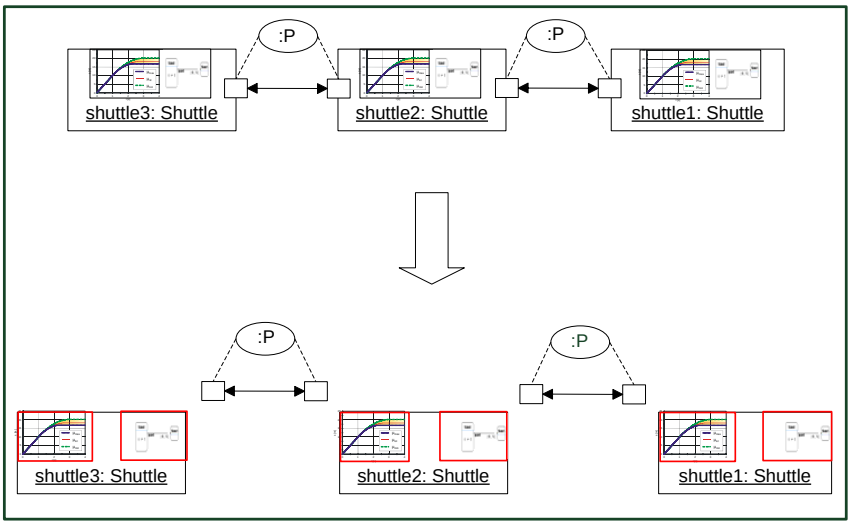
UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik – Informatik – Mathematik
Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Modell-basierte Verifikation von vernetzten mechatronischen Systemen



Martin Hirsch

UNIVERSITÄT PADERBORN

Die Universität der Informationsgesellschaft

Fakultät für Elektrotechnik – Informatik – Mathematik
Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

Modell-basierte Verifikation von vernetzten mechatronischen Systemen

**Genehmigte Dissertation
zur Erlangung des akademischen Grades
„Doktor der Naturwissenschaften“
(Dr. rer. nat.)**

vorgelegt von

Dipl.-Inform. Martin Hirsch

Promotionskommission:
Vorsitzender: Prof. Dr. rer. nat. Wilhelm Schäfer (Universität Paderborn)
Koreferat: Prof. Dr. rer. nat. Heike Wehrheim (Universität Paderborn)
Koreferat: Prof. Dr. rer. nat. Ingolf H. Krüger (University of California at San Diego)
Beisitzer: Prof. Dr.-Ing. Ansgar Trächtler (Universität Paderborn)
Beisitzer: Prof. Dr. rer. nat. Gregor Engels (Universität Paderborn)

Die Dissertation wurde am 15. Juli 2008 bei der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn eingereicht und am 15. September 2008 vor der Promotionskommission verteidigt und durch die Fakultät angenommen.

Dank

An dieser Stelle möchte ich mich bei all den Menschen bedanken, die mir auf dem Weg zur Promotion geholfen, mich ermutigt oder mir freundschaftlich zur Seite gestanden haben.

Als erstes bedanke ich mich bei meinem Doktorvater Prof. Dr. Wilhelm Schäfer. Bei Wilhelm möchte ich mich dafür bedanken, dass er meine Promotion mitbetreut hat und er trotz seines mehr als randvollen Terminkalenders stets Zeit sowohl für wissenschaftliche Diskussionen, als auch für private Gespräche für mich hatte. Danke, Wilhelm!

Prof. Dr. Holger Giese möchte ich danken, dass er mich 2004 zur Promotion „überredet“ hat, meine Promotion mitbetreut und zu jeder Tages- und Nachtzeit für wissenschaftlichen Input gesorgt hat. Danke, Holger!

Bei Prof. Dr. Heike Wehrheim und Prof. Dr. Ingolf Krüger bedanke ich mich, dass sie das Koreferat meiner Dissertation übernommen haben und für die Diskussionen bei der Fertigstellung dieser Arbeit. Des weiteren möchte ich mich bei Ingolf für den Gastaufenthalt in seiner Arbeitsgruppe an der University of California, San Diego, bedanken. Für die Teilnahme an meiner Promotionskommission danke ich weiterhin Prof. Dr.-Ing. Ansgar Trächtler und Prof. Dr. Gregor Engels.

Ganz besonders möchte ich mich bei Stefan Henkler (und seiner Frau Sandra und seinen beiden Kindern Gavin und Jarne, die es „erlaubt haben“, dass Stefan so viel Zeit für mich haben durfte) bedanken! Nicht nur, weil du für mich ein super Bürokollege warst, sondern auch für die zahlreichen wissenschaftlichen und privaten Diskussionen, Gespräche und „geselligen Stunden“, teilweise bei Kaffee, Bier, Sekt und allen möglichen anderen Getränken in und außerhalb der Uni. Ohne dich würde es diese Dissertation nicht geben! Danke, Stefan!

Bürokratische Probleme sind während meiner Promotionszeit in Paderborn dank Jutta Haupt und Sarah Latsch nie aufgetreten. Jürgen (Sammy) Maniera danke ich für die Hilfe bei technischen Problemen und für die Anfangszeit als Bürokollege, sowie Sabrina Clemens, für das anfängliche „Schreibtischsharing“.

Eine Promotion kann nicht alleine und im stillen Kämmerchen geschrieben werden. An dieser Stelle möchte ich mich bei allen anderen Kollegen und Ex-Kollegen Björn Axenath, Dr. Sven Burmester, Dr. Matthias Gehrke, Joel Greenyer, Stefan Henkler, Prof. Dr.

Ekkart Kindler, Florian Klein, Ahmet Mehic, Matthias Meyer, Claudia Priesterjahn, Dr. Vladimir Rubin, Matthias Tichy (MTT), Dr. Daniela Schilling, Oliver Sudmann, Dietrich Travkin, Markus von Detten, Robert Wagner, Dr. Lothar Wendehals bedanken, die immer für wissenschaftliche, aber auch private Gespräche zur Verfügung standen.

Bei Ahmet bedanke ich mich für die vielen Ratschläge aus der „nicht informatischen“ Sicht. Ekkart danke ich für seine vielen Tipps und Hinweise, besonders auch um mal „über den Tellerrand hinauszuschauen“. Claudia, MTT und Stefan danke ich für das Korrekturlesen meiner Arbeit.

Was wäre eine interdisziplinäre Dissertation ohne interdisziplinäre Gespräche – bei meinen Kollegen Eckehard Münch und Henner Vöcking vom Lehrstuhl für Regelungstechnik und Mechatronik bedanke ich mich für interessante Diskussionen. Henner danke ich darüber hinaus für seinen Crashkurs in Regelungstechnik.

Vor allem möchte ich mich ganz herzlich bei meinen Eltern bedanken, dass sie mir meine Ausbildung ermöglicht haben, dass sie immer für mich da waren, mich unterstützt und sie sich mit mir und für mich über Erfolge gefreut haben. Danke, Mama und Papa!

Meinem Bruder Henrik danke ich für zahlreiche Tipps und Hinweise während meines Studiums und meiner Promotion.

Meinem besten Freund Hansjörg und seiner Freundin Manon danke ich ganz herzlich für die vielen vergnüglichen gemeinsamen Stunden, für viele aufmunternde Gespräche, Spaziergänge und die Ablenkung von allem, was mit meiner Dissertation zu tun hat. Danke, Hansjörg und Manon!

Zusammenfassung

Beim Entwurf selbstoptimierender, mechatronischer Systeme stellt die eingebettete Software einen großen Teil der Wertschöpfung dar. Typischerweise werden Regelungen oder Steuerungen in Software umgesetzt. Durch die starke Vernetzung selbstoptimierender Systeme wird Software auch zur nachrichtenbasierten Kommunikation und Koordination zwischen den einzelnen verteilten selbstoptimierenden Systemen eingesetzt. Diese Kommunikation geht über die Aufnahme von System- und Umweltdaten durch Sensorik hinaus. Hier werden ggf. komplexe Zustandsinformationen über entsprechende Protokolle und zugrunde liegende Kommunikationskanäle ausgetauscht, die dann wieder das Verhalten bzw. die zugrunde liegenden Berechnungen der einzelnen Komponenten massiv beeinflussen können. Diese Entwicklung führt zu äußerst komplexer hybrider (diskreter / kontinuierlicher) Software. Des Weiteren werden selbstoptimierende, mechatronische Systeme oftmals in sicherheitskritischen Umgebungen eingesetzt. Hierdurch müssen formale Verfahren zur Verifikation der Korrektheit des Systems gegenüber sicherheitskritischen Eigenschaften eingesetzt werden.

Im Rahmen dieser Dissertation werden nun Konzepte und Methoden zur Modellierung und Verifikation mechatronischer Systeme entwickelt und formal beschrieben. Der hier vorgestellte Ansatz baut auf dem im Sonderforschungsbereichs 614 entwickelten MECHATRONIC UML Ansatz auf. Dieser unterstützt einen kompositionellen Verifikationsansatz für das Echtzeitverhalten von mechatronischen Systemen.

Um eine effiziente Verifikation solcher vernetzten mechatronischen Systeme zu ermöglichen, werden in dieser Arbeit Techniken der *Abstraktion*, *Dekomposition* sowie der *regelbasierten Modellierung* eingeführt. Hierbei werden diese nicht orthogonalen Techniken geschickt miteinander kombiniert. Ziel ist es, die besonders durch die Verwendung domänenübergreifender Modelle, wie sie bei der Modellierung von mechatronischen Systemen vorkommen, entstehenden inhärenten multi-Paradigmenwechsel modellieren zu können. Der hier vorgeschlagene Ansatz zur modell-basierten Verifikation mechatronischer Systeme zeichnet sich durch die Integration effizienter Verifikationstechniken, basierend auf dem Modellwissen und einer geschickten Modellierung, aus.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Intelligente mechatronische Systeme, die autonom und flexibel auf Änderungen in ihrem Umfeld reagieren, sind in unserer Zukunft nicht mehr wegzudenken. Nicht nur in kleinen Anwendungen wie in der intelligenten Lichtsteuerung in modernen Autos, sondern auch in großen Projekten wie in der Entwicklung des „intelligenten, selbst denkenden Hauses“ oder in innovativen Güter- und Personentransportsystemen der Zukunft fließen diese Konzepte maßgeblich mit ein. Um solche Systeme zu realisieren, bedarf es einer engen Verknüpfung der Konzepte und Methoden der in der Mechatronik verankerten Domänen Maschinenbau, Elektronik und Softwaretechnik (siehe Abbildung 1.1). Im Gegensatz zu reinen Softwareanwendungen bekommt der Sicherheitsaspekt in solchen Systemen einen deutlich höheren Stellenwert, da Fehler meist unmittelbar Gefahr für Menschenleben bedeuten [Lev95][Her99].

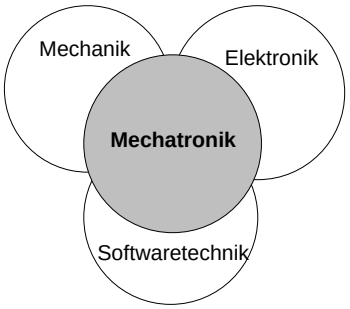


Abbildung 1.1: Die Disziplin Mechatronik ergibt sich aus der Kombination der drei Disziplinen Softwaretechnik, Mechanik und Elektronik

Solche intelligenten, mechatronischen Systeme, wie sie von Schäfer und Wehrheim [SW07] oder auch von Dawson und anderen [DSBB00] beschrieben werden, bestehen heutzutage aus einer Vielfalt von komplexen Einzelkomponenten, welche untereinander verbunden sind und miteinander interagieren. Das Verhalten des Gesamtsystems ist entsprechend durch die Kommunikation und Kooperation der intelligenten Systemelemente

charakterisiert. Aus informationstechnischer Sicht handelt es sich um verteilte Systeme von miteinander kooperierenden Agenten.

1.1 Motivation

Beim Entwurf selbstoptimierender, mechatronischer Systeme stellt die eingebettete Software einen großen Teil der Wertschöpfung dar. Typischerweise werden Regelungen oder Steuerungen in Software umgesetzt. Durch die starke Vernetzung selbstoptimierender Systeme wird Software auch zur nachrichtenbasierten Kommunikation und Koordination zwischen den einzelnen verteilten selbstoptimierenden Systemen eingesetzt. Diese Kommunikation geht über die Aufnahme von System- und Umweltdaten durch Sensorik hinaus. Hier werden ggf. komplexe Zustandsinformationen über entsprechende Protokolle und zugrunde liegende Kommunikationskanäle ausgetauscht, die dann wieder das Verhalten bzw. die zugrunde liegenden Berechnungen der einzelnen Komponenten massiv beeinflussen können. Diese Entwicklung führt zu äußerst komplexer hybrider (diskret / kontinuierlicher) Software. In Abbildung 1.2 ist ein Beispiel für hybrides Verhalten gezeigt. In dem linken Oval ist rein kontinuierliches Verhalten, im rechten Oval rein diskretes Verhalten beschrieben. Das Zusammenspiel, das Umschalten zwischen verschiedenen kontinuierlichen Verhalten, kann nun durch die Integration beider Verhalten vorgenommen werden. Genau dieses wird als hybrid bezeichnet.

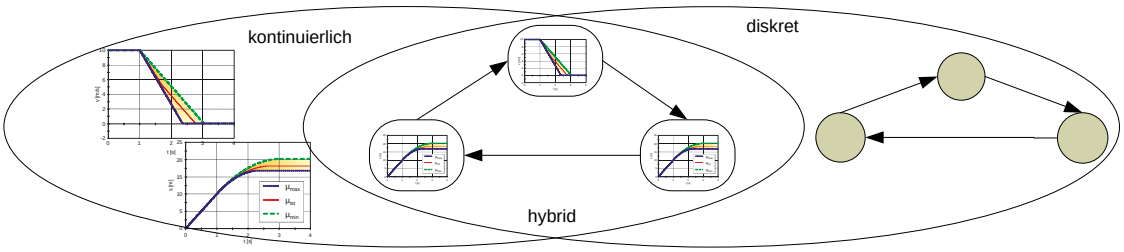


Abbildung 1.2: Hybrides Verhalten

Des Weiteren werden selbstoptimierende, mechatronische Systeme oftmals in sicherheitskritischen Umgebungen eingesetzt. Hierdurch müssen formale Verfahren zur Verifikation der Korrektheit des Systems gegenüber sicherheitskritischen Eigenschaften eingesetzt werden.

Auf Grund der steigenden Komplexität solcher Systeme ist es notwendig, Methoden zu entwickeln, die auf der einen Seite eine geeignete Modellierung erlauben und auf der anderen Seite die Validierung und Verifikation dieser Modelle in akzeptabler Zeit durchführen können. Das Gebiet der *Softwaretechnik* beschäftigt sich mit dieser Thematik.

> Softwaretechnik: Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden, Konzepten, Notationen und Werkzeugen für die arbeitsteilige, ingenieurmäßige Entwicklung und Anwendung von umfangreichen Software-Systemen. (Nach Balzert [Bal98])

Um einen sicheren Betrieb eines mechatronischen Systems zu gewährleisten, müssen die Sicherheitseigenschaften dieses Systems überprüft werden. Die Überprüfung eines mechatronischen Systems durch Testen, d.h. die experimentelle Ausführung des Systems, kann ein sicheres Verhalten alleine nicht ausreichend nachweisen, da durch reines Testen nicht alle Ausführungspfade erreicht werden können. Dabei ist dann nicht auszuschließen, dass Pfade, die durch das Testen nicht überprüft wurden, sicherheitskritische Eigenschaften aufweisen. Außerdem werden durch Testen erhebliche Kosten verursacht, da sie oftmals manuell durchgeführt werden und unvollständig sind [BN03].

In der Softwaretechnik werden formale Methoden [Win90] verwendet, die mathematisch fundierte Techniken zur Spezifikation von Systemen zur Verfügung stellen. In [CW96] wird ein Überblick über formale Methoden und formale Verifikationstechniken gegeben. Diese gehen von manuellen, unvollständigen Testverfahren, über interaktive Theorembeweise, bis hin zu automatischen, vollständigen formalen Verifikationsverfahren. Eine formale Verifikationstechnik ist z.B. das Model Checking. Im Gegensatz zum Testen werden hier alle Pfade des Systems automatisch erstellt und überprüft. Jedoch bringt der Einsatz von formalen Verifikationstechniken eine ganze Reihe von Problemen mit sich.

Die benötigte Rechenzeit und der Speicherbedarf hängen z.B. beim Model Checking von der Größe des zu überprüfenden Systems ab. Daher werden beim Model Checking effiziente Algorithmen eingesetzt, um möglichst große Systeme überprüfen zu können. Die Software eines mechatronischen Systems kann jedoch einen sehr großen oder unendlich großen Zustandsraum besitzen. Deshalb ist eine Überprüfung mechatronischer Systeme in den meisten Fällen auf Grund der Größe des Zustandsraums durch diese Verifikationstechnik alleine nicht möglich. Deshalb müssen weitere Techniken wie z.B. Abstraktion, Approximation und viele andere mit dem Model Checking kombiniert werden, um die Verifikation durchführen zu können. Nicht nur die Größe des zu verifizierenden Systems stellt beim Model Checking ein Problem dar. Die Konstruktion des Modells, die Spezifikation der zu überprüfenden Eigenschaften in der geeigneten temporallogischen Formel, das Problem der Zustandsraumexplosion sowie die Interpretation der Ergebnisse sind weitere Probleme beim Model Checking (siehe [CGP00, Dwy02]).

Der interdisziplinäre Sonderforschungsbreich „SFB 614: Selbstoptimierende Systeme des Maschinenbaus“[1] an der Universität Paderborn beschäftigt sich unter anderem mit dem oben beschriebenen Forschungsgebiet der effizienten Verifikation von mechatronischen Systemen. Im Folgenden wird das Anwendungsbeispiel genau erklärt, welches im Ver-

[1] http://www.sfb614.de

laufe der Arbeit als durchgängiges Beispiel genutzt wird, um die bisherigen Probleme in der Verifikation zu beleuchten und die neuen Konzepte vorzustellen.

1.2 Anwendungsbeispiel

Im Rahmen des Sonderforschungsbereichs 614 „Selbstoptimierende Systeme des Maschinenbaus“ werden Konzepte und Methoden zur Entwicklung von mechatronischen Systemen mit inhärenter Teilintelligenz erforscht. Konkret finden die entwickelten Konzepte im RailCab[2] Forschungsprojekt Anwendung. In diesem Projekt werden innovative Güter- und Personentransportsystemen der Zukunft, so genannte Shuttles, entwickelt. Diese werden durch einen Linear-Motor angetrieben und verfügen über drahtlose Netzwerke zur Kommunikation untereinander. Die Energieübertragung wird durch einen Streckenstator, der einem üblichen Schienennetz hinzugefügt werden kann, erreicht.

Ein *Shuttle* ist eine kleine, autonome, führerlose Einheit (siehe Abbildung 1.3(a)). Diese Shuttles sollen Personen und Güter, auf Nachfrage, individuell von ihrem Ausgangspunkt zu ihrem gewünschten Zielort transportieren. Während der Fahrt können sich einzelne Shuttles zu einem Konvoi zusammenschließen (siehe Abbildung 1.3(b)). Dies spart durch die Windschattenfahrt Energie und erhöht bei stark frequentierten Strecken die Kapazität, da die Züge nicht mehr getrennt fahren müssen. Die Shuttles bleiben während der Konvoifahrt weiterhin autonom und treffen individuelle Entscheidungen. Durch das autonome Verhalten treten jedoch Schwierigkeiten auf. Das Problem besteht darin, die Shuttles so zu koordinieren, dass sie so häufig wie möglich Konvois bilden um den Windwiderstand und hiermit den Energieverbrauch zu reduzieren. Zusätzlich sollte der Abstand zwischen den Fahrzeugen so gering wie möglich sein, um den Effekt noch zu verstärken. Die Erstellung und Auflösung eines Konvois ist ein sicherheitskritisches Manöver, bei dem eine Reihe von Echtzeitbedingungen eingehalten werden müssen. Dabei ist das Verhalten der Beschleunigungsregelung jedes Shuttles je nach aktuellem Szenario verschieden. So könnte es für ein führendes oder einem allein fahrenden Shuttle ein Ziel sein, eine möglichst gleichmäßige Geschwindigkeit zu halten. Dieses Verhalten ist innerhalb des Konvois jedoch nicht optimal. Durch das autonome Verhalten der Shuttles kann es zu kleinen Abweichungen zwischen den Geschwindigkeitseinstellungen der Regler kommen. Sobald ein nachfolgendes Shuttle, welches in einem Abstand von 10 cm folgt, nur 0,01 $\frac{km}{h}$ schneller wäre, würde dies nach nur 36 Sekunden mit dem vorausfahrenden Shuttle kollidieren. Für diesen Fall ist also ein Konvoimodus, in dem der Abstand konstant gehalten wird, besser als eine konstante Geschwindigkeit.

[2] http://www-nbp.upb.de

(a) Shuttles im Konvoi (b) Zwei Shuttles bei der Bildung eines Konvois

Abbildung 1.3: Fallstudie „RailCab – Neue Bahntechnik Paderborn“ (Quelle: NBP)

1.3 Ziel und Lösungsansatz

Ziel dieser Dissertation ist es, Konzepte und Methoden zur Modellierung und Verifikation mechatronischer Systeme zu entwickeln und formal zu beschreiben. Ziel ist es dabei, die besonders durch die Verwendung domänenübergreifender Modelle, wie sie bei der Modellierung von mechatronischen Systemen vorkommen, entstehenden inhärenten multi-Paradigmenwechsel [HH06] modellieren und verifizieren zu können. Der hier vorgeschlagene Ansatz zur modell-basierten Entwicklung mechatronischer Systeme zeichnet sich durch die Integration effizienter Verifikationstechniken, basierend auf dem Modellwissen, Abstraktionstechniken und einer geschickten Modellierung, aus.

Im Rahmen des Sonderforschungsbereichs 614 wurde die MECHATRONIC UML [GHH+08b] entwickelt. Diese erlaubt es, Struktur und Verhalten von mechatronischen Systemen und die Interaktion zwischen mechatronischen Systemen zu spezifizieren und zu verifizieren. Dabei richtet sich die Struktur eines komplexen mechatronischen Systems nach der von Lückel [LHLH01][OHG04][HOG04][Ge05] vorgeschlagenen Struktur. Die konkrete Umsetzung dieser Struktur findet sich im *Operator-Controller-Modul (OCM)* wieder. Die Informationsverarbeitung eines mechatronischen Systems kann als Operator-Controller-Modul (OCM) aufgefasst werden. Ein solches Modul ist in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator unterteilt. Während der Controller direkten Zugriff auf die Aktuatoren des Systems hat, wird der reflektorische Operator dazu verwendet, den Controller zu steuern und die Interaktion mit anderen OCMs zu koordinieren. Die Aufgabe des kognitiven Operators besteht darin Wissen über die Umwelt und das OCM selber zu sammeln und dazu zu nutzen, das Verhalten des OCM besser an die gegebenen Anforderungen anzupassen.

Da die Software des reflektorischen Operators für die Steuerung des Controllers sowie die Interaktion des OCMs mit anderen OCMs verantwortlich ist, ist sie sicherheitskritisch. Deshalb besteht das Ziel dieser Arbeit in der Entwicklung eines ganzheitlichen,

effizienten Ansatzes zur Modellierung und Verifikation für die Software des OCMs sowie für die Koordination zwischen OCMs.

Im Folgenden wird eine erste Idee vermittelt, wie und welche Techniken verwendet werden, um die effiziente Verifikation von durch MECHATRONIC UML beschriebenen vernetzen mechatronischen Systemen zu ermöglichen. In Abbildung 1.4 ist ein Überblick über das Zusammenspiel der in dieser Arbeit verwendeten Techniken „Kompositioneller Aufbau & Verifikation“, „Abstraktion und Verfeinerung“, „Dekomposition“ und „Regelbasierte Modellierung“ gegeben.

