeigten Validierung zu verdeutlichen, werden wir die Nachrichten in den Automaten der Synthese klein schreiben.

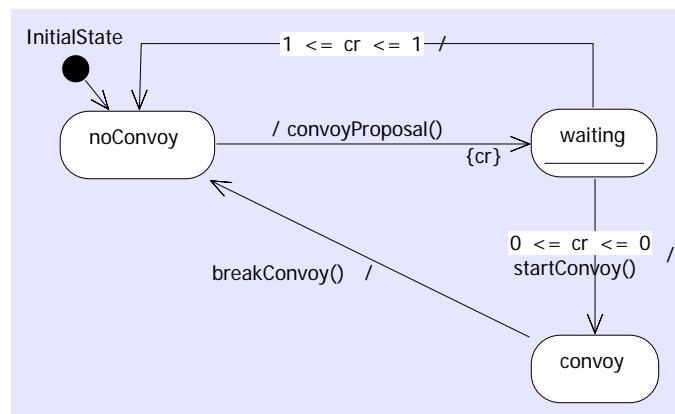


Abbildung 6.39: Vereinfachte rear-Rolle

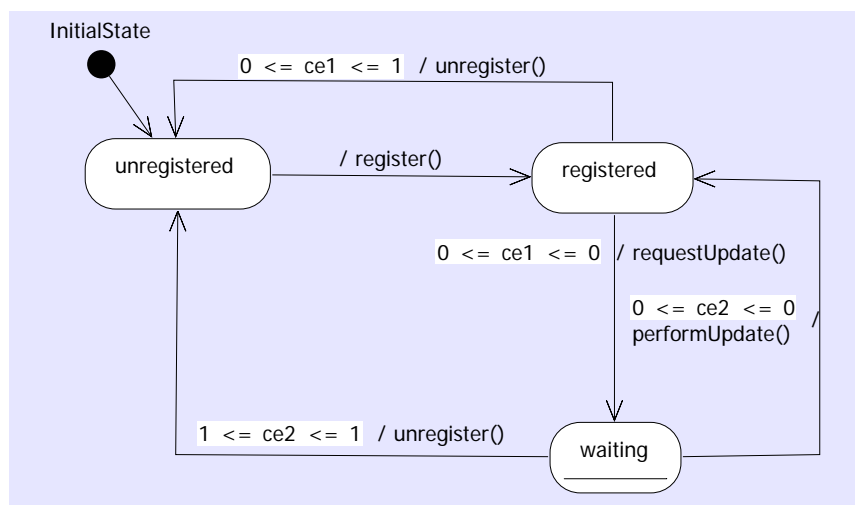


Abbildung 6.40: Vereinfachte registree-Rolle

Das Ergebnis der Synthese ist das rollenkonforme REAL-TIME STATECHART aus Abbildung 6.41. Alle Zustände der registree (Zustände (registree.unregistered,. . .), (registree.registered,. . .) und (registree.waiting,. . .)) werden zu einem Zustand der rear-Rolle zusammengefasst. Der Zustand (registree.unregistered,rear.convoy) wurde durch anwenden der Kompositionsregel entfernt. Die Verfeinerungsbeziehung der Rollenverhalten wurde hierdurch jedoch nicht verletzt.

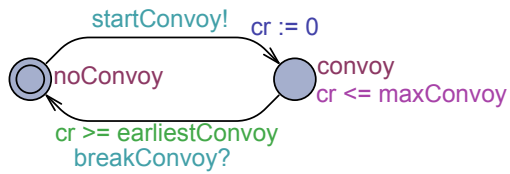


Abbildung 6.42: Parametrisierte rear-Rolle

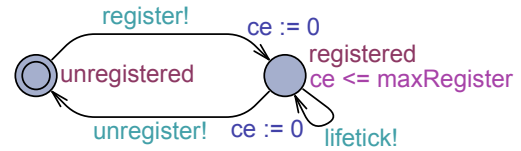


Abbildung 6.43: Parametrisierte registration-Rolle

(earliestConvoy,maxConvoy,maxRegistered)	Ansatz	States	Transitions	Time (ms)
(0,1,1)	FIS	16	100	5
	ZG	16	38	21
(1,2,2)	FIS	33	226	15
	ZG	44	102	68
(4,5,10)	FIS	200	1600	172
	ZG	67	153	104
(1,5,10)	FIS	245	2102	287
	ZG	122	283	344
(9,10,20)	FIS	645	5397	1917
	ZG	195	440	708
(2,10,20)	FIS	848	7683	4208
	ZG	122	283	344
(3,15,20)	FIS	1233	11301	11662
	ZG	254	591	1396
(2,10,40)	FIS	xxx	xxx	xxx
	ZG	192	443	729
(3,15,30)	FIS	xxx	xxx	xxx
	ZG	122	283	338
(5,20,20)	FIS	xxx	xxx	xxx
	ZG	72	168	141

Abbildung 6.44: Komponentenverhaltenssynthese: Evaluierung diskrete und kontinuierliche Zeitsemantik

Ein Vergleich der Anzahl der Zustände, Transitionen und Berechnungsdauer ist in den Abbildungen 6.45, 6.46 und 6.47 dargestellt. Aus dem Vergleich der Zustände und Transitionen kann beobachtet werden, dass mit Zunahme der Zustände in dem diskreten Zeitmodell auch die Anzahl der Transitionen stark zunimmt. In dem Zone Graphen ist diesbezüglich kein beobachtbarer Zusammenhang erkennbar.

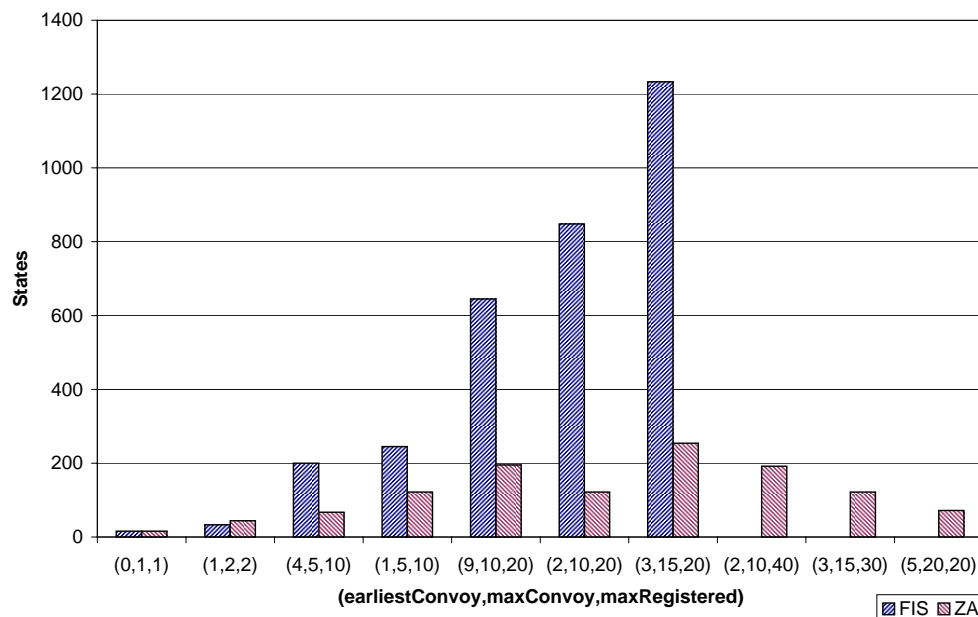


Abbildung 6.45: Vergleich Anzahl der Zustände

Der Vergleich der Berechnungszeit beider Verfahren lässt folgern, dass die der Zone Graphen nicht proportional zu den absoluten Werten der Parameter ansteigt, während dies der Fall für die diskrete Zeitsemantik ist. Die einfache Begründung liegt darin, dass der Zone Graph nicht die absoluten Werte betrachtet, sondern nur die Intervalle. Aus diesen Vergleichen lässt sich folgern, dass der Syntheseansatz mit Zone Graphen praktikabel für die betrachtete Anwendungsdomäne ist.

6.3.1.5 Codegenerierung und WCET-Analyse

Die einzelnen Elemente der Komponenten werden entsprechend dem Komponenten-Metamodell auf Klassen abgebildet. Das Klassenmodell wird dabei während der Modellierung automatisch aus dem Komponentenmodell abgeleitet und beinhaltet die Methoden für die Strukturanpassung. Aus diesem internen Modell wird objektorientierter C++ Code generiert. Für die Strukturanpassungen werden zusätzlich Factory Klassen angelegt, wie in Abbildung 6.48 abgebildet. Die Factory-Klassen ermöglichen das kontrollierte erzeugen (und löschen) der Instanzen.

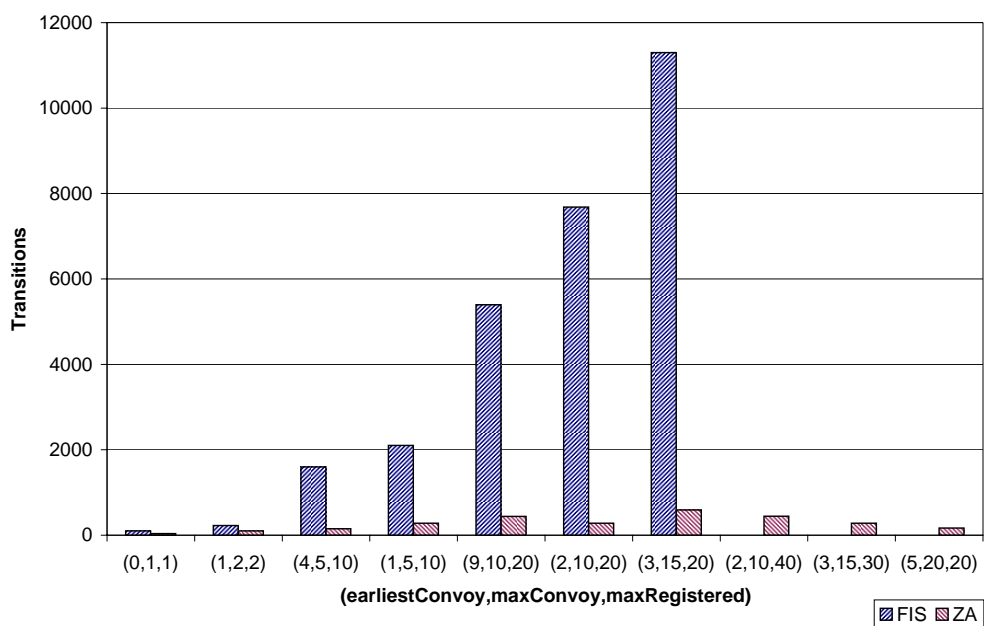


Abbildung 6.46: Vergleich Anzahl der Transitionen

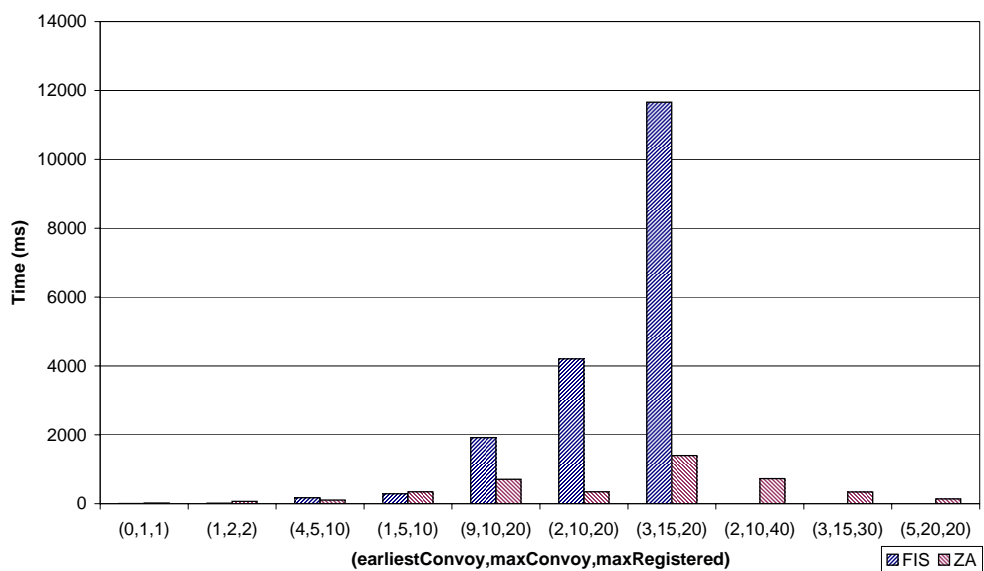


Abbildung 6.47: Vergleich der Berechnungszeit

Wie bereits in Abschnitt 6.2 diskutiert kann die Codegenerierung für die Strukturanpassungen noch nicht vollständig automatisch mit der bisherigen Codegenerierung für HYBRID RECONFIGURATION CHARTS ausgeführt werden, da die hierfür benötigten umfangreichen Metamodellanpassungen noch nicht vollständig in die bisherige Codegenerierung eingeflossen sind. Der generierte Code muss daher manuell angepasst werden, um die Anwendung ausführen zu können. Die manuelle Anpassung beinhaltet im Wesentlichen die direkte Zuordnung von strukturellen Elementen zu Verhalten (z.B. Port zu Portverhalten, Komponente zu Komponentenverhalten). Dies ist momentan nicht gegeben, da die HYBRID RECONFIGURATION CHART Codegenerierung ein Gesamtverhalten für eine Komponente generiert. Wird allerdings unabhängig für die einzelnen Elemente sowie deren Überlappung die Codegenerierung angestoßen, so wird keine manuelle Anpassung benötigt.

Als Validierung haben wir das Hinzufügen und Löschen von Konvoiteilnehmern mit den beschriebenen Verhalten betrachtet und durch ein Testsystem simuliert. Hierbei wurde gleichzeitig die Funktion evaluiert, die Parameter der Profile zur Laufzeit einer neuen Situation anzupassen. Tabelle 6.3 gibt einen Überblick über die Größe und den Speicherverbrauch des generierten Codes für die RailCab Komponente und das ConvoyCoordination-Muster mit der benötigten Factory. Im Vergleich zu der in der Arbeit von Burmester [Bur06] evaluierten HYBRID RECONFIGURATION CHART Codegenerierung ist für die Initialisierung des Systems durch den zusätzlichen Aufwand der Factory Klassen, ein größer Ressourcenbedarf notwendig. Für dynamische Systeme ergibt sich allerdings durch diesen Ansatz ein Vorteil, da nicht alle Ressourcen a priori festgelegt und komplett unveränderbar vorinstanziiert werden müssen.

Als Ergebnis der Validierung kann festgehalten werden, dass eine Anwendung unter den gegebenen Bedingungen generiert und erfolgreich ausgeführt werden kann. Anzumerken ist hierbei, dass wir als Zielsystem nur eine PC-Plattform gewählt haben. Durch die Kapselung der betriebs-systemspezifischen Prozeduren durch die entwickelten Frameworks (siehe Abschnitt 6.1) gehen wir davon aus, dass die Codegenerierung auch für weitere Plattformen anwendbar ist.

Die Codegenerierung ist zudem Grundlage für die WCET-Analyse, die, bis auf die Schleifenanalyse, plattformspezifisch auf Codeebene durchgeführt werden muss. Zuerst wird für die (Timed) Story Diagramme eine WCNI berechnet. Dieses Ergebnis dient dann zusätzlich zu dem übersetzten Code als Eingabe in ein entsprechendes WCET-Analysewerkzeug. Das verwendete Werkzeug Bound-T (siehe Abschnitt 6.1.2.2) ermöglicht die Angabe der WCNI für Schleifen durch sogenannte Assertions, mit denen die Schleifen beschrieben werden, die eingeschränkt werden sollen sowie die maximale Anzahl der Durchläufe für diese Schleife. Die Abbildung der Assertions wurde dabei im Rahmen der Projektgruppe ReCab [BBB⁺09] für das Werkzeug Bound-T automatisiert.

Das Ergebnis der WCET Berechnung wird in einem (Ressourcen) Profil hinterlegt (siehe Abschnitt 6.1.2.1). Für die hier vorgestellten Beispiele benötigte das verwendete WCET-Werkzeug nur wenige Sekunden. Für die Komplexität sowie die Herleitung der WCET Ergebnisse sei auf Abschnitt 6.1.2.2 (Seite 192) verwiesen.

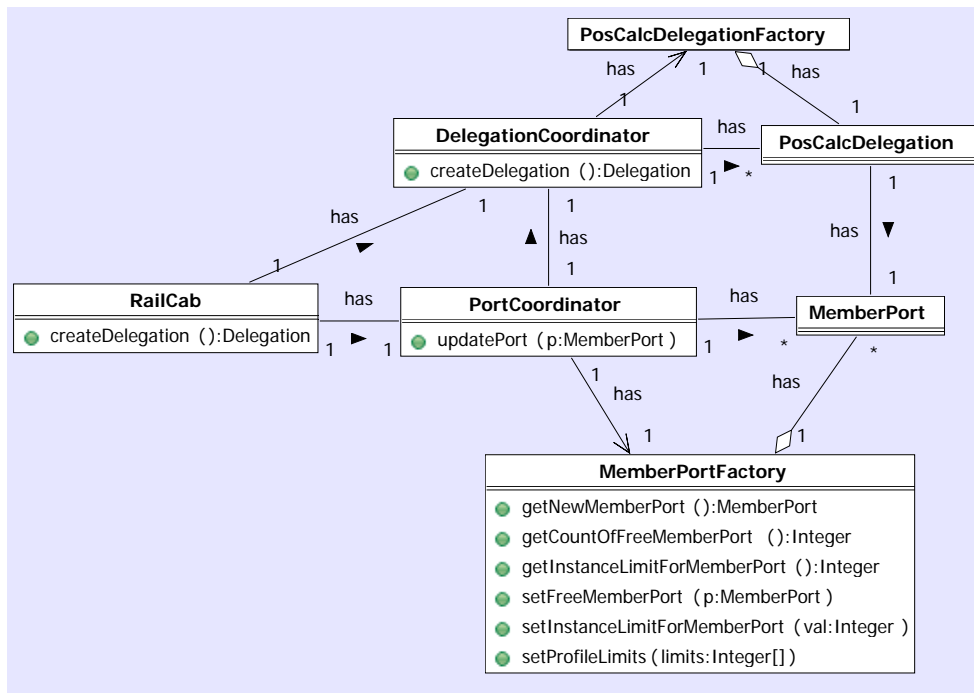


Abbildung 6.48: Generierte Klassen

Typ	Dateien	LoC	Speicher pro Instanz [kBytes]
Component	RailCab.h	23	2,12
	RailCab.cpp	74	
	RailCabStatechart.h	117	47,8
	RailCabStatechart.cpp	1267	
		Σ 1481	
Pattern	Coordinator.h	41	7,74
	Coordinator.cpp	323	
	Member.h	41	2,28
	Member.cpp	90	
		Σ 495	
Factory	CoordinatorFactory.h	26	2,67
	CoordinatorFactory.cpp	91	
		Σ 117	
		Σ 2093	

Tabelle 6.3: Generierte Fujaba Klassen

6.3.2 Weitere Anwendungsszenarien und Fazit

Neben dem Konvoi-Szenario wurden in [May09, May08] die Anwendung des kompositionellen Modellierungsansatzes der MECHATRONIC UML betrachtet, um weitere Szenarien der RailCab-Anwendung zu implementieren. Dabei konnten REAL-TIME COORDINATION PATTERNS für Szenarien wie Bahnübergang oder Weichenfahrt entwickelt werden sowie eine Einteilung dieser Muster bezogen auf die Ebene der Koordination zwischen verschiedenen Komponenten (z.B. Koordination innerhalb eines RailCabs oder zwischen RailCabs).

Der Gesamtansatz der MECHATRONIC UML wurde und wird zudem im Rahmen des Sonderforschungsbereichs 614 in einem ganzheitlichen Ansatz zur Entwicklung von mechatronischen Systemen (von der domänenübergreifenden Entwicklung bis hin zur domänenspezifischen Entwicklung) an zahlreichen Demonstratoren erprobt [ADG⁺09]. Aus diesen Ergebnissen kann geschlossen werden, dass die MECHATRONIC UML für die betrachteten Systeme geeignet ist.

Außerhalb der RailCab-Anwendung wurde die Integration von Altkomponenten an einer Scheibenwischer-Anwendung in Kooperation mit der Hella KGaA Hueck & Co (siehe Abschnitt 6.2) evaluiert [HMSN10a, HMSN10b]. Da wir nicht direkt auf die Altkomponenten zugreifen konnten, wurden diese entsprechend nach Vorgaben (z.B. Patente) nachimplementiert und analysiert. Grundsätzlich scheint daher eine Anwendung der Altkomponentenintegration auch im industriellen Umfeld möglich zu sein. Wünschenswert wäre aber gerade in diesem Bereich eine Validierung mit umfangreicheren industriellen Fallstudien.

Insgesamt lässt sich folgern, dass mit der Werkzeugumgebung Anwendungen aus der betrachteten Domäne mechatronischer Systeme umgesetzt werden können. Die gestellten Anforderungen aus der Einleitung sowie Abschnitt 2.2 wurden dabei erfüllt. Für die Details zu den Evaluierungen der entwickelten Methoden sei auf Abschnitt 6.3.1 verwiesen.

Kapitel 7

Verwandte Arbeiten

Die Beherrschung komplexer Softwaresysteme, wie dies für mechatronische Systeme benötigt wird, verlangt eine modellgetriebene Softwareentwicklung. Durch die Modellbildung wird eine Abstraktion ermöglicht, die zur Reduzierung der Komplexität führt. Die durch die MECHATRONIC UML propagierte Separierung in Echtzeitkoordinationsmusterverhalten und Echtzeitverhalten der Komponenten ermöglicht eine kompositionelle Verifikation großer verteilter Systeme. Diese wohldefinierte Separierung haben wir in dieser Arbeit ausgenutzt, um einen neuen Ansatz zur Unterstützung der Komposition und Wiederverwendung in einem komponentenbasierten, modellgetriebenen Softwareentwicklungsansatz für mechatronische Systeme mit eingebettetem Charakter, kompositionellen Strukturanpassungen und Echtzeitverhalten zu präsentieren.

Es gibt eine ganze Reihe an Ansätzen, die die Anforderungen mechatronischer Systeme adressieren. In diesen Ansätzen ist allerdings die Verhaltensanpassung häufig nur auf einfache Rekonfigurationen beschränkt oder es fehlt gänzlich an einer Unterstützung für die Wiederverwendung von Komponenten mit Strukturanpassungen oder Altkomponenten. Im Unterschied zu diesen Ansätzen ist unser Ansatz in einen nahtlosen Entwicklungsansatz integriert, welcher Echtzeitanforderungen für komplexe, verteilte Systeme garantiert und sogar eine vorhersagbare Codegenerierung für die verifizierten kompositionellen Strukturanpassungen ermöglicht.

Im folgenden Abschnitt 7.1 werden wir zuerst einen Vergleich mit Ansätzen vornehmen, die einen ganzheitlichen Entwurf mechatronischer Systeme adressieren. Anschließend werden wir spezifischer für die vorgestellten Konzepte zur Unterstützung der Komposition und Wiederverwendung Vergleiche in Abschnitt 7.2, 7.3 und 7.4 vornehmen.

7.1 Modellgetriebene Entwicklungsansätze

Die grundlegende Arbeit zu diesem Vergleich wurde in [GH06b] vorgestellt. Als Anwendungsbeispiel haben wir das RailCab betrachtet. Als relevante Vergleichskriterien haben wir die Unterstützung für Modellierung, modellgetriebene Entwicklung sowie die Analyse dieser Modelle identifiziert.

Die Unterstützung für eine *Modellierung* ist eine wichtige Voraussetzung für die Entwicklung von komplexen Systemen. Der Entwickler benötigt eine Unterstützung für angemessene

Abstraktions- und Beschreibungstechniken für die spezifischen Probleme. Neben üblichen Beschreibungstechniken für das Verhalten, wie Zustandsmaschinen, werden Techniken zur Modellierung von Verhaltensanpassungen benötigt sowie eine Möglichkeit Altkomponenten zu betrachten. Die Unterstützung von Modularität ist daher ein wichtiger Bestandteil, um eine Trennung zwischen verschiedenen Komponenten oder allgemein Verhalten zu ermöglichen und dadurch Wartbarkeit und Wiederverwendung zu erleichtern.

Die Unterstützung eines *modellgetriebenen Entwicklungsansatzes (MDD)* ist für komplexe Systeme wichtig, da durch eine Trennung zwischen plattformspezifischen (PSM) und plattformunabhängigen Modellen (PIM) erst eine Wiederverwendung von Modellen (Komponenten) ermöglicht wird. Diese Anforderung geht mit einer Unterstützung von Codegenerierung einher.

Die *Modellanalyse* kann signifikant die Qualität und Entwicklungskosten verbessern (z.B. [Wir04]). Eine wichtige Grundlage für die Modellanalyse ist eine wohldefinierte Semantik, da dies die Voraussetzung für eine Simulation oder eine formale Verifikation ist. Aufgrund der Größe des Zustandsraums komplexer Systeme werden skalierbare Analyseverfahren benötigt.

Betrachtete Ansätze Auf Basis der beschriebenen Kriterien und der Relevanz der Integration zwischen Softwaretechnik und Regelungstechnik für mechatronische Systeme, haben wir (domänenspezifische) Stand der Technik Ansätze, kommende Standards und akademische Ansätze betrachtet. Eine mangelnde Überdeckung der gestellten Anforderungen ist dabei Ausschlusskriterium für die Aufnahme der Ansätze in den Vergleich.

Als de facto Standard in der Industrie betrachten wir MATLAB/Simulink mit Stateflow (im Folgenden mit MATLAB bezeichnet). Domänenspezifische Ansätze mit Softwaretechnik-Hintergrund sind CHARON, Masaccio und Giotto (im Folgenden mit Masaccio bezeichnet), HybridUML in Kombination mit HL3 (im Folgenden mit HybridUML bezeichnet) und das Tripple HyROOM/HyCharts/Hybrid Sequence Charts (im Folgenden mit HyROOM bezeichnet). Zusätzlich betrachten wir auch den Stand der MECHATRONIC UML vor dieser Arbeit als einen Ansatz mit Hintergrund aus der Softwaretechnik. HyVisual/Ptolemy II (im Folgenden mit HyVisual bezeichnet) ist ein Ansatz aus dem klassischen Engineering Bereich. SysML als (kommender) Engineering Standard wird ebenfalls untersucht.

Tabelle 7.1 zeigt einen Überblick der betrachteten Ansätze unter Angabe des Namens, die betrachteten Referenzen und der URL.

Ansatz	Referenzen	URL
CHARON	[ADE ⁺ 01, AIK ⁺ 03, AGLS01]	www.cis.upenn.edu/mobies/charon/
HybridUML	[BBHP04]	www.informatik.uni-bremen.de/agbs/research/hybriduml/
HyROOM	[SPP01, BBP ⁺ 02, GSB98, GKS00]	www4.in.tum.de/~stauner/
HyVisual	[HLL ⁺ 03, BCL ⁺ 05]	ptolemy.eecs.berkeley.edu/
Masaccio	[HKSP02, HHK01, Hen00]	www.eecs.berkeley.edu/~fresco
MATLAB	[ASK04]	www.mathworks.com
MECHATRONIC UML	[GBSO04, BGT05, BGO06, BGK05, BGST05]	www.fujaba.de/projects/realtime/
SysML	[Obj05a]	http://www.omg-sysml.org/
UML ^h	[FNW98, FJW97]	swt.cs.tu-berlin.de/~nordwig/HYFOS/

Tabelle 7.1: Untersuchte modellgetriebene Ansätze

Übersicht Vergleich Tabelle 7.2 zeigt zusammengefasst das Ergebnis unserer Untersuchung. Ausführlich wurde dies in [GH06b] diskutiert. Aus der Tabelle lassen sich folgende Beobachtungen ableiten: 1) Die meisten Ansätze unterstützen nur sehr restriktiv Konzepte für die Modellierung, wie Standardsichten für die Struktur und das Verhalten. Szenario- und Aktivitätssichten, welche für die frühe Entwicklungsphase oder auch zur Beschreibung von Verhaltensanpassungen nützlich sein können, werden häufig nicht unterstützt. Einen musterbasierten Ansatz zur Erhöhung der Wiederverwendung wird nur von der MECHATRONIC UML unterstützt. 2) Weiterhin können wir folgern, dass alle Ansätze in der Unterstützung der PSM Ebene schwächen aufweisen, die allerdings notwendig für eine Integration von Altkomponenten oder der Wiederverwendung von Elementen ist und notwendig für (beinahe) jedes komplexes System. 3) Eine weitere überraschende Beobachtung ist, dass fast alle Ansätze, inklusive MATLAB, eine Codegenerierung anbieten, die keine Garantien für die Einhaltung der spezifizierten Zeitbedingungen und Anforderungen im Modell zusagen. 4) Eine skalierbare formale Verifikation von Sicherheitseigenschaften wird kaum unterstützt, geschweige denn die Berücksichtigung von kompositionellen Strukturanpassungen und Altkomponenten. Wir können daraus folgern, dass keiner dieser Ansätze die identifizierten Anforderungen für die Softwareentwicklung mechatronischer Systeme gerade hinsichtlich der Unterstützung für Wiederverwendung und kompositionellen Strukturanpassungen erfüllen.

7.2 Modellierung und Verfeinerung kompositioneller Strukturanpassungen

In dieser Arbeit haben wir TIMED STORY CHARTS vorgestellt, um einen gemeinsamen Formalismus für Echtzeitverhalten mit kompositionellen Strukturanpassungen anzubieten. Weiterhin haben wir eine Komposition und Wiederverwendung von Lösungen in diesem Formalismus umgesetzt durch eine wohldefinierte Verfeinerung und Verifikation der Verfeinerung. Wir können daher zu einer verwandten Arbeit im Bereich der Modellierung von Systemen mit Strukturanpassungen betrachten. Zum anderen werden wir verwandte Verfeinerungen untersuchen.

7.2.1 Modellierung

Eine abstrakte Vorgehensweise für die Entwicklung dynamischer Architekturen wird in [ZC06] vorgestellt. Hierbei wird nicht explizit auf Sprachen und konkrete Verifikationsansätze eingegangen. Muster werden in diesem Ansatz ebenfalls nicht berücksichtigt. Die grundsätzliche Idee ein zusätzliches Adaptionsverhalten einzuführen ist allerdings ähnlich. Die Notwendigkeit der Modellierung und Analyse von Systemen mit Strukturanpassungen wurde ebenfalls in der Roadmap [CLG⁺09] erkannt, es werden jedoch ebenfalls keine konkreten Ansätze vorgestellt.

Bradbury und weitere geben in [BCDW04] eine Übersicht über Modellierungssprachen für die Modellierung von dynamischen Softwarearchitekturen. Die Übersicht betrachtet Sprachen, die

Legende: X: unterstützt --: nicht unterstützt	Ansatz	MATLAB	CHARON	HybridUML	UML ^h	HyROOM	Masaccio	MechatronicUML	HyVisual	SysML
	Struktur:									
Instanzen und/oder Typen		X	X	X	X	X	X	X	X	X
Deployment		--	--	--	--	--	--	X	--	--
Muster		--	--	--	--	--	--	X	--	--
Prozess/Task Sicht		--	--	--	--	--	--	--	--	--
Verhalten:										
Kontinuierlich		X	X	X	X	X	X	X	X	X
Zustandsmaschine		X	X	X	X	X	X	X	X	X
Szenarien		--	--	--	--	X	--	X	--	X
Aktivitäten		--	--	--	--	--	--	X	--	--
Anpassung		X	X	--	X	X	X	X	X	--
Kompositional		--	--	--	--	--	--	--	--	--
Modularität		X	X	X	X	X	X	X	X	X
MDD Level										
PIM		X	X	X	X	X	X	X	X	X
PSM		--	--	X	--	--	X	X	--	--
Code		X	X	X	X	X	X	X	X	--
Codegenerierung										
Nicht-Echtzeitfähig (nur simulativ)		--	--	--	X	--	--	--	X	--
Echtzeitfähig		X	X	X	--	X	X	X	--	--
Echtzeitfähig + korrekte zeitliche Aktivierung		--	X	X	--	--	X	X	--	--
Echtzeitfähig + korrektes Scheduling (inkl. WCET)		--	--	--	--	--	X	X	--	--
Semantik		X	X	X	X	X	X	X	X	--
Simulation		X	X	X	X	X	--	X	X	--
Scheduling Analyse		--	X	--	--	--	X	X	X	--
Formale Verifikation / Model Checking		X	X	--	--	--	X	X	--	--
Skalierbar		--	X	--	--	--	X	X	--	--
Kompositionale Strukturanpassung		--	--	--	--	--	--	--	--	--
Altkomponenten		--	--	--	--	--	--	--	--	--

Tabelle 7.2: Übersicht Vergleich MDD Ansätze

auf 1) Graphtransformationen basieren, die auf 2) Prozessalgebren basieren und Sprachen, die auf 3) formaler Logik basieren.

Zu den Ansätzen, die auf Graphtransformationen basieren gehören die von Le Métayer [LM98], Hirsch et. al. [HIM98], Taentzer et. al. [TGM00], Gyapay et. al. [GVH03], Rivera et al. [RDV09] und Boronat et. al. [BÖ10].

Die Ansätze von Le Métayer [LM98] und Hirsch et. al. [HIM98] basieren auf einer kontextfreien Grammatik, deren Produktionsregeln als Graphtransformationen spezifiziert sind. Der Ansatz von Taentzer et. al. [TGM00] modelliert Strukturanpassungen über Graphtransformationen. All diese Ansätze betrachten keine Verfeinerung sowie Zeit.

Ansätze zu 1), die Zeit berücksichtigen sind die von Gyapay et. al. [GVH03], Rivera et al. [RDV09] und Boronat et. al. [BÖ10]. Der Ansatz von Gyapay et. al. setzt das Vergehen von Zeit durch diskrete Zeitticks um. Weiterhin können zeitliche Eigenschaften im Vergleich zu unserem Ansatz nicht einzelnen Teilgraphen hinzugefügt werden. In dem Ansatz von Rivera et al. wird Zeit durch eine globale Clock umgesetzt, welche nicht zurückgesetzt werden kann. Weiterhin können Graphtransaktionsregeln nicht durch einen Time Guard eingeschränkt werden. Boronat et. al. stellt einen Ansatz vor, der ebenfalls Zeit durch eine globale Clock umsetzt, die nicht zurückgesetzt werden kann. Zudem unterstützt dieser Ansatz keine Invarianten.

Darwin [MK96], LEDA [CPT99] und Dynamic Wright [ADG98] sind Ansätze, die auf Prozessalgebren basieren. Dynamic Wright unterstützt eine Verhaltensbeschreibung und Verfeinerung ebenfalls über Folgen von externen Nachrichten und ist somit ähnlich zu der in dieser Arbeit verwendeten Verfeinerung. Zeit wird jedoch nicht berücksichtigt. Darwin und Leda sind Ansätze, die auf dem π -Kalkül [MPW92] basieren. Zeit sowie eine Überprüfung einer Verfeinerung wird von diesen Ansätzen nicht unterstützt.

Die Ansätze von Gerel [EW92] und Aguirre et. al. [AM02] sind Beispiele für eine auf formaler Logik basierenden Sprache. Der Ansatz von Gerel beschreibt Vorbedingungen für die Ausführung von Regeln mit Prädikatenlogik (erster Stufe). Der Ansatz von Aguirre et. al. beschreibt das Verhalten von Komponenten mit einer temporalen Logik. Eine Verfeinerung wird durch beide Ansätze nicht berücksichtigt.

Die existierenden Sprachen zur Modellierung von Strukturanpassungen nutzen verschiedenste Formalismen für die Beschreibung des Verhaltens. Bis auf die Ansätze im Bereich der Prozessalgebren wird keine Verfeinerung betrachtet. All die betrachteten Ansätze unterstützen nur sehr eingeschränkt die Modellierung von Zeit.

7.2.2 Verfeinerung

Wir betrachten im Folgenden Ansätze, die eine Verfeinerung für Graphtransformationssysteme oder Timed Automata unterstützen. Im Bereich der Graphtransformationssysteme haben wir die Ansätze von Giese [Gie07], Heckel und Thöne [HT04] sowie Große-Rhode et. al. [GRPS02] untersucht. Giese beschreibt eine Verfeinerung für hybride Graphtransformationssysteme (die

entsprechend kontinuierliche Anteile enthalten). Über die erste Ableitung einer kontinuierlichen Variable lässt sich Zeit darstellen. Dieser Ansatz fordert eine strikte Einhaltung der (Zeit-) Intervalle. Der Erhalt von Protokollverhalten wird nicht betrachtet. Die Ansätze von Heckel und Thöne sowie Große-Rhode et. al. unterstützen keine Zeit. Der Ansatz von Heckel und Thöne fokussiert sich zudem auf eine reine Diensterhaltung und kann keine Verifikationsergebnisse erhalten.

Im Bereich der Timed Automata haben wir die Ansätze von Beyer [Bey02] und Giese et. al. [GTB⁺03, BGH05a, Bur06] untersucht. In [Bey02] und [GTB⁺03] wird jeweils eine Verfeinerung über ganzzahlige Clocks (ein diskretes Zeitmodell) definiert. Die Verfeinerungen werden ebenfalls zu dem in dieser Arbeit vorgestellten Ansatz über Traces definiert. Beide Ansätze unterstützen jedoch nicht eine Relaxierung der Zeitintervalle. In [BGH05a, Bur06] wird eine Abstraktion eines HYBRID RECONFIGURATION CHARTS berechnet und anschließend die Korrektheit der Abstraktion gezeigt. Dieser Ansatz beschreibt eine strikte Einhaltung von Zeitintervallen. Weiterhin wird nicht die Erfüllung des Protokollverhaltens durch die Verfeinerung garantiert.

7.2.3 Verifikation

Im Folgenden schränken wir die Betrachtung von verwandten Ansätzen für die Verifikation auf diejenigen ein, die auch kompositionale Strukturanpassungen unterstützen. Ölveczky hat in [ÖM02, ÖM05, ÖM07] das Werkzeug Real-Time Maude vorgestellt. Real-Time Maude basiert auf textuellen (objektorientierten) Ersetzungsregeln. Dieser Ansatz unterstützt allerdings keine Erzeugungen von Uhren und basiert zudem auf einer diskreten Zeitsemantik.

Rensink hat unter anderem in [Ren08] das Werkzeug GROOVE vorgestellt. GROOVE erlaubt ein Model Checking über eine Graphversion von LTL-Formeln, wobei Gadducci et. al. in [GHK00] die Anwendbarkeit temporallogischer Formeln auf das zu einem Graphtransformationssystem generierte Transitionssystem untersucht hat. Baldan et. al. hat in [BCK08] einen Verifikationsansatz für Graphtransformationssysteme beschrieben, der auf einer Überapproximation des Verhaltens basiert. Die Approximation kann unter Ausnutzung von Gegenbeispielen verfeinert werden [CGJ⁺00]. GROOVE sowie der Ansatz von Baldan et. al. betrachten keine zeitlichen Elemente.

Schilling hat in [Sch06] einen Ansatz für den Nachweis von induktiven Invarianten für Graphtransformationenregeln vorgestellt. Dieser kann ohne die Durchführung einer Erreichbarkeitsanalyse beweisen, dass verbotene Graphsituationen im System nicht erreichbar sind. Becker und Giese haben diesen Ansatz in [BG08] um eine Unterstützung von Zeit bei dem Nachweis von Invarianten erweitert. Diese Ansätze erlauben den Nachweis von strukturellen Eigenschaften. Für Eigenschaften wie Deadlocks, die nicht über eine verbotene Struktur definiert werden können oder für die der gesamte erreichbare Zustandsraum aufgebaut werden muss, werden durch diesen Ansatz nicht unterstützt.

7.3 Analyse von Altkomponenten

Verwandt zu unseren Ansätzen zur Integration von Altkomponenten sind reguläre Inferenzansätze und Modellabstraktionstechniken aus dem Bereich der formalen Verifikation. Wir werden im Folgenden erst verwandte Ansätze im Bereich regulärer Inferenz vorstellen. Anschließend werden wir verwandte Arbeiten im Bereich Modellabstraktion diskutieren.

7.3.1 Reguläre Inferenz

Es existieren verschiedene Ansätze, die auf dem Lernalgorithmus von Angluin (siehe Abschnitt 4.2.1) basieren. Einige Ansätze, wie in [BJLS03, HNS03b, Ber06] beschrieben, erweitern Angluins Algorithmus, zur Verbesserung der Laufzeit für bestimmte Applikationen oder Domänen. Diese Ansätze nutzen Angluin's Algorithmus und fügen z.B. zusätzliche Technologien, wie Testen und Verifikation hinzu. Hierdurch werden primär die Zugehörigkeitsanfragen reduziert.

Hungar et al. [HNS03b, HNS03a, SH03, MNRS04, MRSL07] und Raffelt et al. [RMSM09] optimieren den Algorithmus von Angluin durch domänenspezifische Informationen, wie beispielsweise Präfix-Abgeschlossenheit und die Ausnutzung eines deterministischen Systems. Hierdurch reduzieren sie die Anzahl der Zugehörigkeitsanfragen.

Li and Shahbaz et al. präsentieren in ihrem Ansatz [LGS06b, LGS06a, SLG07] wie Tests genutzt werden können, um parametrisierte Zustandsautomaten zu erlernen. Dieser Ansatz basiert auch auf Angluins Algorithmus. Zunächst wird ein Test für jede Komponente ausgeführt. Anschließend werden die einzelnen Komponenten integriert. Basierend auf den synthetisierten Modellen werden Testfälle generiert und ausgeführt.

Berg et al. zeigen in [BJR06] einen Ansatz, welcher ebenfalls versucht durch reguläre Inferenz Zustandsautomaten mit Parametern zu erstellen. Sie nutzen Angluins L^* Algorithmus, um effizienter auf eine bestimmte Klasse von Systemen arbeiten zu können. Sie optimieren den Ansatz in dem sie für jeden Zustand die Eingangssignale ableiten und zu äquivalenten Klassen zusammenfassen. Dabei gilt die Hypothese, dass alle Eingangssignale die den gleichen Effekt auf einen Zustandsautomaten besitzen in der gleichen äquivalenten Klassen eingeordnet werden.

Die präsentierten Ansätze in [BPG03, CGP03, GP05] basieren auf einem Automatenmodell des Systems/der Komponente. Mit diesem Modell und einer Spezifikation lernen sie die benötigten Annahmen, um die Spezifikation zu garantieren.

Eine Technik, um eine Black Box mittels Model Checking zu überprüfen, wird von Peled und anderen in [PVY99] vorgestellt. Die Idee die zwei Techniken zu kombinieren, ist weiter ausgearbeitet worden zu einer Methode namens Adaptive Model Checking [GPY02]. In [EG⁺06] wird dieser Ansatz zu einem Grey Box Checking erweitert. Hierbei wird vorausgesetzt, dass einige Teile des Systems bereits bekannt sind. Diese Ansätze bieten die Möglichkeit einen Fehler während der Lernphase zu finden.

Grinchtein et al. präsentieren in ihrem Ansatz [GJL04, GJP06] wie der Inferenz Algorithmus von Angluin für zeitbehafte Systeme genutzt werden kann. Präziser berücksichtigen sie Systeme, welche mittels deterministischer Event Recording Timed Automata modelliert werden können. Event Recording Timed Automata [AFH99] sind eine eingeschränkte Klasse von Timed Automata, die für jede Nachricht (Aktion) eine Uhr vorsehen, die die Zeit von dem letzten auftreten der Nachricht erfasst.

Fazit Im Prinzip basieren die hier betrachteten Lernalgorithmen alle auf dem von Angluin. Bis auf [PVY99], versuchen alle Ansätze das ganze Verhalten zu synthetisieren. Erst anschließend werden Konfliktsituationen gefunden. Unser Black-Box-Ansatz betrachtet im Vergleich dazu besonders das enge Zusammenspiel zwischen dem Kontext und der Altkomponente. Somit ist es nicht erforderlich das gesamte Verhalten der Altkomponente zu erlernen. Nur der relevante Teil der Integration ist erforderlich. Ähnlich zu [PVY99] ist unser Ansatz in der Lage reale Fehler nach jedem Lernschritt zu finden. Darüber hinaus ermöglichen wir ein reaktives Verhalten in Form von ein- und ausgehenden Nachrichten sowie Zeit(-bedingungen) zu berücksichtigen. Die Betrachtung von reaktiven Verhalten sowie das Ausnutzen der Präfixabgeschlossenheit ist ähnlich zu den Ansätzen von Hungar et al. [HNS03b]. Die Möglichkeit eingeschränkt Zeit zu betrachten ist ähnlich zu dem Ansatz von Grinchtein et al. [GJL04].

Insgesamt lässt sich folgern, dass keiner der Ansätze all die relevanten Anforderungen der von uns betrachteten Systeme adressieren. Die einzelnen Techniken unseres Black-Box-Ansatzes sind allerdings ähnlich zu den hier betrachteten Verfahren, bzw. basieren hierauf. Unser Gray- und White-Box-Ansatz lassen sich aufgrund der unterschiedlichen zur Verfügung stehenden Informationen, die starke Auswirkung auf das Verfahren haben, nicht direkt mit diesen Ansätzen vergleichen. Bezogen auf den Gray-Box-Ansatz lässt sich am ehesten ein Vergleich mit dem Ansatz von [EG⁺06] erstellen, die ebenfalls davon ausgehen, mehr Informationen als für die reine Black-Box-Analyse zur Verfügung zu haben. Dieser Ansatz basiert allerdings immer noch auf dem von Angluin. Im Vergleich dazu ermöglicht unser Gray-Box-Ansatz direkt das Verhalten zu lernen, ohne Äquivalenzanfragen, sowie reaktive Systeme und Zeit zu betrachten.

7.3.2 Abstraktionstechniken

Abstraktionstechniken sind eine wichtige Technik, um die Explosion des Zustandsraums beim Model Checking zu behandeln. Gegenbeispiele werden dabei oft genutzt, um abstrakte Modelle zu verfeinern. Eine Approximation wird verfeinert, wenn Verhalten der Approximation, welches nicht im ursprünglichen konkreten Modell vorhanden ist, der Grund für ein Gegenbeispiel ist (siehe Abschnitt 7.2.3 - CEGAR).

Ausgehend vom Quellcode ist es Ziel der Abstraktionsansätze ein möglichst abstraktes Modell zu gewinnen, um (realistisch) eine Verifikation zu ermöglichen. In einem ersten Schritt wird dabei eine Überapproximation erstellt (Zustände werden zusammengefasst). Dann wird das Modell so lange verfeinert, bis kein fehlerhaftes Gegenbeispiel auftritt. Zahlreiche Ansätze, wie [Kur94,

LNA99, CGJ⁺03] (siehe auch die von uns angewandten Quellcode Model Checker in Abschnitt 4.3), basieren hierauf.

All diese Ansätze basieren auf einer reinen Quellcodeanalyse. Es werden im Vergleich zu unserem Gray-Box-Ansatz keine Tests durchgeführt, um die Eingabe des Systems zu berücksichtigen. Weiterhin betrachten diese Ansätze im Vergleich zu all unseren keine Interaktion mit der Umgebung (Kontext) sowie Verletzungen von Zeitbedingungen.

7.4 Synthese von Komponentenverhalten

In diesem Abschnitt betrachten wir die verwandten Arbeiten zu unserem Ansatz der Synthese von Komponentenverhalten. In Abschnitt 7.4.1 werden wir Ansätze aus dem Bereich der Controller-Synthese betrachten. In Abschnitt 7.4.2 diskutieren wir Ansätze zur Synthese von nicht zeitbehafteten Komponentenverhalten und anschließend in Abschnitt 7.4.3 zeitbehaftete Ansätze.

7.4.1 Controller-Synthese

Der Bereich der Controller-Synthese [AMP95, AMPS98, AT02, BK06, GGR08] beschäftigt sich mit dem Problem der Synthese von Verhalten für einen *Controller*, welcher mit einer (bestimmten) Umgebung interagiert.

Die Interaktionen eines Controllers werden durch alternierende Aktionen zwischen dem Controller und der Umgebung beschrieben. Die Synthese versucht auf Basis dieser Interaktionen und gegebenenfalls weiteren Anforderungen (Einschränkungen) einen Controller zu synthetisieren, der alle Aktionen mit der Umgebung erfüllen kann. Die hiermit unterliegenden spieletheoretischen Grundlagen führen zur Anwendung von speziellen Verhaltensmodellen wie den Timed Game Automaten [AMP95, MPS95]. In einem solchen Automaten werden Transitionen in kontrollierbar durch den Controller oder der Umgebung eingeteilt.

Der Eingabe Timed Game Automaten ist typischerweise unterspezifiziert (offen), so dass zusätzliche Eigenschaften durch die Synthese integriert werden müssen, um z.B. geforderte Sicherheits- und Lebendigkeitseigenschaften zu erfüllen (wie z.B. [Pnu77, CMP94, ACD90, ACD93]). Diese dienen als weitere Eingabe in die Syntheseverfahren.

Der Hauptunterschied zu unserer Synthese ist, dass die gegebenen Verhaltensmodelle der Controller-Synthese nicht kompositionell sind, sondern als ein Gesamtsystemverhalten aufgefasst werden. In unserem Ansatz ist die Kompositionalität durch die unabhängigen Rollenautomaten gegeben. Konsequenterweise können auch keine Eigenschaften in diesen Ansätzen beschrieben werden, die sich auf eine Komposition beziehen. Insgesamt folgt hieraus, dass andere (Verfeinerungs-) Beziehungen zwischen dem Ursprungsmodell und dem synthetisierten Modell gelten, die wiederum zu einer unterschiedlichen Synthese (Synthesealgorithmus) führen.

7.4.2 Synthese von nicht-zeitbehafteten Komponentenverhalten

Giese und Vilbig haben in [GV06] einen Syntheseansatz für das Verhalten von interagierenden Komponenten vorgestellt. Die Interaktionen werden durch sogenannte Kontrakte spezifiziert [Gie00], die das Protokollverhalten einer Operationen mit Statecharts beschreiben.

Ähnlich zu Koordinationsmustern werden Kontrakte unabhängig voneinander spezifiziert. Nimmt eine Komponente an mehreren Kontrakten teil, ist es ebenfalls möglich, dass Zustandskombination auftreten, die durch gestellte Systemanforderungen verboten sind. Entsprechend wurde in diesem Ansatz Zustandsrestriktionen ohne Zeit definiert, die bestimmte Zustandskombinationen verbieten.

Der Syntheseprozess von Giese und Vilbig beginnt mit einer parallelen Komposition der Kontraktverhalten. Anschließend werden die verbotenen Zustandskombinationen aus dieser parallelen Komposition entfernt. Als letzter Schritt wird überprüft, ob das synthetisierte Verhalten eine Verfeinerung der beteiligten Rollenverhalten ist. Ist dies der Fall, so ist das Ergebnis ein kontraktkonformes Zustandsverhalten unter Berücksichtigung der Restriktionen.

Da der in dieser Arbeit vorgestellte Ansatz historisch auf dem von Giese und Vilbig aufbaut, ist die grundsätzliche Syntheseprozedur sehr ähnlich. Aufgrund der Betrachtung von Zeit durch unseren Ansatz sind die Ansätze jedoch wiederum sehr unterschiedlich.

Die Verhaltensdiagramme der Kontrakte beschreiben eine Sequenz von Nachrichten, unterscheiden jedoch nicht zwischen senden und empfangen von Nachrichten. Für die MECHATRONIC UML ist diese Unterscheidung allerdings inhärent, genauso wie die Beschreibung von Zeitbedingungen.

Ein weiterer Unterschied sind die verschiedenen Verfeinerungsbeziehungen, die entsprechend großen Einfluss auf die Synthese haben. In dem Ansatz von Giese und Vilbig werden sogenannte τ Transitionen eingeführt, um gegenstandslosen Nichtdeterminismus zu beschreiben (repräsentiert irgendein mögliches Verhalten). Für frühe Entwicklungsphasen, wo die abhängigen Verhalten noch nicht konkret bekannt sind, ist dies auch geeignet. Für unseren Fall, mit wohlbekanntem Rollenverhalten, ist dieses Konzept ungeeignet. Jedoch ist die grundsätzliche Idee, dass zwischen den Aktionen eines Rollenverhaltens willkürlich internes Verhalten auftreten kann, ähnlich.

7.4.3 Synthese von zeitbehafteten Komponentenverhalten

Seibel erweitert den Ansatz von Giese und Vilbig in [Sei07] um Timed Automata mit einem diskreten Zeitmodell und Zustandsrestriktionen.

Ähnlich zu unserem Ansatz erweitert Seibel die Zustandsrestriktionen von Giese und Vilbig um Zeit. Zudem werden Restriktionsautomaten definiert, die Nachrichten der Kontraktverhalten (Portverhalten) beobachten können. Für die Synthese wird dann eine diskrete Zeitsemantik definiert, die das Vergehen von Zeit durch Integer-Schritte bestimmt (vergleiche Abschnitt 6.3.1.4). Der Syntheseablauf ist wiederum der gleiche wie bei Giese und Vilbig.

Das von Seibel auf UML Ports [Obj09] erweiterte Konzept ist entsprechend ähnlich zu unserem Ansatz aufgrund der Verwandtschaft zu Giese und Vilbig. Das Port-Konzept ist zudem ähnlich zu unserem Rollen Konzept. Jedoch werden Portverhalten spezifisch für eine Komponente definiert.

Der Hauptunterschied zu der Arbeit von Seibel ist die diskrete Zeitsemantik, die nicht oder nur sehr eingeschränkt für mechatronische Systeme angewandt werden kann (siehe Abschnitt 2.1). Zudem wendet Seibel die parallele Komposition und die Zustandsrestriktionen auf dem diskreten Zeitmodell an, während wir nur die Abstraktion ausnutzen, um die Rollenkonformität zu überprüfen. Das Konzept der τ Transitionen hat Seibel von Giese und Vilbig übernommen, statt das Verhalten der Ports zu betrachten. Daher sind auch die Verfeinerungsbeziehungen und der Synthesealgorithmus verschieden zu unserem und nicht anwendbar für die MECHATRONIC UML.

Kapitel 8

Zusammenfassung und Ausblick

In dieser Arbeit haben wir einen systematischen modellgetriebenen Entwicklungsansatz für selbstoptimierende, mechatronische Systeme vorgestellt, in dessen Mittelpunkt die Komposition und Wiederverwendung von Softwarekomponenten und deren Protokollverhalten zu komplexen hierarchischen Komponentensystemen steht. Die Komposition unterstützt eine kompositionelle Anpassung der Komponentenstruktur unter Berücksichtigung der sich daraus ergebenden Verhaltensanpassung sowie Altkomponenten. Durch eine nahtlose Integration in die MECHATRONIC UML werden Echtzeitanforderungen für komplexe verteilte Systeme garantiert und sogar eine vorhersagbare Codegenerierung für die verifizierten, kompositionellen Strukturanpassungen ermöglicht.