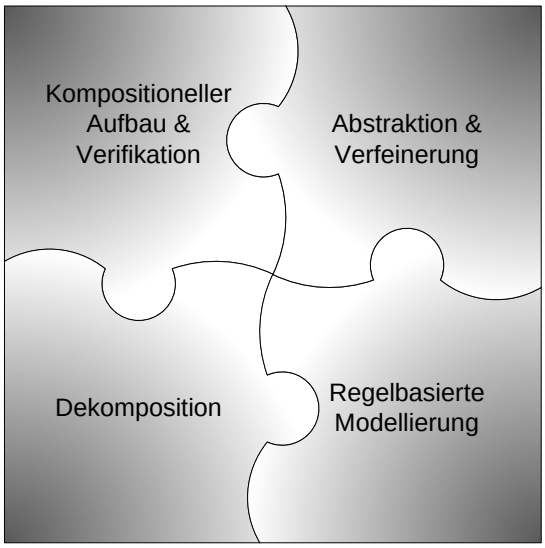


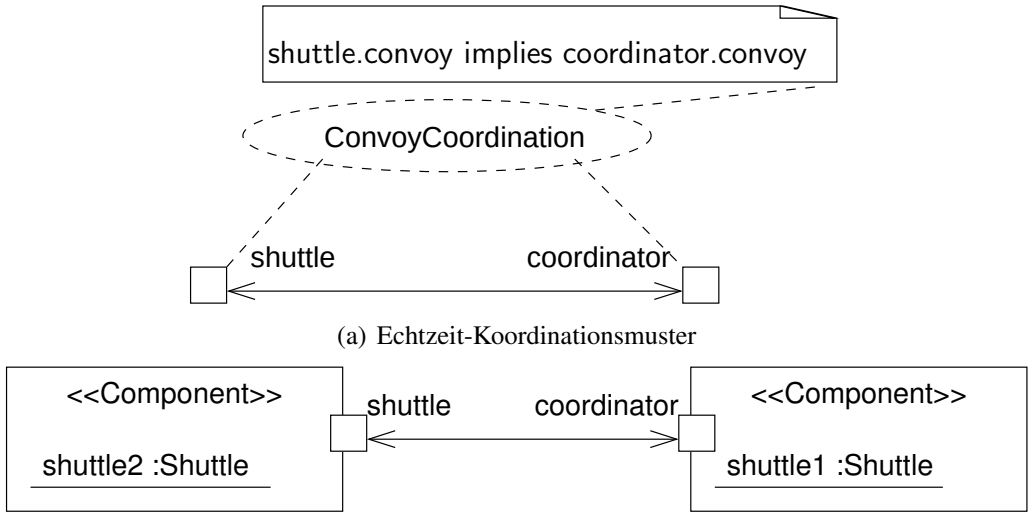
Abbildung 1.4: Die einzelnen „Bausteine“ zu einer effizienten Verifikation von mechatronischen Systemen

Durch den nach innen schwächer werdenden Farbverlauf ist kenntlich gemacht, dass die einzelnen Techniken nicht orthogonal zueinander stehen, sondern ineinander greifen und aufeinander aufbauen. Dies wird in den Folgenden Abschnitten deutlich.

Kompositioneller Aufbau & Verifikation. Die kompositionelle Verifikationsmethode ist eine effiziente Möglichkeit, um große Modelle zu verifizieren [CGP00]. Diese stellen wirkungsvolle Methoden für die Verifikation eines nebenläufigen Systems dar, weil hier direkt der Ursache für den exponentiellen Anstieg des Zustandsraums entgegengewirkt wird. Im Gegensatz zur Überprüfung einer temporallogischen Spezifikation auf dem globalen Zustandsraum wird die kompositionelle Verifikation lediglich auf Teilzustandsräumen mit lokalen temporallogischen Spezifikationen durchgeführt.

Ein Vorteil dieses Verfahrens gegenüber Verfahren, die auf globalem Zustandsraum arbeiten, ist, dass dadurch, dass Komponenten unabhängig voneinander spezifiziert und verifiziert werden können, Komponenten zu verschiedenen Zeitpunkten während der Softwareentwicklung überprüft werden können. Durch diese unabhängige Verifizierung der Komponenten lässt sich der Verifikationsprozess in den Modellierungsprozess integrieren. Dies hat den Vorteil, dass Fehler zu einem sehr frühen Zeitpunkt in der Entwicklungsphase entdeckt und beseitigt werden können. Auch lassen sich so wiederverwendbare Module spezifizieren und verifizieren.

In Abbildung 1.5 ist der in dieser Arbeit verfolgte Ansatz für einen kompositionellen Aufbau anhand einer Komponentenarchitektur dargestellt [GTB+03][HG03][Hir04]. Das Verhalten des Systems ist hierbei zustandsbasiert und rein zeit-kontinuierlich beschrieben. Hier wird in einem ersten Schritt die Echtzeit-Koordination zwischen zwei Komponenten durch ein so genanntes Echtzeit-Koordinationsmuster modelliert (siehe Abbildung 1.5(a)), welches einzeln verifiziert werden kann. In einem nächsten Schritt wird hieraus das Verhalten von Komponenten hergeleitet (siehe Abbildung 1.5(b)), das nun auch separat verifiziert werden kann. Da jetzt sowohl die Kommunikation als auch das Komponentenverhalten verifiziert wurden (und beide in einer bestimmten Verfeinerungsbeziehung (siehe nächster Abschnitt) stehen), ist es möglich, das System durch reine, korrekte syntaktische Anwendungen der Koordinationsmuster und Komponenten zu modellieren.



(a) Echtzeit-Koordinationsmuster

(b) Anwendung des Echtzeit-Koordinationsmusters

Abbildung 1.5: Kompositioneller Ansatz

Abstraktion & Verfeinerung. Eine Abstraktion eines Modells abstrahiert von den internen Eigenschaften und ist das Gegenstück von Verfeinerung. Ist ein System A eine

Abstraktion von B, so ist B eine Verfeinerung von A. Die Eigenschaft der Abstraktion und Verfeinerung kann zur Unterstützung der Verifikation genutzt werden. Kann das ursprüngliche Modell aufgrund seiner Komplexität nicht in einem angemessenen Zeitrahmen verifiziert werden, wird das Modell mit Hilfe der Abstraktion handhabbar gemacht. Das neu erstellte Modell, das vom Umfang her kleiner ist, beinhaltet weiterhin die für eine Verifikation relevanten Eigenschaften und kann schneller verifiziert werden.

Abbildung 1.6 zeigt exemplarisch den Aufbau eines zeit-kontinuierlichen Echtzeitsystems und eines hybriden Systems. Das Verhalten beider Systeme ist durch Echtzeitverhalten charakterisiert. Da eine falsche Spezifikation des Echtzeitverhaltens zum Beispiel zu einem Ausfall des Systems führen kann, muss hier eine geeignete Verifikation durchgeführt werden. Im letzten Abschnitt wurde kompositionelles Model Checking zur Verifikation des Echtzeitverhaltens vorgestellt. Zur Verifikation des Echtzeitverhaltens eines hybriden Systems bietet es sich ebenfalls an, dieses Verfahren zu verwenden. Dazu muss jedoch eine Abstraktion von dem hybriden Verhalten erfolgen. Diese wird mit Hilfe der Verfeinerungsbeziehung zwischen den beiden Systemen erstellt.

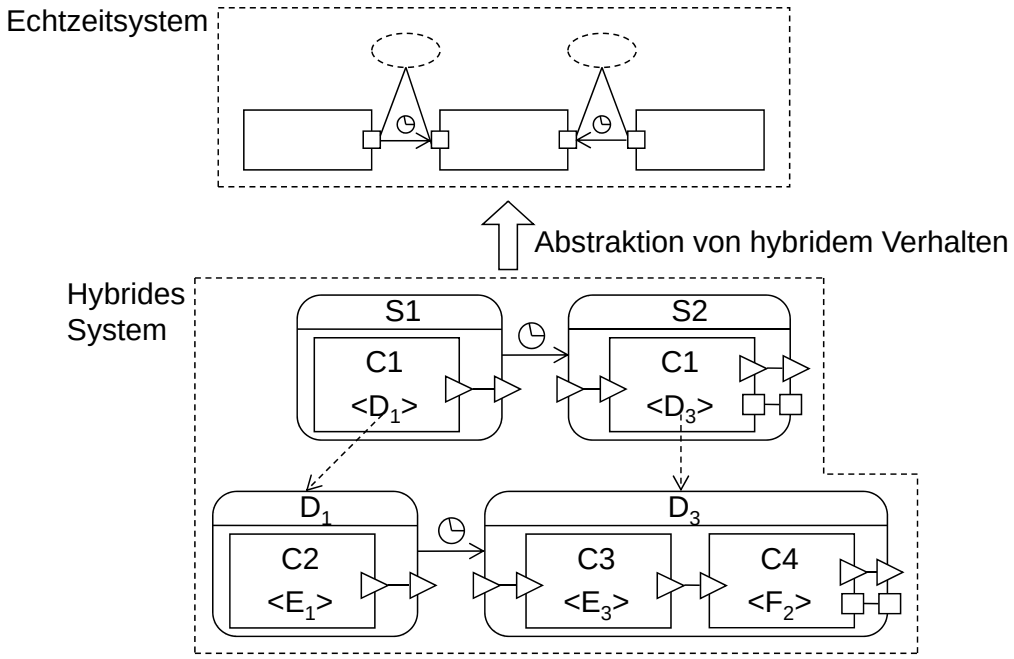


Abbildung 1.6: Abstraktion

Wichtig ist zudem, dass der unterschiedliche Aufbau der beiden Systeme berücksichtigt wird. Während ein Echtzeitsystem auf einer Hierarchieebene spezifiziert wird, sind hybride Systeme hierarchisch aufgebaut [GBSO04]. Abbildung 1.6 verdeutlicht diesen Unterschied. Aufgrund des unterschiedlichen Aufbaus reicht kompositionelles Model Checking

allein zur Verifikation des Echtzeitverhaltens nicht aus. Neben der Spezifikation von Echtzeitverhalten können zudem Inkonsistenzen durch den hierarchischen Aufbau und die dadurch bedingte Rekonfiguration entstehen [GH05b][GH06].

Dekomposition. In hybriden Systemen ist häufig eine klare Trennung zwischen der diskreten und der kontinuierlichen Komponente gegeben. Der kontinuierliche Teil dient der möglichst exakten Abbildung mechatronischer oder physikalischer Abläufe und Zusammenhänge. Die diskrete Komponente muss ihr Verhalten an die kontinuierliche Komponente koppeln, um so z.B. auf Veränderungen der Umwelt zu reagieren.

Durch die bereits erwähnte kompositionelle Modellierung der Echtzeitkoordination und der Abstraktion, die Steuer- und Regelungsalgorithmen entsprechend integriert, ist es möglich, für Verifikationszwecke eine abstrakte Betrachtung des relevanten kontinuierlichen und diskreten Verhaltens der Koordinationslogik vorzunehmen. Dabei werden entsprechend benötigte Eigenschaften der unterlagerten Regelung, die mit klassischen Techniken der Mathematik und der Regelungstechnik verifiziert werden können, als Basis für weitere Betrachtungen verwendet. Darauf aufbauend lässt sich dann durch formale Verifikationstechniken für Echtzeitsysteme eine Verifikation der benötigten Sicherheitseigenschaften der Echtzeitkoordination erreichen [GHH+06c].

In Abbildung 1.7 ist die hierbei zugrunde liegende Idee der Dekomposition skizziert. Die obere Hälfte der Abbildung zeigt die Modellierung des Komponentenverbunds eines Shuttlekonvois. Jede Komponente kommuniziert mit ihrer Nachbarkomponente. In einer Komponente selber ist das interne, sowohl Zeit-kontinuierliche als auch Werte-kontinuierliche Verhalten skizziert. Der untere Teil der Abbildung zeigt die Dekomposition des Modells. Der Komponentenverbund wurde in die Kommunikation und die Komponenten (siehe Kompositioneller Ansatz), aufgeteilt. Weiterhin wurde auch das interne Verhalten dekomponiert. So ist zu erkennen, dass nun das Zeit-kontinuierliche Verhalten von dem Werte-kontinuierlichen Verhalten getrennt ist. Dies ermöglicht, wie schon beschrieben, eine getrennte Verifikation der einzelnen Verhalten.

Regelbasierte Modellierung. In komplexen, vernetzten mechatronischen Systemen stehen nur begrenzte Rechen- und Speicherkapazitäten zur Verfügung. Zusätzlich unterliegt das System zur Laufzeit einer Evolution abhängig vom gegebenem Kontext. Anforderungen an komplexe, mechatronische Systeme sehen deshalb Dynamik vor, d.h. Steuerungssoftware muss zu Laufzeit ausgetauscht werden können.

In Schilling [Sch06] wurde bereits beschrieben, wie Graphtransformationssysteme zur Beschreibung von dynamischen Veränderungen im Kontext von mechatronischen Systemen eingesetzt werden können. So wurde das in Abbildung 1.8 dargestellte Szenario auf der Basis von Graphtransformationssystemen beschrieben. Die obere Hälfte des Bildes

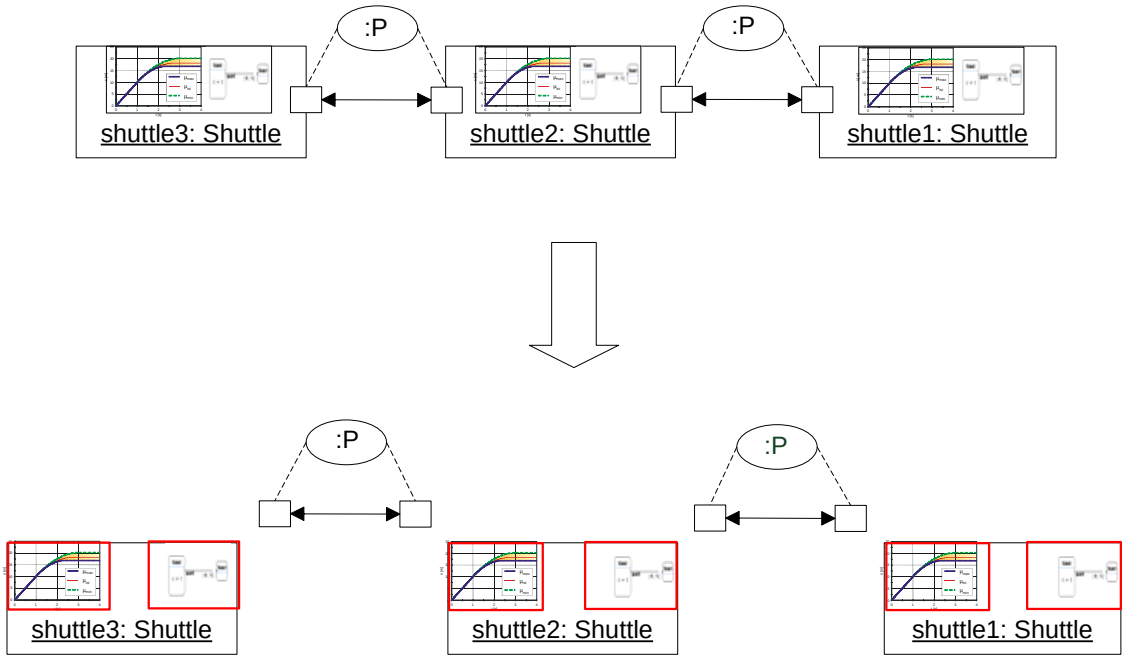


Abbildung 1.7: Dekomposition der Struktur und des internen Verhaltens

zeigt einen aktuellen Systemzustand. Hier fahren zwei Shuttles hintereinander auf zwei verschiedenen Streckenabschnitten. Die untere Hälfte des Bildes zeigt die koordinierte Bewegung des Shuttles auf den Streckenabschnitten, modelliert durch eine graphbasierte Regel. Hier wird beschrieben, welche Objekte existieren und wie miteinander verbunden sind. Im vorliegenden Beispiel existiert eine Instanz des *DistanceCoordinationPattern*, welches die Verhaltenskoordination zweier verbundener Shuttles realisiert.

Neben der reinen Beschreibung der Strukturdynamik eines einzelnen OCMs ist es möglich, eine regelbasierte Modellierung für die Koordination von OCMs in gegebenen Kontexten, wie einem Konvoi, zu verwenden. So ist nachzuvollziehen, dass ein Konvoi, um auch wirklich energieeffizient zu sein, aus mehr als zwei Shuttles bestehen muss. Weiterhin ist die Anzahl der Konvoiteilnehmer zum Zeitpunkt der Instanziierung der initialen Konvoikoordination unbekannt. Mal muss ein und dasselbe Koordinationsprotokoll ein Konvoi der Länge k und im nächsten Moment ein Konvoi der Länge $k + 1$ koordinieren, ohne die Stabilität eines Konvois zu verletzten. Abbildung 1.9 zeigt eine Graphtransformationsregel, die den Zusammenschluss zweier Shuttles zur Laufzeit in einem Konvoi darstellt. Hier ist zu erkennen, welche Modellelemente bei der Konvoibildung erzeugt werden.

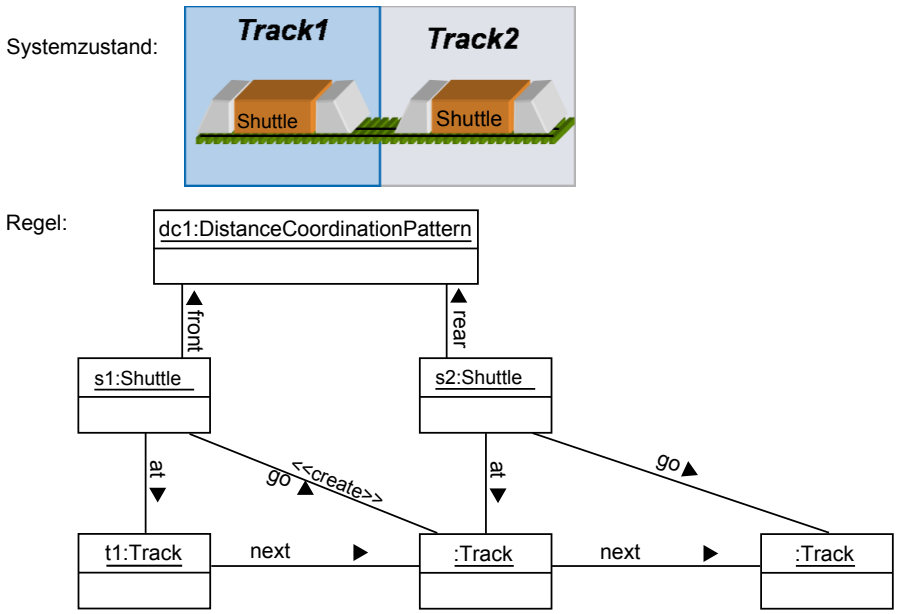


Abbildung 1.8: Beispiel für ein Graphtransformationssystem

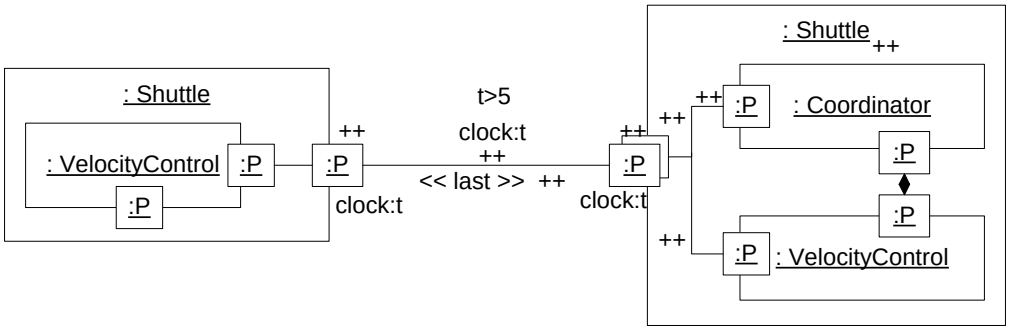


Abbildung 1.9: Regelbasierte Modellierung der Koordination

1.4 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt:

Im nächsten Kapitel 2 werden die für diese Arbeit notwendigen Grundlagen beschrieben. Hier wird zuerst der Aufbau von mechatronischen Systemen beschrieben. Anschließend wird die Idee der modell-basierten Softwareentwicklung beschrieben. An den hier festgemachten Prinzipien werden nun Konzepte, Modelle und Verifikationstechniken für me-

chatronische Systeme vorgestellt, welche die Grundlage für die MECHATRONIC UML, die als letztes vorgestellt wird, bilden.

Im Anschluss wird in Kapitel 3 der neue Ansatz zur Verifikation eines wie in Kapitel 2 eingeführten OCMs beschrieben und diskutiert.

In Kapitel 4 wird darauf eingegangen, wie sich mechatronische Systeme, dessen Hardwareressourcen beschränkt sind und ihr Verhalten kontextabhängig dynamisch anpassen, mit zeitbehafteten Graphtransformationssystemen modellieren lassen und welche Techniken hier zur Verifikation eingesetzt werden.

Kapitel 5 bedient sich der Ansätze aus den vorherigen beiden Kapiteln 3 und 4 und beschreibt die Modellierung und Verifikation von parametrisierten Koordinationsmustern mit Strukturveränderungen in mechatronischen Systemen.

In Kapitel 6 werden verwandte Arbeiten auf dem Gebiet der Verifikation von mechatronischen Systemen diskutiert.

Die vorliegende Dissertation schließt in Kapitel 7 mit einer Zusammenfassung. Dabei werden die Ergebnisse dieser Arbeit zusammengefasst und ein Überblick über mögliche Erweiterungen des Ansatzes wird gegeben.

Kapitel 2

Grundlagen

Dieses Kapitel behandelt die theoretischen Konzepte und Modelle, die als Grundlage zu dieser Arbeit dienen. Im ersten Teil (Abschnitt 2.1) werden mechatronische Systeme, wie sie in dieser Arbeit aufgefasst werden, beschrieben. Daran schließt sich ein Abschnitt über die Grundlagen der Regelungstechnik, wie sie für die Modellierung von mechatronischen Systemen benötigt werden an. Dieser Abschnitt motiviert besonders die Integration der Domäne Regelungstechnik in die Domäne der Softwaretechnik. Nachfolgend schließt sich ein Abschnitt über die Modell-basierte Softwareentwicklung mechatronischer Systeme an (siehe Abschnitt 2.3). In diesem Abschnitt werden Modelle zur Beschreibung von mechatronischen Systemen sowie Verifikationsverfahren grundlegend erklärt. Im Detail werden Automatenmodelle sowie Graphmodelle vorgestellt und deren Gemeinsamkeiten diskutiert. Weiterhin werden hierfür Verifikationsverfahren, Model Checking und Erreichbarkeitsanalysen beschrieben. Der MECHATRONIC UML Ansatz (siehe Abschnitt 2.4) integriert alle bis dahin vorgestellten Modelle und Verfahren zur Modellierung und Verifikation von mechatronischen Systemen. Der MECHATRONIC UML Ansatz bildet die Grundlage für alle in dieser Arbeit neuen Modellierungs- und Verifikationsverfahren. Das Grundlagenkapitel schließt mit einer Zusammenfassung im letzten Abschnitt 2.5.

2.1 Mechatronische Systeme

Der Begriff „Mechatronik“ (Mechanik - Elektronik) wurde 1969 von einer japanischen Firma geprägt [STF96] und bezeichnete zunächst nur die ganzheitliche Betrachtung mechanischer und elektrischer Bestandteile eines technischen Systems. Im Laufe der Zeit wurden immer häufiger Mikrokontroller zu technischen Systemen hinzugefügt, so dass die Mechatronik heute die interdisziplinäre Betrachtung mechanischer, elektrischer und informationstechnischer Bestandteile umfasst. Die stetige Zunahme des informationstechnischer Anteils ermöglicht unter anderem mechatronische Systeme, die ihr Verhalten an geänderte Bedingungen ihrer Umwelt anpassen, also eine Teilintelligenz besitzen.