Zusammenfassung In Abschnitt 2.1 haben wir eine systematische Vorgehensweise für die Entwicklung von hierarchischen Komponentensystemen beschrieben. Hierbei haben wir die Methoden der MECHATRONIC UML mit den neu entwickelten Methoden in dieser Arbeit ganzheitlich integriert dargestellt, um eine Empfehlung für die systematische Entwicklung hierarchischer Komponenten im Kontext mechatronischer Systeme zu geben. Die Unterstützung der Wiederverwendung haben wir dabei in die Bereiche *Verfeinerung in hierarchischen Komponentensystemen*, *Integration von Altkomponenten* und *Synthese von Komponentenverhalten* unterteilt.

Die *Verfeinerung in hierarchischen Komponentensystemen* basiert auf den in Kapitel 2.6 eingeführten TIMED STORY CHART Formalismus. Mit diesem Formalismus begegnen wir der Anforderung, Echtzeitverhalten sowie Strukturanpassungen integriert zu betrachten. Die in Kapitel 3 vorgestellte Verfeinerung und Überprüfung der Verfeinerung nutzt dies aus und zeigt im Vergleich zu bisherigen Ansätzen ein höheres Potential an Wiederverwendung durch eine Relaxierung des Echtzeitverhaltens.

Für die *Integration von Altkomponenten* haben wir drei unterschiedliche Ansätze identifiziert, um für die Altkomponente eine möglichst passende Analyse der Integration zu erreichen. Wir unterscheiden dabei zwischen *Gray Box Checking*, *Black Box Checking* und *White Box Checking* (siehe Kapitel 4). Die Betrachtung von Sicherheits- und Lebendigkeitseigenschaften sowie Analyseverfahren aus der Regelungstechnik, um Reglerverhalten zu identifizieren, führen dazu, dass die gestellten Anforderungen mechatronischer Systeme abgedeckt werden.

Die *Synthese von Komponentenverhalten* rundet die Unterstützung der Wiederverwendung von Komponenten und deren Protokollverhalten ab (siehe Kapitel 5). Mit diesem Ansatz können wir formal Abhängigkeiten zwischen Protokollverhalten beschreiben und automatisch ein konsistentes (protokollkonformes) Gesamtverhalten auf Basis der Protokollverhalten synthetisieren, welches die spezifizierten Abhängigkeiten berücksichtigt.

Um Altkomponenten integrieren zu können und einen modellgetriebenen Ansatz vollständig anzubieten, wird eine automatische Codegenerierung aus den Modellen der MECHATRONIC UML benötigt. Im Rahmen dieser Arbeit haben wir eine Laufzeitumgebung für die Integration von Altkomponenten entwickelt (siehe Kapitel 6.1), die in die bisherige Codegenerierung und Laufzeitumgebung der MECHATRONIC UML [Bur06, BGH⁺07] integriert ist. Hiermit wird eine automatische Analyse für die Integration von Altkomponenten ermöglicht. Zudem haben wir den bisherigen Ansatz erweitert, um eine Vorhersagbarkeit trotz der verwendeten komplexen Objektstrukturen zu ermöglichen, indem wir eine Laufzeitanalyse (WCET Analyse) für Story Diagramme zur Verfügung stellen.

Die durch die Werkzeugunterstützung ermöglichte Validierung (siehe Kapitel 6) hat gezeigt, dass unter richtiger Anwendung der Methoden, selbstoptimierende, mechatronische Systeme erfolgreich umgesetzt werden können.

Ausblick Ausblicke auf weiterführende Arbeiten können aufgrund der stetigen Weiterentwicklung der MECHATRONIC UML im Fachgebiet Softwaretechnik¹ viele gegeben werden.

Naheliegend sind Erweiterungen des eingeführten TIMED STORY CHART Ansatzes und entsprechender Verfeinerung zu einer formalen Verifikation. Hierzu müssen die zu überprüfenden Eigenschaften für die Domäne mechatronischer Systeme insofern angepasst werden, dass sowohl Struktur-, als auch Verhaltenseigenschaften geeignet spezifiziert werden können. Erste Ideen hierzu wurden in [HSJZ10] vorgestellt.

Für die Integration von Altkomponenten und der Synthese von Komponentenverhalten sind an erster Stelle umfassende Evaluierungen notwendig. Für die Integration von Altkomponenten ist besonders zu untersuchen, inwiefern strukturelle und rein statische Reverse Engineering Verfahren die Überprüfung der Integration unterstützen können. Eine Integration mit den an diesem Lehrstuhl entstandenen bisherigen Arbeiten im Bereich Reverse Engineering (z.B. [Wen08]) wäre z.B. denkbar, um strukturelle Information in den Integrationsansatz einfließen zu lassen.

Im Fall der Synthese von Komponentenverhalten ist zu untersuchen, ob die vorgestellten Kompositionsregeln ausreichend sind. Eine Erweiterung um die Möglichkeit der Spezifikation von strukturellen Abhängigkeiten ist ein Ausblick für weiterführende Arbeiten. In diesem Zusammenhang ist zu untersuchen, ob die Unterstützung von Strukturanpassungen einfacher und umfangreicher ermöglicht werden kann, wenn eine Synthese direkt über TIMED STORY CHARTS definiert wird.

¹<http://www.upb.de/cs/ag-schaefer>

Wie schon bereits in Abschnitt 6 diskutiert, ist eine vollständige Integration der Codegenerierung für Story Diagramme mit der für HYBRID RECONFIGURATION CHARTS ein Ausblick. Weiterhin ist eine werkzeugtechnische Integration mit der flexiblen Ressourcenverwaltung (siehe Abschnitt 6.1.2.1) notwendig, um umfangreiche Evaluierungen speziell der Nutzenpotentiale selbstoptimierender, mechatronischer Systeme mit kompositionellen Strukturanpassungen zu untersuchen. Eine Erweiterung der bisherigen Simulation (siehe [BGH⁺07]), um die hiermit ermöglichten komplexeren Szenarien, ist ein Ausblick für eine Validierungsumgebung.

Die MECHATRONIC UML gliedert sich in den Gesamtentwicklungsansatz des Sonderforschungsbereichs 614² als ein domänenspezifischer Entwicklungsansatz für die Softwaretechnik ein. Um eine durchgängige Entwicklung zu unterstützen, wird ein Übergang zwischen der domänenübergreifenden Entwicklung in die Domäne der Softwaretechnik benötigt. In [GGS⁺07, HHKS08] haben wir hierzu erste Ideen vorgestellt, die als Grundlage für weiterführende Arbeiten nützlich sein können (siehe z.B. [GSG⁺09]). Da sich die szenariobasierte Entwicklung in den frühen Phasen als sehr nützlich erwiesen hat, ist der Übergang, unterstützt durch eine Synthese von Zustandsverhalten, wie in [GHHK06, HGH⁺09, Gre10] betrachtet, vielversprechend. Die Einbeziehung von Strukturanpassungen ist ein möglicher Ausblick für eine solche Synthese.

Die MECHATRONIC UML wurde und wird stetig im Rahmen des RailCab-Projektes evaluiert. Eine Anwendung der Konzepte in der Industrie wurde teilweise gezeigt (siehe Abschnitt 6.3.2). Umfangreichere Betrachtungen sind hier wünschenswert. Im Sinne einer tiefgreifenderen Validierung sollte hierbei ebenfalls untersucht werden, ob die Methoden erfolgreich von geschulten Benutzern angewandt werden können.

²<http://www.sfb614.de/>

Anhang A

Timed Story Charts

In Abschnitt 2.6.4 haben wir TIMED STORY CHARTS eingeführt. Im Folgenden werden die einzelnen Elemente (siehe Abschnitt A.1) sowie die zusammengesetzte Ausführung der Elemente (siehe Abschnitt A.2) erläutert.

A.1 Elemente

A.1.1 Statechart

Ein Statechart AB wird definiert durch eine Klasse AB, welche von der Statechart Klasse (siehe Abbildung 2.27) erbt. Der Name der Statechart-Klasse entspricht dem Namen des Statecharts. Ein Statechart wird durch Instanziierung der Statechart-Klasse angelegt. Für Strukturelemente zu denen inhärent ein Verhalten gehört, wird automatisch das dazugehörige Statechart erzeugt. Zu diesen Strukturelementen gehören nach dem Metamodell aus Abbildung 2.20 Component, Part, Delegation und Coordinator. Hierdurch haben auch alle Part Elemente ein Verhalten. Ein Statechart kann ebenfalls in einem ComplexState eingebettet sein. Für Multielemente, die ein parametrisiertes Verhalten verlangen, wird durch die parameter Attribute der Objekte ClockInstance, ActiveState und Synchronization eine eindeutige Zugehörigkeit einer Statechartinstanz zu den Parametern gewährleistet.

Für die Umsetzung von parametrisierten Verhalten gibt es unterschiedliche Möglichkeiten. Es kann z. B. für jede Instanz eines Multielementes eine Instanz des parametrisierten Verhalten angelegt werden. Für Analysezwecke führt dies allerdings zu einem unnötigen Mehraufwand, da die einzelnen Zustandsobjekte die gleichen für jede Instanz sind. Um dies zu umgehen, wird durch unseren Ansatz nur ein Objekt für das Statechart angelegt und für Element des Statecharts, die sich pro Instanz unterscheiden (ClockInstance, ActiveState und Synchronization), wird das parameter Attribut genutzt.

Abbildung A.1 zeigt die Abbildung eines parametrisierten Statecharts AB_Statechart auf ein TIMED STORY CHART. Trotz der Mehrfachinstanziierung des Statecharts, wird nur ein Statechart Objekt AB_Statechart angelegt. Durch das parameter Attribute des ActiveState Objekts wird trotzdem eine eindeutige Unterscheidung zwischen den unterschiedlichen Instanzen ermöglicht.

Die MECHATRONIC UML verlangt, dass zu den Strukturelementen wie Komponenten, Ports und Delegation ein Statechart angelegt werden muss. Weiterhin wird für ein Multielement verlangt, dass für jede Instanz eines Multielements ein Statechart mit entsprechend dazugehörigen Parametern angelegt wird. Ein Statechart muss auch in andere Statecharts eingebettet werden können. Durch die in Abschnitt A.1.1 eingeführten Statechart-Klassen bleibt damit die Semantik eines Statecharts erhalten, da ein Statechart genau durch eine Klasse und Assoziation zu entsprechenden Strukturelementen und komplexen Zuständen, die Statecharts einbetten können, definiert ist. Weiterhin wird durch die parameter Attribute eine Mehrfach-Instanziierung mit den geforderten Eigenschaften unterstützt.

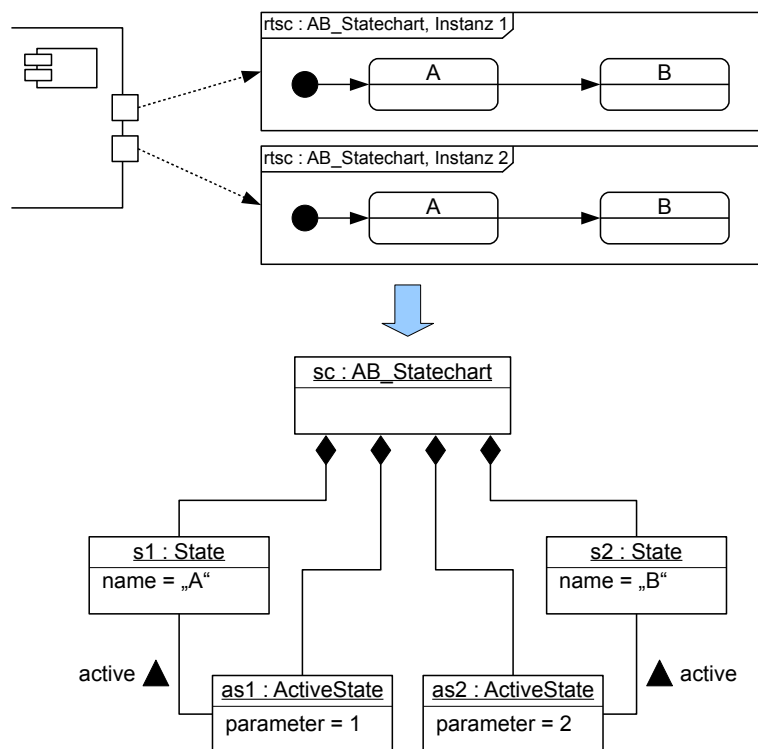


Abbildung A.1: Abbildung eines Statecharts auf einen Objektgraphen.

A.1.2 Zustände

Ein Statechart-Zustand wird über eine Instanz der State Klasse angelegt. Das name Attribute wird mit dem Namen des States initialisiert. Ein AND-State wird durch die Klasse ComplexState spezifiziert, in dem eine Menge von Statecharts dem gleichen ComplexState zugeordnet werden. Ein Zustand ist aktiv, wenn ein Objekt vom Typ ActiveState eine gerichtete Assoziation active zu diesem Zustand hat. Ein Zustandswechsel wird durch Anpassung dieser Assoziation erreicht, indem active zu einen anderen Zustand assoziiert wird. Ein parametrisiertes Statechart besitzt für jede Instanz eine ActiveObjekt Instanz, die mit dem aktiven Zustand der Instanz assoziiert

ist. Das `parameter` Attribut einer `ActiveObjekt` Instanz wird mit dem Wert k der k -ten Instanz des parametrisierten Statecharts instanziiert. Ein `ActiveState` Objekt eines nicht parametrisierten Statecharts wird mit `Parameter` eins instanziiert. Initial assoziieren die `ActiveObjekt` Instanzen die Startzustände eines Statecharts.

Wie bereits in Abschnitt A.1.1 diskutiert, ist durch die Umsetzung von Mehrfach-Instanzen eines parametrisierten Statecharts mittels des parametrisierten `ActiveState` Objekts eine effiziente Möglichkeit, um die Instanzen eines parametrisierten Statecharts zu verwalten, ohne für jede Instanz eine Statechart Instanz anzulegen.

Abbildung A.1 zeigt ein Beispiel für die Abbildung eines parametrisierten Statecharts mit zwei Instanzen auf einen Objektgraphen (siehe auch Abschnitt A.1.1). Für die beiden Instanzen des `AB_Statecharts` wird ein `AB_Statechart` Objekt instanziiert. Die Zustände A und B werden jeweils durch ein `State` Objekt abgebildet. Das Namensattribut ist entsprechend mit A und B initialisiert. Die aktiven Zustände der beiden Instanzen des Statecharts werden über die `ActiveState` Objekte `as1` und `as2` spezifiziert. Das `parameter` Attribut wird mit der eindeutigen Instanznummer initialisiert.

Die endliche Menge an Zuständen S eines `PARAMETERIZED REAL-TIME STATECHART` (siehe Abschnitt 2.4.4) ist durch eine endliche Menge von `State` Objekten definiert. Der Startzustand S^0 ist durch die initiale Menge an `ActiveState` Objekten definiert. Darüber hinaus ist der aktuelle Zustand eines `PARAMETERIZED REAL-TIME STATECHART` durch die Menge der `ActiveState` Objekten bestimmt (siehe auch Definition A.1.3). Das Attribut `parameter` des `ActiveState` Objektes lässt eine Unterscheidung der einzelnen Instanzen eines Statecharts zu und erlaubt so die eindeutige Kodierung des Zustands jeder Statechartinstanz, worüber eine konkrete Instanz eines `Multielements` definiert ist. Die Parametrisierung von Statecharts eines einfachen Elements mit `Parameter` gleich eins verletzt die Semantik nicht, da das Statechart nur einmal instanziiert werden kann und der aktive Zustand sich nur auf diese Instanz beziehen kann. Die Kompositionsbeziehung zwischen Statechart und Zuständen sowie zwischen komplexen Zuständen und eingebetteten Statecharts (Sub-Statecharts) stellt sicher, dass ein Zustand nicht ohne sein Statechart existieren kann und ein eingebettetes Statechart nicht ohne den umgebenden komplexen Zustand.

A.1.3 Transitionen

Eine Transition ist durch ein Story Diagramm definiert. Die einzelnen Stories definieren die Ein- und Ausgehenden-Ereignisse, Bedingungen, Time Guards, Synchronisationskanäle, Seiteneffekte, Clock Resets und Deadlines. Werden Zeitbedingungen definiert, so werden Timed Story Pattern verwendet andernfalls Story Pattern.

Abbildung A.2 zeigt die grundsätzliche Abbildung einer `PARAMETERIZED REAL-TIME STATECHART`-Transition. Eine Transition wird durch ein Story Diagramm mit einer Story für die Transition abgebildet. Die Story stellt lediglich eine Transition von Zustand A nach B dar,

ohne jegliche Bedingungen, Ereignisse und Seiteneffekte. Für diesen einfachen Fall wird der Link durch ein Story Pattern von Zustand A nach Zustand B umstrukturiert.

Die einzelnen Elemente einer Transition eines TIMED STORY CHARTS werden im Folgenden definiert und in Abschnitt A.2 kombiniert.

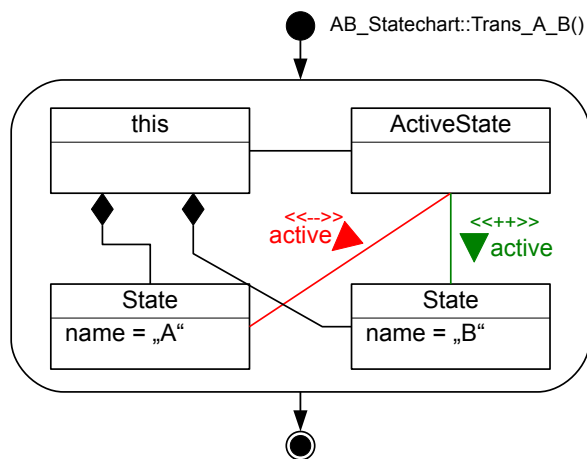


Abbildung A.2: Schalten einer Transition

Eine PARAMETERIZED REAL-TIME STATECHART-Transition T ist definiert durch $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{C}(X) \times 2^X \times \text{Sig}(l) \times \mathcal{S}$, wobei eine einzelne Transition von s nach s' durch ein 6-Tupel beschrieben ist $(s, a, \varphi, \lambda, \text{sig}, s')$ (siehe Abschnitt 2.4.4). Durch die Definition von ActiveObject Objekten lassen sich beliebig Zustandsübergänge zwischen State Objekten beschreiben, die wiederum auf die Zustände S eines Statecharts abbildbar sind (siehe Abschnitt A.1.2). Die Abbildung der anderen Elemente wird im Folgenden erläutert.

A.1.4 Clocks

Eine Clock ist definiert durch ein ClockInstance Objekt (siehe Abschnitt 2.6.2). Das parameter Attribut bezieht sich auf Instanz k des (parametrisierten) Statecharts (siehe Abschnitt A.1.2). Den Namen der Uhr wird durch das id Attribut bestimmt. value gibt den aktuellen Wert der Uhr an.

Da die Elemente des Statecharts nur einmal erzeugt werden, müssen nur einmal unabhängig von der konkreten Instanz Regeln für eine Clock Instanz beschrieben werden.

Eine Abbildung einer Clock auf ein Clock Instanz Objekt ist Abbildung A.3 zu entnehmen. Da es sich hier nur um eine Instanz handelt, wurde die Clock c1 mit dem parameter gleich eins initialisiert (siehe Abschnitt A.1.2).

Die Clocks eines PARAMETERIZED REAL-TIME STATECHART sind definiert über $X := (x_1, \dots, x_n)$ eine endliche Menge an Clockvariablen mit $x_i \in \mathbb{R}^+$. Eine Clockvariable entspricht

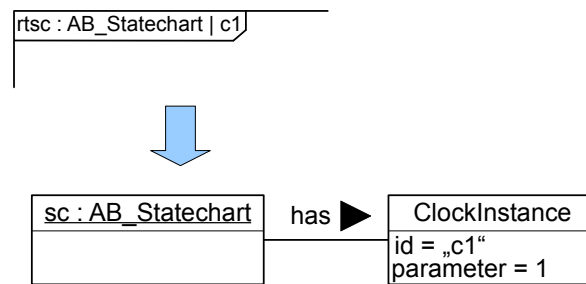


Abbildung A.3: Abbildung einer Clock auf ein ClockInstance Objekt

einem ClockInstance Objekt, welches durch das value Attribut einen Wert aus den reellen Zahlen aufnehmen kann. Eine Clock ist zudem über die Transition von ClockInstance Objekten eine Zuweisung zu Zuständen oder Transitionen ermöglicht, um Zeitbedingungen zu spezifizieren (siehe z. B. Abschnitt A.1.5). Die Definition einer Clock über Clock Instanzen erlaubt zudem, wie bereits in Abschnitt 2.6.2 vorgestellt, mehrere Instanzen einer gleichen Clock anzulegen. Dies ist zwingend notwendig für Multielemente. Eine Clock wird mit Erzeugen des Statecharts initialisiert. Eine Clock Instanz bezieht sich somit immer auf eine konkrete Instanz eines parametrisierten Statecharts. Eine Angabe von Clock Instanzregeln ist für die Instanziierung von Statecharts nicht erforderlich, da die Erstellung der benötigten Clocks direkt in die Regel zum Erstellen des Statecharts mit aufgenommen werden kann (siehe Abschnitt 2.6.2). Dies verletzt nicht die Semantik der PARAMETERIZED REAL-TIME STATECHART, da eine Clock nur mit einem Statechart existiert. Das ClockInstance Objekt erhält den gleichen Parameter wie das zu dem Statechart gehörige ActiveState Objekt.

A.1.5 Guards

Ein Guard einer Transition wird auf eine Boolesche Bedingung des Story Pattern oder TIMED STORY PATTERN abgebildet, welches die Transition beschreibt (siehe Abschnitt A.1.3).

Abbildung A.4 zeigt ein Beispiel für die Abbildung des Guards $railCab.speed \leq 10$. Die Zustände sowie die Transition wird wie zuvor definiert beschrieben. Der Guard wird über das Rail-Cab Objekt rc1 als Story Pattern Bedingung spezifiziert.

Nach der Definition von PARAMETERIZED REAL-TIME STATECHARTS muss ein Guard durch eine Boolesche Bedingung ausgedrückt und evaluiert werden können. Genau dies wird durch eine Boolesche Bedingung eines Story Patterns umgesetzt.

A.1.6 Synchronisationen

Eine Synchronisation ist durch Synchronisationsobjekte der Klasse Synchronization definiert. Das name Attribut gibt den Namen der Synchronisation an. Das parameter Attribut gibt den

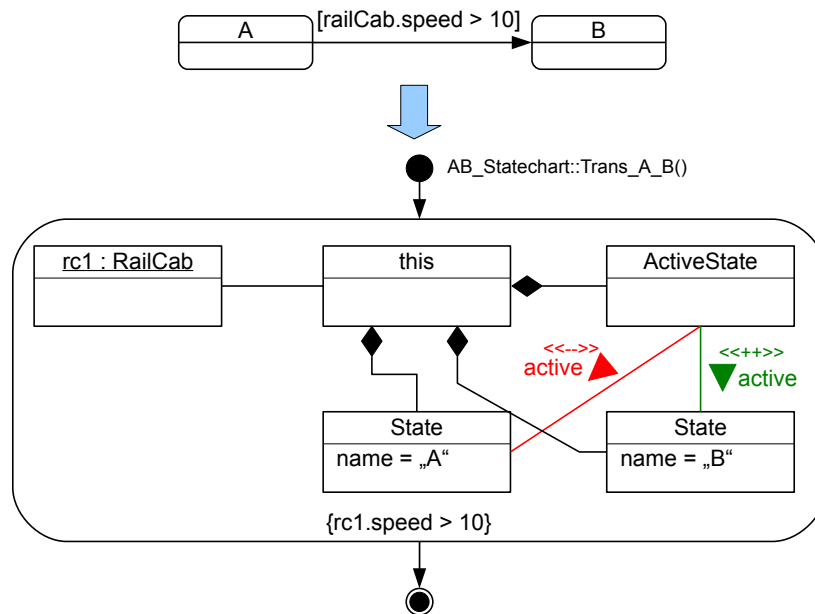


Abbildung A.4: Guard

Parameter der aktuellen Instanz an. Eine Synchronisation findet zwischen zwei Sub-Statecharts eines And-States statt. Ein Story Diagramm schaltet die an der Synchronisation beteiligten Transitionen gleichzeitig. Das Statechart Objekt, in welchem der AND-State eingebettet ist, bietet eine Methode an, die dem Story Diagramm zu Grunde liegt, um die Synchronisation zu schalten. Eine Synchronisation wird ausgeführt, wenn die beteiligten Zustände und das Synchronisationsobjekt gebunden wurden sowie ein Guard, der die Gleichheit der parameter überprüft, um sicherzustellen, dass es sich um die gleichen Instanzen handelt, wahr ausgewertet wird. Eine Synchronisation ist Bidirektional. Um den Lesefluss zu fördern, wird ein sendSrc und ein recvSrc eingeführt (vgl. Syntax von UPPAAL [LPY97])

In Abbildung A.5 wird eine parametrisierte Synchronisation zwischen zwei Transitionen gezeigt. Um die Synchronisation auszuführen, muss das Story Diagramm die Zustände A und C sowie das Synchronisationsobjekt binden. Da es sich hier um eine parametrisierte Synchronisation handelt, muss zudem noch der Parameter der beteiligten Instanzen identisch sein. Das wird über den Guard $as2.parameter = sy.parameter \wedge sy.parameter = as3.parameter$ sichergestellt.

Eine Synchronisation kann nur dann durchgeführt werden, wenn die Zustände aktiviert sind, die die Synchronisationsobjekte als ausgehende Transition schalten. Die Definition der Synchronisationskanäle ermöglicht zudem nur lokale Synchronisation innerhalb eines AND-States, wie durch die Definition von PARAMETERIZED REAL-TIME STATECHARTS gefordert.

Synchronisationsobjekte dürfen nur dann aktiviert sein, wenn die Synchronisation tatsächlich durch entsprechende aktive Zustände aktiviert wird. Ist dies nicht der Fall, so darf ein Synchronisationsobjekt nicht aktiv sein. Dies wird durch die in Abbildung A.6 und A.7 dargestellten Regeln ermöglicht.