In der Entwicklung mechatronischer Systeme wird zunächst ein Modell des Systems erstellt. Dieses Modell wird dann in ein reales System überführt. Aufgrund der Komplexität mechatronischer Systeme hat sich in den letzten Jahren eine komponentenbasierte Modellierung bewährt. Hierbei wird das System als Menge von Komponenten dargestellt, die Informationen verarbeiten. Diese Komponenten sind untereinander verbunden, sodass sich das Verhalten des Gesamtsystems aus der Interaktion der einzelnen Komponenten ergibt. Jede dieser Komponenten ist durch die von ihr zur Verfügung gestellten und benötigten Informationen, deren Verarbeitung, sowie die Parameter dieser Verarbeitung eindeutig charakterisiert.

Im Rahmen des SFB 614 wurde der in Abbildung 2.1 dargestellte hierarchische, modulare Aufbau von mechatronischen Systemen entwickelt. Abbildung 2.1 zeigt den Aufbau eines komplexen mechatronischen Systems nach Lückel [LHLH01][Ge05][GHH+08b].

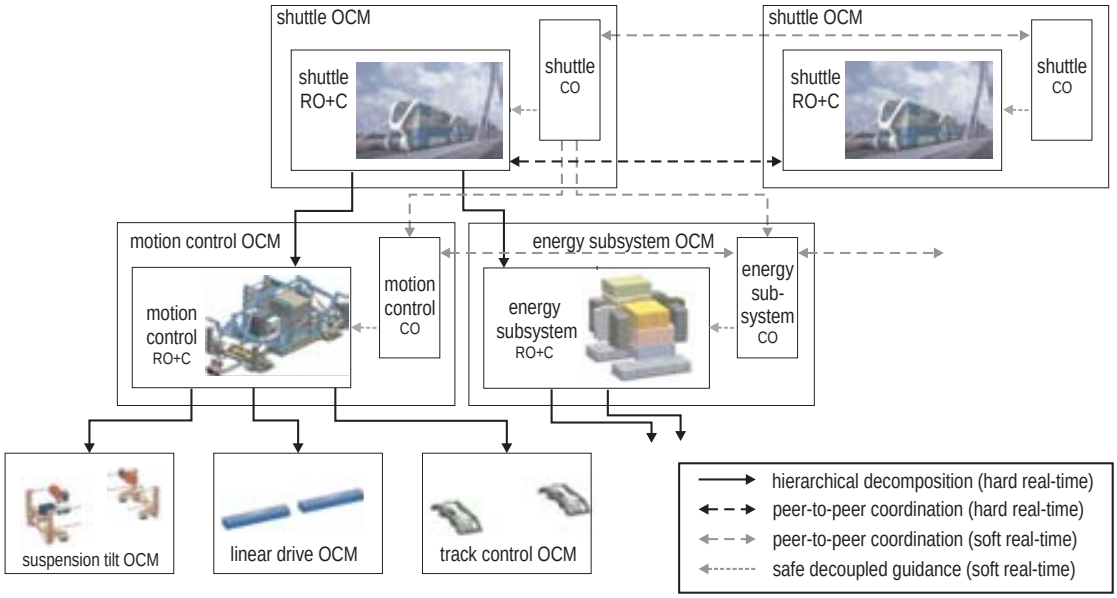


Abbildung 2.1: Hierarchische Struktur eines mechatronischen Systems nach Lückel

Die Basis bildet das *mechatronische Funktionsmodul* (MFM). Es ist aus einer mechanischen Grundstruktur, Sensoren und Aktoren und einer lokalen Informationsverarbeitung aufgebaut. Die mechanische Grundstruktur führt die Aufgaben des mechatronischen Systems in der realen Welt aus. Dazu gehört zum Beispiel das Heben einer Last oder wie in dem in Abbildung 2.1 dargestellten Beispiel das Neigen eines Fahrzeugs (*suspension tilt OCM*). Die Steuerung des Systems übernimmt die Informationsverarbeitung. Sie kommuniziert über Sensoren und Aktoren mit der mechanischen Grundstruktur. *Autonome mechatronische Systeme* (AMS) sind aus MFM aufgebaut, die informationstechnisch oder

mechanisch gekoppelt sind. Die Informationsverarbeitung eines AMS übernimmt übergeordnete Aufgaben. Dazu gehören zum Beispiel die Überwachung mit Fehlerdiagnose und Instandhaltungsentscheidungen (*motion control OCM*). Außerdem werden Vorgaben für die lokale Informationsverarbeitung generiert. Aus den AMS werden vernetzte mechatronische Systeme (VMS) gebildet. Sie entstehen durch die Verbindung von AMS über die Informationsverarbeitung. Im dargestellten Beispiel entspricht das Feder-Neige-Modul dem MFM, das Shuttle einem AMS und ein Fahrzeugverband einem VMS. In der Abbildung ist zu sehen, dass jede Schicht, MFM, AMS sowie VMS durch OCMs beschrieben wird.

Abbildung 2.2 zeigt ein Operator-Controller-Modul (OCM) [Ge05][HOG04]. Das OCM ist in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator aufgeteilt. Der Controller bildet die unterste Ebene. Er arbeitet direkt mit der mechanischen Grundstruktur, verarbeitet auf direkte Weise die Messsignale, ermittelt daraus Stellsignale und gibt sie an die mechanische Grundstruktur weiter. Der Controller arbeitet kontinuierlich und unter harten Echtzeitbedingungen. Der reflektorische Operator bildet die mittlere Ebene. Er steuert den Controller und unterliegt ebenfalls harten Echtzeitbedingungen. Er agiert nicht direkt mit dem System, sondern beeinflusst den Controller durch Initiierung von Parameter- und Strukturänderungen. Die oberste Ebene bildet der kognitive Operator. Auf dieser Ebene kann das System Wissen über sich und die Umgebung zur Verbesserung des eigenen Verhaltens nutzen. Der kognitive Operator unterscheidet sich von den anderen beiden Ebenen vor allem dadurch, dass er weichen Echtzeitanforderungen unterliegt.

Das dargestellte System kann also in zwei Teile untergliedert werden: Ein Teil, der unter harten Echtzeitbedingungen arbeitet und den Controller und den reflektorischen Operator umfasst und ein Teil, der unter weichen Echtzeitbedingungen arbeitet. Zum letzteren gehört der kognitive Operator. Um zu verstehen, wie das Werte-kontinuierliche Verhalten des Controllers spezifiziert ist, werden im Folgenden Grundlagen der Regelungstechnik beschrieben.

2.2 Regelungstechnik

Technische Systeme lassen sich im regelungstechnischen Sinn durch Zustandsgrößen beschreiben. In vielen Fällen will man diese Zustandsgrößen gezielt beeinflussen, um gewünschtes Verhalten zu erzielen. Das Ziel ist es, die Zustandsgrößen an einen bestimmten Wert zu ändern oder sie an einem Wert zu halten (z.B. die Rotationsgeschwindigkeit soll immer $100\frac{rad}{s}$ betragen).

In der Regelungstechnik liegt der Schwerpunkt nicht auf der Konstruktion eines Systems, sondern auf der Beschreibung der kontinuierlichen Zustandsgrößen durch Differen-

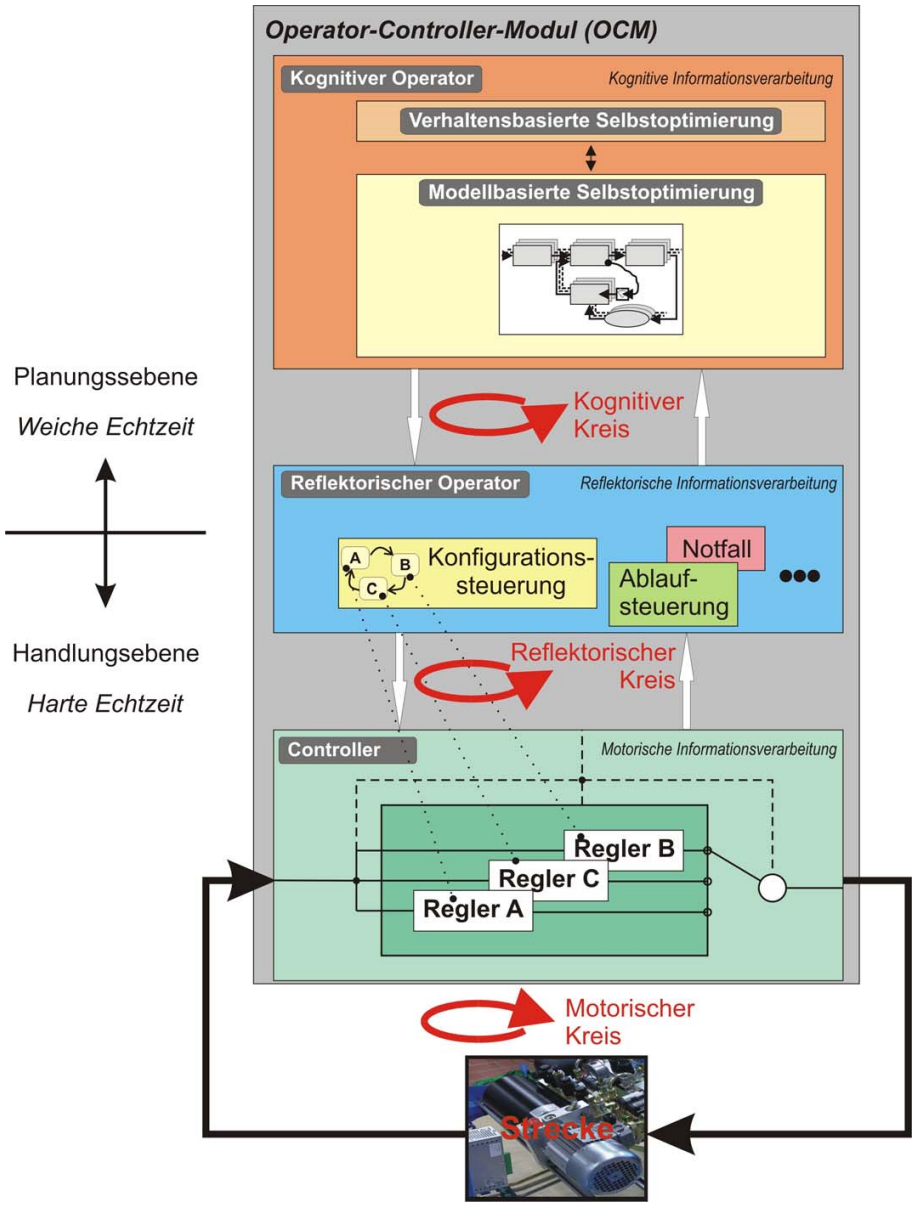


Abbildung 2.2: Operator-Controller-Modul

tialgleichungen. Hierbei wird das dynamische Verhalten der Regler durch Differentialgleichungen beschrieben, die dafür sorgen, dass sich das System wie gewünscht verhält [Föl05].

In Abbildung 2.3 ist die generelle Struktur einer Steuerung dargestellt. 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 Auf-

gabe 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. x und y sind Elemente eines Vektors $\vec{x}$ bzw. $\vec{y}$. Da Systeme mit mehr als einer Zustandsgröße und einer Stellgröße nach demselben Prinzip funktionieren, werden im Folgenden nur Systeme mit x und y betrachtet.

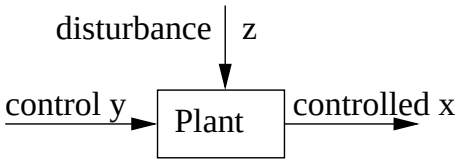


Abbildung 2.3: Generelle Struktur einer Steuerung

Grundsätzlich wird zwischen Steuerungen (Feed-Forward-Regler] und Reglern (Feed-Back-Reglern) unterschieden. Steuerungen reagieren schneller auf a priori bekannte Störungen, allerdings nicht auf unbekannte Störungen. Regler reagieren durch den Regelkreis 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. (siehe Abbildung 2.4).

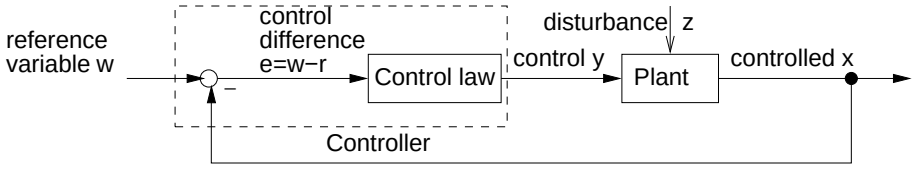


Abbildung 2.4: Einfacher Regelkreis

Die Entscheidung für einen bestimmten Regler-Typ hängt von den Zeiteigenschaften und der Genauigkeit des Systems ab. In der Literatur wird hier zwischen drei Regler-Typen unterschieden: Proportionalregler (P), Proportional-Integral-Regler (PI) und Proportional-Integral-Differential-Regler (PID). Letzterer wird am meisten in der Praxis verwendet [HPPS03]. Je nach Eigenschaft der Strecke und der vorgegebenen Anforderungen werden nun die Regler ausgesucht, um eine möglichst hohe Stabilität des Systems zu erreichen dabei Überschwingen zu vermeiden und Reaktionszeiten zu verbessern.

2.2.1 Adaptive Regler

Einige Systemänderungen sind nicht vorhersagbar und gewöhnliche Regelsysteme können möglicherweise nicht richtig reagieren, wenn die Eingangs- und Ausgangsrelation sich verändert. Manchmal können diese Effekte durch herkömmliche Regelungstechniken geregelt werden, jedoch nicht immer [IML92].

Adaptive Regelungen können hier helfen, die Stabilität sowie die Reaktionszeit zu verbessern. Dieser Ansatz verändert die Regler-Algorithmen in Echtzeit, um sich an die Änderungen der Umgebung anzupassen. Allgemein beobachtet der Regler periodisch die System Eingabe- und Ausgaberelation und ändert den Regler-Algorithmus. Das Ziel ist es, dadurch den Regler so robust wie möglich zu haben, und damit das dynamische System so unempfindlich gegen Störungen wie möglich zu halten.

In der Literatur gibt es drei Hauptansätze, um adaptive Feed-Back Regler zu modellieren: Der erste, triviale Ansatz legt z.B. in einer Datenbank a priori fest, wie sich der Regler bei bestimmten Änderungen zu verhalten hat. Neben der trivialen adaptiven Regelung gibt es noch den *Model Reference Adaptive Control* (MRAC) und *Self-Tuning Regulators* (STRs) Ansatz. Beim MRAC Ansatz beschreibt ein Referenzmodell die Systemeigenschaften. Der adaptive Regler ist hier so aufgebaut, dass das System bzw. die Strecke sich so verhält, wie das Referenzmodell. Die Ausgaben des Modells werden mit den tatsächlichen Ausgaben verglichen und die Differenz wird verwendet, um die Regler-Parameter anzupassen. Im dritten Ansatz STRs werden selbsteinstellende Regler verwendet. Diese nehmen ein lineares Modell an. Die Regler verwenden die zugrunde liegenden Reglergesetze, um ihre Koeffizienten zu verändern.

2.2.2 Rekonfiguration

Bis jetzt wurden nur Regler betrachtet, die immer aktiv sind. Zusätzlich wurde angenommen, dass für eine Aufgabe immer ein Regler zur Verfügung steht. Für das Shuttle System wird Rekonfiguration benötigt, um die Regler-Algorithmen auszutauschen, wenn z.B. ein Konvoi gebildet wird um die beste Strategie zu fahren. Da die Ressourcen in eingebetteten Systemen typischerweise begrenzt sind, ist es erforderlich, diese soweit wie möglich einzusparen. Hierfür wird Rekonfiguration verwendet, um zwischen verschiedenen Rollen, z.B. Führungsfahrzeug oder letztes Shuttle im Konvoi, hin und her zuschalten. Um dies zu ermöglichen, muss eine Logik, welche die Reglerstruktur steuert, hinzugefügt werden. Durch Hybride Rekonfigurations Charts (siehe Kapitel 2.4) kann modelliert werden, welche Regler in welcher Situation aktiv oder inaktiv sein sollen.

2.2.3 Block Diagramme

Eine gängige Technik für die Modellierung von Reglerstrukturen sind hierarchische Block Diagramme. Block Diagramme bestehen aus Grund-Blöcken, die das Verhalten modellieren und hierarchischen Blöcken, welche die Grund-Blöcke oder andere hierarchische Blöcke beinhalten, um die visuelle Komplexität zu reduzieren. Jeder Block besitzt Eingabe- und Ausgabesignale. Blöcke sind durch gerichtete Verbindungen untereinander verbun-

den, um den Informationsfluss darzustellen. Z.B. kann das Ausgabesignal eines Blocks als Eingabesignal eines anderes Blocks verwendet werden.

Definition 1
Ein kontinuierlicher Block M wird durch ein 7 Tupel $(V^x, V^u, V^y, F, G, C, X^0)$ definiert:

- V^x*: Menge der Zustandsvariablen,*
- V^u*: Menge der Eingabevariablen,*
- V^y*: Menge der Ausgabevariablen,*
- $F \subseteq EQ(V^{\dot{x}} \bigcup V^a, V^x \bigcup V^u \bigcup V^a)$ *mit* $V^a = V^y \bigcap V^u$*: beschreibt den Fluss der Zustandsvariablen,*
- $G \subseteq EQ(V^{\dot{y}} \bigcup V^a, V^x \bigcup V^u \bigcup V^a)$*: bestimmt die Ausgabevariablen,*
- $C \in COND(V^x)$*: Invariante, welche die Menge der zulässigen Zustände bestimmt und*
- X^0*: Menge der Anfangszustände.*

Die Menge $EQ(V_l, V_r)$ bezeichnet alle Gleichungen der Form $v_l = f^i(v_r^1, ..., v_r^n)$, mit den Funktionen f^i, die n Argumente besitzen, und den Variablen $v_l \in V_l$ und $v_r^1, ..., v_r^n \in V_r$. Die Menge $COND(V)$ beinhaltet alle Bedingungen über die Variable V.

Ein Block M ist wohl-definiert, wenn für alle Differentialgleichungen des Systems $F \bigcup G$ gilt, dass es keine zyklischen Abhängigkeiten gibt, keine doppelten Zuweisungen, alle undefinierten Verweise auf Variablen in $V^u - V^y$ enthalten sind und jeder Zustandsvariablen V^x und Ausgabevariablen V^y ein Wert zugewiesen ist.

2.2.3.1 Beispiel

Wie in Kapitel 1.2 beschrieben, wird die Konvoifahrt genutzt, um möglichst energieeffizient zu fahren. Das ist nur dann möglich, wenn die Shuttles möglichst nah hintereinander fahren. Hierbei muss die Distanz ständig geregelt werden, um Auffahrunfälle zu vermeiden. Die Geschwindigkeit des Shuttles muss also ständig in Bezug auf die Distanz zum Führungsfahrzeug angepasst werden. Um dies zu erreichen, benötigt das hinterherfahrende Shuttle zwei Regler, einen Distanzregler und einen Geschwindigkeitsregler. Das Führungsshuttle hingegen benötigt nur einen Geschwindigkeitsregler.

Als erstes wird das Modell der Strecke beschrieben. Dieses ist in Abbildung 2.5 dargestellt. Es gilt: $m\dot{v} + bv = u, y = v$, wobei m die Masse eines Shuttles, v die Geschwindigkeit, $\dot{v}$ die Ableitung der Geschwindigkeit v, bv die Reibungkraft, u die Antriebskraft und y die Zustandsgröße ist.

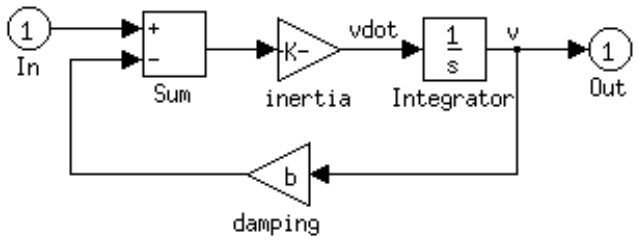


Abbildung 2.5: Model der Strecke

Als nächstes wird der PID Regler beschrieben. Der Unterschied zwischen dem gewünschten Eingabewert und dem tatsächlichen Ausgabewert wird durch den Fehler e dargestellt (siehe Abbildung 2.4). e wird an den PID Regler weitergeleitet, der daraufhin die Ableitung sowie das Integral von e berechnet. Das Signal u ist nun, nachdem es zum Regler weitergeleitet wurde, gleich dem Proportionalglied K_P multipliziert mit dem Betrag des Fehlers plus dem Integrationsglied K_I multipliziert mit dem Integral des Fehlers plus dem Differenzialglied K_D multipliziert mit der Ableitung des Fehlers. Das Signal u wird danach an die Strecke weitergegeben. Der neue Ausgabewert x wird entsprechend hergeleitet. Der neue Ausgabewert x wird erneut zurück an den Sensor geleitet, um den neuen Fehler e zu bekommen. Der Regler bekommt diesen neuen Fehler und berechnet die Ableitung und das Integral erneut (siehe Abbildung 2.6). Der PID Regler wurde entsprechend am linearisierten Modell der Strecke ausgelegt.

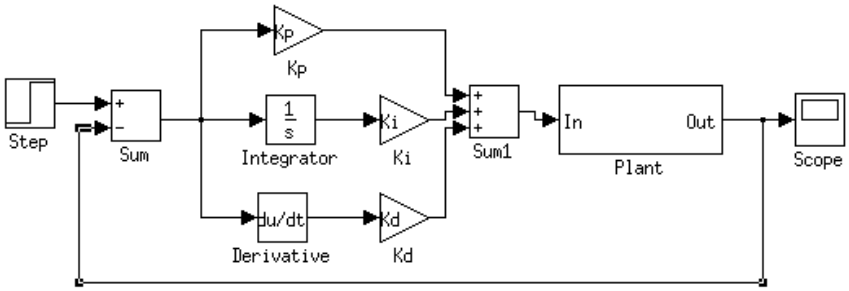


Abbildung 2.6: PID Geschwindigkeitsregler

2.2.3.2 Rekonfiguration

Abbildung 2.7 zeigt die Logik der Abstandsregelung in einem Stateflow Diagramm. Das Modell besteht aus drei Modi. Initial ist das Shuttle im Modus NoConvoy. Die Ereignisse convoyFront und convoyRear erzwingen den Wechsel von NoConvoy zum Modus ConvoyFront oder ConvoyRear. Das Ereignis breakConvoy löst den Konvoi auf und erzwingt

den Wechsel in den Modus NoConvoy. In den Modi NoConvoy und ConvoyFront ist die Geschwindigkeitsregelung implementiert. Im Zustand ConvoyRear ist die Distanzregelung implementiert.

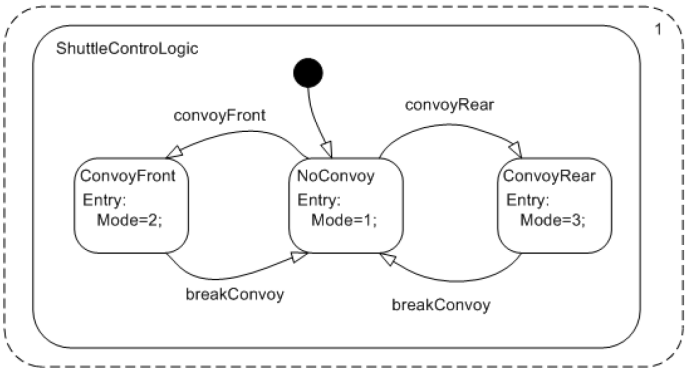


Abbildung 2.7: Stateflow Diagramm der Shuttlesteuerung

In Abbildung 2.8 ist das hybride Modell einer Shuttleregelung für die Konvoiregelung als Simulink Diagramm dargestellt. Es beinhaltet die Reglerlogik (oben links), die Reglergesetze für die Distanz- und Geschwindigkeitsregelung (mitte) und die Beschreibung des physikalischen Modells (rechts). Der Block zur Distanzkontrolle beinhaltet sowohl den Distanzregler als auch den Geschwindigkeitsregler. Der schwarze Balken (rechts) ist ein Umschalter. Durch die Mode/Konfigurationseingabe beider Kontroller und dem Umschalter ist es möglich, den gerade benötigten Regler zu aktivieren und einen anderen zu deaktivieren.

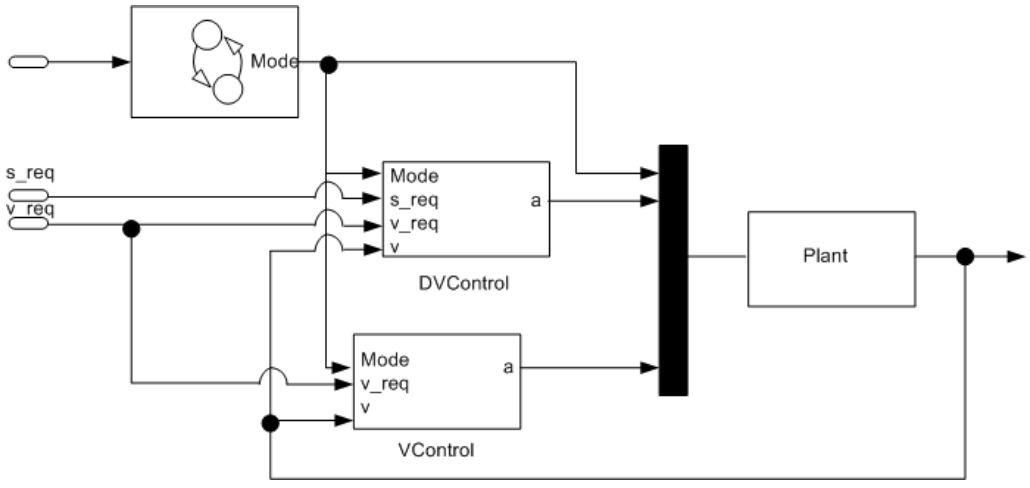


Abbildung 2.8: Hybrides Modell der Shuttelsteuerung

2.3 Modell-basierte Softwareentwicklung

Die Techniken der modell-basierten Softwareentwicklung wurden eingeführt, um die Komplexität eines Problems zu reduzieren und ein abstrakteres Modell zu erhalten, welches frei von Implementierungsdetails ist [Fav05]. Damit wird das Modell übersichtlicher und es lassen sich einfacher Operationen durchführen. Drei wesentliche Schritte in der modell-basierten Entwicklung sind die (1) Modellierung, (2) Verifikation und (3) Codesynthese.

Die UML hat sich als Standardmodellierungssprache für die modell-basierte Softwareentwicklung durchgesetzt. Sie bietet eine große Palette von Diagrammen, um ein Systemmodell zu erstellen und ihr Verhalten zu spezifizieren. Um allerdings komplexe, vernetzte mechatronische Systeme, wie sie bereits vorgestellt wurden, adäquat zu modellieren, reicht die UML nicht mehr aus. Um dem domänenübergreifenden Charakter von mechatronischen Systemen gerecht zu werden, müssen bei der modell-basierten Entwicklung auch die entsprechenden Techniken der verschiedenen Domänen miteinander integriert werden [Bur06][HHKS08][GHH+08b][BGH+07]. Neben der Modellierung steht bei der modell-basierten Entwicklung auch die Verifikation im Vordergrund. Da nun Modelle des eigentlichen System vorliegen, ist es möglich, diese formal zu verifizieren. Jedoch wirft die Welt der komplexen, vernetzen mechatronischen Systeme auch hier Probleme auf. Bisherige Ansätze lassen sich nicht anwenden, da sie mit der Vielfalt der Modelle der anderen Domänen nicht entsprechend klar kommen oder nicht skalieren [GH06].

Im Folgenden werden nun zuerst Verhaltens- und Strukturmodelle vorgestellt, die bei der klassischen Modellierung und Verifikation von mechatronischen Systemen bisher verwendet wurden. Die Auswahl der aufeinander aufbauenden Modelle wurde getroffen, da sie die syntaktische und semantische Grundlage für die später in der MECHATRONIC UML eingeführten Modelle bilden.

2.3.1 Automaten

Um das Verhalten von reaktiven Systemen zu modellieren, können endliche Automaten verwendet werden. Von den verschiedenen existierenden Formen von Automaten werden hier die endlichen Automaten betrachtet, welche auch bei [CGP00] als Grundlage für das dort definierte Modell eines Timed Automaton dienen. Für weitergehende Informationen zu endlichen Automaten siehe [HU79]. Ein endlicher Automat setzt sich zusammen aus einzelnen Knoten, welche über Kanten miteinander verbunden sind. Die Knoten stellen einzelne Zustände und die Kanten Transitionen zwischen diesen dar.

Definition 2
Ein endlicher Automat M ist ein Quadrupel $M := (\Sigma, \mathcal{Q}, \Delta, \mathcal{Q}^0)$, wobei Σ ein endliches Eingabealphabet, $\mathcal{Q}$ eine endliche Menge an Zuständen, $\Delta \subseteq \mathcal{Q} \times \Delta \times \mathcal{Q}$ eine Transitionsrelation und $\mathcal{Q}^0$ die endliche Menge der initialen Startzustände darstellt.

Ein endlicher Automat M besteht aus $\mathcal{Q}$, einer endlichen Menge an Knoten bzw. Zuständen, Δ der Menge an Kanten, welche die Transitionen zwischen den Zuständen abbilden, aus $\mathcal{Q}^0$, einer endlichen Menge an initialen Startzuständen sowie aus Σ. Σ setzt sich zusammen aus einer Menge an möglichen Eingaben, bei welchen über die Transitionen $\delta \in \Delta$ zwischen den Zuständen aus $\mathcal{Q} \cup \mathcal{Q}^0$ gewechselt werden kann. Ein Beispiel für einen solchen Automaten zeigt die Abbildung 2.9. Der abgebildete Automat setzt sich zusammen aus der Eingabemenge $\Sigma = \{a, b\}$, den Zuständen $\mathcal{Q} = \{s_0, s_1\}$ und dem Startzustand $\mathcal{Q}^0 = \{s_0\}$. Die Menge der Übergangstransitionen Δ lässt sich beschreiben durch die Tripel (s_0, a, s_1) und (s_1, b, s_0).

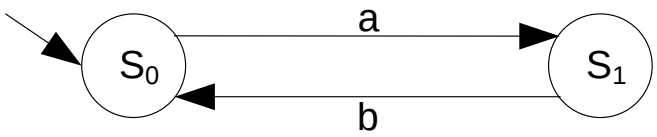


Abbildung 2.9: Ein endlicher Automat mit den Zuständen s_0, s_1 und zwei Kanten, welche mit a und b beschriftet sind.

Die in diesem Abschnitt vorgestellten Modelle eignen sich zur Beschreibung von Systemen, welche durch ein diskretes Zeitmodell (siehe [CGP00], Kapitel 16) beschrieben werden können. Um Zeit-kontinuierliche Eigenschaften zu modellieren, müssen die Modelle erweitert werden. Es gibt verschiedene Möglichkeiten, um Zeit in einem Modell abzubilden. Dies bezieht sich z.B. auf das Verhalten von Uhren (eine solche Uhr wird nachfolgend Clock genannt), durch welche das Fortschreiten der Zeit abgebildet wird. So wird bei [CGP00][Kop97] unterschieden zwischen diskreter und kontinuierlicher Echtzeit, wobei in der vorliegenden Arbeit Modelle mit kontinuierlichem Echtzeit-Verhalten betrachtet werden. Als Gemeinsamkeit für die hier verwendeten Modelle gilt weiterhin, dass alle dort auftretenden Uhren mit der gleichen Geschwindigkeit voranschreiten und somit synchron laufen. In [GHH06a] wird beschrieben, weshalb diese Annahme für mechatronische Systeme, wie hier beschrieben, gültig ist.

2.3.2 Timed Automata

Synchrone Modelle basieren meistens auf einem diskreten Zeitmodell. Ein Beispiel ist das gleichmäßige Takten einer Hardwareeinheit. Dies hat zur Folge, dass als Uhrwerte

nur positive ganzzahlige Werte zur Verfügung stehen und Ereignisse nur zu ganzzahligen Zeitpunkten auftreten können. Will man nun asynchrone Modelle untersuchen, muss man ein kontinuierliches Zeitmodell zu Grunde legen. Hier können z.B. Ereignisse in beliebig kleinen Abständen hintereinander auftreten. Um solche Systeme zu diskretisieren, ist es nötig, kleine, a priori vorgegebene minimale Zeitintervalle, festzulegen. Abstände zwischen Ereignissen können nun als Vielfache dieser Maßeinheit ausgedrückt werden. Dieses ist jedoch schwer von Beginn an festzulegen und schränkt zusätzlich die Genauigkeit eines Systems ein. Brzozowski und Seger [BS91] haben sogar gezeigt, dass das Erreichbarkeitsproblem für asynchrone Schaltkreise unter der Annahme von festen Zeitintervallen, Zeit wird diskretisiert, nicht korrekt gelöst werden kann. Hinzu kommt, dass der Zustandsraum bei der Verwendung von sehr kleinen Intervallen, wie sie bei einer möglichst exakten Modellierung eines asynchronen Systems vorkommen, schlagartig explodiert. Dadurch wird die Verifikation undurchführbar. Obgleich viele verschiedene Ansätze zur Modellierung von Systemen mit einem kontinuierlichen Zeitsystem gemacht wurden, hat sich das Modell der Timed Automata von Alur, Courcoubetis und Dill [ACD90] etabliert, welches im Folgenden beschrieben wird.

Ein Timed Automaton ist ein endlicher Automat, der über eine feste Anzahl von Clocks verfügt. Eine Clock kann dabei einen Wert aus den positiven reellen Zahlen annehmen, wodurch Modelle mit kontinuierlichen Echtzeit-Bedingungen formuliert werden können. Die Transitionsrelation Δ des Timed Automaton verfügt zusätzlich zu den endlichen Automaten über zeitliche Bedingungen, welche erfüllt sein müssen, damit der jeweilige Übergang stattfinden kann. Diese Bedingungen werden als Guards bezeichnet. Weiterhin kann eine solche Transition über so genannte Resets verfügen, welche beim Schalten dafür sorgen, dass einzelne Clocks auf den Zahlenwert 0 zurück gesetzt werden.

Ähnlich wie die Kanten (Kanten repräsentieren die Transitionen) über Guards verfügen können, kann jeder Knoten ebenfalls zeitliche Bedingungen besitzen, so genannte Invarianten. Diese müssen erfüllt sein, damit bei dem jeweiligen Knoten verweilt werden kann. Entsprechend ist es möglich bei einem Knoten zu verweilen, während die Transitionen zwischen den einzelnen Knoten in Null-Zeit passiert.

Im Unterschied zu den einfachen endlichen Automaten, bei denen die Zustände durch die Menge $\mathcal{Q}$ beschrieben werden können, hängt der Zustand beim Timed Automaton zusätzlich von den geltenden Bedingungen über die einzelnen Clocks ab. Somit wird hier nicht von der Menge der Zustände $\mathcal{Q}$, sondern von so genannten Locations $\mathcal{S}$ gesprochen, welche den Knoten des Automaten zugeordnet sind. Dabei kann es innerhalb einer Location für die einzelnen Clocks nicht nur eine Belegung geben, sondern eine unendlich große Menge an möglichen Belegungen. Diese Eigenschaft ergibt sich aus der Verwendung von Ungleichungen bei den Invarianten und Guards, durch welche komplette zeitliche Erreichbarkeitsräume aufgebaut werden können. So ist es möglich innerhalb einer Location für eine Zeitspanne zu verweilen. Dabei können die Clockvariablen Werte aus den posi-

tiven reellen Zahlen annehmen. Auf diese Eigenschaft der Locations wird im Folgenden noch genauer eingegangen.

Ein Beispiel für einen entsprechenden Timed Automaton zeigt die Abbildung 2.10. Der Automat verfügt über die Locations s_0, s_1, die zwei Clock-Variablen x_1, x_2 sowie über die Transition a, welche s_0 mit s_1 verbindet und die Transition b, welche s_1 mit s_0 verbindet. Der Automat startet in der Location s_0 und kann dort höchstens solange verweilen, bis die Clock x_2 den Wert 2 erreicht hat. Dies wird durch die Invariante $x_2 \leq 2$ in s_0 vorgegeben. Sobald die Clockvariable x_2 den Wert 1 erreicht oder überschreitet, kann der Automat über die Transition a zu der Location s_1 wechseln. Dieser Wechsel muss spätestens vollzogen werden, wenn die Clock x_2 den Wert 2 erreicht. Diese Eigenschaft ergibt sich durch die Kombination des entsprechenden Guards $x_2 \geq 1$ an der Transition a, zusammen mit der Invariante innerhalb der Location s_0. Beim Übergang von s_0 nach s_1 wird die Clock-Variable x_2 durch den Reset $x_2 := 0$ zurück gesetzt. Ähnlich verfügt die Location s_1 über die Invarianten $x_2 \leq 3$ und $x_1 \leq 2$, sowie die Transition b über den Guard $x_1 \geq 2$ und den Clock-Reset $x_1 := 0$.

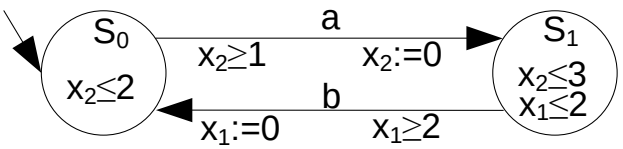


Abbildung 2.10: Ein Timed Automaton, der über zwei Location, drei Invarianten und zwei Kanten mit jeweils einem Guard und einem Clockreset verfügt.

Formal ist ein Timed Automaton nach [CGP00] wie folgt definiert:

Definition 3
Ein Timed Automaton A ist ein 6-Tupel $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, T)$, wobei Σ ein endliches Eingabealphabet, $\mathcal{S}$ eine endliche Menge an Locations, $\mathcal{S}^0 \subseteq \mathcal{S}$ eine endliche Menge von Start-Locations, $X := (x_1, .., x_n)$ 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, und T ist die Menge der Transitionen. $\mathcal{C}(X)$ ist eine Menge von Bedingungen über die 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 $T \subseteq \mathcal{S} \times \Sigma \times \mathcal{C}(X) \times 2^X \times \mathcal{S}$. Eine Transition von Location s nach s' läßt sich durch ein 5-Tuple $(s, a, \varphi, \lambda, s')$ 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 zurück gesetzt werden.

2.3.3 Hybride Automaten

Mit einem hybriden Automaten kann das Verhalten von hybriden Systemen modelliert werden. Hybride Systeme sind dadurch gekennzeichnet, dass sie sowohl aus einem diskreten wie auch einem kontinuierlichen Teil bestehen. Der hybride Automat stellt eine Erweiterung zum endlichen Automaten und zum Timed Automaton dar, da er zusätzlich zu einem diskreten Teil auch einen kontinuierlichen Teil besitzt.

Jeder hybride Automat besteht aus Locations und Transitionen, die einen Übergang zwischen zwei Locations ermöglichen. Wie auch beim endlichen Automaten existiert im hybriden Automaten eine ausgezeichnete Teilmenge der Locations, welche die Anfangslocations bilden. Das Hauptmerkmal des hybriden Automaten ist, dass er es erlaubt, in jeder Location eine Differentialgleichung, die einen kontinuierlichen Regler repräsentiert, einzubetten. Der Übergang zwischen zwei Locations erfolgt durch diskrete Übergänge. Durch diese Einbettung und die Übergänge kann mit Hilfe eines hybriden Automaten das Verhalten von hybriden Systemen modelliert werden. Des Weiteren wird durch den Wechsel von einer Location in eine andere ein Austausch von kontinuierlichen Reglern erreicht.

Neben der Einbettung von kontinuierlichen Reglern ermöglicht der hybride Automat wie auch der Timed Automaton die Spezifikation von Zeitangaben. Innerhalb einer Location können Zeitinvarianten in Bezug auf eine Uhr angegeben werden. Eine Zeitinvariante drückt aus, bis wann eine Location spätestens verlassen sein muss. Ein Locationübergang kann nur stattfinden, wenn der spezifizierte Timeguard wahr ist und das entsprechende Event anliegt. Während des Schaltvorgangs einer Transition ist es möglich, Uhren auf den Wert Null zu setzen. Nach [BGH05a] ist ein *hybrider Automat* M wie folgt definiert:

Definition 4
Ein hybrider Automat *ist durch ein 6-Tupel* (L, D, I, O, T, S^0) *definiert. Dabei ist* L *eine endliche Menge von Locations,* D *eine Funktion über* L*, die jeder Location* $l \in L$ *ein kontinuierliches Modell* $D(l) = (V^x, V^u, V^y, F(l), G(l), C(l), X^0(l))$ *wie in Definition 1 beschrieben, zuweist,* I *eine endliche Menge von Eingabesignalen,* O *eine endliche Menge von Ausgabesignalen,* T *eine endliche Menge von Transitionen und* $S^0 \subseteq \{(l, x) | l \in L \wedge x \in X^0(l)\}$ *die Menge der Anfangslocations.*

Für jede Transition $(l, g, g', a, l') \in T$ *ist* $l \in L$ *die Startlocation,* $g \in COND(V^x \cup V^u)$ *der kontinuierliche Guard, der eine Bedingung über die Zustands- oder die Eingabevariablen angibt,* $g' \in \wp(I \cup O)$ *der I/O-Guard,* $a \in [[V^x \rightarrow \mathbb{R}] \rightarrow [V^x \rightarrow \mathbb{R}]]$ *die kontinuierliche Aktualisierung und* $l' \in L$ *Ziellocation.*

Abbildung 2.11 aus [Bur06] zeigt einen hybriden Automaten, der das Fahrverhalten eines Shuttles modelliert. Ein Shuttle kann entweder alleine oder im Konvoi fahren. Abhängig von der Situation ist ein bestimmter Regler aktiv. Nutzt das Shuttle den Geschwindig-

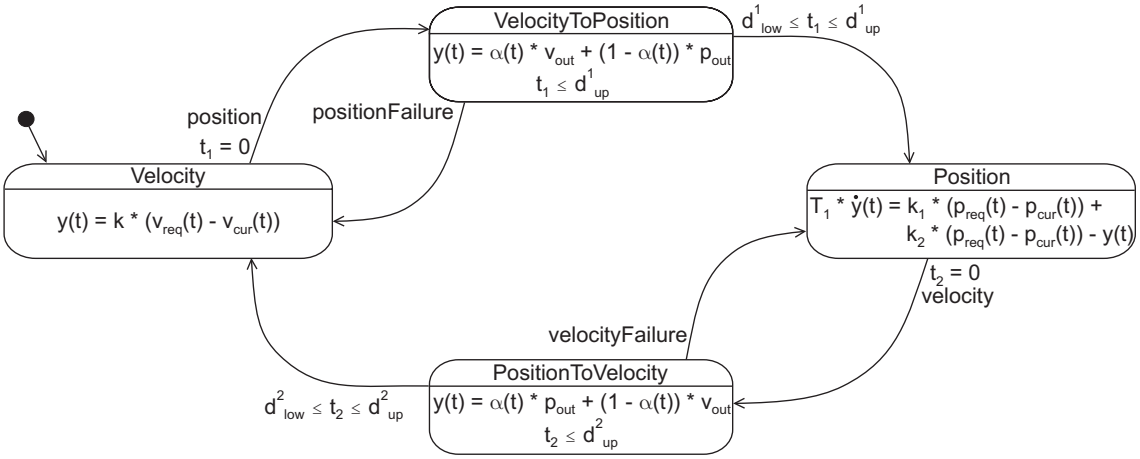


Abbildung 2.11: Hybrider Automat

keitsregler, bewegt es sich mit einer bestimmten Geschwindigkeit fort, die sich innerhalb eines vorgegebenen Intervalls befindet. Wenn es mit einem anderen Shuttle in einem Konvoi fährt, wird die Geschwindigkeit des Shuttles anhand seiner Position geregelt. Hierbei stellt der Positionsregler sicher, dass der Abstand zwischen den beiden Shuttles nicht zu groß bzw. nicht zu klein wird.

Zu Anfang befindet sich das Shuttle in der Location *Velocity*. Diese Location beinhaltet eine Differentialgleichung, die den Geschwindigkeitsregler repräsentiert. Anhand der Differentialgleichung ist zu erkennen, dass der Ausgang *y* des Reglers von den beiden Eingängen v_{req} und v_{cur} und einer Konstante *k* abhängt.

Liegt das Event *position* an, wird vom Geschwindigkeitsregler zum Positionsregler gewechselt. Um den Wechsel zwischen diesen beiden Reglern zu ermöglichen, existiert ein so genannter Überblendzustand. Dieser Überblendzustand enthält eine Funktion, die den Ausgang des Geschwindigkeitsreglers auf den Ausgang des Positionsreglers überblendet. Diese Funktion wiederum beinhaltet die Überblendfunktion $\alpha(t)$ aus [Voe03] mit: $\alpha(t) = -2(\frac{t-t_0}{t_{dauer}})^3 + 3(\frac{t-t_0}{t_{dauer}})^2$

Wird von der Location *Velocity* in die Location *VelocityToPosition* gewechselt, wird der Wert der Uhr t_1 auf Null gesetzt. In der Location *VelocityToPosition* befindet sich eine Zeitinvariante, die aussagt, dass die Location verlassen sein muss, wenn $t_1 > d^1_{up}$ gilt. Liegt das Event *positionFailure* an, ist während des Überblendvorgangs ein Fehler aufgetreten und der hybride Automat wechselt wieder in die Location *Velocity*. Liegt kein Event an und ist der Timeguard $d^1_{low} \leq t_1 \leq d^1_{up}$ wahr, verlässt der hybride Automat die Location *VelocityToPosition* und wechselt in die Location *Position*. Diese Location beinhaltet eine Differentialgleichung, die den Positionsregler repräsentiert. Der Ausgang *y* der Differentialgleichung ist von den beiden Eingängen p_{req} und p_{cur} und den Konstanten k_1 und k_2 abhängig.

Liegt das Event *velocity* an, muss vom Positionsregler auf den Geschwindigkeitsregler gewechselt werden. Hierfür existiert der Überblendzustand *PositionToVelocity*. Wenn die Location *Position* verlassen wird, wird der Wert der Uhr t_2 auf Null gesetzt. In der Location *PositionToVelocity* ist eine Funktion eingebettet. Diese wiederum beinhaltet die Überblendfunktion $\alpha(t)$. Die Zeitinvariante $t_2 \leq d_{up}^2$ sagt aus, bis wann die Location spätestens verlassen sein muss. Die Location kann verlassen werden, wenn das Event *velocityFailure* anliegt oder der Timeguard $d_{low}^2 \leq t_2 \leq d_{up}^2$ wahr ist.

Wie dieses Beispiel zeigt, vereinen hybride Automaten kontinuierliche Regler und diskrete Zustände. Durch diese Kombination ermöglichen sie unter anderem die Simulation von physikalischen Gesetzen, die häufig für hybride Systeme benötigt werden. Zudem ist eine automatische, formale Überprüfung von Invarianten möglich.

Allerdings werden alle Komponenten durch einen einzigen hybriden Automaten modelliert. Dies führt schnell zu komplexen Modellen, die schwer zu verifizieren sind. Des Weiteren ermöglicht das Modell des hybriden Automaten keine Rekonfiguration, wie sie in [FGK+04][Bur06] definiert ist. Durch einen Wechsel der Locations werden zwar die Regler ausgetauscht, aber die Struktur und der interne Zustand der Regler können von einem hybriden Automaten nicht beeinflusst werden. Wie auch beim Timed Automaton findet beim hybriden Automaten ein Schaltvorgang in Null Zeit statt. Dies entspricht jedoch nicht der Realität, da ein Locationwechsel oft eine gewisse Zeit benötigt.

2.3.4 Graphen

Die in den vorherigen Abschnitten vorgestellten Modelle der Timed Automata und der Hybriden Automaten verfügen über eine zeitliche Komponente, mit welcher es möglich ist, kontinuierliches (Echtzeit-)Verhalten zu modellieren. Das Verhalten ist dabei auf eine vorgegebene Zustandsstruktur festgelegt. In komplexen, vernetzten mechatronischen Systemen stehen nur begrenzte Rechen- und Speicherkapazitäten zur Verfügung. Zusätzlich unterliegt das System zur Laufzeit einer Evolution abhängig vom gegebenem Kontext. Anforderungen an komplexe, mechatronische Systeme sehen deshalb Dynamik vor. So müssen z.B. zur Laufzeit Softwarekomponenten instanziiert und deinstanziiert werden.