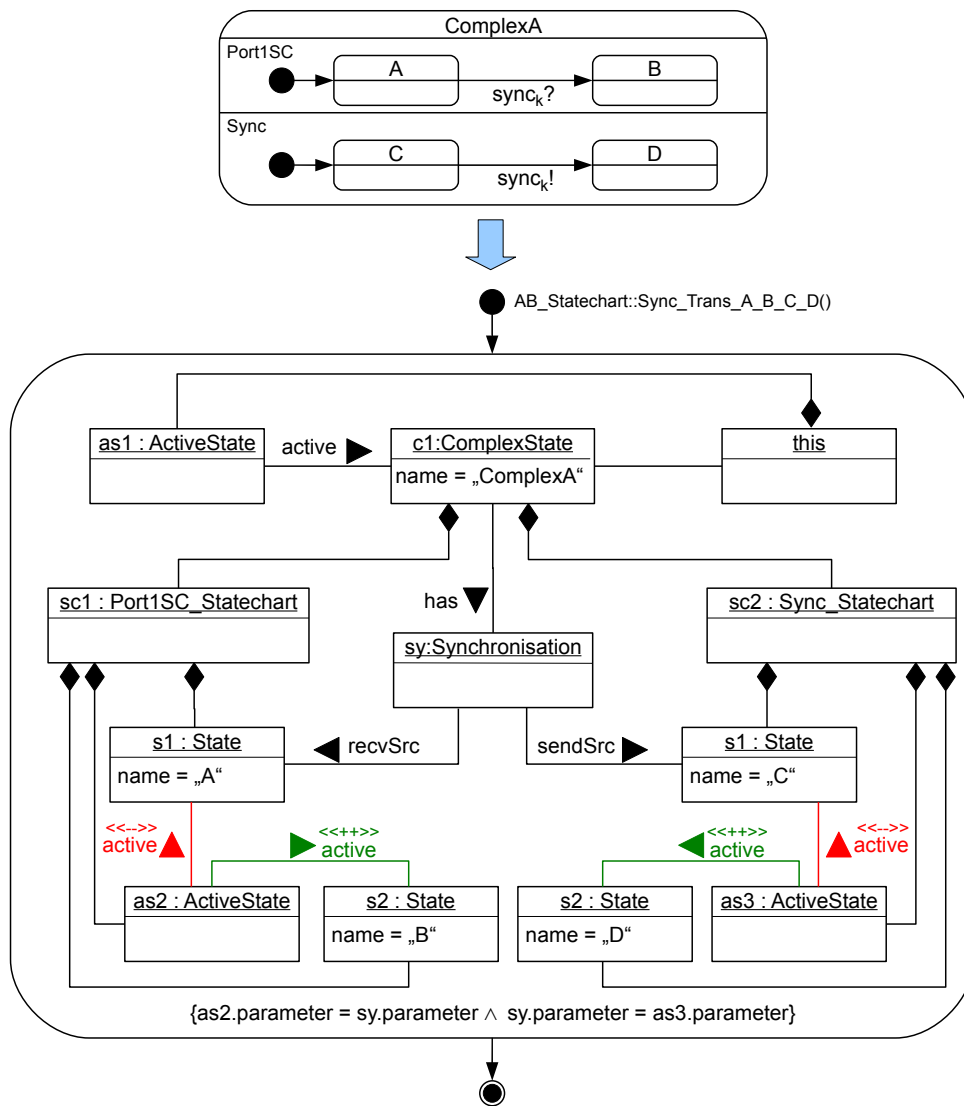


Abbildung A.5: Synchronisation von zwei Transitionen

Das Story Pattern aus Abbildung A.6 stellt nur dann ein Synchronisationsobjekt zur Verfügung, wenn der zu dem Synchronisationsobjekt gehörige Zustand aktiv ist. Falls ein entsprechendes Synchronisationsobjekt bereits angelegt wurde, so wird nur eine Assoziation zu diesem Objekt erzeugt. Andernfalls wird auch das Synchronisationsobjekt erzeugt. In diesem Fall ist die zugehörige Transition sendend und es wird eine Assoziation vom Typ sendSrc angelegt.

Abbildung A.7 zeigt, wie ein Synchronisationsobjekt deaktiviert wird, indem die Referenz zu diesem Objekt gelöscht wird. Für jeden Synchronisationskanal einer ausgehenden Transition des Zustands, der verlassen wird, wird diese Story ausgeführt. Falls ein Synchronisationsobjekt keine Assoziation zu einem Zustand besitzt, wird dieses Objekt entfernt (siehe Abbildung A.8).

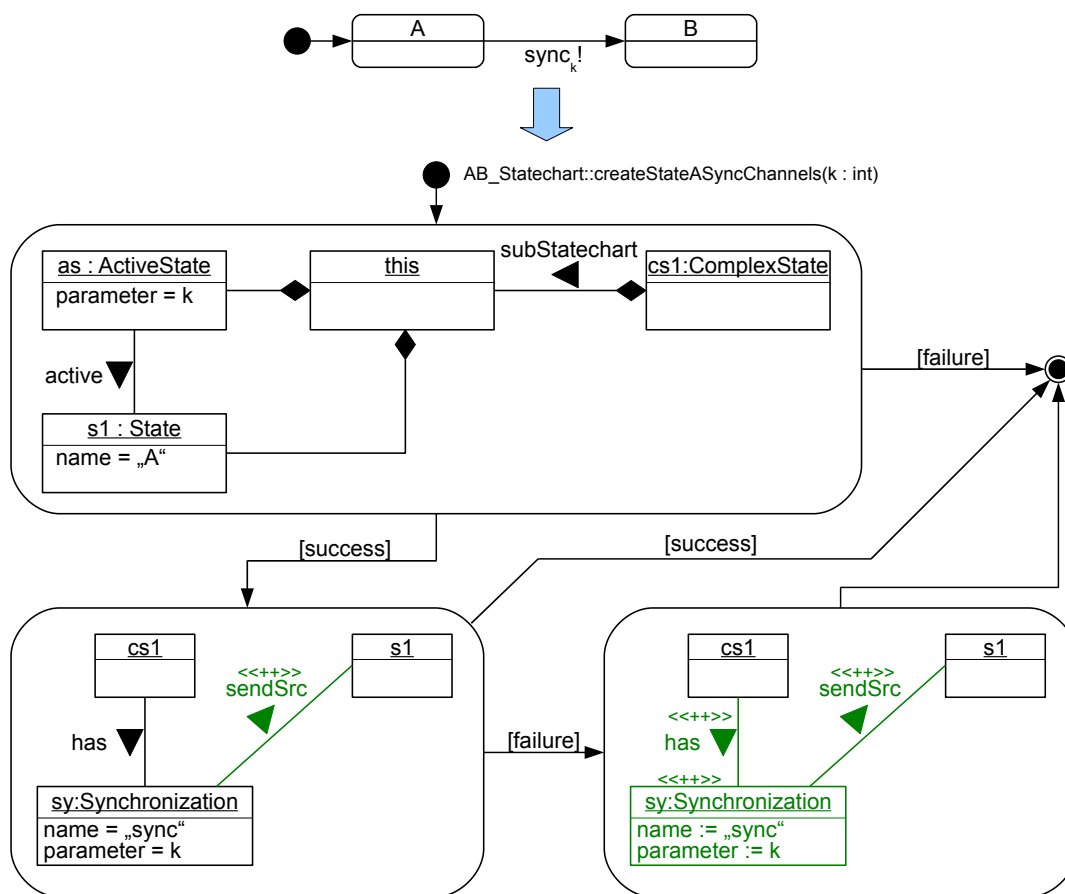


Abbildung A.6: Erstellung von Synchronisationskanälen beim Betreten eines Zustands

Die bisher vorgestellten Synchronisationsobjekte lassen sich auf Synchronisationskanäle der PARAMETERIZED REAL-TIME STATECHART einfach abbilden. Ein Synchronisationsobjekt entspricht dabei genau einem Synchronisationskanal (vgl. auch [GB03]). Nach der Definition von PARAMETERIZED REAL-TIME STATECHARTS, bzw. REAL-TIME STATECHARTS, kann eine Transition allerdings auch eine Menge an Synchronisationskanälen schalten. Dies ist durch den vorgestellten Ansatz ebenfalls gegeben, da lediglich mehrere Synchronisationsobjekte instanziiert werden müssen.

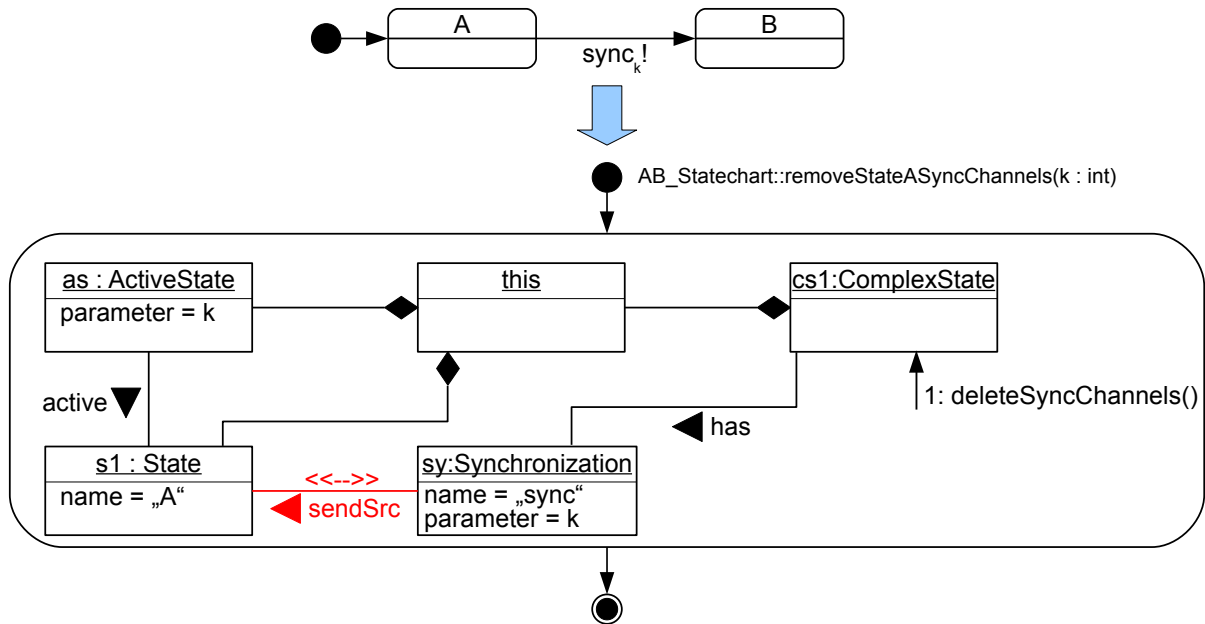


Abbildung A.7: Entfernen von Synchronisationskanälen beim Verlassen eines Zustands

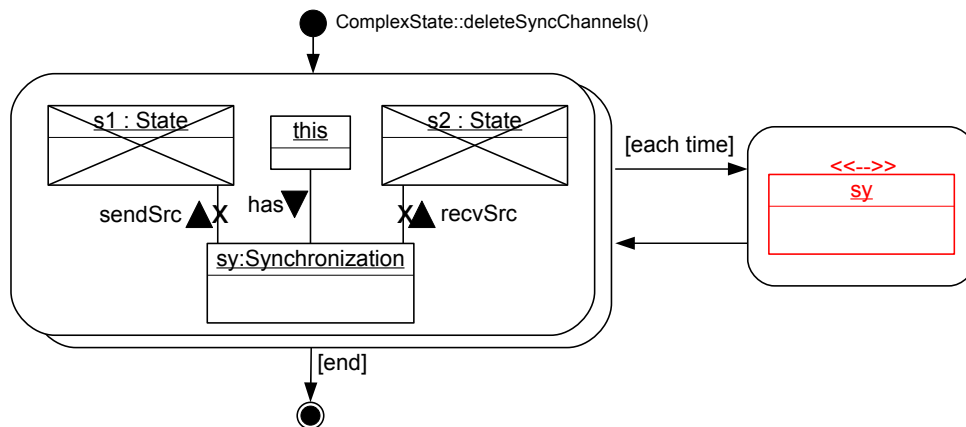


Abbildung A.8: Löschen von Synchronisationskanälen ohne Assoziation zu einem Zustand

A.1.7 Invariante

Eine Invariante für einen Zustand wird durch eine TIMED STORY PATTERN-Invariante definiert (siehe Abschnitt 2.6.2).

Abbildung A.9 zeigt die Abbildung einer parametrisierten Invariante $c1 \leq ub$. Die Abbildung definiert zum einen, dass der entsprechende Zustand und die dazugehörige Clock gebunden werden muss. Der Zustand muss aktiv sein. Weiterhin müssen die Parameter übereinstimmen ($as.parameter = ci.parameter$). Ist dies der Fall, kann die Invariante $c1 \leq ub$ überprüft werden.

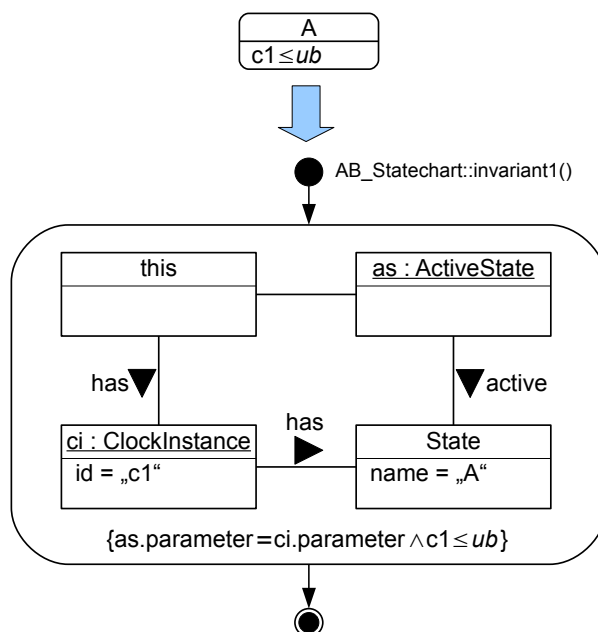


Abbildung A.9: Abbildung einer Time Invariante eines Zustands

Eine Invariante I ist eine Funktion $I \rightarrow \mathcal{C}(X)$, welche eine Menge von Ungleichungen den Zuständen zuweist. Eine Invariante limitiert das Verweilen in einem Zustand über die obere Schranke hinaus. Die gezeigte Abbildung stellt zum einen eine Verbindung zwischen Invarianten und Zuständen sowie zwischen Invarianten und Clocks her. Die Invariante des TIMED STORY PATTERN ist zudem dadurch definiert, dass der Zustand nur so lange aktiv sein darf, wie die Invariante gültig ist. Danach muss der Zustand durch eine Transition verlassen werden. Falls dies nicht möglich ist, erhält man einen Time-Stopping-Deadlock. Die Berücksichtigung der Invarianten wird in der Berechnung des Folgezustandes, wie sie in [Hir08] definiert und in Abschnitt TIMED STORY PATTERN übernommen wurde, erzwungen.

A.1.8 Time Guards

Ein Time Guard einer Transition ist definiert durch einen Time Guard eines Story Pattern (siehe Abschnitt 2.6.2).

Abbildung A.10 zeigt die Abbildung eines parametrisierten Time Guards. Voraussetzung, um den Guard zu überprüfen ist, dass die beteiligten Zustände der Transition und die Clock ($c1$), über die der Guard Einschränkungen trifft gebunden sind. Wie üblich für eine Parametrisierung, wird zudem überprüft, ob die Parameter zueinander passen. Dann kann überprüft werden, ob der Guard erfüllt ist ($lb \leq c1 \wedge c1 \leq ub$).

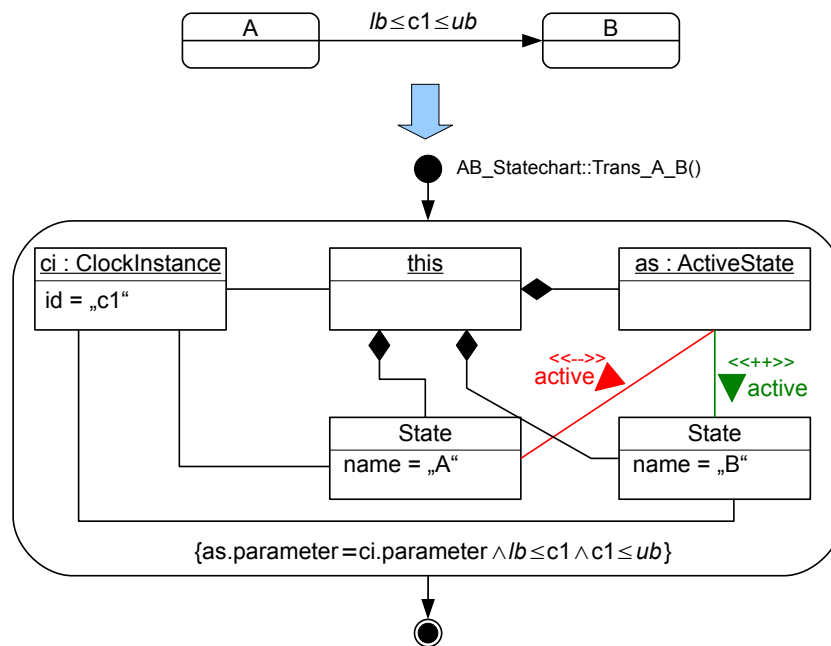


Abbildung A.10: Abbildung eines Time Guards einer Transition

Ein Time Guard ist definiert durch $\phi ::= x \sim n \mid x - y \sim n \mid \phi \wedge \phi \mid true \mid false$, mit $x, y \in C$, $\sim \in \{\leq, <, =, >, \geq\}$, $n \in \mathbb{N}$. Eine Transition des TIMED STORY CHART kann nur schalten, wenn der Time Guard erfüllt ist. Somit bleibt die Semantik eines Time Guards für PARAMETERIZED REAL-TIME STATECHARTS erhalten.

A.1.9 Clock Resets

Ein Clock Reset ist definiert durch ein TIMED STORY PATTERN Clock Reset (siehe Abschnitt 2.6.2).

Abbildung A.11 zeigt die Abbildung eines Clock Resets. Voraussetzung für einen Clock Reset ist, dass die Zustände, die den Clock Reset durch das Schalten einer Transition auslösen und die Clock, die zurückgesetzt werden soll, gebunden sind. Weiterhin müssen die Parameter der

Clock und des aktiven Zustands übereinstimmen. Ist dies der Fall, wird durch das Binden des ClockReset Objekts die Clock zurückgesetzt.

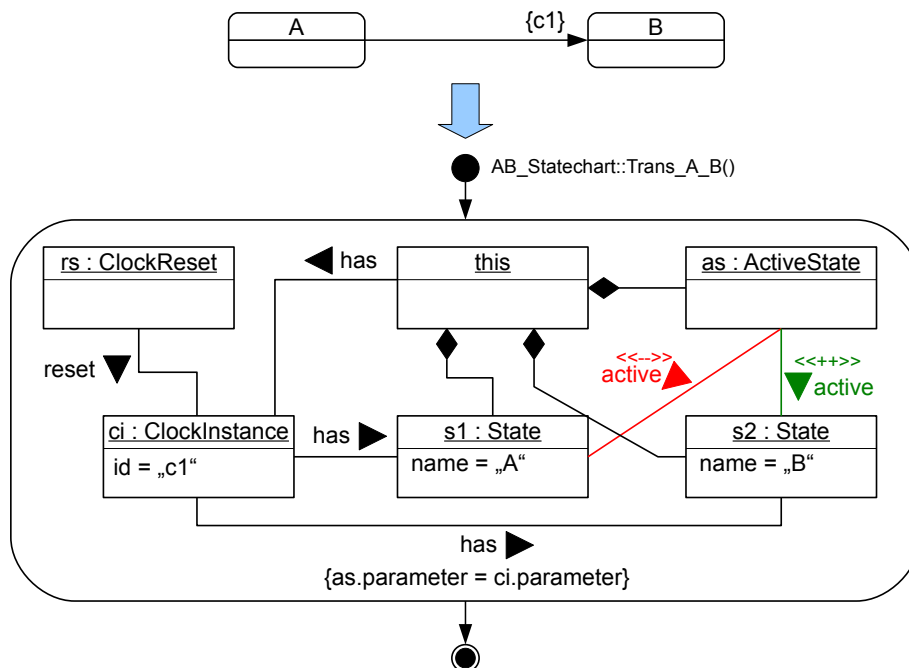


Abbildung A.11: Abbildung eines Clock Resets einer Transition

Ein Clock Reset ist definiert durch $\lambda \subseteq X$ ist eine Menge von Variablen, die auf null gesetzt werden, wenn die Transition schaltet. Dies ist durch die gezeigte Abbildung ebenfalls der Fall. Die Transition wird durch Binden des Ziel- und Quellzustands der Transition ermittelt. Die Menge der Clocks kann so beliebig einer Transition zugewiesen werden. Dadurch, dass jedem ClockInstance Objekt bei der Initialisierung ein Clock Reset Objekt zugewiesen wird, ist die Semantik unverändert.

A.1.10 Deadlines

Eine Deadline restriktiert das Verweilen in einer Transition durch eine zeitliche Unter- und Obergrenze. Eine Deadline ist definiert durch ein ClockInstance Objekt, welches mit dem Schalten der Transition angelegt wird, einer Zwischenstory, in dem die Untergrenze durch einen Time Guard definiert ist und einer weiteren Story, die eine Invariante mit der Obergrenze der Deadline definiert.

Abbildung A.12 zeigt die Abbildung einer Deadline. Die Ausführung der Transition mit einer Deadline wird in drei Story Diagramme aufgeteilt. Das erste Story Diagramm in Abbildung A.12 bindet die Vorbedingung für die Ausführung und schaltet in den Zwischenzustand executingTransAB um. Dabei wird ein ClockInstance Objekt für die Deadline erzeugt und mit 0 initialisiert.

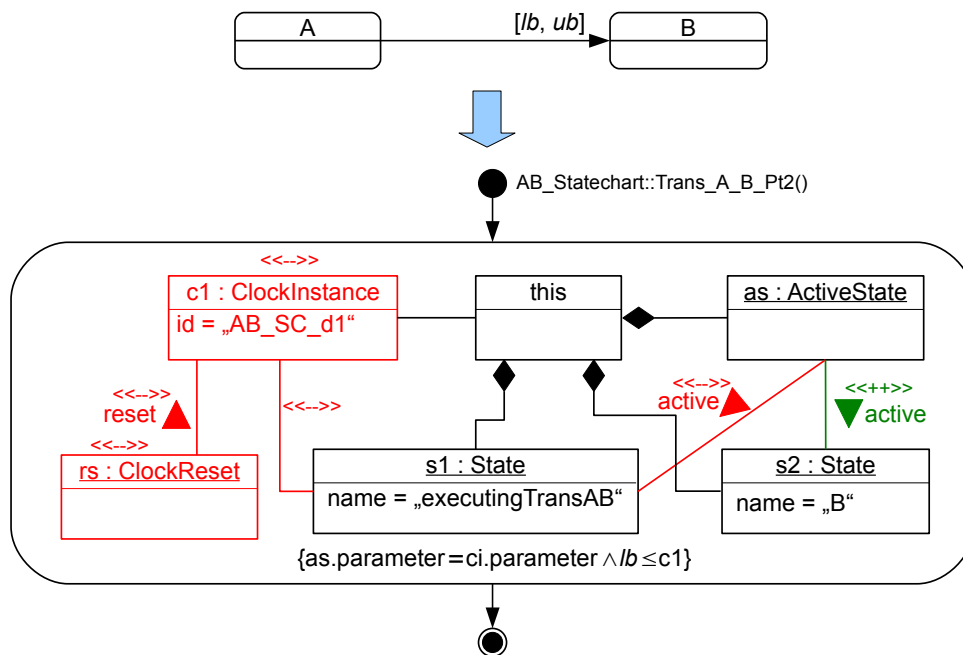


Abbildung A.13: Abbildung einer Deadline (Teil 2).

Abbildung A.14 zeigt die Abbildung von Seiteneffekten, Entry und Exit Action. Abbildung A.15 zeigt die Abbildung einer Do Action. Ein Seiteneffekt, Entry und Exit Action wird abgebildet durch binden der Zustände, die an der Transition beteiligt sind (Zustand A und Zustand B). Ist dies der Fall, kann der Quellzustand (Zustand A) verlassen werden und die Exit Action (`exitAction1()`) ausgeführt werden. Anschließend kann der Seiteneffekt ausgeführt werden (`sideEffect1()`). Das schalten des Seiteneffekts wird durch einen Zwischenzustand `executingTransAB` simuliert. Nachdem der Seiteneffekt ausgeführt wurde, wird der Zwischenzustand verlassen und der Zielzustand (Zustand B) betreten und die `entryAction1()` ausgeführt.

Eine Do Action hat die Besonderheit, dass sie periodisch ausgeführt werden kann. Daher erfolgt die Abbildung durch zwei Stories (siehe Abbildung A.15). Die Story `do1_execute()` stellt eine Abbildung der `doAction()` und der unteren Ausführungsgrenze `lb` da. Die zweite Story `do1_invariant()` stellt sicher, dass die Obergrenze `ub` eingehalten wird. Ist die Ausführung beendet, wird die Clock wieder zurückgesetzt. Die Invariantenregel stellt zudem sicher, dass die Do Action periodisch ausgeführt wird.

Eine Exit Action wird nach Definition der PARAMETERIZED REAL-TIME STATECHARTS ausgeführt, wenn der Zustand verlassen wird. TIMED STORY PATTERN müssen daher erst die Zustände, die an der Transition beteiligt sind, binden. Ist dies der Fall, kann der Quellzustand verlassen werden und die Collaboration Message, die die Exit Action implementiert, ausgeführt werden. Ein Seiteneffekt wird beim Schalten der Transition ausgeführt. Um dies zu ermöglichen wird ein Zwischenzustand eingeführt, der beim Verlassen der ersten Story erzeugt wird. Der Zwischenzustand ist ein Stellvertreter Objekt für eine Transition. Das Ausführen des Seiteneffekts simuliert

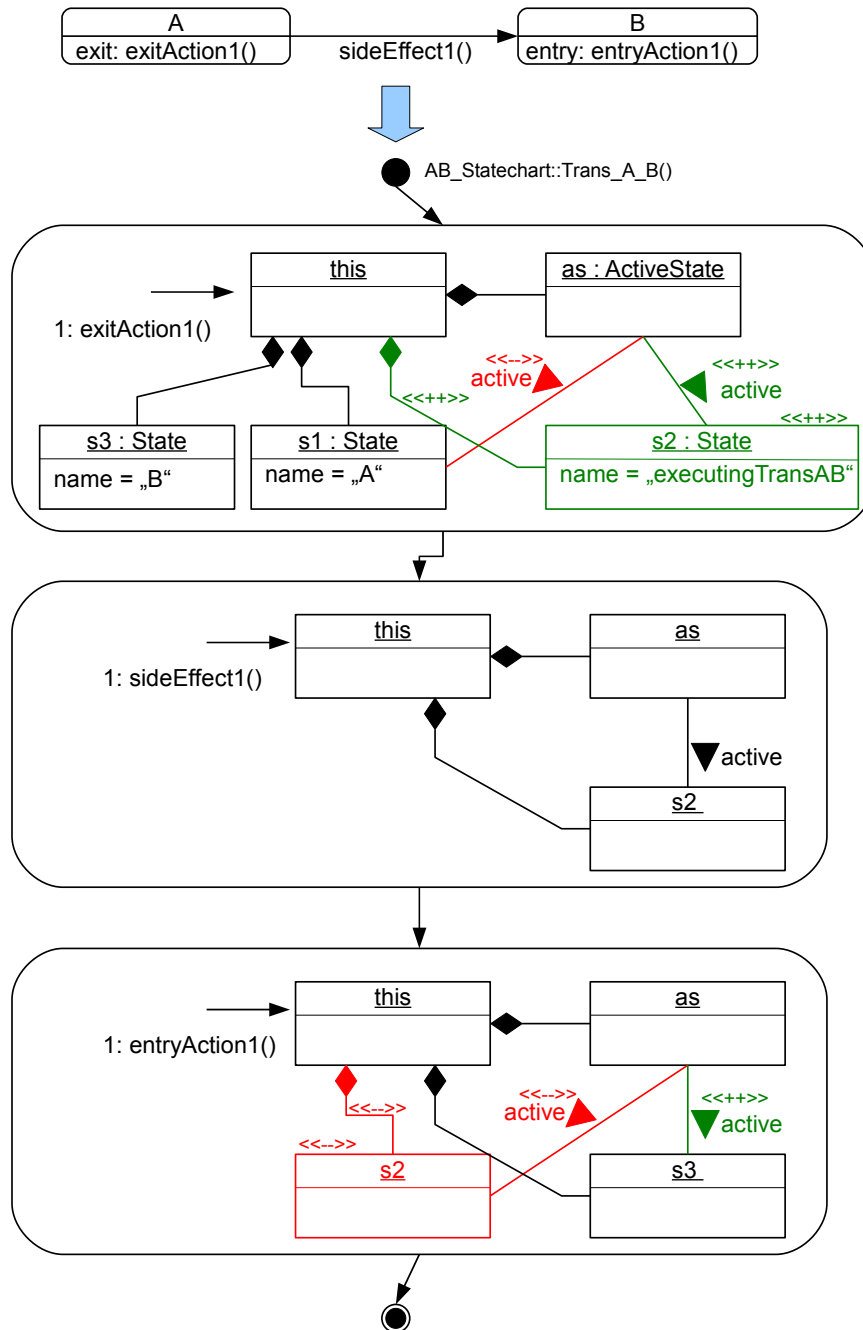


Abbildung A.14: Ausführung von Entry Action, Exit Action und Seiteneffekt

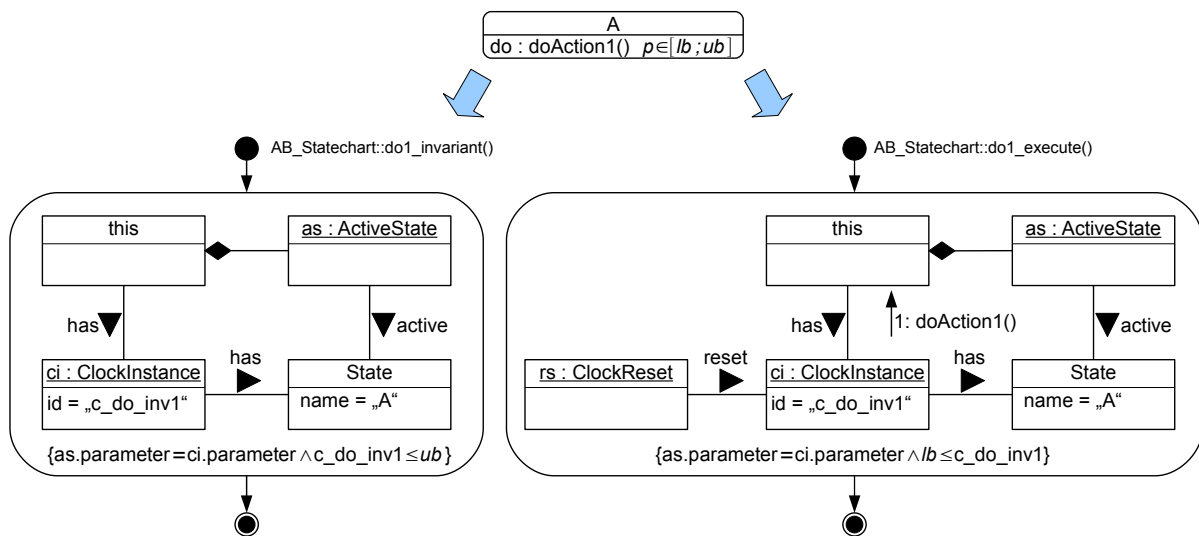


Abbildung A.15: Ausführung einer Do Action eines Zustandes.

daher genau die Situation des Schaltens einer Transition. Eine Entry Action wird ausgeführt, wenn der Zielzustand betreten wird. Voraussetzung dafür ist also, dass der Zwischenzustand verlassen wird und der Zielzustand aktiv ist. Ist dies der Fall kann die Entry Action ausgeführt werden. Dies wird durch unsere Definition umgesetzt. Die Semantik der Collaboration Messages und die Aufspaltung in drei Stories erfüllt damit die Semantik der PARAMETERIZED REAL-TIME STATECHARTS.

Eine Do Action muss einmal in jeder Periode innerhalb der angegebenen Schranken ausgeführt werden. Durch die Abbildung über TIMED STORY CHART-Invarianten bleibt damit die Semantik erfüllt (siehe Abschnitt A.1.7).

A.1.12 WCET und Prioritäten

Eine WCET nach [GB03] wird für alle Action und Seiteneffekte definiert, um ein Scheduling bestimmen zu können. Eine WCET kann einfach durch ein extra WCET Objekt der Story hinzugefügt werden, indem eine Collaboration Message ausgeführt wird. Um eine eindeutige Assoziation der WCET zu einer Collaboration Message zu ermöglichen, erhält das WCET Objekt den gleichen Namen, wie die Collaboration Message.

Eine Priorität an einer Transition gibt an, dass eine Transition mit höherer Piorität geschaltet wird, falls mehrere Transitionen gleichzeitig schalten können. Eine Abbildung auf TIMED STORY CHARTS erfolgt durch Hinzufügen von Prioritäts Objekten, die durch eine extra Story überprüft werden, falls mehrere Transitionen schalten können.

Da die TIMED STORY CHARTS bisher im Wesentlichen für Analysen von plattformunabhängigen Modellen eingesetzt wurden, sind die hierfür notwendigen Klassen noch nicht in das Metamodel aus Abbildung 2.23 eingeflossen.

A.2 Zusammengesetzte Ausführung

Im vorherigen Abschnitt wurden die einzelnen Elemente eines TIMED STORY CHARTS beschrieben. Hierbei wurde gezeigt, dass die PARAMETERIZED REAL-TIME STATECHART-Elemente auf TIMED STORY CHART-Elemente abbildbar sind. Hierdurch wird allerdings nicht vermieden, dass die einzelnen Elemente in einer beliebigen Reihenfolge ausgeführt werden, bzw. es ist nicht beschrieben, wie die verschiedenen Elemente zusammen angewandt werden, ohne die Ausführungssemantik der PARAMETERIZED REAL-TIME STATECHARTS zu verletzen.

In diesem Abschnitt soll entsprechend das Zusammenspiel der einzelnen TIMED STORY CHART-Elemente betrachtet werden. Die Kombination der Elemente muss wiederum der korrespondierenden Kombination der PARAMETERIZED REAL-TIME STATECHARTS-Elemente entsprechen. Die grundsätzliche Umsetzung der Kombination der Elemente ist einfach über ein Story Diagramm realisierbar. Die einzelnen definierten Elemente werden dabei als Stories in einer wohldefinierten Reihenfolge verschaltet. Dieser modulare Aufbau erlaubt eine einfache Anpassung der Ausführungsreihenfolge.

Zum einen müssen wir zeigen, dass das Verschalten von Zustandselementen eines TIMED STORY CHARTS dem Zustand eines PARAMETERIZED REAL-TIME STATECHARTS entspricht. Zum anderen betrachten wir, dass das verschalten von Transitionselementen eines TIMED STORY CHARTS einer Transition des PARAMETERIZED REAL-TIME STATECHARTS entspricht. Andersherum formuliert, wird ein Zustand und eine Transition eines PARAMETERIZED REAL-TIME STATECHARTS in eine oder mehrere Schaltregeln des TIMED STORY CHARTS mit mehreren Stories transformiert. Wie bereits einleitend in diesem Kapitel erläutert beschreiben wir hier lediglich informell die Semantik. Die für die Verfeinerung benötigte formale Semantik der TIMED STORY CHARTS wird in Abschnitt 3.1.2 beschrieben.

Im Folgenden betrachten wir zunächst das Verschalten der TIMED STORY CHART Elemente, um einen Zustand zu bestimmen. Anschließend werden wir Transitionen betrachten.

A.2.1 Zustände

Abbildung A.16 zeigt einen Zustand eines REAL-TIME STATECHARTS, bzw. PARAMETERIZED REAL-TIME STATECHARTS. Die Ausführungssemantik eines Zustands nach [GB03] ist durch vier Schritte definiert:

- I. Beim betreten des Zustands wird die entry-Methode ausgeführt.

- II. Anschließend wird überprüft, ob die Invariante erfüllt ist. Ist dies der Fall, wird überprüft, ob die Clock des Zustands kleiner oder gleich der Invariante minus der oberen Grenze der Periode ist. Wenn diese Bedingung positiv ausgewertet wird, kann mit III fortgefahren werden und andernfalls wird mit IV fortgefahren.
- III. Sind die Voraussetzungen der Invariante erfüllt, wird die do Methode mit spezifizierter Periode (die durch eine untere und obere Schranke angegeben wird) ausgeführt.
- IV. Ist die Invariante abgelaufen, so wird die exit Methode ausgeführt.

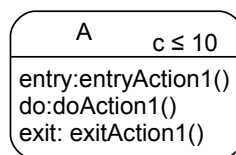


Abbildung A.16: Zustand eines Real-Time Statecharts

Das in Abbildung A.17 dargestellte Story Diagramm setzt diese Schritte um. Als 1. Story wird die Entry Action, wie in Abschnitt A.1.11 vorgestellt, ausgeführt. Voraussetzung, um die Entry Action auszuführen ist, dass der dazugehörige Zustand gebunden ist. Die Entry Action wird als Collaboration Message definiert, so dass diese wiederum durch ein Story Diagramm spezifiziert wird.

Für eine spezifische Plattform muss zudem gezeigt werden, dass die WCET der Entry Action kleiner der Invariante ist. Allgemein müssen alle Action und Seiteneffekte eine kleinere WCET als die korrespondierenden Zeitbedingungen haben. In Kapitel 6.1 betrachten wir plattformspezifische Informationen, um z. B. die Laufzeit von Story Diagrammen zu bestimmen. Für die in diesem Kapitel betrachteten plattformunabhängigen Modelle ist das nicht relevant.

Nachdem die Entry Action ausgeführt wurde, wird durch die 2. Story nach Abschnitt A.1.7 die Invariante überprüft. Hierfür wird wiederum der Zustand sowie die zu der Invarianten gehörende Clock Instanz gebunden.

Ist die Invariante abgelaufen, so wird die Exit Action ausgeführt und der Zustand verlassen.

Ist dies nicht der Fall, wird überprüft, ob die Do Action ausgeführt werden kann. Hierfür wird überprüft, ob die angegebene obere Schranke ($p.up$) noch innerhalb des offenen Zeitintervalls der Invariante ausgeführt werden kann. Ist dies nicht der Fall, wird ebenfalls der Zustand über die Exit Action verlassen.

Die 3. Story führt die Do Action nach Abschnitt A.1.11 aus. Die Do Action wird periodisch ausgeführt. Dabei wird sichergestellt, dass die Do Action innerhalb des angegebenen Periodenintervalls ausgeführt wird.

Die 4. Story führt die Exit Action des Zustands nach Abschnitt A.1.11 aus. Die Exit Action ist der Startpunkt für das Schalten einer Transition, wie in Abbildung A.14 gezeigt.

Die definierte Verschaltung der Stories setzt damit die Ausführungssemantik eines Zustands um. Im Folgenden werden wir die Ausführungssemantik einer Transition betrachten.

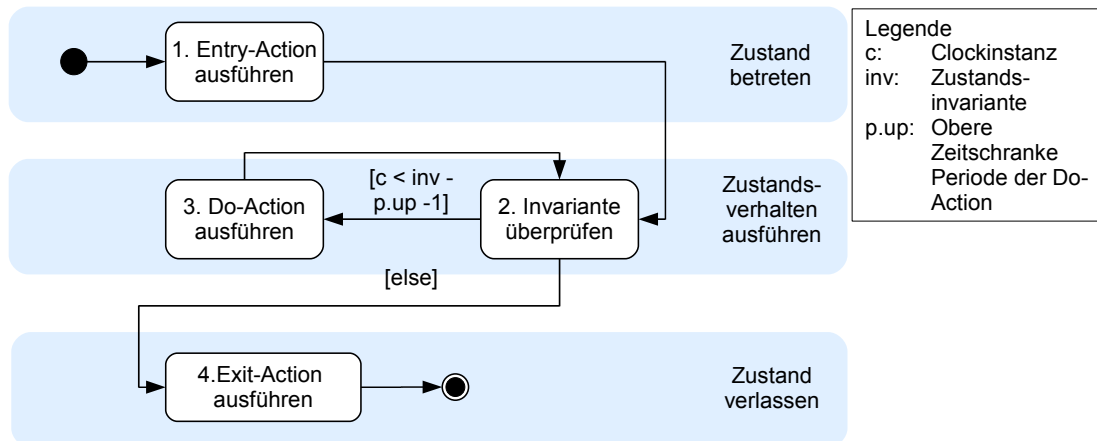


Abbildung A.17: Zustand eines Timed Story Chart

A.2.2 Transitionen

Im vorherigen Abschnitt haben wir die Ausführungssemantik eines Zustands erläutert. Die Entry und Exit Action sind der Übergang von einem Zustand in eine Transition, bzw. von einer Transition in einen Zustand. In diesem Abschnitt werden wir die Semantik der Ausführung einer Transition durch Abbildung der PARAMETERIZED REAL-TIME STATECHART-Transitionen auf TIMED STORY CHART-Transitionen definieren.

Abbildung A.18 zeigt eine Transition eines PARAMETERIZED REAL-TIME STATECHARTS, mit allen relevanten Elementen für die Abbildung auf TIMED STORY CHARTS. Die Entry und Exit Action der Zustände A und B stellen den angesprochenen Übergang von einem Zustand zu einer Transition und umgekehrt dar.

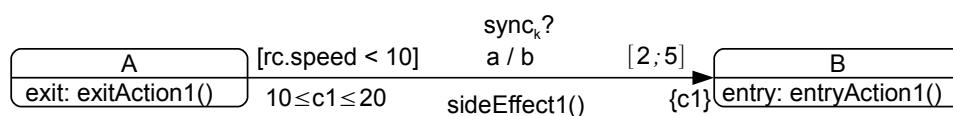


Abbildung A.18: Transition eines Realtime Statecharts

Die Ausführung einer solchen Transition ist nach [GB03] definiert durch die folgenden Schritte:

- I. Als erstes werden die Vorbedingungen zum Schalten einer Transition überprüft. Die Vorbedingungen sind, dass das Trigger-Event vorliegt, der Synchronisationskanal schaltbereit ist und der Guard sowie Time Guard erfüllt sind.
- II. Das Trigger-Event wird aus dem Event-Puffer genommen. Falls der Quellzustand weitere ausgehende Transitionen besitzt, werden diese nicht weiter berücksichtigt.
- III. Als nächstes wird der Quellzustand der Transition verlassen und die Exit Action ausgeführt.

- IV. Ist eine relative Deadline spezifiziert worden, so wird eine Clock angelegt.
- V. Während des Schaltens der Transition wird der Seiteneffekt der Transition ausgeführt, der auf die Parameter des Trigger-Events zugreifen kann. Nach der Ausführung des Seiteneffekts steht das Event nicht mehr zur Verfügung.
- VI. Anschließend werden die Deadlines der Transition überprüft.
- VII. Nach der Ausführung des Seiteneffektes wird das RaisedEvent der Transition erzeugt.
- VIII. Anschließend werden die Clock Resets ausgeführt.
- IX. Mit Betreten des Zielzustandes werden die Synchronisationskanäle von ausgehenden Transitionen des Zielzustandes verfügbar gemacht.
- X. Die letzte Aktion des Zustandswechsels ist die Ausführung der Entry Action des Zielzustandes.

Abbildung A.19 zeigt das Story Diagramm, welches die Ausführungsreihenfolge für TIMED STORY CHARTS auf Basis der PARAMETERIZED REAL-TIME STATECHARTS festlegt. Die einzelnen Stories repräsentieren die zuvor definierten einzelnen Elemente eines TIMED STORY CHART.

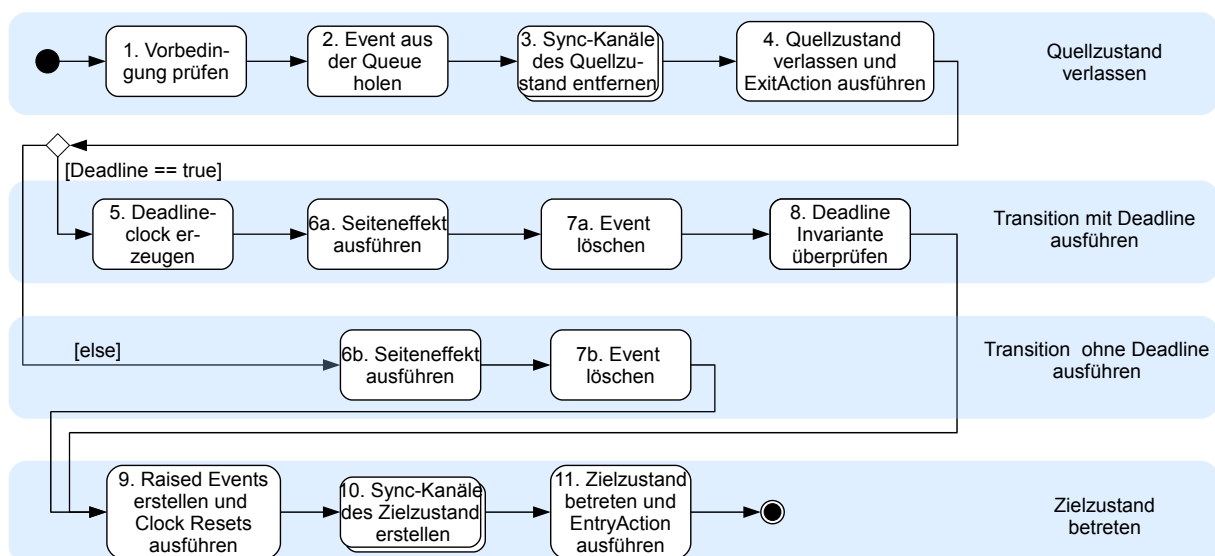


Abbildung A.19: Timed Story Chart Transition

Die 1. Story überprüft die Vorbedingung zum Schalten der Transition. Dazu zählt das Binden des Quellzustandes als aktiven Zustand, das Binden des Trigger-Events sowie der Synchronisationskanäle. Bei einer Synchronisation werden die an der Synchronisation beteiligten Transitionen gemäß Abschnitt A.1.6 in einem Story Diagramm geschaltet, um die Gleichzeitigkeit der Transitionenübergänge zu gewährleisten. Falls eines dieser Objekte nicht gebunden werden kann, kann das Story Diagramm nicht ausgeführt werden und die Transition wird nicht geschaltet. Außerdem werden alle Guards und Time Guards der Transition als Bedingungen in die erste Story

aufgenommen. Der Guard sowie der Time Guard müssen ebenfalls erfüllt sein, damit das Story Diagramm ausgeführt werden kann. Zusammen ergibt sich damit genau Punkt I der Ausführungssemantik einer Transition eines PARAMETERIZED REAL-TIME STATECHART.

Die 2. Story ruft die dequeue Methode des Statecharts auf und nimmt das Trigger-Event aus der Queue. Das Trigger-Event bleibt bis zum Ende der Ausführung des Seiteneffektes im System erhalten, um die Parameter des Events verarbeiten zu können.

Die 3. Story entfernt die Synchronisationskanäle des verlassenen Zustands gemäß Abschnitt A.1.6. Dies entspricht zusammen mit der 2. Story genau Punkt II der Ausführungssemantik eines PARAMETERIZED REAL-TIME STATECHARTS. Die Multi-Story spezifiziert zudem, dass mehr als ein Synchronisationskanal entfernt werden kann.

In Story 4. wird die Exit Action des Quellzustands ausgeführt und ein Zwischenzustand angelegt. Dies entspricht Punkt III der Ausführungssemantik eines PARAMETERIZED REAL-TIME STATECHART.

Story 5. legt für relative Deadlines, wie in Abschnitt A.1.10 beschrieben, Clock Instanzen an. Dies entspricht Regel IV eines PARAMETERIZED REAL-TIME STATECHARTS.

Story 6a. und 6b. sind identisch. Die Unterscheidung wird getroffen, um zwischen einer Transition mit und ohne Deadline zu unterscheiden, da dies zu einer unterschiedlichen Folgeaktion führt. Beide Stories führen den Seiteneffekt aus. Der Seiteneffekt kann dabei auf die Parameter des Trigger-Event zurückgreifen und in Story 7a. und 7b, die ebenfalls identisch sind, wird das Trigger-Event gelöscht. Die Stories entsprechen damit Punkt V der Ausführungssemantik eines PARAMETERIZED REAL-TIME STATECHARTS.

Die 8. Story überprüft, ob das Deadlineintervall nicht verletzt wurde (siehe Abschnitt A.1.10). Dies entspricht Regel VI eines PARAMETERIZED REAL-TIME STATECHART.

In der 9. Story werden die Raised-Events und die Clock Resets der Transition ausgeführt. Dabei müssen nach der Semantik der PARAMETERIZED REAL-TIME STATECHARTS die Raised-Events vor dem Zurücksetzen der Clock Instanzen generiert werden. Da die Definition der TIMED STORY PATTERN sicherstellt, dass Clock Resets im Anschluss an die Ausführung der Graphtransformation ausgeführt werden, bleiben die Punkte VII und VIII der Ausführungssemantik eines PARAMETERIZED REAL-TIME STATECHARTS erhalten.

Die 10. Story aktiviert die Synchronisationskanäle der von dem Zielzustand ausgehenden Transitionen. Dies geschieht vor dem eigentlichen Betreten des Zielzustandes in der 11. Story.

In der 11. Story wird der Zielzustand betreten und der Zwischenzustand entfernt. Abschließend wird die Entry Action des Zielzustandes ausgeführt. Zusammen mit der Generierung der Synchronisationskanäle in der 10. Story werden die Regeln IX und X erfüllt.

Abbildungsverzeichnis

1.1	RailCab Konvoi	6
	(a) RailCab-Konvoi in der Simulation	6
	(b) RailCab-Konvoi auf der Teststrecke	6
1.2	Ausschnitt der RailCab-Komponentenarchitektur	7
2.1	Übersicht Entwicklungsansatz	11
2.2	RailCab Komponente	16
2.3	Convoy-Koordinationsmuster	19
2.4	Registration-Koordinationsmuster	19
2.5	REAL-TIME STATECHARTS der Rolle front	20
2.6	REAL-TIME STATECHARTS der Rolle rear	20
2.7	REAL-TIME STATECHART der Rolle registrar	20
2.8	REAL-TIME STATECHART der Rolle registree	20
2.9	Verhalten RailCab Komponente	22
2.10	Difference Bound Matrice für eine Clock Zone mit einer Clock.	26
2.11	Erweiterung des Konvois um ein RailCab mit einem Time Guard und einem Clock Reset	32
2.12	Eine Invariantenregel über einen Teilgraphen	33
2.13	Ein Story Pattern zur Erweiterung des Konvois um ein RailCab	34
2.14	Ein Story Pattern zur Reduzierung des Konvois um ein RailCab	34
2.15	Ein Story Diagramm zur Erweiterung des Konvois um ein RailCab	36
2.16	HYBRID RECONFIGURATION CHART für die rear Rolle	44
2.17	Beispiel Konvoirestrukturierung	46
2.18	Konvoirestrukturierung: Überblick	49
2.19	Konvoirestrukturierung: Story	50
2.20	Komponenten und -parts Metamodell	51
2.21	Multi-Part, -Port, und -Delegation	52
2.22	Beispiel-Klassendiagramm	52
2.23	Erweiterung Komponentenmetamodell um Zeit	54
2.24	Definition einer Clock-Instanz und eines Clock Resets	55
2.25	Clock Reset und Time Guard	56
2.26	Invariante	56
2.27	Metamodell für die Abbildung von Realtime Statecharts auf Story Diagramme . .	60
2.28	Schalten einer Transition mit Events	60
2.29	Dequeue der Event Handling Queue	62

2.30	Enqueue der Event Handling Queue	63
3.1	Verhaltensmodell Coordinator-Komponente	68
3.2	Coordinator-Komponente mit eingebetteten Regler	68
3.3	Verfeinertes UpdatePort Timed Story Diagramm	69
3.4	Anforderungen an die Verfeinerung	72
3.5	Beziehung zwischen RTBS und (Timed) Bisimulation	73
3.6	Zeitintervall-Verfeinerung	78
3.7	Beispiel für eine Strukturverfeinerung: (a) zeigt eine gültige Strukturverfeinerung, (b) eine ungültige	84
3.8	Überprüfung Kreise	95
4.1	Architektur mit LegacyRailCab	98
4.2	Iteratives Lernen und Überprüfen: Gray Box Checking	101
4.3	Maximal chaotisches Verhalten: der chaotische Automat	107
4.4	Trivialer initialer Automat, der den bekannten initialen Zustand berücksichtigt (4.4(a)) und das initiale Verhalten einer Altkomponente (4.4(b))	109
	(a)	109
	(b)	109
4.5	Bekanntes Kontextverhalten	110
4.6	Synthetisiertes Verhalten: Konflikt mit der Umgebung	114
4.7	Korrekt synthetisiertes Verhalten in Bezug auf den Kontext	117
4.8	Iteratives Lernen und Überprüfen: Black Box Checking	118
4.9	Parameter White Box Checking	135
5.1	Beispiel Konvoirestrukturierung mit Basisstation	141
5.2	Kombination von separaten Protokollen in der MECHATRONIC UML	142
5.3	Vereinfachte rear-Rolle	142
5.4	Vereinfachte Registree-Rolle	142
5.5	Synchronisationsverhalten Komponente: Anwendungsfälle	144
5.6	Ansatz Komponentenverhaltenssynthese	145
5.7	Nachrichten-Kompositionsregel eca_1	148
5.8	Beispiel eines parallelen Kompositionsautomaten (Rollen rear und registree)	151
5.9	Anwendung von Zustands-Kompositionsregel r_1	152
5.10	Anwendung von Zustands-Kompositionsregel r_2	152
5.11	Anwendung von Nachrichten-Kompositionsregel eca_1	156
5.12	Ausschnitt eines Zone Graphen des Konvoi-Beispiels (siehe Abbildung 5.11)	160
5.13	Modifizierte einfache rear-Rolle	168
5.14	Modifizierte einfache registree-Rolle	168
5.15	Kompositions-konformer Automat der vereinfachten Rollenautomaten	169
5.16	Zone Graph des vereinfachten kompositions-konformen Timed Automaten (siehe Abbildung 5.14)	170
5.17	Modifizierter rollen-konformer Automat aus Abbildung 5.15	171

5.18	Deadlock-freier Zone Graph des modifizierten Automaten aus Abbildung 5.17 . . .	172
6.1	Schichtenarchitektur der Laufzeitumgebung	176
6.2	Basis-Klassen der Komponentenschicht	177
6.3	Ausführungssequenz einer Komponente	178
6.4	Basis-Klassen der Portsicht	178
6.5	Eine Nachrichtensequenz	179
6.6	Beobachtung und Kontrolle der Ausführung einer MECHATRONIC UML Kom- ponentenarchitektur	180
6.7	Laufzeitumgebung mit Simulationsschicht	180
6.8	Externe Ereignisse einer Komponente	181
6.9	Nachrichtensequenz einer deterministischen Wiederholung	182
6.10	Integration plattformspezifischer Informationen	184
6.11	Einfaches Adaptionsverhalten zur Erzeugung einer Musterbeziehung	185
6.12	Story Diagram zur Beschreibung des initiateCoordination Seiteneffekts	185
6.13	Unterliegendes Klassendiagramm des initiateCoordination Seiteneffekts	186
6.14	Parametrisiertes Profil	193
6.15	Abstraktes Partitionierungsbeispiel [Bur06]	195
6.16	Übersicht Werkzeugarchitektur	198
6.17	REAL-TIME COORDINATION PATTERN DistanceCoordination	202
6.18	REAL-TIME STATECHART front-Rolle	203
6.19	PARAMETERIZED REAL-TIME STATECHART coordinator-Rolle	205
6.20	RailCab Komponententyp	206
6.21	PARAMETERIZED REAL-TIME STATECHART PosCalc-Port	207
6.22	REAL-TIME STATECHART Delegation	208
6.23	Erzeugen einer Delegation	208
6.24	Eigenschaften Altkomponente	209
6.25	RailCab-Konvoi mit Altkomponente	209
6.26	Laufzeit der Erreichbarkeitsanalyse	210
6.27	Anzahl expandierter Graphen und maximale Anzahl der Knoten	211
6.28	Laufzeit Verfeinerung	212
6.29	Legacy Checking	213
6.30	Parameter White Box Checking	213
6.31	Gegenbeispiel White Box Checking	215
6.32	Erlerner Automat der Altkomponente	216
6.33	Altkomponente: laden der (kontinuierlichen) Daten	216
6.34	Starte Systemidentifikation	217
6.35	Erkannte Regler / Transferfunktion	217
6.36	Erlerner Automat der Altkomponente mit Reglerkonfigurationen	218
6.37	Evaluierungsergebnisse Black Box Checking	219
6.38	Laufzeiten Black Box Checking mit einer Periode von 400 ms	219
6.39	Vereinfachte rear-Rolle	224
6.40	Vereinfachte registree-Rolle	224