An dieser Stelle wird das Modell der Graphtransformationssysteme vorgestellt, welches über keine vergleichbare zeitliche Komponente verfügt. Allerdings ist es im Gegensatz zu den vorgestellten Automatenmodellen möglich, dynamische Bestandteile abzubilden und damit eine Zustandsstruktur dynamisch zu erzeugen. Mit Hilfe von Graphtransformationssystemen werden strukturelle Veränderungen auf Graphen herbeigeführt.

Definition 5
Ein gerichteter Graph ist ein Tupel $G := (V, E, E_s, E_t)$, bestehend aus einer Menge von Knoten V sowie einer Menge von Kanten E. Eine Kante $e \in E$ verbindet einzelne Knoten

$v_s, v_t \in V$ miteinander. Die Funktion E_s ordnet jeder Kante $e \in E$ einen Startknoten v_s zu. Entsprechend ordnet die Funktion E_t jeder Kante e einen Zielknoten v_t zu.

Zusätzlich zu den bis hier beschriebenen Eigenschaften gibt es die Möglichkeit, für die einzelnen Knoten und Kanten eines Graphen eindeutige IDs zu vergeben, um eine eindeutige Zuordnung aller Elemente vornehmen zu können.

Definition 6
Ein gerichteter Graph mit Knoten und Kanten IDs besteht aus $G := (V, E, E_s, E_t, V_i, E_i)$, wobei V, E, E_s, E_t wie beim gerichteten Graphen in Definition 5 definiert sind. Zusätzlich existiert eine injektive Funktion V_i, welche jedem Knoten aus V ein eindeutiges n_i zuordnet, sowie E_i eine injektive Funktion, die jeder Kante aus E eine eindeutiges e_i zuordnet. Dabei sind $n_i, e_i \in \mathbb{N}^+$.

Eine Eigenschaft, welche für den Umgang mit den hier verwendeten Graphen notwendig ist, ist die Teilgraphbeziehung.

Definition 7
$G = (V, E, E_s, E_t)$ ist ein Teilgraph von $G' := (V', E', E'_s, E'_t)$ $(G \leq G')$ wenn gilt, $V \subseteq V'$, $E \subseteq E'$, sowie das die Zuordnungsfunktionen E_s und E'_s für die Kanten $V \cap V'$ identisch sind. Ebenfalls muss gelten E_t und E'_t sind identisch für die Menge $V \cap V'$.

In Abbildung 2.12 sind zwei Graphen abgebildet, wobei der rechte Graph einen Teilgraph des linken Graphen darstellt. Die Knoten n_1, n_2 sowie die Kante e_2 des rechten Graphen sind ebenfalls im linken vorhanden.

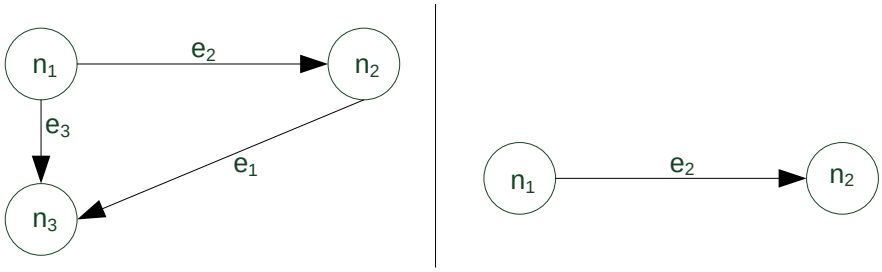


Abbildung 2.12: Die Abbildung zeigt zwei Graphen, wobei der rechte im linken Graphen enthalten ist.

Um über Graphtransformationssysteme zu reden, ist es notwendig, Relationen zwischen Graphen zu definieren. Eine Relation wäre z.B. die gerade erwähnte Teilgraph-Relation. Um Abbildungen zwischen einzelnen Graphen vornehmen zu können, muss hierfür ein Graphmorphismus für Graphtransformationssysteme definiert werden:

Definition 8
Für einen Graphmorphismus $m := (m_v, m_e)$ *mit* $m_v \in V \rightarrow V'$ *und* $m_e \in E \rightarrow E'$*, welcher einen Graphen* $G := (V, E, E_s, E_t)$ *auf* $G' := (V', E', E'_s, E'_t)$ *abbildet gilt:*

- $\forall v : v \in V \rightarrow m_v(v) \in V'$
- $\forall e : e \in E \rightarrow m_e(e) \in E'$
- $\forall e : e \in E \rightarrow m_v(E_s(e)) = E'_s(m_e(e)) \wedge m_v(E_t(e)) = E'_t(m_e(e))$

Die einzelnen, innerhalb der Graphtransformationssysteme vorhandenen Graphen haben eine statische Struktur. Damit ein dynamisches Strukturverhalten ermöglicht wird, finden Übergänge zwischen einzelnen Graphen statt, wobei die einzelnen Graphen Zustände des Graphtransformationssystems repräsentieren. Um diese Übergänge zu ermöglichen, werden Regeln verwendet, die durch ihre Anwendung Strukturen innerhalb von Graphen suchen und Veränderungen herbeiführen. Diese Regeln werden nachfolgend Graphtransformationsregeln genannt.

Definition 9
Eine Graphtransformationsregel $P := (P_l, P_r, h)$ *besteht aus den Graphen* P_l, P_r *und dem partiellen Graphhomomorphismus* $h \in p_l \rightarrow p_r$*. Dabei gilt:* p_l *ist ein Teilgraph von* P_l*, so wie* p_r *ein Teilgraph von* P_r *ist. Die Menge* $d(P) := \{V_{P_l} \setminus V_{P_r}\} \cup \{E_{P_l} \setminus E_{P_r}\}$ *beschreibt die Elemente, welche durch* P *gelöscht werden und* $n(P) := \{V_{P_r} \setminus V_{P_l}\} \cup \{E_{P_r} \setminus E_{P_l}\}$ *die Elemente, welche durch die Anwendung von* P *hinzugefügt werden. Dabei sind* E_{P_l}, V_{P_l} *die Knoten und Kanten aus* P_l *und* E_{P_r}, V_{P_r} *die Knoten und Kanten aus* P_r*.*

Ein Graph G kann mit Hilfe einer Graphtransformationsregel P in einen Nachfolgegraphen G' überführt werden. Den Graphen G, auf welchem die Graphtransformationsregel angewendet wird, bezeichnet man auch als Wirts- oder Muttergraph. Entsprechend wird der resultierende Graph G' als Tochtergraph bezeichnet.

Damit die Anwendung einer Graphproduktion $P(P_l, P_r, h)$ auf einen Wirtsgraphen G möglich ist, aus der ein Tochtergraph G' resultiert, muss ein Teilgraph g aus G, sowie ein Graphmorphismus m existieren, welcher P_l auf g abbildet. Der resultierende Tochtergraph G' ergibt sich aus den Kanten $E_{G'}$ und Knoten $V_{G'}$:

$$V_{G'} = \{V_G \setminus m_v(v) : v \in d(P) \cap V_l\} \cup \{v : v \in n(P) \cap V_r\}$$
$$E_{G'} = \{E_G \setminus m_e(e) : e \in d(P) \cap E_l\} \cup \{e : e \in n(P) \cap E_r\}$$

V_l und E_l sind die Knoten und Kanten der linken Seite P_l und V_r, E_r die Knoten und Kanten aus P_r, der rechten Seite von P. Ein Beispiel für eine derartige Regelanwendung zeigt die Abbildung 2.13, bei der aus dem linken Graphen der Knoten n_3, sowie die Kanten e_1, e_3, e_4 entfernt werden. Bei der Anwendung wird die Kante e_5 hinzugefügt.

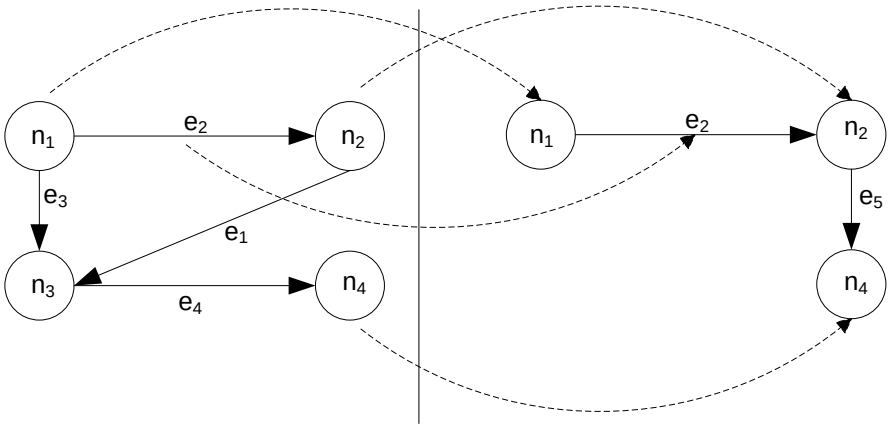


Abbildung 2.13: Schematische Darstellung einer Regelanwendung.

Dabei ist es möglich, die einzelnen Graphtransformationsregeln mit Prioritäten zu versehen. Eine solche priorisierte Graphtransformationsregel erlaubt es dann in dem Fall, dass auf einen Wirtsgraphen G zwei unterschiedliche Regeln P_1 und P_2 angewendet werden können. Die Reihenfolge der Anwendung wird durch die Prioritäten geregelt. Hierdurch wird Nichtdeterminismus vermieden.

Definition 10
Eine Graphtransformationsregel $P := (P_l, P_r, h, r)$ besteht zusätzlich zu der in Definition 9 definierten Graphtransformationsregel $P := (P_l, P_r, h)$ aus einer Priorität r, wobei $r \in \mathbb{N}^+$. Ein Regel p_i mit einem zugehörigen r_i , wird gegenüber einer Regel P_j mit zugehörigem r_j bevorzugt, falls $r_i > r_j$.

Am Beispiel bedeutet dies auch, dass falls zwei Regeln P_i, P_j auf einen Graphen G angewendet werden können, eine Regel mit höherer Priorität die andere verdrängt. Mit Hilfe der bis hier vorgestellten Definitionen lässt sich ein Graphtransformationssystem wie folgt formulieren:

Definition 11
Ein Graphtransformationssystem $S := (\mathcal{G}, \mathcal{G}^0, \mathcal{P})$ besteht aus einer potentiell unendlichen Menge an Zuständen in Form der Graphen $\mathcal{G}$, einer endlichen Anzahl an Startzuständen in Form der Graphen $\mathcal{G}^0$ sowie $\mathcal{P} := (P_1, ..., P_n)$ einer endlichen Anzahl an Graphtransformationsregeln P_i, mit $i \in \mathbb{N}$.

Bei der Anwendung von Graphtransformationsregeln kommt es vor, dass einzelne Kanten und Knoten aus dem Wirtsgraphen entfernt werden. Dabei kann es passieren, dass Kanten entstehen, bei welchen kein Zielknoten, bzw. kein Startknoten vorhanden ist (siehe

Abbildung 2.14). Eine solche Kante wird als Dangling-Edge bezeichnet. Zur Behandlung dieses Problems gibt es zwei unterschiedliche Vorgehensprinzipien, den so genannten *single-pushout approach* (SPO) und den *double-pushout approach* (DPO) [Roz97]. In der vorliegenden Arbeit wird der SPO verwendet. Wird bei einem Wirtsgraphen durch die Anwendung des linken Teils einer Graphtransformationsregel ein Knoten $e \in E$ entfernt, so werden beim SPO alle mit dem Knoten inzidenten Kanten $v \in V$ gelöscht. Somit ist die Entstehung von Dangling-Edges zwar ausgeschlossen, allerdings kann es passieren, dass Elemente aus einem Graphen entfernt werden, obwohl dies nicht beabsichtigt ist.

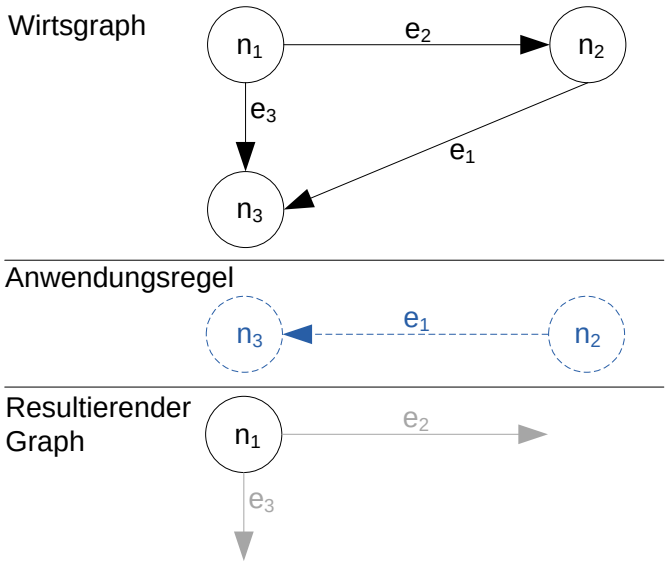


Abbildung 2.14: Die Abbildung zeigt oben einen Wirtsgraphen, auf den eine Regel (blau gestrichelt) angewendet wird und durch das Entfernen von Elementen ein resultierender Graph mit zwei Dangling-Edges entsteht (grau markierte Kanten im unteren Graph).

Um zu vermeiden, dass zu viele Knoten und Kanten unbeabsichtigt durch eine Graphtransformationsregel gelöscht werden, besteht die Möglichkeit, eine sogenannte Negative-Application-Condition (NAC) zu verwenden (siehe [HHT96]). Mit Hilfe einer NAC kann formuliert werden, welche Bedingung im Wirtsgraphen nicht vorhanden sein darf, damit eine Graphtransformation angewendet werden kann. Eine Graphtransformationsregel kann entsprechend mit einer zusätzlichen NAC versehen werden, wobei es sich bei dieser ebenfalls um einen Graphen handelt. Kann die NAC im Wirtsgraphen an der gleichen Stelle aufgefunden werden wie die linke Seite P_l, so wird die zugehörige Graphtransformation an dieser Stelle nicht angewendet. Eine Graphtransformationsregel mit NAC wird wie folgt definiert:

Definition 12
Eine Graphtransformationsregel $P := (P_l, P_r, P_{NAC}, h)$ mit negativer Anwendungsbedingung besteht aus einem linken Anwendungsteil P_l, welcher die Vorbedingung darstellt, sowie einem rechten Anwendungsteil P_r, welcher die Nachbedingung abbildet. Zusätzlich verfügt P über P_{NAC}, eine negative Anwendungsbedingung, welche nicht im Wirtsgraphen an der gleichen Stelle auffindbar sein darf, damit die Graphtransformationsregel P an dieser Stelle angewendet werden kann. P_{NAC} kann entweder P_l vollständig enthalten und um zusätzliche Knoten und Kanten erweitern oder P_{NAC} entspricht dem leeren Graphen.

Die Anwendung einer Graphtransformationsregel mit Negative-Application-Condition ist wie folgt definiert:

Definition 13
Eine Graphtransformationsregel $P := (P_l, P_r, P_{NAC}, h)$ mit Negative-Application-Condition kann auf einen Wirtsgraphen $G := (V, E, E_s, E_t)$ angewendet werden, wenn die Bedingungen erfüllt sind, dass:

$$\exists g : g \leq G$$

$$\exists h : h : P_l \rightarrow g$$

$$\nexists h_{NAC} : h_{NAC} : P_{NAC} \rightarrow g' \wedge g \leq g'$$

2.3.5 Verifikation

Bei der Analyse und Verifikation von Modellen, so auch bei den Automaten und Graphtransformationssystemen, können unterschiedliche Eigenschaften untersucht werden. Dabei wird zwischen der Berechnung der erreichbaren Zustände und dem erweiterten Verfahren des Model Checking unterschieden. Nachfolgend wird hier kurz auf die unterschiedlichen Verfahren eingegangen.

2.3.5.1 Erreichbarkeitsanalyse

Um die erreichbaren Zustände, bzw. den Zustandsraum eines Modells zu erzeugen, ist es notwendig, basierend auf den initial gegebenen Zuständen eine Erreichbarkeitsanalyse durchzuführen. So auch bei den Modellen des Timed Automaton und der Graphtransformationssysteme. Hierbei ist von Interesse, welche Zustände innerhalb des Modells erreichbar sind, oder sicher zu stellen, dass bestimmte Zustände ausgeschlossen werden können. Nachfolgend wird für die beiden Modelle des Timed Automaton und der Graphtransformationssysteme jeweils ein Verfahren vorgestellt, mit dem eine solche Erreichbarkeitsanalyse durchgeführt werden kann.

2.3.5.2 Model Checking

Ein Verifikationsverfahren ist das so genannte Model Checking [CGP00], bei dem auch komplexere Aussagen in Form einer speziellen Logik gegen ein Modell geprüft werden. Bei dem zu überprüfenden Modell kann es sich beispielsweise um einen Automaten oder ähnliches handeln. Es existieren dabei verschiedene Logiken um Aussagen zu formulieren, sowie unterschiedliche Verfahren um diese gegen das entsprechende Modell zu prüfen. So gibt es Logiken und Verfahren für diskrete sowie kontinuierliche Modelle und Eigenschaften. Model Checking kann als folgendes Entscheidungsproblem formuliert werden:

> Sei M ein zu testendes Modell und Ψ eine Spezifikation. Erfüllt das Modell die Spezifikation, gilt also $M \models \Psi$?

Die Eigenschaften, die sich mit Hilfe einer solchen Logik formulieren lassen, können wesentlich komplexer sein als die einfache Erreichbarkeit von Zuständen. So kann überprüft werden, ob die einzelnen Zustände in einer bestimmten Reihenfolge innerhalb des Modells erreicht werden können. Im Vergleich zu der Erreichbarkeitsanalyse, kann dort nicht nur überprüft werden, welche Zustände erreicht werden. Beim Model Checking gibt es die Möglichkeit, Aussagen zu formulieren und zu überprüfen, in welcher Reihenfolge diese Zustände erreicht werden müssen. Dabei ist Voraussetzung für eine derartige Überprüfung, dass diese entsprechenden Zustände bereits ermittelt wurden.

Jedoch hat auch das Verfahren des Model Checking seine Grenzen. Abbildung 2.15 zeigt noch einmal die generelle Vorgehensweise beim Model Checking. Werden jedoch die Eingabemodelle, wie in Abbildung 2.16 beschrieben, immer komplexer, skaliert das Verfahren des Model Checkings aufgrund des Problems der Zustandsraumexplosion nicht mehr. In 2.15 ist verdeutlicht, dass Model Checking für hybride Modelle nicht in akzeptabler Zeit durchgeführt werden kann.

Um ein Gefühl für die Komplexität solcher Zustandsräume, die bei der Verifikation erstellt werden, zu bekommen, wird im folgenden die Konstruktion für das Eingabemodell der Timed Automata (siehe Abschnitt 2.3.5.3) und anschließend für Graphtransformationssystem (siehe Abschnitt 2.3.5.4) gezeigt.

2.3.5.3 Zustandsraum des Timed Automaton

Bei einem Timed Automaton $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, T)$ hängen die Zustände nicht nur von der jeweiligen Location ab, der aktuelle Zustand ergibt sich zusätzlich aus der Belegung der einzelnen Clocks. Einer Location $s \in \mathcal{S}$ wird durch eine Zuordnungsfunktion ein $\mathcal{C}(X)$ zugeordnet. Dabei ist $\mathcal{C}(X)$ eine Teilmenge der Bedingungen über die Clockvariablen aus X. Dies bedeutet, dass sich der Zustand eines Timed Automaton aus der

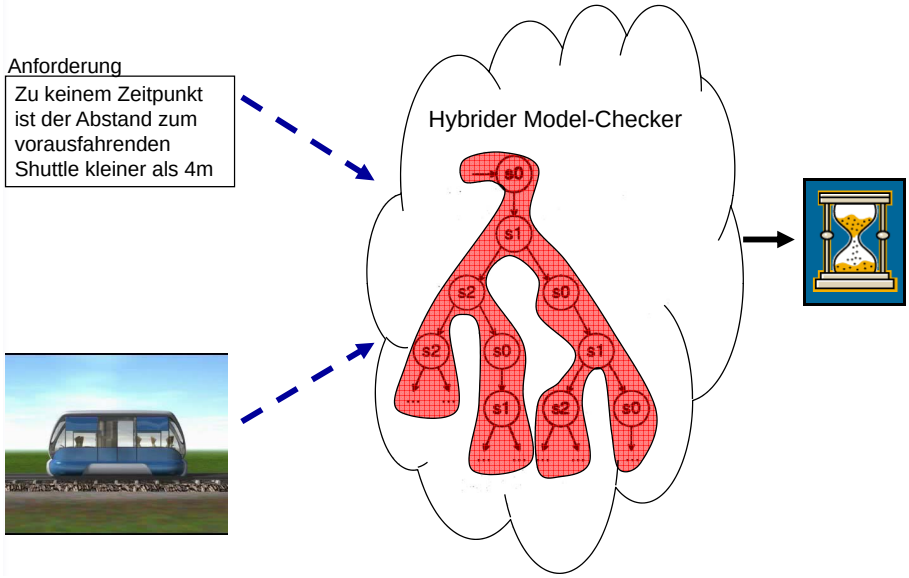


Abbildung 2.15: Problem: Hybrides Model Checking

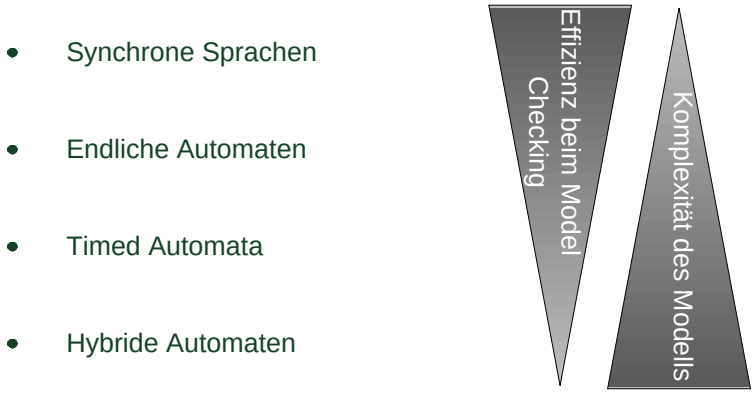


Abbildung 2.16: Mächtigkeit des Eingabemodells vs. Effizienz beim Model Checking

aktuellen Location, sowie den dort aktuell geltenden zeitlichen Bedingungen über die einzelnen Clocks zusammen setzt.

Im Beispiel des Automaten aus Abbildung 2.10 setzt sich der initiale Zustand aus der Location s_0, sowie der dort vorhandenen Invariante zusammen. Die Invariante, welche die Clock-Variable x_2 beinhaltet, definiert einen zeitlichen Erreichbarkeitsraum, in dem gilt, dass der Wert von $x_2 \leq 2$ ist. Über die Clock x_1 ist im initialen Zustand in der Location s_0 keine einschränkende Bedingung vorhanden, allerdings haben die im Automaten existierenden Clocks die Eigenschaft, dass diese in der Zeit synchron voranschreiten. Hierdurch ergibt sich eine Abhängigkeit zwischen x_1 und x_2 in der Art und Weise, dass beide den

gleichen Wert besitzen müssen. Diese Abhängigkeit zwischen x_1 und x_2 gilt solange, bis durch einen Reset eine der beiden Clocks auf den Wert 0 zurück gesetzt wird. Auch dann laufen beide weiterhin synchron, allerdings können die einzelnen Werte sich unterscheiden. Solange in der initialen Location s_0 verweilt wird, sind die entsprechenden Werte identisch und somit x_1 und x_2 jeweils kleiner oder gleich dem Wert 2.