6.41	Synthetisiertes Komponentenverhalten für die vereinfachten Rollen	225
6.42	Parametrisierte rear-Rolle	226
6.43	Parametrisierte registration-Rolle	226
6.44	Komponentenverhaltenssynthese: Evaluierung diskrete und kontinuierliche Zeitsemantik	226
6.45	Vergleich Anzahl der Zustände	227
6.46	Vergleich Anzahl der Transitionen	228
6.47	Vergleich der Berechnungszeit	228
6.48	Generierte Klassen	230
A.1	Abbildung eines Statecharts auf einen Objektgraphen.	250
A.2	Schalten einer Transition	252
A.3	Abbildung einer Clock auf ein ClockInstance Objekt	253
A.4	Guard	254
A.5	Synchronisation von zwei Transitionen	255
A.6	Erstellung von Synchronisationskanälen beim Betreten eines Zustands	256
A.7	Entfernen von Synchronisationskanälen beim Verlassen eines Zustands	257
A.8	Löschen von Synchronisationskanälen ohne Assoziation zu einem Zustand	257
A.9	Abbildung einer Time Invariante eines Zustands	258
A.10	Abbildung eines Time Guards einer Transition	259
A.11	Abbildung eines Clock Resets einer Transition	260
A.12	Abbildung einer Deadline (Teil 1).	261
A.13	Abbildung einer Deadline (Teil 2).	262
A.14	Ausführung von Entry Action, Exit Action und Seiteneffekt	263
A.15	Ausführung einer Do Action eines Zustandes.	264
A.16	Zustand eines Real-Time Statecharts	266
A.17	Zustand eines Timed Story Chart	267
A.18	Transition eines Realtime Statecharts	267
A.19	Timed Story Chart Transition	268

Tabellenverzeichnis

6.1	Ergebnisse der Evaluierung der Verbesserungsvorschläge für die korrekte Integration (Iterationen / Einzelschritte).	223
6.2	Ergebnisse der Evaluierung der Verbesserungsvorschläge für die fehlerhafte Integration (Iterationen / Einzelschritte).	223
6.3	Generierte Fujaba Klassen	230
7.1	Untersuchte modellgetriebene Ansätze	234
7.2	Übersicht Vergleich MDD Ansätze	236

Literaturverzeichnis

Eigene Veröffentlichungen

- [AB11] ANDREAS BAUMGART, Matthias Büker Werner Damm Günter Ehmen Tayfun Gezgün Stefan Henkler Hardi Hungar Bernhard Josko Markus Oertel Thomas Peikenkamp Philipp Reinkemeier Ingo Stierand Raphael W. Eckard Böde B. Eckard Böde: *Architecture Modeling / OFFIS*. 2011. – Forschungsbericht 144
- [ACE⁺08] ALHAWASH, Kahtan ; CEYLAN, Toni ; ECKARDT, Tobias ; FAZAL-BAQAIE, Masud ; GREENYER, Joel ; HEINZEMANN, Christian ; HENKLER, Stefan ; RISTOV, Renate ; TRAVKIN, Dietrich ; YALCIN, Coni: The Fujaba Automotive Tool Suite. In: AŠMANN, Uwe (Hrsg.) ; JOHANNES, Jendrik (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proceedings of the 6th International Fujaba Days 2008, Dresden, Germany* Bd. TUD-FI08-09, Technische Universität Dresden, September 2008, S. 36–39 10, 17, 198
- [ADG⁺09] ADEL, P. ; DONOTH, J. ; GAUSEMEIER, Jürgen ; GEISLER, J. ; HENKLER, Stefan ; KAHL, Sascha ; KLÖPPER, B. ; KRUPP, A. ; MÜNCH, E. ; OBERTHÜR, Simon ; PAIZ, C. ; PODLOGAR, H. ; PORRMANN, M. ; RADKOWSKI, R. ; ROMAUS, C. ; SCHMIDT, Alexander ; SCHULZ, B. ; VÖ, H. ; WITKOWSKI, U. ; WITTING, K. ; ZNAMENSHCHYKOV, O.: *Selbstoptimierende Systeme des Maschinenbaus – Definitionen, Anwendungen, Konzepte..* Bd. Band 234. Paderborn : HNI-Verlagsschriftenreihe, 2009 14, 231
- [BGH05] BURMESTER, Sven ; GIESE, Holger ; HENKLER, Stefan: Visual Model-Driven Development of Software Intensive Systems: A Survey of available Techniques and Tools. In: *Proceedings of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA, 2005*, S. 11–18 1
- [BGH⁺07] BURMESTER, Sven ; GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; TICHY, Matthias ; GAMBUZZA, Alfonso ; MÜNCH, Eckehard ; VÖCKING, Henner: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMEL-View. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA, IEEE Computer Society Press, May 2007*, S. 801–804 5, 175, 176, 183, 199, 201, 246, 247

- [BGH⁺08] BRENNER, Christian ; GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia: Integration of Legacy Components in Mechatronic UML Architectures. In: AŠMANN, Uwe (Hrsg.) ; JOHANNES, Jendrik (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proceedings of the 6th International Fujaba Days 2008, Dresden, Germany* Bd. TUD-FI08-09, Technische Universität Dresden, September 2008, S. 52–56 100, 138, 199
- [EH09] ECKARDT, Tobias ; HENKLER, Stefan: Synthesis of Component Behavior. In: GORP, Pieter V. (Hrsg.): *Proceedings of the 7th International Fujaba Days*. Eindhoven University of Technology, The Netherlands : Technische Universiteit Eindhoven, November 2009, S. 40–45 144, 199
- [EH10a] ECKARDT, Tobias ; HENKLER, Stefan: Component Behavior Synthesis for Critical Systems. In: GIESE, Holger (Hrsg.): *Architecting Critical Systems* Bd. 6150. Springer Berlin / Heidelberg, 2010, S. 52–71 5, 144
- [EHH⁺11] ECKARDT, Tobias ; HEINZEMANN, Christian ; HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia ; SCHÄFER, Wilhelm: Modeling and verifying dynamic communication structures based on graph transformations. In: *Computer Science - Research and Development* (2011), july, S. 1–20. – ISSN 1865–2034 63
- [GGS⁺07] GAUSEMEIER, Jürgen ; GIESE, Holger ; SCHÄFER, Wilhelm ; AXENATH, Björn ; FRANK, Ursula ; HENKLER, Stefan ; POOK, Sebastian ; TICHY, Matthias: Towards the Design of Self-Optimizing Mechatronic Systems: Consistency between Domain-Spanning and Domain-Specific Models. In: *Proceedings of the 16th International Conference on Engineering Design (ICED), Paris, France, 2007*, S. 25–38 247
- [GH06a] GIESE, Holger ; HENKLER, Stefan: Architecture-Driven Platform Independent Deterministic Replay for Distributed Hard Real-Time Systems. In: *Proceedings of the 2nd International Workshop on The Role of Software Architecture for Testing and Analysis (ROSATEA2006)*. New York, NY, USA : ACM Press, July 2006, S. 28–38 5, 176, 181, 183, 200
- [GH06b] GIESE, Holger ; HENKLER, Stefan: A Survey of Approaches for the Visual Model-Driven Development of Next Generation Software-Intensive Systems. In: *Journal of Visual Languages and Computing* Bd. 17, 2006, S. 528–550 1, 2, 3, 14, 200, 233, 235
- [GHH06a] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: Analysis and Modeling of Real-Time with Mechatronic UML taking Clock Drift into Account. In: *Proceedings of the International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES), Satellite Event of the 9th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML2006, Genova, Italy* Bd. 343. University of Oslo, October 2006 (Research Report), S. 41–60 38

- [GHH06b] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: A PlugIn for the Development of Resource Aware Components with Mechatronic UML. In: GIESE, Holger (Hrsg.) ; WESTFECHTEL, Bernhard (Hrsg.): *Proceedings of the fourth International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2006 (Technical Report), S. 51–55 183
- [GHH⁺06c] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; TICHY, Matthias ; VÖCKING, Henner: Modellbasierte Entwicklung vernetzter, mechatronischer Systeme am Beispiel der Konvoifahrt autonom agierender Schienenfahrzeuge. In: *Proceedings of the Fourth Paderborner Workshop Entwurf mechatronischer Systeme* Bd. 189, 2006 (HNI-Verlagsschriftenreihe), S. 457–473 6, 27
- [GHH08a] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: Combining Compositional Formal Verification and Testing for Correct Legacy Component Integration in Mechatronic UML. In: LEMOS, Rogério de (Hrsg.) ; GIANDOMENICO, Felicitad D. (Hrsg.) ; GACEK, Cristina (Hrsg.) ; MUCCINI, Henry (Hrsg.) ; VIEIRA, Marlon (Hrsg.): *Architecting Dependable Systems V* Bd. 5135, Springer Verlag, 2008 (Lecture Notes in Computer Science (LNCS)), S. 248–273 5, 100
- [GHH08b] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: A Multi-Paradigm Approach Supporting the Modular Execution of Reconfigurable Hybrid Systems / Computer Science Department, University of Paderborn. 2008 (tr-ri-08-297). – Forschungsbericht 183
- [GHH⁺08c] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; ROUBIN, Vladimir ; TICHY, Matthias: Modeling Techniques for Software-Intensive Systems. In: TIAKO, Dr. Pierre F. (Hrsg.): *Designing Software-Intensive Systems: Methods and Principles*. Langston University, OK, 2008, S. 21–58 1
- [GHH11] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: A multi-paradigm approach supporting the modular execution of reconfigurable hybrid systems. In: *SIMULATION - Transactions of the Society for Modeling and Simulation International* 87 (2011), Nr. 9, S. 775–808 5, 183, 196
- [GHHK06] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; KLEIN, Florian: Nobody's perfect: Interactive Synthesis from Parametrized Real-Time Scenarios. In: *Proceedings of the 5th ICSE 2006 Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06), Shanghai, China*, ACM Press, May 2006, S. 67–74 10, 17, 247
- [GHHP07] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia: Model-Based Testing of Mechatronic Systems. In: GEIGER, Leif (Hrsg.) ; GIESE, Holger (Hrsg.) ; ZÜNDORF (Hrsg.): *Proceedings of the fifth International Fujaba Days 2007, Kassel, Germany*, University of Kassel, September 2007 (Technical Report), S. 51–55 100

- [HBB⁺09] HENKLER, Stefan ; BREIT, Moritz ; BRINK, Christopher ; BÖGER, Markus ; BRENNER, Christian ; BRÖKER, Kathrin ; POHLMANN, Uwe ; RICHTERMEIER, Manel ; SUCK, Julian ; TRAVKIN, Oleg ; PRIESTERJAHN, Claudia: FRiTS^{Cab}: Fujaba Re-Engineering Tool Suite for Mechatronic Systems. In: GORP, Pieter V. (Hrsg.): *Proceedings of the 7th International Fujaba Days*. Eindhoven University of Technology, The Netherlands : Eindhoven University of Technology, November 2009, S. 25–29 100, 128, 138, 183, 184, 199, 200
- [Hen05] HENKLER, Stefan: *Laufzeitunterstützung für Test, Überwachung und Diagnose bei der modellbasierten Entwicklung mit Mechatronic UML*, University of Paderborn, Software Engineering Group, Diplomarbeit, June 2005 176
- [HGH⁺09] HENKLER, Stefan ; GREENYER, Joel ; HIRSCH, Martin ; SCHÄFER, Wilhelm ; ALHAWASH, Kahtan ; ECKARDT, Tobias ; HEINZEMANN, Christian ; LÖFFLER, Renate ; SEIBEL, Andreas ; GIESE, Holger: Synthesis of Timed Behavior from Scenarios in the Fujaba Real-Time Tool Suite. In: *Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), May 16-24, 2009, Vancouver, Canada*. Washington, DC, USA : IEEE Computer Society, May 2009. – ISBN 978–1–4244–3453–4, S. 615–618 5, 144, 199, 201, 247
- [HH06] HENKLER, Stefan ; HIRSCH, Martin: A Multi-Paradigm Modeling Approach for Reconfigurable Mechatronic Systems. In: *Proceedings of the International Workshop on Multi-Paradigm Modeling: Concepts and Tools (MPM06), Satellite Event of the the 9th International Conference on Model-Driven Engineering Languages and Systems MoDELS/UML2006, Genova, Italy* Bd. 2006/1. Budapest University of Technology and Economics, October 2006 (BME-DAAI Technical Report Series), S. 15–25 15
- [HH07] HENKLER, Stefan ; HIRSCH, Martin: Compositional Validation of Distributed Real Time Systems. In: GEHRKE, Matthias (Hrsg.) ; GIESE, Holger (Hrsg.) ; STROOP, Joachim (Hrsg.): *Proceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany, 30.-31.10.2007* Bd. tr-ri-07-286, University of Paderborn, October 2007, S. 52–56 100
- [HH08a] HENKLER, Stefan ; HIRSCH, Martin: Iterative Behavior Synthesis by Combining Formal Verification and Model-Based Testing. In: *Postproceedings of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany, 2008*, S. 39–51 5, 100
- [HH08b] HENKLER, Stefan ; HIRSCH, Martin: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMEL-View. In: SCHÄTZ, Bernhard (Hrsg.) ; GIESE, Holger (Hrsg.) ; NICKEL, Ulrich (Hrsg.) ; HUHN, Michaela (Hrsg.): *Proceedings of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 7.3.-12.3.2008, Schloss Dagstuhl, Germany*. Technische Universität Braunschweig, April 2008 (Informatik-Bericht 2008-02), S. 78–87 199

- [HH11] HEINZEMANN, Christian ; HENKLER, Stefan: Reusing dynamic communication protocols in self-adaptive embedded component architectures. In: *Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. New York, NY, USA : ACM, 2011 (CBSE '11). – ISBN 978–1–4503–0723–9, S. 109–118 4, 73, 80, 84
- [HHG08] HIRSCH, Martin ; HENKLER, Stefan ; GIESE, Holger: Modeling collaborations with dynamic structural adaptation in mechatronic UML. In: *SEAMS '08: Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*. New York, NY, USA : ACM, 2008. – ISBN 978–1–60558–037–1, S. 33–40 6, 27, 38, 47, 48, 57, 73, 173, 202
- [HHH10] HEINZEMANN, Christian ; HENKLER, Stefan ; HIRSCH, Martin: Refinement Checking of Self-Adaptive Embedded Component Architectures / Computer Science Department, University of Paderborn. 2010 (tr-ri-10-313). – Forschungsbericht 4, 27, 47, 48, 57, 73, 75
- [HHKS08] HENKLER, Stefan ; HIRSCH, Martin ; KAHL, Sascha ; SCHMIDT, Alexander: Development of Self-optimizing Systems: Domain-spanning and Domain-specific Models exemplified by an Air Gap Adjustment System for Autonomous Vehicles. In: *ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, August 3-6, 2008, New York, USA*. New York, USA : ASME, September 2008, S. 1–11 247
- [HHP08] HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia: Hybrid Model Checking with the FUJABA Real-Time Tool Suite. In: AŠMANN, Uwe (Hrsg.) ; JOHANNES, Jendrik (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proceedings of the 6th International Fujaba Days 2008, Dresden, Germany*, Technische Universität Dresden, September 2008, S. 40–43 45
- [HHPS10] HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia ; SCHÄFER, Wilhelm: Modeling and Verifying Dynamic Communication Structures based on Graph Transformations. In: ENGELS, Gregor (Hrsg.) ; LUCKEY, Markus (Hrsg.) ; SCHÄFER, Wilhelm (Hrsg.): *Software Engineering 2010 - Fachtagung des GI-Fachbereichs Softwaretechnik, 22.-26.2.2010 in Paderborn* Bd. 159, GI, 2010 (LNI). – ISBN 978–3–88579–253–6, S. 153–164 27, 38, 63
- [HHZ09] HEINZEMANN, Christian ; HENKLER, Stefan ; ZÜNDORF, Albert: Specification and Refinement Checking of Dynamic Systems. In: GORP, Pieter V. (Hrsg.): *Proceedings of the 7th International Fujaba Days*. Eindhoven University of Technology, The Netherlands, November 2009, S. 6–10 47, 73, 199
- [HMS⁺10] HENKLER, Stefan ; MEYER, Jan ; SCHÄFER, Wilhelm ; DETTEN, Markus von ; NICKEL, Ulrich: Legacy Component Integration by the Fujaba Real-Time Tool Suite. In: *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. New York, NY, USA : ACM, 2010. – ISBN 978–1–60558–719–6, S. 267–270 5, 100, 128, 138, 199

- [HMSN10a] HENKLER, Stefan ; MEYER, Jan ; SCHÄFER, Wilhelm ; NICKEL, Ulrich: Reverse Engineering mechatronischer Systeme. In: *Seventh Paderborner Workshop Entwurf mechatronischer Systeme, eingereicht*, Heinz Nixdorf Institut, Universität Paderborn, 2010 (HNI-Verlagsschriftenreihe), S. 1–16 99, 100, 128, 138, 231
- [HMSN10b] HENKLER, Stefan ; MEYER, Jan ; SCHÄFER, Wilhelm ; NICKEL, Ulrich: Reverse Engineering vernetzter automotiver Softwaresysteme. In: GIESE, Holger (Hrsg.) ; HUHN, Michaela (Hrsg.) ; PHILLIPS, Jan (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VI, Schloss Dagstuhl, Germany, 2010, Tagungsband Modellbasierte Entwicklung eingebetteter Systeme*, fortiss GmbH, München, 2010, S. 77–86 100, 128, 138, 231
- [HOGS10] HENKLER, Stefan ; OBERTHÜR, Simon ; GIESE, Holger ; SEIBEL, Andreas: Model-Driven Runtime Resource Predictions for Advanced Mechatronic Systems with Dynamic Data Structures. In: *ISORC '10: Proceedings of the 2010 13th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*. Washington, DC, USA : IEEE Computer Society, 2010. – ISBN 978-0-7695-4037-5, S. 58–65 183
- [HOGS12] HENKLER, Stefan ; OBERTHÜR, Simon ; GIESE, Holger ; SEIBEL, Andreas: Model-Driven Runtime Resource Predictions for Advanced Mechatronic Systems with Dynamic Data Structures. In: *International Journal of Computer Systems Science and Engineering* 26 (2012), Nr. 6, S. 1–16 5, 183
- [HSG08] HENKLER, Stefan ; SEIBEL, Andreas ; GIESE, Holger: Synthesis of Real-Time Component Behavior / University of Paderborn, Software Engineering Group. 2008 (tr-ri-08-296). – Forschungsbericht 144
- [OMT⁺08] OSMIC, Semir ; MÜNCH, Eckehard ; TRÄCHTLER, Ansgar ; HENKLER, Stefan ; SCHÄFER, Wilhelm ; GIESE, Holger ; HIRSCH, Martin: Safe Online-Reconfiguration of Self-Optimizing Mechatronic Systems. In: GAUSEMEIER, Jürgen (Hrsg.) ; RAMMIG, Franz (Hrsg.) ; SCHÄFER, Wilhelm (Hrsg.): *Selbst-optimierende mechatronische Systeme: Die Zukunft gestalten. 7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik*. Paderborn : HNI-Verlagsschriftenreihe, February 2008, S. 411–426 43
- [PTH⁺09] PRIESTERJAHN, Claudia ; TICHY, Matthias ; HENKLER, Stefan ; HIRSCH, Martin ; SCHÄFER, Wilhelm: Fujaba4Eclipse Real-Time Tool Suite. In: *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)* Bd. 6100. Springer Verlag, 2009. – ISBN 978-3-642-16276-3, S. 1–7 197

Betreute Arbeiten

- [ACE⁺08] ALHAWASH, Kahtan ; CEYLAN, Toni ; ECKARDT, Tobias ; FAZAL-BAQAIE, Masud ; HEINZEMANN, Christian ; RISTOV, Renate ; YALCIN, Coni ; SOFTWARE ENGI-

- NEERING GROUP, UNIVERSITY OF PADERBORN (Hrsg.): *Abschlussarbeit der Projektgruppe Mauritius: Fujaba Automotive Tool Suite*. Software Engineering Group, University of Paderborn, 2008 198, 201
- [BBB⁺09] BREIT, Moritz ; BÖGER, Markus ; BRENNER, Christian ; BRÖKER, Kathrin ; POHLMANN, Uwe ; RICHTERMEIER, Manel ; SUCK, Julian ; TRAVKIN, Oleg ; SOFTWARE ENGINEERING GROUP, UNIVERSITY OF PADERBORN (Hrsg.): *Abschlussarbeit der Projektgruppe ReCab: Re-Engineering mechatronischer Systeme*. Software Engineering Group, University of Paderborn, 2009 48, 70, 100, 128, 138, 184, 198, 199, 200, 201, 229
- [Bre08] BRENNER, Christian: *Verhaltenssynthese von Legacy-Komponenten auf Basis modellbasierter Testverfahren*, Software Engineering Group, University of Paderborn, Bachelor Thesis, Juni 2008 100, 199
- [Bre10] BRENNER, Christian: *Analyse von mechatronischen Systemen mittels Testautomaten*, Software Engineering Group, University of Paderborn, Master Thesis, August 2010 73, 85, 94, 199
- [Dor08] DOROCIAK, Rafal: *Hybride Verifikation von Mechatronic UML Modellen durch Integration des Modelcheckers PHAVer*, Software Engineering Group, University of Paderborn, Bachelor Thesis, Januar 2008 45
- [Eck09] ECKARDT, Tobias: *Synthesis of Reconfiguration Charts*, Software Engineering Group, University of Paderborn, Master Thesis, September 2009 144, 199
- [Hei09] HEINZEMANN, Christian: *Verifikation von Protokollverfeinerungen*, Software Engineering Group, University of Paderborn, Master Thesis, Oktober 2009 47, 73, 199, 210
- [May08] MAY, Karl A.: *Identifikation von Koordinationsmustern in autonomen mechatronischen Echtzeitsystemen*, Software Engineering Group, University of Paderborn, Bachelor Thesis, September 2008 231
- [May09] MAY, Karl A.: *Verifikation und Klassifizierung einer Koordinationsmustersammlung für Autonome Schienenfahrzeuge*, Software Engineering Group, University of Paderborn, Bachelor Thesis - Aufbauarbeit, März 2009 59, 70, 85, 231
- [Poh08] POHLMANN, Uwe: *Modellierung und Implementierung von Umschaltverfahren in Hybrid Reconfiguration Charts*, Software Engineering Group, University of Paderborn, Bachelor Thesis, Mai 2008 198
- [Pri07] PRIESTERJAHN, Claudia: *Modellbasiertes Testen von Mechatronic UML Modellen mit Gegenbeispielen*, Software Engineering Group, University of Paderborn, Master Thesis, November 2007 100
- [Ric08] RICHTERMEIER, Manuel: *Worst Case Execution Time Berechnung von Story Diagrammen für mechatronische Systeme*, Software Engineering Group, University of Paderborn, Bachelor Thesis, September 2008 183

- [Suc08] SUCK, Julian: *Entwurf und Implementierung einer formalen Verifikation für parametrisierte Echtzeitkoordinationsmuster*, Software Engineering Group, University of Paderborn, Bachelor Thesis, Dezember 2008 38

Literatur

- [ABBL03] ACETO, Luca ; BOUYER, Patricia ; BURGUEÑO, Augusto ; LARSEN, Kim G.: The power of reachability testing for timed automata. In: *Theory of Computer Science* 300 (2003), Nr. 1-3, S. 411–475. – ISSN 0304–3975 96
- [ACD90] ALUR, Rajeev ; COURCOUBETIS, Costas ; DILL, David L.: Model-Checking for Real-time Systems. In: *Proceedings of the Fifth Annual Symposium on Logic in Computer Science (LICS '90), 4-7 June 1990, Philadelphia, Pennsylvania, USA*, IEEE Computer Society, June 1990, S. 414–425 241
- [ACD93] ALUR, Rajeev ; COURCOUBETIS, Costas ; DILL, David L.: Model-Checking in Dense Real-time. In: *Information and Computation* 104 (1993), Nr. 1, S. 2–34 37, 71, 241
- [ACH94] ALUR, Rajeev ; COURCOUBETIS, Costas ; HENZINGER, Thomas A.: The Observational Power of Clocks. In: *CONCUR '94: Proceedings of the Concurrency Theory*. London, UK : Springer-Verlag, 1994. – ISBN 3–540–58329–7, S. 162–177 85
- [AD90] ALUR, Rajeev ; DILL, David L.: Automata for Modeling Real-time Systems. In: *Proceedings of the Seventeenth International Colloquium on Automata, Languages and Programming* Bd. 443. New York, NY, USA : Springer-Verlag New York, Inc., 1990 (Lecture Notes in Computer Science (LNCS)). – ISBN 0–387–52826–1, S. 322–335 21, 53, 85
- [AD94] ALUR, Rajeev ; DILL, David L.: A Theory of Timed Automata. In: *Theoretical Computer Science* 126 (1994), Nr. 2, S. 183–235 32, 53, 54, 55, 70, 71
- [ADE⁺01] ALUR, Rajeev ; DANG, Thao ; ESPOSITO, Joel M. ; FIERRO, Rafael B. ; HUR, Yerang ; IVANCIC, Franjo ; KUMAR, Vijay ; LEE, Insup ; MISHRA, Pradyumna ; PAPPAS, George J. ; SOKOLSKY, Oleg: Hierarchical Hybrid Modeling of Embedded Systems. In: *EMSOFT '01: Proceedings of the First International Workshop on Embedded Software*. London, UK : Springer-Verlag, 2001. – ISBN 3–540–42673–6, S. 14–31 234
- [ADG98] ALLEN, Robert ; DOUENCE, Rémi ; GARLAN, David: Specifying and Analyzing Dynamic Software Architectures. In: *Lecture Notes in Computer Science* 1382 (1998), S. 21–37 237
- [AFH99] ALUR, Rajeev ; FIX, Limor ; HENZINGER, Thomas A.: Event-clock automata: a determinizable class of timed automata. In: *Theoretical Computer Science* 211