Die Wertemengen, welche durch die möglichen Clock-Belegungen aufgebaut werden, sind in der Regel unendlich groß. Bei der Analyse des Erreichbarkeitsraumes ist es nicht möglich, jeden Zustand einzeln abzubilden, da nicht jeder Zustand und die daraus resultierenden Folgezustände in endlicher Zeit betrachtet werden können. Somit ist es notwendig, eine endlich große Repräsentation dieser Erreichbarkeitsräume vorzunehmen. Um eine endliche Repräsentation dieser Bereiche zu ermöglichen, existieren verschiedene Datenstrukturen. Eine davon ist das Modell der sogenannten Clock-Regions, welches bei [CGP00] sowie [BBF$^+$01] genauer beschrieben wird.

Die entsprechende Clock-Region für den initialen Zustand des Timed Automaton aus dem Beispiel im Abschnitt 2.3.2, lässt sich grafisch wie in Abbildung 2.17 darstellen. Dabei stellen die grau markierten Bereiche die Wertemenge dar, welche die Clock-Variablen x_1 und x_2 annehmen können. In dieser Menge sind alle Werte enthalten, die auf der Winkelhalbierenden des ersten Quadranten des Koordinatensystems liegen und kleiner oder gleich dem Wert 2 sind.

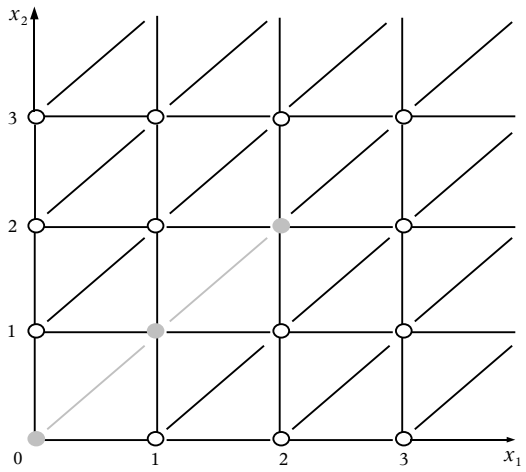


Abbildung 2.17: Der zeitliche Erreichbarkeitsraum des initialen Zustandes des Automaten aus Abbildung 2.10. Die grau markierten Bereiche entsprechen der Menge der Werte, welche die Clocks x_1 und x_2 annehmen können.

Zusammen mit der Location s_0 bildet diese Clock-Region den initialen Zustand des Beispiel-Automaten aus Abbildung 2.10. In der vorliegenden Arbeit wird allerdings mit

einer anderen Form zur Darstellung der zeitlichen Erreichbarkeitsräume gearbeitet. Dies sind die sogenannten Clockzones, die in dieser Arbeit Verwendung finden.

Clockzone Hier wird kurz auf die Datenstruktur und Operationen der Clockzones eingegangen. Eine genauere Definition für die hier vorgestellte Datenstruktur ist bei Clarke und anderen [CGP00] zu finden.

Eine Clockzone definiert eine Reihe an Bedingungen über einzelne Clock-Variablen, wobei hierzu Ungleichungen ähnlich wie beim Modell des Timed Automaton (siehe Definition 3) verwendet werden:

$$i, j \in \mathbb{N}^+, d \in \mathbb{Z}, x_i \in \mathbb{R}^+, \prec \in \{<, \leq\} : x_j \prec d, d \prec x_j, x_i - x_j \prec d.$$

Zusätzlich verfügt jede Clockzone über eine Referenz-Clock x_0, welche zu jedem Zeitpunkt den Wert 0 besitzt. Die gesamte Clockzone ergibt sich durch die Konjunktion der einzelnen Ungleichungen über die Clock-Variablen. Am Beispiel aus der Abbildung 2.10 wird die initiale Clockzone durch die folgenden Ungleichungen aufgebaut:

$$x_2 \leq 2, x_0 - x_2 \leq 0 \wedge x_1 - x_2 \leq 0 \wedge x_2 - x_1 \leq 0 \wedge x_0 - x_1 \leq 0$$

Dabei resultiert die erste Ungleichung aus der Invariante der initialen Location s_0. Die restlichen Bedingungen entstehen aus der Eigenschaft, dass alle vorhandenen Clocks $x_i \in X$ des Beispiel-Automaten synchron in der Zeit voranschreiten, sowie dass diese immer ≥ 0 seien müssen.

Definition 14
Eine Clockzone Z hat eine Menge X an Clock-Variablen x_i. Jedes x_i kann Werte aus $\mathbb{R}^+ \cup 0$ annehmen, wobei $i \in \mathbb{N}^+$ und $i > 0$. Zusätzlich existiert eine Referenz-Clock x_0, die immer den Wert 0 besitzt, sowie eine Anzahl von Bedingungen $c \in C$ in Form von Ungleichungen der Art $x_j \prec d, d \prec x_j, x_i - x_j \prec d$, mit $i, j \in \mathbb{N}^+$, $d \in \mathbb{Z}$ und $\prec \in \{<, \leq\}$. Die Clockzone ergibt sich aus der Konjunktion über die Bedingungen aus C.

Eine Clockzone mit k verschiedenen Clocks wird als k-dimensional bezeichnet, da jede Clock im euklidischen Raum eine eigene Dimension aufspannt (ohne die Referenz-Clock x_0). Der hierbei aufgespannte Raum erfüllt zusätzlich die Eigenschaft, dass dieser konvex ist. Die Abbildung 2.18 zeigt in grafischer Form die Clockzone ϕ mit den zwei Clocks x_1, x_2 und den folgenden Bedingungen:

$$x_1 \leq 4 \wedge x_2 \leq 5 \wedge 2 \leq x_1 \wedge 1 \leq x_2 \wedge x_1 \leq x_2$$

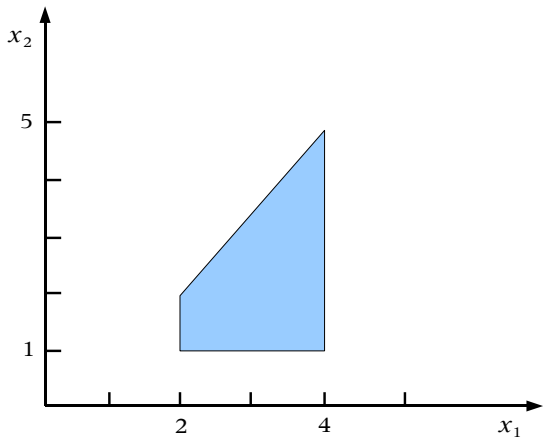


Abbildung 2.18: Die Clockzone ϕ

Es existieren eine Anzahl an Operatoren, welche auf eine, bzw. mehrere Clockzones angewendet werden können. Dabei werden für die zeitliche Erreichbarkeitsanalyse, wie diese etwa beim Timed Automaton vorgenommen wird, drei spezielle Operatoren benötigt. Hierzu zählt die Vereinigung von zwei Clockzones, das Verstreichen lassen von Zeit und das Zurücksetzen von einzelnen Clocks:

- Vereinigung von zwei Clockzones: $\phi_1 \wedge \phi_2$
- Verstreichen lassen von Zeit (Up-Operation): $\phi \Uparrow$
- Zurücksetzen von Uhren (Clock-Reset): $\phi[\gamma]$ mit $\gamma \subseteq X$, wobei X die Menge der Clock-Variablen in ϕ ist.

Bei der Vereinigung von zwei Clockzones ist es auch möglich, dass ϕ_1 oder ϕ_2 nur aus einer einzigen Bedingung besteht, wie diese etwa durch einen Guard oder eine Invariante gegeben ist.

Die entsprechenden drei Operatoren werden benötigt, um die Erreichbarkeitsanalyse beim Timed Automaton durchzuführen. So etwa beim Zustandswechsel innerhalb des Automaten. Dabei setzt sich ein Zustand aus der jeweiligen Location sowie der dort momentan vorhandenen Belegung der einzelnen Clockvariablen zusammen. Diese Belegung der Clockvariablen kann durch eine entsprechende Clockzone dargestellt werden. So existiert etwa für die Location s eine Clockzone ϕ und der Zustand des Automaten ergibt sich aus dem Tupel $\langle s, \phi \rangle$. Um den Nachfolgezustand $succ(s, \phi, t)$ beim Wechsel von der Location s über eine Transition t zu einer Location s' zu berechnen, sind die vorgestellten Operatoren auf die Clockzone ϕ anzuwenden. Dabei wird hier zur Berechnung der Folge-Clockzone ϕ' die Funktion $succ_\phi$ eingeführt, welcher ϕ selbst, die Invarianten I der Location s, sowie die Guards φ der Transition t übergeben werden.

Algorithmus 2.1 procedure $\phi_{succ} = succ_{\phi}(\phi, I, \varphi)$

procedure $\phi_{succ} = succ_{\phi}(\phi, I, \varphi)$

1: Bilden der Schnittmenge: $\phi = \phi \wedge I$
2: Verstreichen lassen von Zeit: $\phi \Uparrow$
3: Wiederum Bilden der Schnittmenge: $\phi = \phi \wedge I$
4: Vereinigen mit φ: $\phi = \phi \wedge \varphi$

Falls die resultierende Clockzone ϕ_{succ} nicht leer[1] ist, kann über die Transition t zur Location s' gewechselt werden. Eine Clockzone ϕ ist leer, wenn für alle $c \in \phi$ gilt: Es gibt keine mögliche Belegung für die Clocks x_i aus c, so dass alle Ungleichungen c erfüllt sind.

Anschließend müssen noch die Clockresets λ der Transition t ausgeführt werden, um aus ϕ_{succ}, ϕ' zu erhalten. Dies wird durch den Algorithmus 2.2 $succ'_{\phi}$ beschrieben.

Algorithmus 2.2 procedure $\phi' = succ'_{\phi}(\phi_{succ}, \lambda)$

procedure $\phi' = succ'_{\phi}(\phi_{succ}, \lambda)$

1: Zurücksetzen der Clocks λ: $\phi' = \phi_{succ}[\lambda := 0]$

Bevor ϕ' zu s' als Folge-Clockzone hinzugefügt wird, muss die Schnittmenge ϕ' mit den Invarianten $\phi' \wedge I(s')$ von s' gebildet werden. Diese Schnittmenge ergibt zusammen mit s' den Folgezustand, welcher über die Transition t erreicht wird. Dabei kann es vorkommen, dass durch die Berücksichtigung der Invarianten von s', beim Bilden der Schnittmenge $\phi' \wedge I(s')$ eine Clockzone entsteht, die leer ist. Der hierdurch abgebildete Zustand wird auch als Timedeadlock bezeichnet.

Ein Timedeadlock bezeichnet einen erreichten Zustand, bei dem aufgrund der zeitlichen Bedingungen aus diesem kein Folgezustand erreicht werden kann. Für eine Clockzone ϕ mit einem Timedeadlock gilt, dass ϕ einerseits erreicht wurde und es andererseits mindestens eine Ungleichung der Form $x_1 - x_2 \sim d$ gibt, welche zusammen mit den restlichen Ungleichungen aus ϕ nicht erfüllt werden kann.

Die Funktionen $succ_{\phi}$, die Überprüfung ob die resultierende Clockzone leer ist, sowie die Funktion $succ'_{\phi}$ werden zu der Funktion $succ(\phi, I(s), \varphi, \lambda, I(s'))$ zusammengefasst. Die Berechnung ist in Algorithmus 2.3 beschrieben.

Mit Hilfe von $succ(\phi, I(s), \varphi, \lambda, I(s'))$ wird nun die Funktion $reach(A)$ zur Berechnung der erreichbaren Zustände formuliert. Die Berechnung ist in Algorithmus 2.4 beschrieben.

[1] Siehe [CGP00] zur Definition einer leeren Clockzone und wie dies festgestellt werden kann.

Algorithmus 2.3 procedure $\phi' = succ(\phi, I(s), \varphi, \lambda, I(s'))$

procedure $\phi' = succ(\phi, I(s), \varphi, \lambda, I(s'))$

1: $\phi_{succ} = succ_{\phi}(\phi, I(s), \varphi))$
2: **if** ϕ_{succ} ist nicht leer **then**
3: $\quad \phi' = succ'_{\phi}(\phi_{succ}, \lambda)$
4: $\quad \phi' = \phi' \wedge I(s')$
5: **end if**

Algorithmus 2.4 procedure $reach(A)$

procedure $reach(A)$

1: $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, T)$
2: $Open = \emptyset, Close = \emptyset$
3: **for all** $s \in S_0$ **do**
4: $\quad z := \langle s, I(s) \rangle$
5: $\quad Open = Open \cup z$
6: **end for**
7: **while** $Open \neq \emptyset$ **do**
8: $\quad$ **for all** $t \in T : t := s \times C(X) \times 2^X \times s'$ **do**
9: $\quad\quad \phi_t = succ(s, \phi, t)$
10: $\quad\quad$ **if** $\phi_t \neq empty$ **then**
11: $\quad\quad\quad \phi' = \phi_t \wedge I(s')$
12: $\quad\quad\quad z' = \langle s', \phi' \rangle$
13: $\quad\quad\quad Open = Open \cup z'$
14: $\quad\quad$ **end if**
15: $\quad$ **end for**
16: $\quad Open = Open \setminus z$
17: $\quad Close = Close \cup z$
18: **end while**

Das Ergebnis der Berechnung ist die Menge $Close$, welche sich aus einzelnen Tupeln $\langle s, \phi \rangle$ zusammen setzt. Diese Tupel beschreiben die Menge der erreichbaren Zustände.

Bei jedem berechneten Folgezustand $\langle s', \phi' \rangle$ stammt s' immer aus der Menge S des Timed Automaton A. Somit sind alle Locations und hierdurch auch die Struktur des durch A abgebildeten Modells bereits fest vorgegeben. Strukturdynamische Eigenschaften werden hier somit nicht abgebildet, im Gegensatz zu den Graphtransformationssystemen. Die Datenstruktur der Difference-Bound-Matrices erlaubt es, Clockzones effizient zu kodieren und zu manipulieren. Ein Vorteil der Datenstruktur Difference-Bound-Matrice liegt darin, dass eine kanonische Form herleitbar ist, welche es möglich macht mehrere Difference-Bound-Matrices effizient miteinander zu vergleichen. Für weitergehende Informationen zu der Datenstruktur der Difference-Bound-Matrice siehe [CGP00].

2.3.5.4 Zustandsraum des Graphtransformationssystems

Im Gegensatz zu dem Modell des Timed Automaton lassen sich mit Hilfe von Graphtransformationssystemen dynamische Strukturen modellieren. Diese Strukturen werden in Form von Graphen, während der Erreichbarkeitsanalyse, durch Anwendung der einzelnen Graphtransformationsregeln aufgebaut. Eine zeitliche Komponente, wie sie beim Timed Automaton vorhanden ist, gibt es hingegen nicht.

Der Ausgangspunkt ist dabei ein Graphtransformationssystem $S := (\mathcal{G}, \mathcal{G}_0, \mathcal{P})$, mit einer Menge an Graphen $\mathcal{G}$, einer Menge an initialen Graphen $\mathcal{G}^0$ und einer Menge an Graphtransformationsregeln $\mathcal{P}$. Damit der erreichbare Zustandsraum ermittelt werden kann, wird hier davon ausgegangen, dass alle Mengen endlich sind[2].

Um, ausgehend von einem gegebenen Wirtsgraphen G, über eine Graphtransformationsregel P die Menge der daraus resultierenden Tochtergraphen $\mathcal{G}'$ zu berechnen, wird die Funktion $prod(G, P)$ eingeführt. Diese liefert die Menge der Graphmorphismen $m \in M$ zurück, welche P_l aus der Graphtransformationsregel P auf einem Teilgraphen g des Wirtsgraphen G abbilden. Für diese Morphismen $m := (m_v, m_e)$ gelten die Eigenschaften, die in Abschnitt 2.3.4 bei der Anwendung einer Graphtransformationsregel beschrieben wurden.

Der Tochtergraph G' ergibt sich aus der Anwendung von P zusammen mit dem jeweiligen Morphismus m:

$$G \xrightarrow{m,P} G'$$

Um die Notation abzukürzen wird die Funktion $prod(G, P)$ verwendet, um sowohl die Morphismen m als auch die Folgegraphen G' herzuleiten. Für weitergehende Details, z. B. wie genau eine Graphtransformationsregel P auf einen Wirtsgraphen nach dem Prinzip des SPO angewendet wird, siehe [Roz97].

Mit Hilfe der Funktion $prod$ wird ein Algorithmus formuliert, mit dem das gesamte Graphtransformationssystem aufgebaut werden kann. Der hier angegebene Algorithmus 2.5 entspricht einer Breitensuche über ein Graphtransformationssystem S.

Um auch die Priorität einer einzelnen Graphtransformationsregel zu berücksichtigen, muss die Zeile 5 des Algorithmus angepasst werden. Dort werden alle Graphproduktionen in einzelnen Mengen Q_r zusammengefasst, die über die gleiche Priorität r verfügen. Aus diesen Mengen werden ihrer Priorität entsprechend absteigend, die einzelnen Regeln wie in Zeile 6 auf den Wirtsgraphen G angewendet. Die einzelnen Mengen Q_r werden solange verarbeitet, bis mindestens eine Regel aus Q_r auf G angewendet werden kann, also die Funktion $prod$ nicht die leere Menge zurück geliefert hat. Falls dieser Fall eintrifft,

[2]Falls eine der Mengen $\mathcal{G}$, $\mathcal{G}_0$ oder $\mathcal{P}$ nicht endlich ist, würde das hier vorgestellte Verfahren nicht terminieren.

Algorithmus 2.5 procedure $Open = reachGTS(S)$

procedure $Open = reachGTS(S)$

```
1:  S := (G, G_0, P)
2:  Open = G_0, Close = ∅
3:  while Open ≠ ∅ do
4:      for all G ∈ Open do
5:          for all P ∈ P do
6:              Open = Open ∪ prod(G, P)
7:          end for
8:      end for
9:      Close = Close ∪ G
10:     Open = Open \ G
11: end while
12: return Close
```

wird die innere Schleife in Zeile 5 verlassen und mit der äußeren in Zeile 3 fortgefahren. Das hieraus resultierende Graphtransformationssystem verfügt in Form der durch die Transitionen erzeugten Tochtergraphen über die hinzugekommenen dynamischen Strukturbestandteile allerdings über keine zeitliche Komponente, wie dies beim Modell des Timed Automaton der Fall ist.

Nachdem nun die Modelle zur Beschreibung der Struktur und des Verhaltens von regelungstechnischen Systemen (Abschnitt 2.2) sowie von Softwaresystemen beschrieben wurden, wird im nächsten Abschnitt nun auf die Integration der Domäne des Software Engineering mit der Domäne der Regelungstechnik eingegangen. Der nun im Folgenden beschriebene MECHATRONIC UML Ansatz vereint beide Domänen.

2.4 Mechatronic UML

Der MECHATRONIC UML Ansatz [GHH+08b] ermöglicht die modell-basierte Entwicklung von mechatronischen Systemen (siehe Abschnitt 2.1). MECHATRONIC UML unterstützt dabei die Spezifikation von Softwarestrukturen und Strukturänderungen [BBG+06], die Koordination von komplexem Echtzeitverhalten [BGS05], formale Verifikation von Sicherheitseigenschaften [BGH+05b][GTB+03][Hir04][HG03] und die Fehleranalyse [GT06]. Neben diesen unterstützt MECHATRONIC UML auch die Integration der Modellierung von Regelungstechnik, kontinuierlichen Verhalten durch die Einbettung von Reglerstrukturen in die Zustände, ohne dabei die Verifikationsergebnisse der Echtzeiteigenschaften zu verletzen. [BGH05a].

2.4.1 Echtzeitverhalten

Das Echtzeit-Koordinationsverhalten beschreibt die nachrichtenbasierte Kommunikation zwischen verschiedenen mechatronischen Komponenten unter harten Echtzeitbedingungen. Zuallererst wird bei dem Ansatz das Kommunikationsverhalten verschiedener Szenarien durch die Verwendung von Echtzeit-Sequenzdiagrammen spezifiziert. Die Menge von spezifizierten Szenarien, z.B. die Bildung des Konvois von Shuttles, kann automatisch zu Realtime Statecharts synthetisiert werden, wobei die beteiligten Rollen auf Software Komponenten abgebildet werden. Realtime Statecharts sind eine Erweiterung von UML State Machines um spezielle Echtzeiteigenschaften für die periodische Ausführung, Echtzeitverhalten, Wort Case Ausführungszeiten usw. zu modellieren. Die Semantik der Realtime Statecharts ist über die Semantik der Timed Automata (siehe Abschnitt 2.3.2 und [HG03][Hir04][BGHS04]) definiert.

Die Realtime Statecharts der verschiedenen Software Komponenten werden aus den so genannten *Echtzeit-Koordinationsmustern* abgeleitet. Realtime Statecharts spezifizieren das Verhalten einer bestimmen Rolle in einer Koordination. Am Beispiel des Shuttlekonvois lassen sich zwei Rollen aufzeigen, die Rolle eines Führungsshuttles und die Rolle eines hinterherfahrenden Shuttles. Da so ein Koordinationsverhalten in mechatronischen Systemen immer sicherheitskritischen Charakter hat, stellt der MECHATRONIC UML Ansatz zur Analyse der Echtzeit-Koordinationsmuster formale Verifikationstechniken zur Verfügung.

Wie schon angedeutet, leitet sich das Verhalten der Komponenten durch die Verwendung der verfeinerten Rollen ab. MECHATRONIC UML baut auf einer Verfeinerungsbeziehung auf, die garantiert, dass, falls eine Komponente eine Rolle verfeinert, die Verifikation der Rolle ebenfalls für die Komponente gilt. Als letztes werden Regler in die Zustände der Statecharts einer Komponente eingebettet. Hierdurch wird die Verbindung zu der Domäne der Regelungstechnik geschlagen. Im Folgenden werden die bereits skizzierten Schritte anhand der Modellierung des Konvoiszenarios detailliert beschrieben.

2.4.2 Echtzeit-Koordinationsmuster

In Abbildung 2.19 ist ein Echtzeit Sequenzdiagramm modelliert. Es zeigt die Bildung eines Shuttlekonvois an einer Weiche [GHHK06]. Hierbei wird die Nachrichteninteraktion zwischen Komponenten modelliert. Durch die Integration von Zuständen und Zeit ist es möglich, die Nachrichteninteraktion in eine globale zeitliche und lokale kausale Ordnung zu bringen.

Durch den Syntheseansatz aus [GB04][GHHK06] können aus diesen Szenarien Rollen eines Echtzeit-Koordinationsmusters generiert werden, die durch Realtime Statecharts

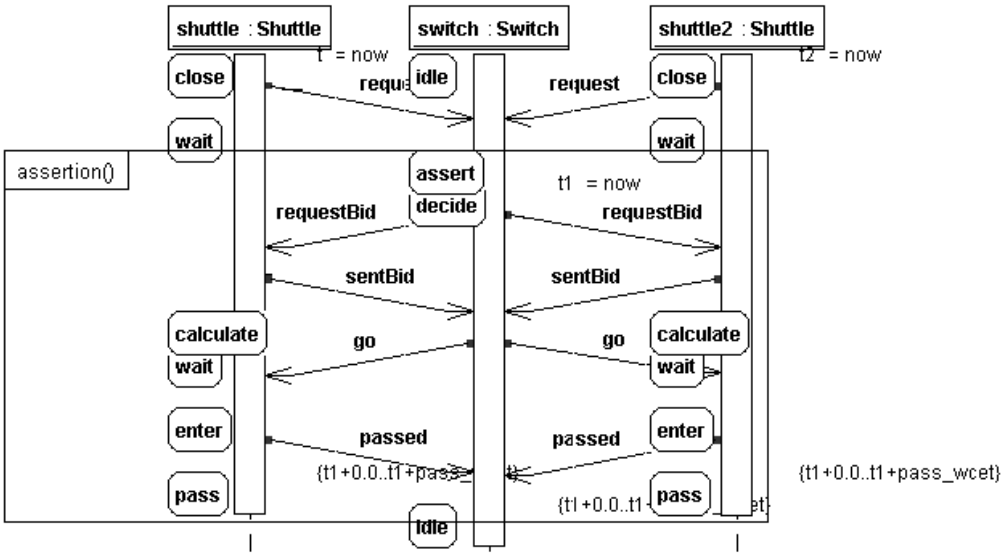


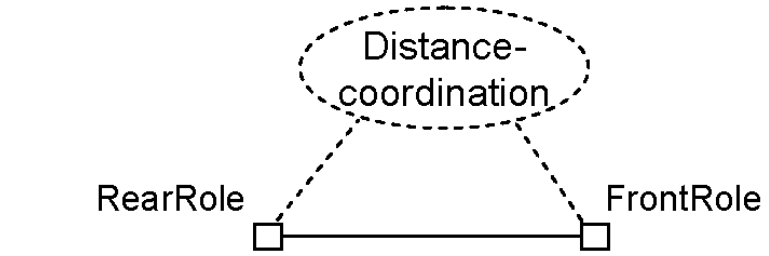
Abbildung 2.19: Echtzeit Sequenzdiagramm

spezifiziert sind. Natürlich können Rollen auch direkt spezifiziert werden. Eine Rolle spezifiziert das Verhalten des Führungsshuttles, die andere Rolle die des hinterherfahrenden Shuttles (siehe Abbildung 2.20). Zusätzlich wird das Verhalten des Connectors, der das Kommunikationsmedium darstellt, durch ein Realtime Statechart beschrieben.