- (1999), Nr. 1-2, S. 253 – 273. – ISSN 0304–3975 240
- [AGLS01] ALUR, Rajeev ; GROSU, Radu ; LEE, Insup ; SOKOLSKY, Oleg: Compositional Refinement of Hierarchical Hybrid Systems. In: *Proceedings of the Fourth International Conference on Hybrid Systems: Computation and Control (HSCC'01)* Bd. 2034, Springer Verlag, 2001 (Lecture Notes in Computer Science), S. 33–48 234
- [AIK⁺03] ALUR, Rajeev ; IVANCIC, Franjo ; KIM, Jesung ; LEE, Insup ; SOKOLSKY, Oleg: Generating embedded software from hierarchical hybrid models. In: *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, ACM Press, 2003, S. 171–182 234
- [Alu92] ALUR, Rajeev: *Techniques for automatic verification of real-time systems*. Stanford, CA, USA, Stanford University, Diss., 1992 146
- [Alu99] ALUR, Rajeev: Timed Automata. In: HALBWACHS, Nicolas (Hrsg.) ; PELED, Doron (Hrsg.): *Proceedings of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy* Bd. 1633, Springer Verlag, 1999 (Lecture Notes in Computer Science (LNCS)), S. 8–22 24, 25, 32, 53, 54, 55, 75, 87
- [AM02] AGUIRRE, Nazareno ; MAIBAUM, Tom: A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In: *Automated Software Engineering, International Conference on 0* (2002), S. 271–274. – ISSN 1527–1366 237
- [AMP95] ASARIN, Eugene ; MALER, Oded ; PNUELI, Amir: Symbolic Controller Synthesis for Discrete and Timed Systems. In: *Hybrid Systems II*. London, UK : Springer-Verlag, 1995. – ISBN 3–540–60472–3, S. 1–20 241
- [AMPS98] ASARIN, Eugene ; MALER, Oded ; PNUELI, Amir ; SIFAKIS, Joseph: Controller Synthesis for Timed Automata. In: *Proceedings of the 5th IFAC Cconference on System Structure and Control (SSC'98)*, Elsevier Science, Juli 1998, S. 469–474 70, 71, 181, 241
- [Ang87] ANGLUIN, Dana: Learning regular sets from queries and counterexamples. In: *Information and Computation* 75 (1987), Nr. 2, S. 87–106. – ISSN 0890–5401 117, 119
- [ASK04] AGRAWAL, Aditya ; SIMON, Gyula ; KARSAI, Gabor: Semantic Translation of Simulink/Stateflow Models to Hybrid Automata Using Graph Transformations. In: *Electronic Notes Theoretic Computer Science* 109 (2004), S. 43–56. – ISSN 1571–0661 234
- [AT02] ALTISEN, Karine ; TRIPAKIS, Stavros: Tools for Controller Synthesis of Timed Systems. In: PETERSSON, Paul (Hrsg.) ; YI, Wang (Hrsg.): *Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, 2002, S. 1–12 241

- [BBHP04] BERKENKÖTTER, Kirsten ; BISANZ, Stefan ; HANNEMANN, Ulrich ; PELESKA, Jan: Executable HybridUML and its Application to Train Control Systems. In: EHRIG, Hartmut (Hrsg.) ; DAMM, Werner (Hrsg.) ; DESEL, Jörg (Hrsg.) ; GROSSE-RHODE, Martin (Hrsg.) ; REIF, Wolfgang (Hrsg.) ; SCHNIEDER, Eckehard (Hrsg.) ; WESTKÄMPER, Engelbert (Hrsg.): *Integration of Software Specification Techniques for Applications in Engineering* Bd. 3147, Springer Verlag, 2004 (Lecture Notes in Computer Science (LNCS)), S. 145–173 234
- [BBP⁺02] BENDER, K. ; BROY, M. ; PETER, I. ; PRETSCHNER, A. ; STAUNER, T.: Model based development of hybrid systems. In: *Modelling, Analysis, and Design of Hybrid Systems* Bd. 279. Springer Verlag, July 2002, S. 37–52 234
- [BCDW04] BRADBURY, Jeremy S. ; CORDY, James R. ; DINGEL, Juergen ; WERMELINGER, Michel: A survey of self-management in dynamic software architecture specifications. In: *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA : ACM, 2004. – ISBN 1–58113–989–6, S. 28–33 235
- [BCH⁺04] BEYER, Dirk ; CHLIPALA, Adam J. ; HENZINGER, Thomas A. ; JHALA, Ranjit ; MAJUMDAR, Rupak: The Blast Query Language for Software Verification. In: GIACOBAZZI, Roberto (Hrsg.): *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings* Bd. 3148, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–22791–1, S. 2–18 131
- [BCK08] BALDAN, Paolo ; CORRADINI, Andrea ; KÖNIG, Barbara: A framework for the verification of infinite-state graph transformation systems. In: *Information and Computation* 206 (2008), Nr. 7, S. 869–907. – ISSN 0890–5401 238
- [BCL⁺05] BROOKS, C. ; CATALDO, A. ; LEE, E. A. ; LIU, J. ; LIU, X. ; NEUENDORFFER, S. ; ZHENG, H. ; UNIVERSITY OF CALIFORNIA, BERKELEY (Hrsg.): *HyVisual: A Hybrid System Visual Modeler*. CA 94720: University of California, Berkeley, 2005. – Technical Memorandum UCB/ERL M05/24 234
- [BDL04] BEHRMANN, Gerd ; DAVID, Alexandre ; LARSEN, Kim G.: A Tutorial on Uppaal. In: BERNARDO, Marco (Hrsg.) ; CORRADINI, Flavio (Hrsg.): *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*, Springer-Verlag, September 2004 (LNCS 3185), S. 200–236 23
- [Bec08] BECKER, Steffen: *Coupled Model Transformations for QoS Enabled Component-Based Software Design*, Department of Computer Science, University of Oldenburg, Diss., March 2008 200
- [Ber06] BERG, Therese: *Regular Inference for Reactive Systems*, it, Licentiate thesis, April 2006. – 132 S. 127, 239
- [Bey02] BEYER, Dirk: *Formale Verifikation von Realzeit-Systemen mittels Cottbus Timed Automata*. Mensch & Buch Verlag, Berlin, 2002. – ISBN 3–89820–450–2 65, 223,

238

- [BG08] BECKER, Basil ; GIESE, Holger: On Safe Service-Oriented Real-Time Coordination for Autonomous Vehicles. In: *In Proceedings of 11th International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC)*, IEEE Computer Society Press, 5 2008. – ISBN 978-0-7695-3132-8, S. 203–210 238
- [BGG004a] BURMESTER, Sven ; GEHRKE, Matthias ; GIESE, Holger ; OBERTHÜR, Simon: Making Mechatronic Agents Resource-aware in order to Enable Safe Dynamic Resource Allocation. In: GEORGIO, B. (Hrsg.): *Proceedings of Fourth ACM International Conference on Embedded Software 2004 (EMSOFT 2004)*, Pisa, Italy, ACM Press, September 2004, S. 175–183 184, 186, 193
- [BGG004b] BURMESTER, Sven ; GIESE, Holger ; GAMBUTTA, Alfonso ; OBERSCHELP, Oliver: Partitioning and Modular Code Synthesis for Reconfigurable Mechatronic Software Components. In: BOBEANU, C. (Hrsg.): *Proceedings of European Simulation and Modelling Conference (ESMc'2004)*, Paris, France, EOROSIS Publications, October 2004, S. 66–73 196
- [BGH05a] BURMESTER, Sven ; GIESE, Holger ; HIRSCH, Martin: Syntax and Semantics of Hybrid Components / Software Engineering Group, University of Paderborn. 2005 (tr-ri-05-264). – Forschungsbericht 43, 238
- [BGH⁺05b] BURMESTER, Sven ; GIESE, Holger ; HIRSCH, Martin ; SCHILLING, Daniela ; TICHY, Matthias: The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis, Missouri, USA, ACM Press, May 2005, S. 670–671 201
- [BGK05] BURMESTER, Sven ; GIESE, Holger ; KLEIN, Florian: Synthesis of Parameterized UML Real-Time Patterns from Multiple Parameterized Real-Timed Scenarios. In: BORDELEAU, Francis (Hrsg.) ; LEUE, Stefan (Hrsg.) ; SYSTÄ, Tarja (Hrsg.): *Scenarios: Models, Algorithms and Tools* Bd. 3371. Springer Verlag, April 2005, S. 193–211 10, 201, 234
- [BGO06] BURMESTER, Sven ; GIESE, Holger ; OBERSCHELP, Oliver: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: BRAZ, JOSÉ (Hrsg.) ; ARAÚJO, HELDER (Hrsg.) ; VIEIRA, ALVES (Hrsg.) ; ENCARNAÇÃO, BRUNO (Hrsg.): *Informatics in Control, Automation and Robotics I*. Springer Netherlands, 2006. – ISBN 978-1-4020-4543-1, S. 281–288 234
- [BGST05] BURMESTER, Sven ; GIESE, Holger ; SEIBEL, Andreas ; TICHY, Matthias: Worst-Case Execution Time Optimization of Story Patterns for Hard Real-Time Systems. In: *Proceedings of the 3rd International Fujaba Days 2005, Paderborn, Germany*, 2005, S. 71–78 234

- [BGT05] BURMESTER, Sven ; GIESE, Holger ; TICHY, Matthias: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In: ASSMANN, Uwe (Hrsg.) ; RENSINK, Arend (Hrsg.) ; AKSIT, Mehmet (Hrsg.): *Model Driven Architecture: Foundations and Applications* Bd. 3599, Springer Verlag, August 2005 (Lecture Notes in Computer Science), S. 47–61 234
- [BJLS03] BERG, Therese ; JONSSON, Bengt ; LEUCKER, Martin ; SAKSENA, Mayank: Insights to Angluin’s Learning. In: *Proceedings of the International Workshop on Software Verification and Validation (SVV 2003)* Bd. 118, 2003 (Electronic Notes in Theoretical Computer Science), S. 3–18 239
- [BJR06] BERG, Therese ; JONSSON, Bengt ; RAFFELT, Harald: Regular Inference for State Machines with Parameters. In: BARESI, Luciano (Hrsg.) ; HECKEL, Reiko (Hrsg.): *Fundamental Approaches to Software Engineering* Bd. 3922. Springer Berlin / Heidelberg, 2006, S. 107–121 97, 239
- [BK06] BONAKDARPOUR, Borzoo ; KULKARNI, Sandeep S.: Automated Incremental Synthesis of Timed Automata. In: BRIM, Lubos (Hrsg.) ; HAVERKORT, Boudevijn R. (Hrsg.) ; LEUCKER, Martin (Hrsg.) ; POL, Jaco van d. (Hrsg.): *Formal Methods: Applications and Technology, 11th International Workshop, FMICS 2006 and 5th International Workshop PDMC 2006, Bonn, Germany, August 26-27, and August 31, 2006, Revised Selected Papers* Bd. 4346, Springer-Verlag Berlin Heidelberg, 2006 (Lecture Notes in Computer Science (LNCS)), S. 261–276 241
- [BK08] BAIER, Christel ; KATOEN, Joost-Pieter: *Principles of Model Checking*. MIT Press, 2008 36, 74, 86, 104
- [BN03] BROEKMAN, Bart ; NOTENBOOM, Edwin: *Testing Embedded Software*. Addison-Wesley, 2003 98
- [BÖ10] BORONAT, A. ; ÖLVECZKY, P. C.: Formal Real-Time Model Transformations in MOMENT2. In: *Proc. of the 13th Intern. Conf. on Fundamental Approaches to Software Engineering, FASE 2010*, 2010, S. 29–43 237
- [BPG03] BARRINGER, H. ; PASAREANU, Corina S. ; GIANNAKOPOLOU, D.: Proof Rules for Automated Compositional Verification through Learning. In: *International Workshop on Specification and Verification of Component Based Systems, Finland*, 2003, S. 14–21 239
- [Bur06] BURMESTER, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems*, Software Engineering Group, University of Paderborn, Diss., 8 2006 2, 15, 45, 65, 70, 136, 175, 176, 181, 183, 195, 196, 229, 238, 246, 273
- [But05] BUTTAZZO, Giorgio C.: *Real-Time Systems Series*. Bd. 23: *Hard Real-time Computing Systems: Predictable Scheduling Algorithms and Applications*. 2. Springer, 2005. – ISBN 978-0-387-23137-2 70, 186

- [BY03] BENGTTSSON, Johan ; YI, Wang: Timed Automata: Semantics, Algorithms and Tools. In: *Lectures on Concurrency and Petri Nets*, 2003, S. 87–124 24, 25, 75, 86, 87, 163
- [CDH⁺00] CORBETT, James C. ; DWYER, Matthew B. ; HATCLIFF, John ; LAUBACH, Shawn ; PĂSĂREANU, Corina S. ; ROBBY ; ZHENG, Hongjun: Bandera: extracting finite-state models from Java source code. In: *ICSE '00: Proceedings of the 22nd international conference on Software engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–206–9, S. 439–448 100
- [CGJ⁺00] CLARKE, Edmund ; GRUMBERG, Orna ; JHA, Somesh ; LU, Yuan ; VEITH, Helmut: Counterexample-Guided Abstraction Refinement. In: EMERSON, E. (Hrsg.) ; SISTLA, A. (Hrsg.): *Computer Aided Verification* Bd. 1855. Springer Berlin / Heidelberg, 2000, S. 154–169 238
- [CGJ⁺03] CLARKE, Edmund ; GRUMBERG, Orna ; JHA, Somesh ; LU, Yuan ; VEITH, Helmut: Counterexample-guided abstraction refinement for symbolic model checking. In: *J. ACM* 50 (2003), Nr. 5, S. 752–794. – ISSN 0004–5411 241
- [CGP00] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking*. MIT Press, 2000 26, 32, 36, 40, 41, 70, 80, 104
- [CGP03] COBLEIGH, Jamieson M. ; GIANNAKOPOULOU, Dimitra ; PASAREANU, Corina S.: Learning Assumptions for Compositional Verification. In: *Tools and Algorithms for the Construction and Analysis of Systems* Bd. Volume 2619/2003, Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–00898–9, S. 331–346 239
- [Cho78] CHOW, T. S.: Testing Software Design Modeled by Finite-State Machines. In: *IEEE Transactions on Software Engineering* 4 (1978), Nr. 3, S. 178–187. – ISSN 0098–5589 117, 118, 127
- [CKL04] CLARKE, Edmund M. ; KROENING, Daniel ; LERDA, Flavio: A Tool for Checking ANSI-C Programs. In: JENSEN, Kurt (Hrsg.) ; PODELSKI, Andreas (Hrsg.): *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Procee* Bd. 2988, Springer, 2004 (Lecture Notes in Computer Science). – ISBN 3–540–21299–X, S. 168–176 130
- [CKSY05] CLARKE, Edmund ; KROENING, Daniel ; SHARYGINA, Natasha ; YORAV, Karen: SATABS: SAT-based Predicate Abstraction for ANSI-C. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005)* Bd. 3440, Springer Verlag, 2005 (Lecture Notes in Computer Science). – ISBN 3–540–25333–5, S. 570–574 130
- [CLG⁺09] CHENG, Betty H. C. ; LEMOS, Rogério de ; GIESE, Holger ; INVERARDI, Paola ; MAGEE, Jeff ; ANDERSSON, Jesper ; BECKER, Basil ; BENCOMO, Nelly ; BRUN, Yuriy ; CUKIC, Bojan ; SERUGENDO, Giovanna Di M. ; DUSTDAR, Schahram ;

- FINKELSTEIN, Anthony ; GACEK, Cristina ; GEIHS, Kurt ; GRASSI, Vincenzo ; KARSAI, Gabor ; KIENLE, Holger M. ; KRAMER, Jeff ; LITOIU, Marin ; MALEK, Sam ; MIRANDOLA, Raffaella ; MÜLLER, Hausi A. ; PARK, Sooyong ; SHAW, Mary ; TICHY, Matthias ; TIVOLI, Massimo ; WEYNS, Danny ; WHITTLE, Jon: *Software Engineering for Self-Adaptive Systems: A Research Roadmap*. In: *Software Engineering for Self-Adaptive Systems* Bd. 5525, Springer Berlin / Heidelberg, 2009 (Lecture Notes in Computer Science). – ISBN 978–3–642–02160–2, S. 1–26 235
- [CMP94] CHANG, Edward Y. ; MANNA, Zohar ; PNUELI, Amir: *Compositional Verification of Real-time Systems*. In: *Proceedings of the 9th Annual IEEE Symposium on Logic in Computer Science, 4-7 July 1994, Paris, France, 1994*, S. 458–465 241
- [CPT99] CANAL, Carlos ; PIMENTEL, Ernesto ; TROYA, José M.: *Specification and Refinement of Dynamic Software Architectures*. In: *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 1999. – ISBN 0–7923–8453–9, S. 107–126 237
- [Crn02] CRNKOVIC, Ivica ; LARSSON, Magnus (Hrsg.): *Building Reliable Component-Based Software Systems*. Norwood, MA, USA : Artech House, Inc., 2002. – ISBN 1580533272 1, 3, 10
- [Dij76] DIJKSTRA, Edsger W.: *A Discipline of Programming*. Prentice-Hall, 1976 139
- [Dil89] DILL, David L.: *Timing Assumptions and Verification of Finite-State Concurrent Systems*. In: *Automatic Verification Methods for Finite State Systems, International Workshop, Grenoble, France, June 12-14, 1989, Proceedings* Bd. 407/1990, Springer-Verlag Berlin / Heidelberg, 1989 (Lecture Notes in Computer Science (LNCS)). – ISBN 0–387–52148–8, S. 197–212 26
- [DKU06] DUARTE, Lucio ; KRAMER, Jeff ; UCHITEL, Sebastian: *Model Extraction Using Context Information*. In: NIERSTRASZ, Oscar (Hrsg.) ; WHITTLE, Jon (Hrsg.) ; HAREL, David (Hrsg.) ; REGGIO, Gianna (Hrsg.): *Model Driven Engineering Languages and Systems* Bd. 4199. Springer Berlin / Heidelberg, 2006, S. 380–394 100
- [DMY02] DAVID, Alexandre ; MÖLLER, M. O. ; YI, Wang: *Formal Verification of UML Statecharts with Real-Time Extensions*. In: *Fundamental Approaches to Software Engineering* Bd. 2306. Springer Berlin / Heidelberg, 2002, S. 208–241 19
- [Dou99] DOUGLASS, Bruce P.: *Doing hard time: developing real-time systems with UML, objects, frameworks, and patterns*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1999. – ISBN 0–201–49837–5 176
- [Dou02] DOUGLASS, Bruce P.: *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0201699567 70, 176, 177, 178

- [Dun02] DUNN, William R.: *Practical Design of Safety-Critical Computer Systems*. Reliability Press, 2002 100
- [EFR08] EUSGELD, Irene (Hrsg.) ; FREILING, Felix C. (Hrsg.) ; REUSSNER, Ralf (Hrsg.): *Dependability Metrics: Advanced Lectures [result from a Dagstuhl seminar, October 30 - November 1, 2005]*. Bd. 4909. Springer, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–68946–1 200
- [EG⁺06] ELKIND, Edith ; GENEST, Blaise ; ; PELED, Doron ; AND, Hongyang Q.: Grey-Box Checking. In: *Formal Techniques for Networked and Distributed Systems - FORTE 2006* Bd. Volume 4229/2006, Springer Berlin / Heidelberg, 2006. – ISBN 978–3–540–46219–4, S. 420–435 239, 240
- [EW92] ENDLER, M. ; WEI, J.: Programming generic dynamic reconfigurations for distributed applications. In: *International Workshop on Configurable Distributed Systems*, 1992, S. 68–79 237
- [FGK⁺04] FRANK, Ursula ; GIESE, Holger ; KLEIN, Florian ; OBERSCHELP, Oliver ; SCHMIDT, Andreas ; SCHULZ, Bernd ; VÖCKING, Henner ; WITTING, Katrin ; GAUSEMEIER, Jürgen (Hrsg.): *Selbstoptimierende Systeme des Maschinenbaus: Definitionen und Konzepte*. HNI-Verlagsschriftenreihe, Band 155, Paderborn, Germany, 2004 14
- [Fid96] FIDGE, Colin: Fundamentals of Distributed System Observation. In: *IEEE Software* 13 (1996), Nr. 6, S. 77–83. – ISSN 0740–7459 183
- [FJW97] FRIESEN, Viktor ; JÄHNICHEN, Stefan ; WEBER, Matthias: Specification of software controlling a discrete-continuous environment. In: *ICSE '97: Proceedings of the 19th international conference on Software engineering*. New York, NY, USA : ACM, 1997. – ISBN 0–89791–914–9, S. 315–325 234
- [FNW98] FRIESEN, Viktor ; NORDWIG, André ; WEBER, Matthias: Object-Oriented Specification of Hybrid Systems Using UMLh and ZimOO. In: *Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation, Berlin, Germany* Bd. 1493, Springer Verlag, 1998 (Lecture Notes in Computer Science (LNCS)), S. 328–346 234
- [Föl05] FÖLLINGER, Otto: *Regelungstechnik. Einführung in die Methoden und ihre Anwendung*. Hüthig, 2005 42
- [FPW98] FRANKLIN, G. F. ; POWELL, J. D. ; WORKMAN, M.: *Digital Control of Dynamic Systems*. 3. Addison-Wesley Longman Publishing Co., Inc., 1998 136
- [GAO95] GARLAN, David ; ALLEN, Robert ; OCKERBLOOM, John: Architectural Mismatch: Why Reuse Is So Hard. In: *IEEE Software* 12 (1995), Nr. 6, S. 17–26. – ISSN 0740–7459 2
- [Gar03] GARLAN, David: Formal Modeling and Analysis of Software Architecture: Components, Connectors, and Events. In: *Formal Methods for Software Architectures*

- Bd. 2804, Springer Berlin / Heidelberg, 2003 (Lecture Notes in Computer Science). – ISBN 978-3-540-20083-3, S. 1–24 2
- [GB03] GIESE, Holger ; BURMESTER, Sven: Real-Time Statechart Semantics / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Germany, 6 2003 (tr-ri-03-239). – Forschungsbericht 19, 21, 23, 24, 59, 61, 256, 261, 264, 265, 267
- [GBSO04] GIESE, Holger ; BURMESTER, Sven ; SCHÄFER, Wilhelm ; OBERSCHELP, Oliver: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: *Proceedings of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, ACM Press, November 2004, S. 179–188 234
- [GGR08] GEIST, Stephanie ; GROMOV, Dmitry ; RAISCH, Jörg: Timed Discrete Event Control of Parallel Production Lines with Continuous Outputs. In: *Discrete Event Dynamic Systems* 18 (2008), Nr. 2, S. 241–262. – ISSN 0924-6703 241
- [GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns*. Boston, MA : Addison-Wesley, 1995. – ISBN 0201633612 58
- [GHK00] GADDUCCI, Fabio ; HECKEL, Reiko ; KOCH, Manuel: A Fully Abstract Model for Graph-Interpreted Temporal Logic. In: *Theory and Application of Graph Transformation* Bd. 1764. Springer Berlin / Heidelberg, 2000, S. 310–322 238
- [Gie00] GIESE, Holger: Contract-Based Component System Design. In: *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8, 4-7 January, 2000, Maui, Hawaii*. Washington, DC, USA : IEEE Computer Society, 2000. – ISBN 0-7695-0493-0, S. 8051–8060 242
- [Gie03] GIESE, Holger: A Formal Calculus for the Compositional Pattern-Based Design of Correct Real-Time Systems. / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Deutschland, July 2003 (tr-ri-03-240). – Forschungsbericht 10, 70, 79, 80
- [Gie07] GIESE, Holger: Modeling and Verification of Cooperative Self-adaptive Mechatronic Systems. In: *Reliable Systems on Unreliable Networked Platforms* Bd. 4322, Springer Berlin / Heidelberg, 2007 (Lecture Notes in Computer Science). – ISBN 978-3-540-71155-1, S. 258–280 65, 85, 237
- [GJL04] GRINCHTEIN, Olga ; JONSSON, Bengt ; LEUCKER, Martin: Learning of Event-Recording Automata. In: LAKHNECH, Yassine (Hrsg.) ; YOVINE, Sergio (Hrsg.): *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems* Bd. 3253. Springer Berlin / Heidelberg, 2004, S. 77–82 240
- [GJM91] GHEZZI, Carlo ; JAZAYERI, Mehdi ; MANDRIOLI, Dino: *Fundamentals of software engineering*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1991. – ISBN 0-13-820432 1

- [GJP06] GRINCHTEIN, Olga ; JONSSON, Bengt ; PETTERSSON, Paul: Inference of Event-Recording Automata Using Timed Decision Trees. In: *CONCUR 2006 Concurrency Theory* Bd. Volume 4137/2006, Springer Berlin / Heidelberg, 2006. – ISBN 978-3-540-37376-6, S. 435–449 240
- [GKS00] GROSU, Radu ; KRÜGER, Ingolf ; STAUNER, Thomas: Hybrid Sequence Charts. In: *Proceedings of the 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 2000)*, IEEE Computer Society, 2000. – ISBN 0-7695-0607-0, S. 104–112 234
- [Gol78] GOLD, E. M.: Complexity of Automaton Identification from Given Data. In: *Information and Control* 37 (1978), Nr. 3, S. 302–320 119
- [Gom00] GOMAA, Hassan: *Designing Concurrent, Distributed, and Real-Time Applications with Uml*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2000. – ISBN 0201657937 176
- [GP05] GIANNAKOPOULOU, Dimitra ; PASAREANU, Corina S.: Learning-Based Assume-Guarantee Verification (Tool Paper). In: GODEFROID, Patrice (Hrsg.): *Model Checking Software, 12th International SPIN Workshop, San Francisco, CA, USA, August 22-24, 2005, Proceedings* Bd. 3639, Springer, 2005 (Lecture Notes in Computer Science). – ISBN 3-540-28195-9, S. 282–287 239
- [GPY02] GROCE, Alex ; PELED, Doron ; YANNAKAKIS, Mihalis: Adaptive Model Checking. In: *Tools and Algorithms for the Construction and Analysis of Systems* Bd. Volume 2280/2002, Springer Berlin / Heidelberg, 2002. – ISBN 978-3-540-43419-1, S. 269–301 127, 239
- [Gra72] GRAUPE, Daniel: *Identification of Systems*. Krieger Pub. Co., 1972. – ISBN 0882753592 136
- [Gre10] GREENYER, Joel: Synthesizing Modal Sequence Diagram Specifications with Uppaal-Tiga / Software Engineering Group, University of Paderborn. 2010 (trri-10-310). – Forschungsbericht 247
- [GRPS02] GROŠE-RHODE, Martin ; PRESICCE, Francesco P. ; SIMEONI, Marta: Formal software specification with refinements and modules of typed graph transformation systems. In: *J. Comput. Syst. Sci.* 64 (2002), Nr. 2, S. 171–218. – ISSN 0022-0000 65, 85, 86, 237
- [GS03] GÖSSLER, Gregor ; SIFAKIS, Joseph: Component-Based Construction of Deadlock-Free Systems. In: PANDYA, Paritosh K. (Hrsg.) ; RADHAKRISHNAN, Jaikumar (Hrsg.): *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science* Bd. 2914, Springer Berlin / Heidelberg, 2003, S. 420–433 139
- [GSB98] GROSU, Radu ; STAUNER, Thomas ; BROY, Manfred: A Modular Visual Model for Hybrid Systems. In: *Formal Techniques in Real Time and Fault Tolerant Systems*

- (*FTRTFT'98*), Springer Verlag, 1998. – ISBN 3–540–65003–2, S. 75–91 234
- [GSG⁺09] GAUSEMEIER, Jürgen ; SCHÄFER, Wilhelm ; GREENYER, Joel ; KAHL, Sascha ; POOK, Sebastian ; RIEKE, Jan: Management of Cross-Domain Model Consistency During the Development of Advanced Mechatronic Systems. In: BERGENDAHL, Margareta N. (Hrsg.) ; GRIMHEDEN, Martin (Hrsg.) ; LEIFER, Larry (Hrsg.): *Proceedings of the 17th International Conference on Engineering Design (ICED'09)* Bd. 6. University of Stanford, CA, USA : Design Society, August 2009, S. 1–12 247
- [GTB⁺03] GIESE, Holger ; TICHY, Matthias ; BURMESTER, Sven ; SCHÄFER, Wilhelm ; FLAKE, Stephan: Towards the Compositional Verification of Real-Time UML Designs. In: *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-11)*, ACM Press, September 2003, S. 38–47 3, 17, 18, 35, 37, 38, 65, 70, 238
- [GV06] GIESE, Holger ; VILBIG, Alexander: Separation of Non-Orthogonal Concerns in Software Architecture and Design. In: *Software and System Modeling (SoSyM)* 5 (2006), June, Nr. 2, S. 136 – 169 139, 242
- [GVH03] GYAPAY, Szilvia ; VARRÓ, Dániel ; HECKEL, Reiko: Graph transformation with time. In: *Fundamenta Informaticae* 58 (2003), Nr. 1, S. 1–22. – ISSN 0169–2968 237
- [Har87] HAREL, David: Statecharts: A visual formalism for complex systems. In: *Sci. Comput. Program.* 8 (1987), Nr. 3, S. 231–274. – ISSN 0167–6423 57, 59
- [Hen92] HENZINGER, Thomas A.: Sooner is Safer than Later. In: *Information Processing Letters* 43 (1992), Nr. 3, S. 135–141. – ISSN 0020–0190 145
- [Hen96] HENZINGER, T. A.: The theory of hybrid automata. In: *Logic in Computer Science, Symposium on 0* (1996), S. 278. – ISSN 1043–6871 42, 43
- [Hen00] HENZINGER, Thomas A.: Masaccio: A Formal Model for Embedded Components. In: *Proceedings of the First IFIP International Conference on Theoretical Computer Science (TCS), Lecture Notes in Computer Science 1872*, Springer-Verlag, 2000, 2000, S. 549–563 234
- [HHK01] HENZINGER, Thomas A. ; HOROWITZ, Benjamin ; KIRSCH, Christoph M.: Giotto: A Time-triggered Language for Embedded Programming. In: *Proceedings of the IEEE 91:84-99, 2003. A preliminary version appeared in the Proceedings of the First International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2211*, Springer-Verlag, 2001, S. 166–184 234
- [HIM98] HIRSCH, Dan ; INVERARDI, Paola ; MONTANARI, Ugo: Graph grammars and constraint solving for software architecture styles. In: *ISAW '98: Proceedings of the third international workshop on Software architecture*. New York, NY, USA :

- ACM, 1998. – ISBN 1–58113–081–3, S. 69–72 237
- [Hir08] HIRSCH, Martin: *Modell-basierte Verifikation von vernetzten mechatronischen Systemen*, Software Engineering Group, University of Paderborn, Diss., September 2008 3, 4, 5, 27, 28, 31, 32, 45, 53, 55, 56, 57, 59, 81, 86, 87, 258
- [HJMS03] HENZINGER, Thomas ; JHALA, Ranjit ; MAJUMDAR, Rupak ; SUTRE, Grégoire: Software Verification with BLAST. In: BALL, Thomas (Hrsg.) ; RAJAMANI, Sriram (Hrsg.): *Model Checking Software* Bd. 2648. Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–40117–9, S. 624–624 129
- [HKK04] HARDUNG, Bernd ; KÖLZOW, Thorsten ; KRÜGER, Andreas: Reuse of software in distributed embedded automotive systems. In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–860–1, S. 203–210 1, 3, 10
- [HKP05] HAREL, David ; KUGLER, Hillel ; PNUELI, Amir: Synthesis Revisited: Generating Statechart Models from Scenario-Based Requirements. In: *Formal Methods in Software and Systems Modeling*. Berlin/Heidelberg, Germany : Springer-Verlag, 2005, S. 309–324 10, 173
- [HKPV95] HENZINGER, Thomas A. ; KOPKE, Peter W. ; PURI, Anuj ; VARAIYA, Pravin: What's decidable about hybrid automata? In: *STOC '95: Proceedings of the twenty-seventh annual ACM symposium on Theory of computing*. New York, NY, USA : ACM, 1995. – ISBN 0–89791–718–9, S. 373–382 45
- [HKSP02] HENZINGER, Thomas A. ; KIRSCH, Christoph M. ; SANVIDO, Marco A. ; PREE, Wolfgang: From Control Models to Real-Time Code Using Giotto. In: *IEEE Control Systems Magazine* 23(1):50-64, 2003. A preliminary report on this work appeared in C.M. Kirsch, M.A.A. Sanvido, T.A. Henzinger, and W. Pree, *A Giotto-based helicopter control system, Proceedings of the Second International Workshop on Embedded Software (EMSOFT), Lecture Notes in Computer Science 2491, Springer-Verlag, 2002*, 2002, S. 46–60 234
- [HLL⁺03] HYLANDS, Christopher ; LEE, Edward ; LIU, Jie ; LIU, Xiaojun ; NEUENDORFFER, Stephen ; XIONG, Yuhong ; ZHAO, Yang ; ZHENG, Haiyang: Overview of the Ptolemy Project / Department of Electrical Engineering and Computer Science, University of California, Berkeley. 2003 (UCB/ERL M03/25). – Forschungsbericht 234
- [HN96] HAREL, David ; NAAMAD, Amnon: The STATEMATE semantics of statecharts. In: *ACM Transaction on Software Engineering Methodology* 5 (1996), Nr. 4, S. 293–333. – ISSN 1049–331X 59
- [HNS03a] HUNGAR, Hardi ; NIESE, Oliver ; STEFFEN, Bernhard: Domain-Specific Optimization in Automata Learning. In: *Computer Aided Verification* Bd. 2725. Springer Berlin / Heidelberg, 2003, S. 315–327 100, 239

- [HNS03b] HUNGAR, Hardi ; NIESE, Oliver ; STEFFEN, Bernhard: Domain-Specific Optimization in Automata Learning. In: *Computer Aided Verification* Bd. Volume 2725/2003, Springer Berlin / Heidelberg, 2003. – ISBN 978–3–540–40524–5, S. 315–327 97, 118, 239, 240
- [Hoa85] HOARE, C. A. R.: *Communicating Sequential Processes*. Prentice-Hall International, 1985 (Series in Computer Science) 102
- [Hon98] HONEKAMP, Uwe: *IPANEMA - Verteilte Echtzeit-Informationsverarbeitung in mechatronischen Systemen*, University of Paderborn, Diss., 1998 176, 195, 196
- [HPSS87] HAREL, D. ; PNUELI, A. ; SCHMIDT, J. P. ; SHERMAN, R.: On the Formal Semantics of Statecharts. In: *Proceedings of the 2nd IEEE Symposium Logic in Computer Science (LICS 1987)*, IEEE, 1987, S. 54–64 57
- [HS99] HOLZMANN, Gerard J. ; SMITH, Margaret H.: A practical method for verifying event-driven software. In: *ICSE '99: Proceedings of the 21st international conference on Software engineering*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1999. – ISBN 1–58113–074–0, S. 597–607 100
- [HSE10] HEINZEMANN, C. ; SUCK, J. ; ECKARDT, T.: Reachability Analysis on Timed Graph Transformation Systems. In: *Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)*, 2010 212
- [HSJZ10] HEINZEMANN, Christian ; SUCK, Julian ; JUBEH, Ruben ; ZÜNDORF, Albert: Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study. In: GORP, Pieter V. (Hrsg.) ; MAZANEK, Steffen (Hrsg.) ; RENSINK, Arend (Hrsg.): *Transformation Tool Contest*. Malaga, 2010, S. 1–15 63, 199, 211, 246
- [HT04] HECKEL, Reiko ; THÖNE, Sebastian: Behavioral Refinement of Graph Transformation-Based Models. In: *Proceedings of the ICGT 2004 Workshop on Software Evolution through Transformations (SETra 04)*, Electronic Notes in Theoretical Computer Science, 2004, S. 139–151 65, 84, 91, 237
- [HT05] HECKEL, Reiko ; THÖNE, Sebastian: Behavioral Refinement of Graph Transformation-Based Models. In: *Electronic Notes in Theoretical Computer Science* 127 (2005), Nr. 3, S. 101–111 84, 85
- [Ise92] ISERMANN, Rolf: *Identifikation dynamischer Systeme*. Springer-Verlag, 1992 97, 136
- [IWFY00] INVERARDI, Paola ; WOLF, Alexander L. ; YANKELEVICH, Daniel: Static checking of system behaviors using derived component assumptions. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9 (2000), Nr. 3, S. 239–272. – ISSN 1049–331X 2
- [JLS00] JENSEN, Henrik E. ; LARSEN, Kim G. ; SKOU, Arne: Scaling up Uppaal. In: *Formal Techniques in Real-Time and Fault-Tolerant Systems* Bd. 1926, Springer

- Berlin / Heidelberg, 2000 (Lecture Notes in Computer Science). – ISBN 978–3–540–41055–3, S. 641–678 23, 37, 39, 41, 65, 85, 99, 131
- [Ken02] KENT, Stuart: Model Driven Engineering. In: BUTLER, Michael (Hrsg.) ; PETRE, Luigia (Hrsg.) ; SERE, Kaisa (Hrsg.): *Integrated Formal Methods* Bd. 2335. Springer Berlin / Heidelberg, 2002, S. 286–298 2
- [KM98] KRAMER, Jeff ; MAGEE, Jeff: Analysing Dynamic Change in Software Architectures: A Case Study. In: *CDS '98: Proceedings of the International Conference on Configurable Distributed Systems*. Washington, DC, USA : IEEE Computer Society, 1998. – ISBN 0–8186–8451–8, S. 91 45, 100
- [Kop97] KOPETZ, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. 1. Springer, 1997. – ISBN 978–0792398943 16, 70
- [Kur94] KURSHAN, Robert P.: *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton, NJ, USA : Princeton University Press, 1994. – ISBN 0–691–03436–2 241
- [LAK92] LAPRIE, J. C. C. (Hrsg.) ; AVIZIENIS, A. (Hrsg.) ; KOPETZ, H. (Hrsg.): *Dependability: Basic Concepts and Terminology*. Secaucus, NJ, USA : Springer-Verlag New York, Inc., 1992. – ISBN 0387822968 1
- [Lam77] LAMPORT, Leslie: Proving the Correctness of Multiprocess Programs. In: *IEEE Transactions on Software Engineering* SE-3 (1977), March, Nr. 2, S. 125–143. – ISSN 0098–5589 145
- [Lam09] LAMSWEERDE, Axel: Reasoning About Alternative Requirements Options. (2009), S. 380–397. ISBN 978–3–642–02462–7 201
- [LGS06a] LI, Keqin ; GROZ, Roland ; SHAHBAZ, Muzammil: Integration Testing of Components Guided by Incremental State Machine Learning. In: *TAIC-PART '06: Proceedings of the Testing: Academic & Industrial Conference on Practice And Research Techniques*. Washington, DC, USA : IEEE Computer Society, 2006. – ISBN 0–7695–2672–1, S. 59–70 239
- [LGS06b] LI, Keqin ; GROZ, Roland ; SHAHBAZ, Muzammil: Integration Testing of Distributed Components Based on Learning Parameterized I/O Models. In: NAJM, Elie (Hrsg.) ; PRADAT-PEYRE, Jean (Hrsg.) ; DONZEAU-GOUGE, Véronique (Hrsg.): *Formal Techniques for Networked and Distributed Systems - FORTE 2006* Bd. 4229. Springer Berlin / Heidelberg, 2006, S. 436–450 239
- [Lju98] LJUNG, Lennart: *System Identification: Theory for the User (2nd Edition)*. Prentice Hall PTR, 1998. – ISBN 0136566952 136, 137
- [Lju10] LJUNG, Lennart: Perspectives on system identification. In: *Annual Reviews in Control* 34 (2010), Nr. 1, S. 1–12. – ISSN 1367–5788 136
- [LM98] LE MÉTAYER, Daniel: Describing Software Architecture Styles Using Graph Grammars. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 7, S.

521–533. – ISSN 0098–5589 237

- [LNA99] LIND-NIELSEN, Jørn ; ANDERSEN, Henrik R.: Stepwise CTL Model Checking of State/Event Systems. In: *CAV '99: Proceedings of the 11th International Conference on Computer Aided Verification*. London, UK : Springer-Verlag, 1999. – ISBN 3–540–66202–2, S. 316–327 241
- [LO08] LICHTER, Hermann-Simon ; OBERTHÜR, Simon: Schedulability Criteria and Analysis for Dynamic and Flexible Resource Management. In: *Electron. Notes Theor. Comput. Sci.* 200 (2008), Nr. 2, S. 3–19. – ISSN 1571–0661 186, 187
- [LPY97] LARSEN, Kim G. ; PETTERSSON, Paul ; YI, Wang: Uppaal in a Nutshell. In: *International Journal on Software Tools for Technology Transfer* 1 (1997), Oktober, Nr. 1-2, S. 134–152 254
- [Mil89] MILNER, Robin: *Communication and Concurrency*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1989 (Prentice Hall International Series in Computer Science) 23, 24, 139, 155
- [MJS⁺00] MÜLLER, Hausi A. ; JAHNKE, Jens H. ; SMITH, Dennis B. ; STOREY, Margaret-Anne ; TILLEY, Scott R. ; WONG, Kenny: Reverse engineering: a roadmap. In: *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*. New York, NY, USA : ACM, 2000. – ISBN 1–58113–253–0, S. 47–60 100
- [MK96] MAGEE, Jeff ; KRAMER, Jeff: Dynamic structure in software architectures. In: *SIGSOFT Software Engineering Notes* 21 (1996), Nr. 6, S. 3–14. – ISSN 0163–5948 237
- [MNRS04] MARGARIA, Tiziana ; NIESE, Oliver ; RAFFELT, H. ; STEFFEN, Bernhard: Efficient test-based model generation for legacy reactive systems. In: *HLDVT '04: Proceedings of the High-Level Design Validation and Test Workshop, 2004. Ninth IEEE International*. Washington, DC, USA : IEEE Computer Society, 2004. – ISBN 0–7803–8714–7, S. 95–100 239
- [MPS95] MALER, Oded ; PNUELI, Amir ; SIFAKIS, Joseph: On the Synthesis of Discrete Controllers for Timed Systems (An Extended Abstract). In: MAYR, Ernst W. (Hrsg.) ; PUECH, Claude (Hrsg.): *Proceedings of the 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS 95), Munich, Germany, March 2-4, 1995* Bd. 900, Springer Verlag, 1995 (Lecture Notes in Computer Science (LNCS)), S. 229–242 70, 71, 241
- [MPW92] MILNER, Robin ; PARROW, Joachim ; WALKER, David: A calculus of mobile processes. In: *Information and Computation* 100 (1992), Nr. 1, S. 1–77. – ISSN 0890–5401 237
- [MRSL07] MARGARIA, Tiziana ; RAFFELT, Harald ; STEFFEN, Bernhard ; LEUCKER, Martin: The LearnLib in FMICS-jETI. In: *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems*. Washington,

- DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2895–3, S. 340–352
239
- [Obj05a] OBJECT MANAGEMENT GROUP (Hrsg.): *Systems Modeling Language (SysML) Specification*. Object Management Group, Januar 2005 234
- [Obj05b] OBJECT MANAGEMENT GROUP (Hrsg.): *UML 2.0 Superstructure Specification*. Object Management Group, Juli 2005. – Document: formal/2005-07-04 2, 9, 10, 16, 19, 34
- [Obj09] OBJECT MANAGEMENT GROUP (Hrsg.): *UML 2.2 Superstructure Specification*. Object Management Group, April 2009. – Document: formal/2009-02-02 10, 243
- [OGBG04] OBERSCHELP, Oliver ; GAMBUZZA, Alfonso ; BURMESTER, Sven ; GIESE, Holger: Modular Generation and Simulation of Mechatronic Systems. In: CALLAOS, N. (Hrsg.) ; LESSO, W. (Hrsg.) ; SANCHEZ, B. (Hrsg.): *Proceedings of the 8th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI), Orlando, USA*, International Institute of Informatics and Systemics (IIIS), July 2004, S. 1–6 196
- [ÖM02] ÖLVECKZY, Peter C. ; MESEGUER, José: Specification of real-time and hybrid systems in rewriting logic. In: *Theoretical Computer Science* 285 (2002), Nr. 2, S. 359 – 405. – ISSN 0304–3975 238
- [ÖM05] ÖLVECKZY, Peter C. ; MESEGUER, José: Real-Time Maude 2.1. In: *Electronic Notes in Theoretical Computer Science* 117 (2005), S. 285 – 314. – ISSN 1571–0661. – Proceedings of the Fifth International Workshop on Rewriting Logic and Its Applications (WRLA 2004) 238
- [ÖM07] ÖLVECKZY, Peter C. ; MESEGUER, José: Abstraction and Completeness for Real-Time Maude. In: *Electronic Notes in Theoretical Computer Science* 176 (2007), Nr. 4, S. 5 – 27. – ISSN 1571–0661. – Proceedings of the 6th International Workshop on Rewriting Logic and its Applications (WRLA 2006) 65, 238
- [OZKV08] OBERTHÜR, Simon ; ZNAMENSHCHYKOV, Alex ; KLÖPPER, Benjamin ; VÖCKING, Henner: Improved Flexible Resource Management by Means of Look-Ahead Scheduling and Bayesian Forecasting. In: GAUSEMEIER, Jürgen (Hrsg.) ; RAMMIG, Franz J. (Hrsg.) ; SCHÄFER, Wilhelm (Hrsg.): *Self-optimizing Mechatronic Systems: Design the Future*. Paderborn : Heinz Nixdorf Institut, Universität Paderborn, February 2008 (HNI-Verlagsschriftenreihe, Paderborn), S. 361–376 189, 193
- [OZL10] OBERTHÜR, Simon ; ZARAMBA, Leszek ; LICHTER, Hermann-Simon: Flexible Resource Management for Self-X Systems: An Evaluation. In: *Proceedings of First IEEE Workshop on Self-Organizing Real-Time Systems – SORT 2010* IEEE, IEEE CS Press, May 2010, S. 1–10 193

- [Pet99] PETERSSON, Paul: *Modelling and Verification of Real-Time Systems Using Timed Automata: Theory and Practice*, Department of Computer Systems, Uppsala University, Diss., February 1999 23
- [Pit89] PITT, Leonard: Inductive inference, DFAs, and computational complexity. In: *Analogical and Inductive Inference* Bd. 397, Springer Berlin / Heidelberg, 1989 (Lecture Notes in Computer Science). – ISBN 978–3–540–51734–4, S. 18–44 119
- [Pnu77] PNUELI, Amir: The Temporal Logic of Programs. In: *Proceedings of the 18th Annual Symposium on Foundations of Computer Science, 1977*, IEEE Computer Society Press, 1977, S. 46–57 241
- [Pul01] PULLUM, Laura L.: *Software Fault Tolerance*. ARTECH HOUSE, INC., 2001 100
- [PVY99] PELED, Doron ; VARDI, Moshe Y. ; YANNAKAKIS, Mihalis: Black Box Checking. In: *FORTE XII / PSTV XIX '99: Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX)*. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 1999. – ISBN 0–7923–8646–9, S. 225–240 97, 127, 239, 240
- [RDV09] RIVERA, J. E. ; DURAN, F. ; VALLECILLO, A.: A graphical approach for modeling time-dependent behavior of DSLs. In: *Visual Languages - Human Centric Computing 0* (2009), S. 51–55. ISBN 978–1–4244–4876–0 237
- [Ren08] RENSINK, Arend: Explicit State Model Checking for Graph Grammars. In: *Concurrency, Graphs and Models* Bd. 5065, Springer Berlin / Heidelberg, 2008 (Lecture Notes in Computer Science). – ISBN 978–3–540–68676–7, S. 114–132 238
- [Rie09] RIETH, Peter: Das mechatronische Fahrwerk der Zukunft. In: WINNER, Hermann (Hrsg.) ; HAKULI, Stephan (Hrsg.) ; WOLF, Gabriele (Hrsg.): *Handbuch Fahrerassistenzsysteme*. Vieweg+Teubner, 2009. – ISBN 978–3–8348–9977–4, S. 626–631 1
- [RMSM09] RAFFELT, Harald ; MERTEN, Maik ; STEFFEN, Bernhard ; MARGARIA, Tiziana: Dynamic testing via automata learning. In: *International Journal on Software Tools for Technology Transfer (STTT)* 11 (2009), October, Nr. 4, S. 307–324. – ISSN 1433–2779 239
- [Roz97] ROZENBERG, Grzegorz: *HANDBOOK of GRAPH GRAMMARS and COMPUTING by GRAPH TRANSFORMATION, Volume 1: Foundations*. World Scientific, 1997. – ISBN 9810228848 29, 30
- [Sch06] SCHILLING, Daniela: *Kompositionale Softwareverifikation mechatronischer Systeme*, Software Engineering Group, University of Paderborn, Diss., Februar 2006 238
- [Sei05] SEIBEL, Andreas: *Story Diagramme für eingebettete Echtzeitsysteme*, Software Engineering Group, University of Paderborn, Bachelor Thesis, 2005 194

-
- [Sei07] SEIBEL, Andreas: *Behavioral Synthesis of Potential Component Real-Time Behavior*, Software Engineering Group, University of Paderborn, Diploma Thesis, June 2007 242
- [SGW94] SELIC, Bran ; GULLEKSON, Garth ; WARD, Paul T.: *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994. – ISBN 0471599174 10, 48
- [SH03] STEFFEN, Bernhard ; HUNGAR, Hardi: Behavior-Based Model Construction. In: *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation* Bd. 2575/2003, Springer Berlin / Heidelberg, 2003 (Lecture Notes in Computer Science). – ISBN 978-3-540-00348-9, S. 5-19 239
- [SLG07] SHAHBAZ, Muzammil ; LI, Keqin ; GROZ, Roland: Learning Parameterized State Machine Model for Integration Testing. In: *COMPSAC '07: Proceedings of the 31st Annual International Computer Software and Applications Conference - Vol. 2- (COMPSAC 2007)*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0-7695-2870-8, S. 755-760 239
- [SPP01] STAUNER, Thomas ; PRETSCHNER, Alexander ; PÉTER, Istran: Approaching a Discrete-Continuous UML: Tool Support and Formalization. In: *Workshop of the pUML-Group held together with the UML'2001 on Practical UML-Based Rigorous Development Methods - Countering or Integrating the eXtremists*. Toronto, Canada : GI, October 2001. – ISBN 3-88579-335-0, S. 242-257 234
- [Sta08] STALLMANN, Florian: *A model-driven approach to multi-agent system design*, Software Engineering Group, University of Paderborn, Diss., April 2008 58
- [Sto96] STOREY, Neil R.: *Safety Critical Computer Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1996. – ISBN 0201427877 1, 100
- [Sto02] STOELINGA, Mariëlle: *Alea jacta est: verification of probabilistic, real-time and parametric systems*, University of Nijmegen, the Netherlands, Diss., April 2002 71
- [TFCB90] TSAI, J. J. P. ; FANG, K. Y. ; CHEN, H. Y. ; BI, Y. D.: A Noninterference Monitoring and Replay Mechanism for Real-Time Software Testing and Debugging. In: *IEEE Transactions on Software Engineering* 16 (1990), Nr. 8, S. 897-916. – ISSN 0098-5589 182
- [TGM00] TAENTZER, Gabriele ; GOEDICKE, Michael ; MEYER, Torsten: Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. In: *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*. London, UK : Springer-Verlag, 2000. – ISBN 3-540-67203-6, S. 179-193 237
- [Tic09] TICHY, Matthias: *Gefahrenanalyse selbstoptimierender Systeme*, Software Engineering Group, University of Paderborn, Diss., May 2009 3, 12, 50

- [TOHS99] TARR, Peri ; OSSHER, Harold ; HARRISON, William ; SUTTON, JR., Stanley M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: *Proceedings of the 21st International Conference on Software Engineering (ICSE '99)*. New York, NY, USA : ACM, 1999. – ISBN 1–58113–074–0, S. 107–119 3, 10, 139, 143
- [TY01] TRIPAKIS, Stavros ; YOVINE, Sergio: Analysis of Timed Systems Using Time-Abstracting Bisimulations. In: *Formal Methods in System Design* 18 (2001), January, Nr. 1, S. 25–68 80, 164
- [Vas73] VASILEVSKII, M. P.: Failure diagnosis of automata. In: *Cybernetics and Systems Analysis* 9 (1973), July, Nr. 4, S. 653–665. – ISSN 1060–0396 117, 118, 127
- [Wen08] WENDEHALS, Lothar: *Struktur- und verhaltensbasierte Entwurfsmustererkennung*, Software Engineering Group, University of Paderborn, Diss., Januar 2008 246
- [Wir04] WIRSING, Martin (Hrsg.): *Report on the EU/NSF Strategic Workshop on Engineering Software-Intensive Systems*. Edinburgh, GB, May 2004 1, 234
- [WL97] WEISE, Carsten ; LENZKES, Dirk: Efficient Scaling-Invariant Checking of Timed Bisimulation. In: REISCHUK, Rüdiger (Hrsg.) ; MORVAN, Michel (Hrsg.): *Proceedings of the 14th Annual Symposium on Theoretical Aspects of Computer Science (STACS 97), Lübeck, Germany, February 27 - March 1, 1997* Bd. 1200, Springer Verlag Berlin Heidelberg, 1997 (Lecture Notes in Computer Science (LNCS)), S. 177–188 39, 164
- [YJ94] YI, Wang ; JONSSON, Bengt: Decidability of timed language-inclusion for networks of real-time communicating sequential processes. In: *Foundation of Software Technology and Theoretical Computer Science* Bd. 880, Springer Verlag, 1994 (Lecture Notes in Computer Science (LNCS)), S. 243–255 74, 75, 85
- [YPD94] YI, Wang ; PETTERSSON, Paul ; DANIELS, Mats: Automatic Verification of Real-time Communicating Systems by Constraint-solving. In: HOGREFE, Dieter (Hrsg.) ; LEUE, Stefan (Hrsg.): *Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Formal Techniques, Berne, Switzerland, 1994* Bd. 6, Chapman & Hall, 1994 (IFIP Conference Proceedings), S. 243–258 23, 155
- [Zam99] ZAMBONELLI, Franco: An Efficient Logging Algorithm for Incremental Replay of Message. In: *IPPS '99/SPDP '99: Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*. Washington, DC, USA : IEEE Computer Society, 1999. – ISBN 0–7695–0143–5, S. 392–398 182
- [ZC06] ZHANG, Ji ; CHENG, Betty H. C.: Model-based development of dynamically adaptive software. In: *ICSE '06: Proceeding of the 28th international conference on Software engineering*. New York, NY, USA : ACM, 2006. – ISBN 1–59593–375–1, S. 371–380 45, 48, 100, 235

- [Zün01] ZÜNDORF, Albert: *Rigorous Object Oriented Software Development*, Software Engineering Group, University of Paderborn, Habilitation, 2001 29, 33, 34, 46, 53, 56, 57, 58, 261
- [Zün09] ZÜNDORF, Albert: Model Checking the Leader Election Protocol with Fujaba. In: LEVENDOVSKY, Tihamer (Hrsg.) ; RENSINK, Arend (Hrsg.) ; GORP, Pieter V. (Hrsg.): *Fifth International Workshop on Graph Based Tools (GraBaTs)*, 2009, S. 1–11 198
-