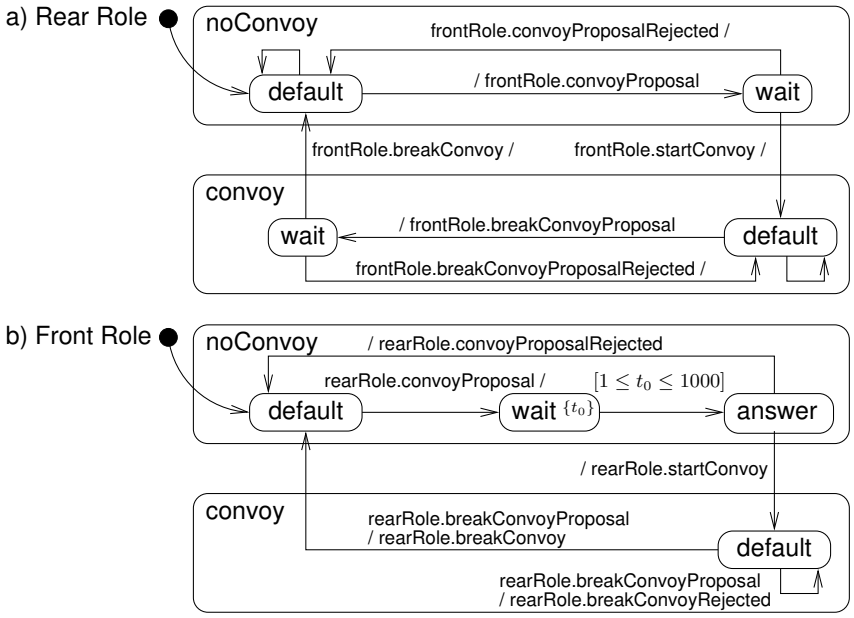
Beide Rollen befinden sich initial in dem Zustand noConvoy::default, der die Bedeutung, von zwei allein, nicht im Konvoibetrieb fahrenden Shuttles, darstellt. Die Rolle rear kann nun nicht-deterministisch wählen, ob ein Konvoibetrieb aufgenommen werden soll oder nicht. Für den positiven Fall, dass ein Konvoibetrieb aufgenommen werden soll, sendet die Rolle eine Nachricht an das andere Shuttle, bzw. Rolle front. Die Rolle front entscheidet nun ebenfalls nicht-deterministisch, den Vorschlag abzuweisen oder anzunehmen. Im negativen Fall schalten beide Realtime Statecharts zurück in den Zustand noConvoy::default. Im positiven Fall schalten beide in den Zustand convoy::default.

Für den Fall, dass das hinterherfahrende Shuttle nicht-deterministisch entscheidet, zu bremsen, sendet dies dieses Ereignis an das Führungsshuttle. Auch hier kann das Führungsshuttle den Vorschlag abweisen oder akzeptieren. Wenn der Vorschlag abgelehnt wird, bleiben beide Shuttles im Konvoizustand, andernfalls wechseln beide Shuttles in den Zustand noConvoy::default.

Der Connector zwischen beiden Rollen repräsentiert eine Funkverbindung. In diesem Beispiel wird der Connector nicht explizit durch ein Realtime Statechart beschrieben, sondern nur durch QoS Angaben spezifiziert. In [Hof07] ist ein Verfahren vorgestellt, hieraus automatisch Connectoren zu synthetisieren. In dem Beispiel wird angenommen, dass Nach-



(a) Echtzeit-Koordinatiosmuster



(b) Rollenverhalten

Abbildung 2.20: Echtzeit-Koordinationsmuster für die Konvoikoordination

richten um 1 bis 5 Zeiteinheiten verzögert werden können. Außerdem ist der Connector nicht zuverlässig, so dass Nachrichten verloren gehen können.

Um die Sicherheitseigenschaften zu überprüfen, müssen diese zuerst spezifiziert werden. Hierfür kann z.B. TCTL verwendet werden. In diesem Beispiel soll gelten: rear.convoy implies front.convoy. Befindet sich das hinterherfahrende Shuttle im Konvoimodus, muss sich auch das Führungsshuttle im Konvoimodus befinden. Andernfalls kann es in einer Notfallsituation zu einem Unfall kommen, da das Führungsshuttle für die Situation nicht geeignete Entscheidungen treffen könnte.

In [Gie03, BGHS04, BGH+05b] wurde gezeigt, dass die Eigenschaft erfüllt ist. Nach der erfolgreichen Verifikation des Echtzeit-Koordinationsmusters können diese nun zu Komponenten verfeinert werden.

2.4.3 Komponenten

Die im letzten Abschnitt vorgestellten Echtzeit-Koordinationsmuster werden bei der Verhaltenserstellung der Softwarekomponenten verwendet. Die von MECHATRONIC UML verwendeten Komponentendiagramme basieren auf denen der UML 2.0. Diskrete Komponenten werden von kontinuierlichen Komponenten unterschieden. Das Verhalten von diskreten Komponenten wird durch Realtime Statecharts modelliert, das Verhalten von kontinuierlichen Komponenten durch Blockdiagramme aus der Domäne der Regelungstechnik. Dieselbe Unterscheidung wird auch bei Ports gemacht. Die Definition der Komponenten unterstützt die Verwendung von Interface Statecharts [BGO06] für die Spezifizierung von verschiedenen Ports und Konfigurationen/Zuständen der Komponenten.

Diskrete Komponenten verfeinern einfach nur das Verhalten der Rollen. Der Entwickler muss lediglich ein Synchronisationsverhalten der beteiligten Rollen modellieren, um das Verhalten der Rollen zu koordinieren. Danach werden die Rollen als Ports zu den Komponenten hinzugefügt (siehe Abbildung 2.21).

Eine Komponente spezifiziert ebenfalls, ob noch weitere Komponenten eingebettet sind. In dem Beispiel sind dies noch die beiden kontinuierlichen Komponenten - eine um die Geschwindigkeit zu kontrollieren (*Velocity Controller*) und eine, um den Abstand zum vorausfahrenden Shuttle zu kontrollieren (*Distance Controller*). Da die kontinuierlichen Komponenten typischerweise aus der Regelungstechnik kommen, verdeutlicht dies sehr deutlich die domänübergreifende Integration [HH06].

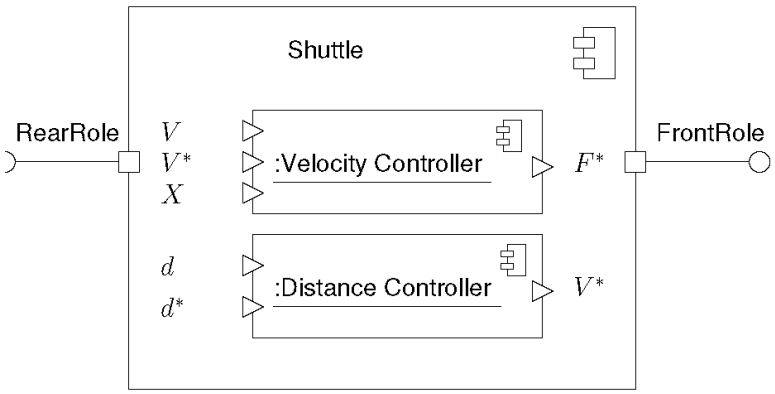


Abbildung 2.21: Shuttle Komponente

Während die Rollen der Komponenten hinzugefügt werden, können diese durch den Entwickler verfeinert werden, indem sie z.B. im Verhalten eingeschränkt werden. Im Beispiel kann der Entwickler z.B. festlegen, dass ein Shuttle nur als rear einem Konvoi beitritt.

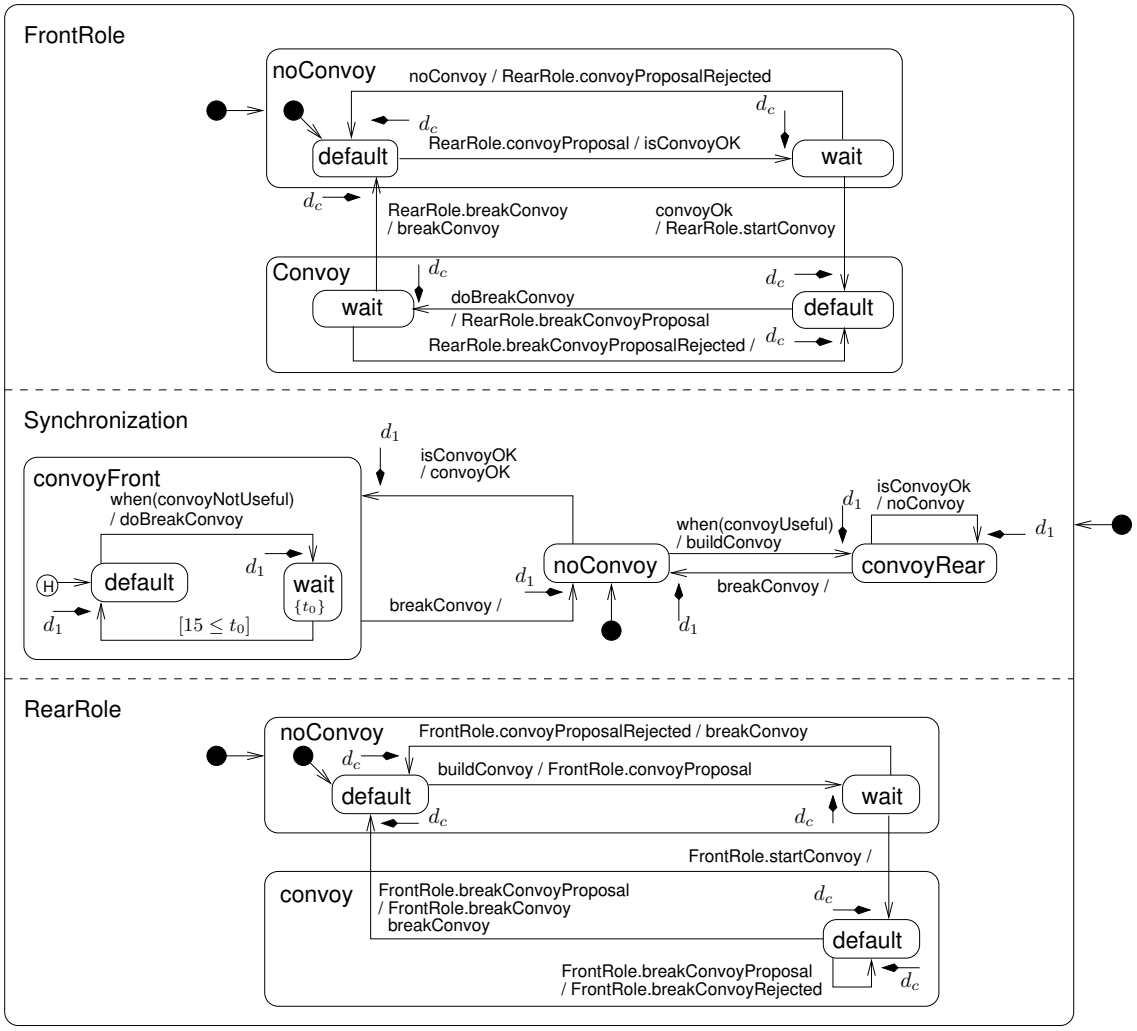


Abbildung 2.22: Realtime Statechart der Shuttle Komponente

In Abbildung 2.22 ist das Verhalten der Komponente Shuttle aus Abbildung 2.21 dargestellt. Das Realtime Statechart besteht aus drei orthogonalen Zuständen FrontRole, RearRole und Synchronization. Die Zustände FrontRole und RearRole sind Verfeinerungen des Rollenverhaltens aus Abbildung 2.20 und spezifizieren das Kommunikationsverhalten im Detail, um einen Konvoi zu erzeugen oder aufzulösen. Im Zustand Synchronization wird das Kommunikationsverhalten und die Initiierung eines Konvois modelliert. Die drei Unterzustände des Zustandes Synchronization stellen dar, ob sich das Shuttle gerade als Führungsfahrzeug (convoyFront), als letztes Fahrzeug (convoyRear) oder als allein fahrendes Shuttle (noConvoy) verhält.

Das ganze Statechart ist eine Verfeinerung beider Rollen und es wurde nur der Nicht-Determinismus der Rollen aus Abbildung 2.20 durch hinzufügen von Synchronisation aufgelöst.

Wie schon erwähnt, erfüllen Komponenten in der Domäne der mechatronischen Systeme eine Menge von Echtzeitbedingungen. In dem Beispiel muss gelten, dass RearRole die Nachricht startConvoy innerhalb einer bestimmten Zeit verschickt, nachdem sie convoyOK empfangen hat.

Um dieses adäquat durch Realtime Statecharts abbilden zu können, muss es möglich sein, an Transitionen Zeit zu spezifizieren - in der Realität passiert auch keine Aktion in Nullzeit. Hierfür werden die folgenden Deadline Konstrukte verwendet: In Abbildung 2.22 wird das Deadlineintervall d_c und d_1 verwendet, um eine minimale und maximale Schaltzeit einer Transition anzugeben. Zum Beispiel muss das Senden der Nachricht convoyProposalRejected innerhalb der Deadline d_c passieren, nachdem die Nachricht noConvoy im Zustand FrontRole::noConvoy::wait empfangen wurde. Ein weiteres Beispiel ist der Wechsel im Statechart Synchronization von noConvoy nach convoyFront, der innerhalb von d_1 beendet sein muss.

Für eine Komponente, die mehrere Echtzeit-Koordinationsmuster anwendet, sind die Verifikationsergebnisse der einzelnen Echtzeit-Koordinationsmuster immer noch erfüllt [GTB+03]. Aufgrund dieses kompositionellen Ansatzes lassen sich mechatronische Systeme sehr einfach komponentenbasiert entwickeln.

2.4.4 Einbettung hybrider Komponenten

Die Typdefinition der Komponenten aus Abbildung 2.21 spezifiziert, dass die Shuttle Komponenten zwei verschiedene Unterkomponenten einbettet. Eingebettete Komponenten sind im Kontext von mechatronischen Systemen zur Ressourceneinsparung nicht immer aktiv. Die Aktivierung und Deaktivierung, oder auch Rekonfiguration, wird von Software übernommen.

Das Modell der Realtime Statecharts aus Abbildung 2.22 wird dementsprechend erweitert zu so genannten Hybriden Rekonfigurations Charts. Die Komponenten werden Zuständen zugeordnet, so dass hier von Zustandskonfigurationen gesprochen wird. Der Velocity Regler ist aktiv in dem Zustand convoyFront und noConvoy. Beide Regler sind aktiv in dem Zustand convoyRear, wobei der Wert des Distance Reglers noch als Eingabe für den Velocity Regler dient. Das das Modell der Hybriden Rekonfigurations Charts für den weiteren Verlauf der Arbeit essentiell ist, wird es im Folgenden detailliert beschrieben.

Das klassische hybride Automatenmodell aus Definition 4 ermöglicht keine modulare Rekonfiguration zur Laufzeit. Dieser Nachteil kann durch den hybriden Rekonfigurations

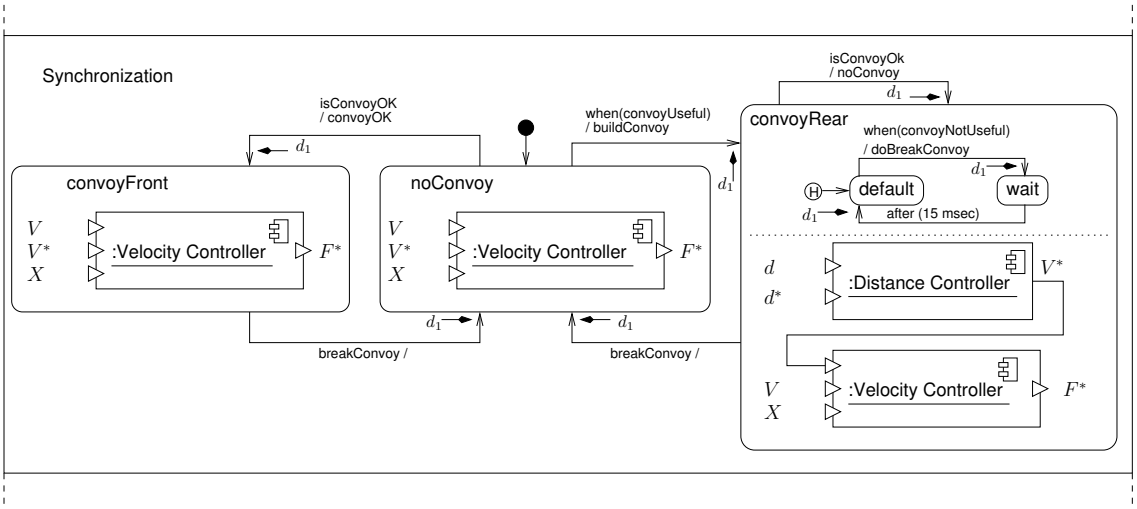


Abbildung 2.23: Einbettung von kontinuierlichen Unterkomponenten in ein hybrides Rekonfigurations Chart

Automaten aus [BGH05a] aufgehoben werden. Wie auch beim hybriden Automaten bettet ein hybrider Rekonfigurations Automat Komponenteninstanzen in die Zuständen ein und tauscht diese durch einen Zustandswechsel aus. Jedoch bietet der hybride Rekonfigurations Automat zusätzlich die Möglichkeiten, die Struktur und den internen Zustand der Komponenten durch einen Wechsel der Locations zu modifizieren. Durch diese beiden Möglichkeiten erlaubt das Modell eine Rekonfiguration des Systems. Ein weiterer Vorteil des hybriden Rekonfigurations Automaten gegenüber dem hybriden Automaten liegt darin, dass die Ports immer in Abhängigkeit von einer Location aktiv sind. Ports, die innerhalb einer Location keine Signale empfangen, werden in dieser Location auch nicht mehr dargestellt. Durch diese Möglichkeit der Modellierung wird dem Betrachter sofort ersichtlich, von welchen Eingangsports der Ausgangsport abhängt.

Um die beschriebenen Vorteile umzusetzen, verwendet der hybride Rekonfigurations Automat im Gegensatz zum hybriden Automaten ein verändertes kontinuierliches Modell. In diesem Modell werden die Zustands-, Eingabe- und Ausgabevariablen in Abhängigkeit der jeweiligen Location angegeben.

Definition 15
Formal ist das kontinuierliche Modell $D(l)$ des hybriden Rekonfigurations-Automaten durch ein 7 Tupel $(V^x(l), V^u(l), V^y(l), F(l), G(l), C(l), X^0(l))$ definiert:

- $V^x(l)$*: Menge der Zustandsvariablen,*
- $V^u(l)$*: Menge der Eingabevariablen,*
- $V^y(l)$*: Menge der Ausgabevariablen,*

- $F(l) \subseteq EQ(V^{\dot{x}} \bigcup V^a, V^x \bigcup V^u \bigcup V^a)$*: beschreibt den Fluss der Zustandsvariablen,*
- $G(l) \subseteq EQ(V^{\dot{y}} \bigcup V^a, V^x \bigcup V^u \bigcup V^a)$*: bestimmt die Ausgabevariablen,*
- $C(l) \in COND(V^x)$*: Invariante, welche die Menge der zulässigen Zustände bestimmt und*
- $X^0(l)$*: Menge der Anfangszustände.*

Ein Nachteil ist, dass durch das Einführen von Zuständen, die den Austausch von Reglern realisieren, die Zustandsmenge zunimmt. Dies hat zur Folge, dass hybride Rekonfigurations Automaten sehr umfangreich werden. Des Weiteren kann in einem hybriden Rekonfigurations Automaten keine Schaltdauer angegeben werden. Automatenmodelle bieten ebenfalls keine Möglichkeit, hierarchische Zustände zu verwenden. Hybride Rekonfigurations Chart, wie sie im Folgenden eingeführt werden, integrieren alle Konzepte.

Hybride Rekonfigurations Charts [BGH05a] bauen wie schon erwähnt auf dem Konzept der Realtime Statecharts und der hybriden Rekonfigurations Automaten auf. Mit ihnen ist es möglich, das Verhalten hybrider Komponenten, die sich durch die Kombination von diskreten und kontinuierlichen Komponenten auszeichnen, zu modellieren. Sie besitzen zusätzlich zu allen Eigenschaften der Realtime Statecharts die Möglichkeit, zwischen *atomaren* Umschalttransitionen und *nicht-atomaren* Umschalttransitionen zu unterscheiden. Außerdem ist es möglich, in den Zuständen Komponenteninstanzdiagramme einzubetten, so dass Rekonfiguration ermöglicht wird. Da Rekonfiguration nicht nur allein durch den Austausch von Komponenteninstanzen erreicht werden kann, erlaubt es das Konzept der hybriden Rekonfigurations Charts zusätzlich, die interne Struktur und den Zustand der Komponenten zur Laufzeit zu ändern. In jedem Zustand, ausgenommen Start- und Stopzustand, können Komponenteninstanzdiagramme eingebettet werden.

Die Komponenteninstanzdiagramme bestehen aus den Komponenteninstanzen, die in der Komponente eingebettet sind, deren Verhalten durch das hybride Rekonfigurations Chart beschrieben wird. Für jede Komponenteninstanz kann angegeben werden, ob sie in einem Zustand existiert oder nicht. Falls eine hybride bzw. eine diskrete Komponenteninstanz in einem Zustand vorhanden ist, wird zusätzlich spezifiziert, in welchem Zustand sich diese Komponenteninstanz befindet. Daraus ergibt sich ebenfalls, welche Ports aktiv und welche inaktiv sind. Bei der Einbettung einer kontinuierlichen Komponente in einen Zustand wird kein interner Zustand spezifiziert. Folglich sind auch alle Ports der kontinuierlichen Komponente aktiv. Neben der Einbettung von Komponenteninstanzen wird zusätzlich spezifiziert, welche Delegations und Assemblys in diesem Zustand aktiv sind. Durch diese vielfältige Spezifikation wird für jeden Zustand des hybriden Rekonfiguration Charts ein Komponenteninstanzdiagramm erstellt. Verlässt bzw. betritt die Komponente einen Zustand, wird bei der Modellierung vorausgesetzt, dass die eingebetteten Komponenteninstanzen ihren internen Zustand zeitgleich verlassen bzw. betreten.

Die Transitionen des hybriden Rekonfigurations Charts können zusätzlich zu den Transitionen eines Realtime Statecharts eine Umschaltfunktion besitzen. Diese ermöglicht es, die Ausgänge der eingebetteten Komponenteninstanzen in einem Zustand auf die Ausgänge der eingebetteten Komponenteninstanzen in einem anderen Zustand umzuschalten. Eine Umschalttransitionen besteht aus einer Funktion f_{fade} und einem Intervall $d = [d_{low}, d_{up}]$, das die minimale und maximale Dauer des Umschaltens spezifiziert, sowie der Ports, die ineinander übergeblendet werden. Die Dauer der Umschaltfunktion wird als Deadline für die Umschalttransitionen spezifiziert. Durch das Konzept der Umschaltfunktion existieren im Gegensatz zum hybriden Rekonfigurations Automaten keine Zustände, die das Überblenden zwischen zwei Reglern realisieren. Dies führt dazu, dass ein hybrides Rekonfigurations Chart nicht so komplex ist wie ein hybrider Rekonfigurations Automat. Des Weiteren ermöglicht ein hybrides Rekonfigurations Chart die Spezifikation von mehreren Hierarchieebenen.

Abschließend wird die formale Definition eines hybriden Rekonfigurations Chart gegeben. Die Definition baut auf der Definition eines hybriden Automaten (siehe Definition 4) auf.

Definition 16
Ein Hybrides Rekonfigurations Chart *wird durch ein 6-Tuple* (L, D, I, O, T, S^0) *beschrieben. Dabei ist* L *eine endliche Menge von Locations,* D *eine Funktion über* L*, die jedem* $l \in L$ *ein kontinuierliches Modell zuordnet,* $D(l) = (V^x(l), V^u(l), V^y(l), F(l), G(l), C(l), X^0(l))$ *ist ein kontinuierlicher Block wie in Definition 15 beschrieben,* I *ist eine endliche Menge von Eingabesignalen,* O *eine endliche Menge von Ausgabesignalen,* T *eine endliche Menge von Transitionen und* $S^0 \subseteq \{(l, x) | l \in L \wedge x \in X(l)\}$ *die Menge der initialen Locations.*

Für jede Transition $(l, g, g^i, a, l') \in T$ *gilt, dass* $l \in L$ *die Sourcelocations,* $g \in COND(V^x(l) \cup V^u(l))$ *ein kontinuierlicher Guard,* $g^i \in \wp(I \cup O)$ *der I/O-Guard,* $a \in [[V^x(l) \to \mathbb{R}] \to [V^x(l') \to \mathbb{R}]]$ *die Aktualisierung kontinuierlicher Daten und* $l' \in L$ *die Targetlocation ist. Für jedes* $l \in L$ *wird gefordert, dass* $D(l)$ *wohl-definiert ist.*

Das Hybride Rekonfigurations Charts erlaubt, dass jeder Location eine eigene Variablenmenge besitzt. Mit V^x *wird die Vereinigung aller* $V^x(l)$ *beschrieben.* V^u *und* V^y *werden analog bestimmt. Mit* $V^x(F(l))$ *werden die* Variablenmengen der Locations *beschrieben. Alle zugewiesenen Ausgabevariablen werden analog als* provided Ausgabevariablenmenge *(*$V^y(F(l) \cup G(l))$*) und alle Eingabevariablen als* required Eingabevariablenmenge *(*$V^u(F(l) \cup G(l))$*) bezeichnet.*

Für eine Einbettung benötigt eine übergeordnete Komponente jedoch nicht alle Informationen, welche die hybriden Rekonfigurations Charts der eingebetteten Komponenteninstanzen liefern. Es sind lediglich die nach außen sichtbaren Schnittstellen notwendig. Diese werden von einem Interface Statechart beschrieben.

Definition 17
Ein Interface Statechart für einen hybriden Rekonfigurations Automat $M = (L, D, I, O, T, S^0)$ *existiert genau dann, wenn für das kontinuierliche Modell* D *folgendes gilt:*

- $V^y \cap V^u = \emptyset$,
- *alle* $v \in V^x$ *sind Uhren:* $\dot{v} = 1$,
- *der Seiteneffekt a für jede Transition* (l, g, g^i, a, l') *ist beschränkt durch* OP_{const}, *wobei* OP_{const} *die Menge der möglichen Konstanten ist,*
- *das kontinuierliche Verhalten von* V^y *ist unbestimmt.*

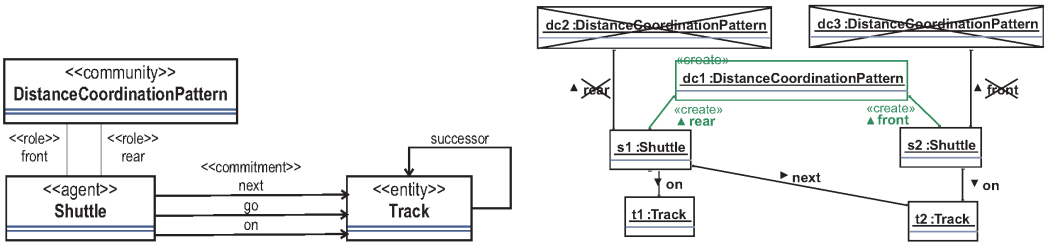
2.4.5 Anpassung der Softwarestruktur

Echtzeit-Koordinationsmuster spezifizieren die Koordination zwischen unterschiedlichen mechatronischen Komponenten. Während der Laufzeit können Komponenten allerdings aktiviert und deaktiviert werden. Um dies auch zu berücksichtigen, muss ein geeignetes Modell verwendet werden. dass die Echtzeit-Koordinationsmuster entsprechend integriert. So ein Modell modelliert die Strukturanpassung von Software. Bei Betrachtung wird ein Systemzustand durch eine Konfiguration aus Komponenten und Echtzeit-Koordinationsmuster charakterisiert. Die Erzeugung und Löschung von Echtzeit-Koordinationsmustern sowie der Austausch von Komponenten kann also durch ein Graphtransformationssystem formalisiert werden [Sch06].

Der MECHATRONIC UML Ansatz unterstützt die Spezifikation von Strukturen durch Klassendiagramme. Strukturänderungen können durch Story Diagramme modelliert werden. Der Ansatz unterstützt ferner die Verifikation von strukturellen Invarianten [BBG+06].

Für das Shuttlebeispiel ist ein Klassendiagramm in Abbildung 2.24(a)) dargestellt. Es stellt die Komponenten des physikalischen Modells dar (Shuttles und Schienenabschnitte). Auf ein Schienenstück passt genau ein Shuttle. Die Position eines Shuttles ist durch die on Assoziation modelliert; die go Assoziation modelliert die physikalische Bewegung auf einem Schienenstück.

Wie gerade erwähnt findet die Kollisionsvermeidung durch das DistanceCoordination-Pattern statt. Das DistanceCoordinationPattern wird erzeugt, sobald ein Shuttle sich einem anderen Shuttle nähert. Die Instanziierungsregel createDC erzeugt das Echtzeit-Koordinationsmuster, sofern zwei unverbundene Shuttle da sind (siehe Abbildung 2.24(b)).

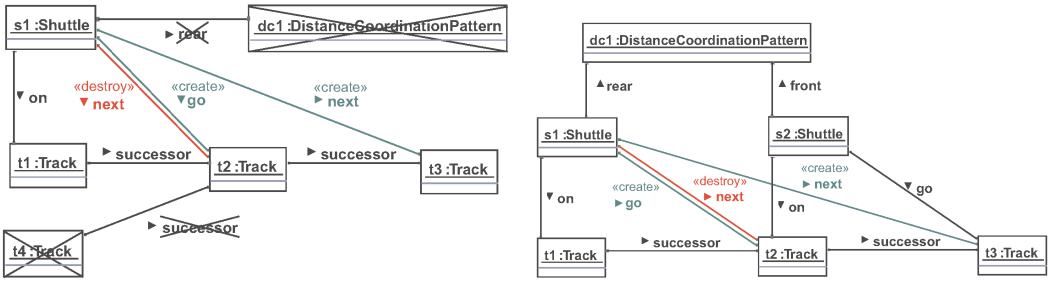


(a) Klassendiagramm

(b) Instanziierungsregel: Erzeugung des DistanceCoordinationPattern

Abbildung 2.24: Klassendiagramm und Instanziierung eines Echtzeit-Koordinationsmuster

Zwei Regeln sind spezifiziert: (1) goSimple1 (siehe Abbildung 2.25(a)) beschreibt die Bewegung eines einzelnen Shuttles von einem Schienenstück zum nächsten. (2) goDC1 (siehe Abbildung 2.25(b)) erlaubt dem rear Shuttle sich nur zu bewegen, wenn das front Shuttle sich auch bewegt.



(a) Ein freies Shuttle bewegt sich

(b) Koordinierte Fahrweise zweier Shuttles

Abbildung 2.25: Verhaltensregeln

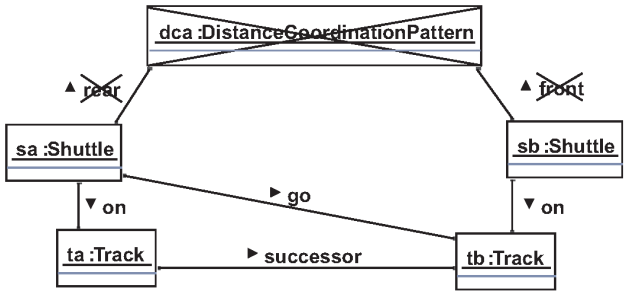


Abbildung 2.26: Invariante: Keine unkontrollierte Bewegung zweier benachbarter Shuttles

Eine in diesem Kontext angewendete Invariante ist, die es gilt zu verifizieren, dass ein Shuttle nie versucht, ein bereits besetztes Schienenstück zu befahren, ohne sich mit dem

anderen Shuttle abgesprochen zu haben. Abbildung 2.26 zeigt diese Invariante als Negative Anwendungsbedingung.

2.5 Zusammenfassung

In diesem Kapitel wurden die Grundlagen dieser Arbeit beschrieben. Hierzu wurde zuerst die grundlegende hierarchische Struktur eines mechatronischen System nach Lückel, wie es in dieser Arbeit aufgefasst wird, diskutiert. Ein Operator-Controler-Modul beschreibt die Informationsverarbeitung eines mechatronischen System. Es ist in die drei Ebenen Controller, Reflektorischer Operator und Kognitiver Opertor aufgebaut. Die Software eines OCMs ist sowohl für die Koordination innerhalb als auch für die Koordination mehrerer OCMs verantwortlich. Da die Software komplex und sicherheitskritisch ist, muss diese verifiziert werden.

Anhand des Vorgehens der modell-basierten Entwicklung, wurden Modelle und Verfahren zur Verifikation von mechatronischen Systemen vorgestellt. Diese sind jedoch für die komplexen, vernetzten mechatronischen Systeme, wie sie hier beschrieben sind, nicht alleine anwendbar. Der MECHATRONIC UML Ansatz wird hier als Lösung vorgeschlagen. Dieser kombiniert die Techniken aus den verschiedenen Domänen der Softwaretechnik und der Regelungstechnik, die es erlauben, solche Modelle adäquat zu modellieren und zu verifizieren. Die Architektur der Software wird mittels Komponentendiagrammen und Echtzeit-Koordinationsmustern beschrieben. Zur Spezifikation der komponenteninternen Struktur werden Klassendiagramme verwendet. Das Verhalten wird durch Realtime Statecharts und Hybride Rekonfigurations Charts beschrieben. Die dynamischen Strukturänderungen werden durch Story Pattern beschrieben.

Aufbauend auf den Techniken des MECHATRONIC UML Ansatzes werden nun in den nächsten Kapiteln Erweiterungen und neue Verfahren für die Modellierung, als auch für die Verifikation, komplexer, vernetzter mechatronischer Systeme vorgestellt.

Kapitel 3

Verifikation eines OCM

In Kapitel 2.1 wurde die Struktur eines mechatronischen Systems (siehe Abbildung 2.1) beschrieben. Der Fokus dieses Kapitels liegt nun auf der Verifikation eines OCMs, genauer gesagt, auf der Verifikation des korrekten Zusammenspiels hinsichtlich der hierarchischen Rekonfiguration (siehe Abschnitt 2.4.4) zwischen dem *Reflektorischen Operator* und dem *Controller*. In Abbildung 3.1 ist die Aufgabe der Verifikation eines OCM, wie sie in diesem Kapitel im Folgenden beschrieben wird, skizziert. Hierzu wird zuerst anhand der in den Grundlagen vorgestellten Modellierungskonzepte der MECHATRONIC UML (siehe Abschnitt 2.4) informal ein Beispiel eingeführt (siehe Abschnitt 3.1). Danach wird die syntaktische Verifikation für die hierarchische Rekonfiguration bedingt durch rein lokal relevante Zeitbedingungen für die Rekonfiguration innerhalb eines OCMs beschrieben (Abschnitt 3.2). Anschließend wird in Abschnitt 3.3 das Beispiel um nicht lokale Eigenschaften bezüglich der Zeitbedingungen sowie um nicht-deterministisches Verhalten bezüglich der Rekonfiguration erweitert. Anhand dieses Beispiels werden die Konzepte zur Verifikation für die sichere Rekonfiguration in Abschnitt 3.4 erklärt. Das Kapitel schließt mit einer Zusammenfassung in Abschnitt 3.6.

3.1 Beispiel

Das *Feder-Neige-Modul* ist ein Teilsystem des in Abschnitt 1.2 vorgestellten Shuttlesystems der *Neuen Bahntechnik Paderborn*. Das Feder-Neige-Modul ist ein Beispiel für ein komplexes mechatronisches System. Die Aufgabe des Feder-Neige-Moduls ist es, einen maximalen Fahrkomfort für die Passagiere eines Shuttles zu ermöglichen. Befährt ein Shuttle einen Schienenabschnitt, sammelt es Informationen über die Streckenverhältnisse und sendet diese nach Befahren des Streckenabschnittes an eine Streckenabschnittskontrolle. Die Streckenabschnittskontrolle kann diese Information weiteren Shuttles zur Verfügung stellen, so dass diese anhand der Streckeninformationen Unebenheiten ausgleichen können und somit der Fahrkomfort erhöht wird. Die Verarbeitung der Streckeninformationen sowie das damit verbundene Erreichen des maximalen Fahrkomforts werden

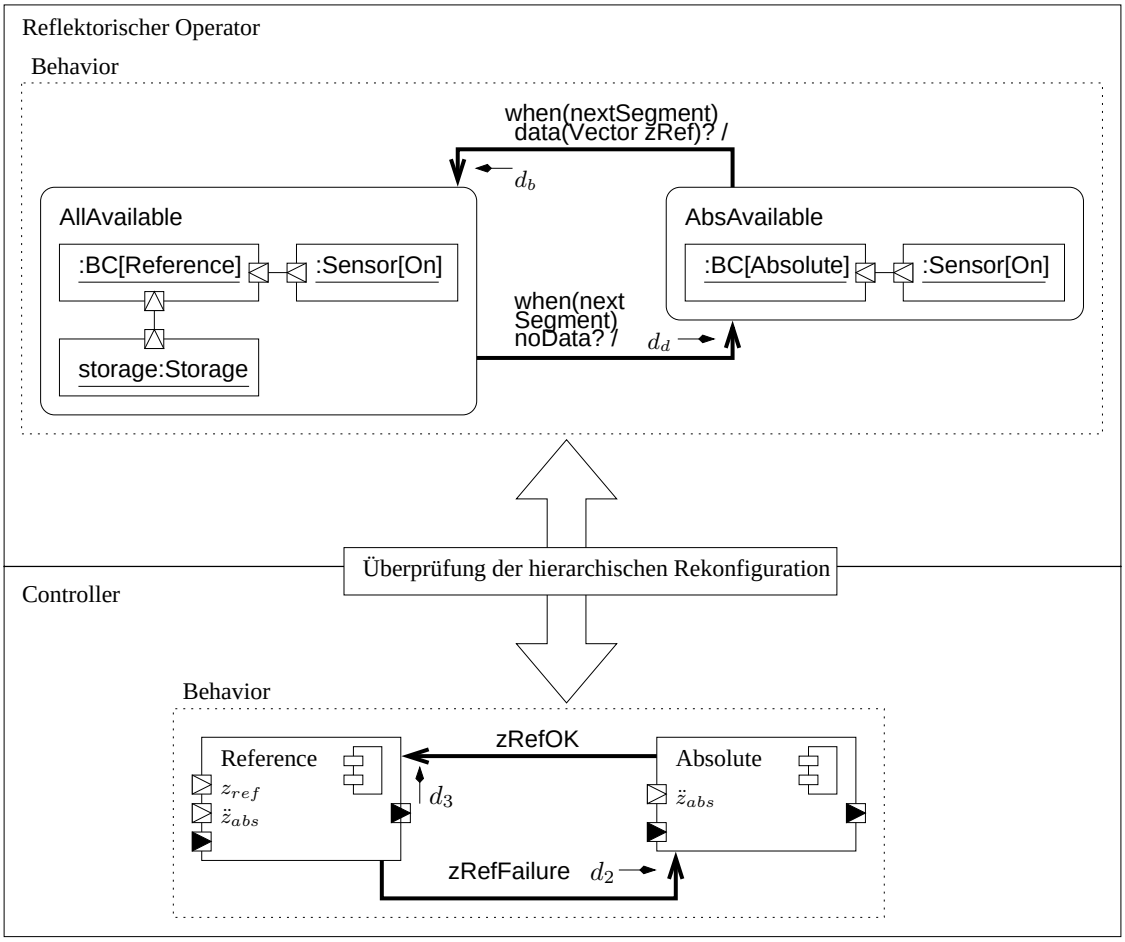


Abbildung 3.1: Verifikation eines OCM

vom Feder-Neige-Modul ermöglicht [TMV06]. Der physikalische Aufbau aller relevanten Teilmodule ist in Abbildung 3.2 dargestellt und wird im Folgenden im Detail beschrieben.

Das Feder-Neige-Modul besteht unter anderem aus einem Aufbau, der über Luftfedern mit dem Fahrwerk verbunden ist. Zudem enthält das Feder-Neige-Modul verschiedene Regler, welche die Position der drei hydraulischen Zylinder A, B und C aufgrund von Werten der Sensoren regeln. Die Zylinder wiederum beeinflussen aktiv den Aufbau und damit auch das Fahrwerk [HSE02].

Die Sensoren messen die Streckenverhältnisse und liefern diese Werte als Eingabe für den jeweiligen Regler. Dieser führt anhand der Eingabewerte Berechnungen durch, um die Ergebnisse anschließend an die Zylinder weiterzuleiten. Je nach Ergebnis ihrer Berechnung wird die Position der Zylinder verändert, so dass sich das Feder-Neige-Modul

der Schienenumgebung anpassen kann und damit den für die Strecke höchstmöglichen Fahrkomfort liefert.

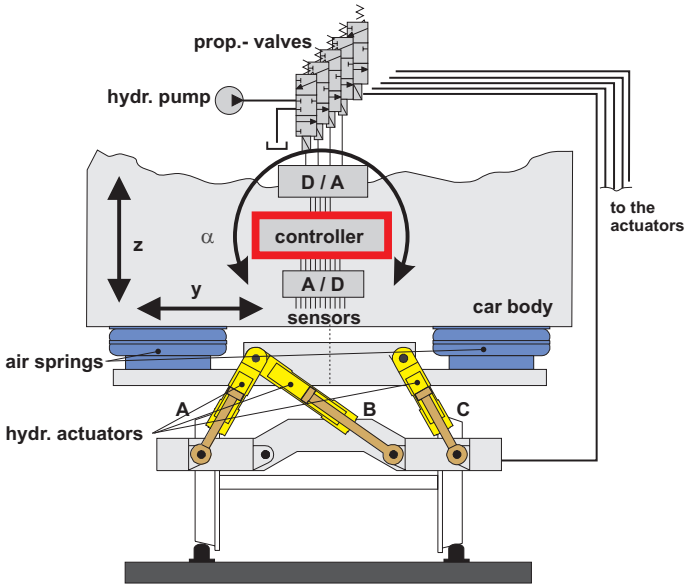


Abbildung 3.2: Schematische Darstellung des Feder-Neige-Moduls

Das verwendete CAE Werkzeug CAMeL [Ric96] ermöglicht die Beschreibung des kompletten kontinuierlichen Parts des Modells durch strikte, hierarchische Blockdiagramme mit nichtlinearen Gleichungen der Art

$$\dot{\underline{x}} = \underline{f}(\underline{x}, \underline{u}, t) \quad \text{und} \quad \underline{y} = \underline{g}(\underline{x}, \underline{u}, t)$$

wobei $\underline{x}$ der Zustandsvektor, $\dot{\underline{x}}$ die erste Ableitung des Zustandsvektors, $\underline{y}$ der Ergebnisvektor, $\underline{u}$ der Eingabevektor und t die Zeit ist.

Das Feder-Neige-Modul besteht aus drei Reglern. Der Regler Reference stellt den höchsten Komfort zur Verfügung, indem er durch die Vorgabe einer Trajektorie die Bewegung des Aufbaus beschreibt, um Unebenheiten der Strecke auszugleichen. Um die Stabilität des Systems zu gewährleisten und um damit Entgleisungen des Shuttles entgegenzuwirken, müssen alle Sensoren immer korrekte Werte liefern (siehe Grundlagenkapitel 2.2). Falls dies beim Regler Reference nicht mehr der Fall ist, wird der Regler Absolute verwendet, der als Eingabe nur die vertikale Beschleunigung des Aufbaus benötigt. Falls auch dieser Sensor ausfällt, wird der Regler Robust aktiv, der den geringsten Komfort zur Verfügung stellt. Dieser Regler benötigt nur die Standardeingabewerte, um Stabilität zu gewährleisten.

Das Blockdiagramm der Regler ist in Abbildung 3.3 dargestellt. Die Komponente body control (BC) ist verantwortlich für die übergeordnete Regelung des Feder-Neige-Moduls

und besteht aus den eben beschriebenen drei Reglern. Abhängig von den Eingabesignalen wird zwischen den Reglern umgeschaltet. Das Referenzsignal ist mit z_{ref} und die absolute Beschleunigung mit $\ddot{z}_{abs}$ bezeichnet. Die Ausgabewerte sind $X_{Z,A,ref}, \ldots, X_{Z,C,ref}$ und geben die Position der hydraulischen Zylinder an.

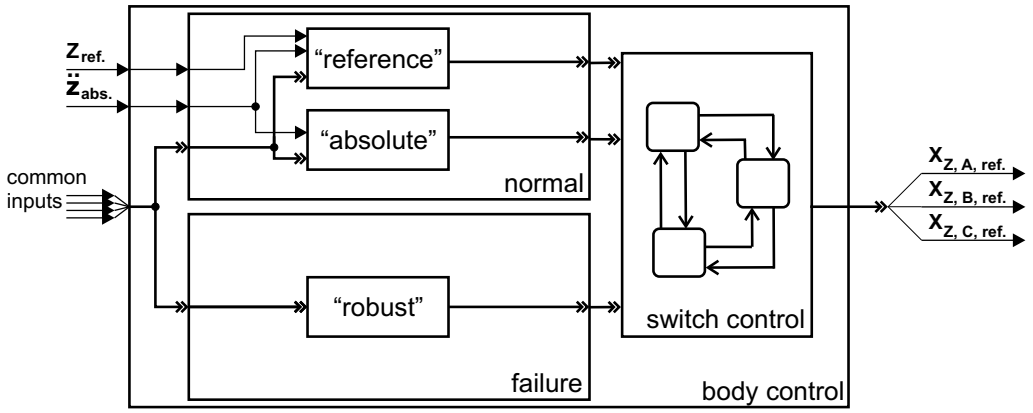


Abbildung 3.3: Blockdiagramm der Regler

Beim Wechsel zwischen zwei Reglern wird zwischen zwei Fällen unterschieden: *Atomares Umschalten* und *nicht-atomares Umschalten*. Im ersten Fall findet der Wechsel ganz normal zwischen zwei Berechnungsschritten statt. Im Beispiel wäre der Wechsel vom Block normal zum Block failure atomar. Im anderen Fall ist es notwendig, eine Umschaltfunktion $f_{switch}(t)$ und eine Umschaltdauer zu spezifizieren [OMT+08]. Im Beispiel wäre das der Wechsel zwischen dem Regler reference und dem Regler absolut.

3.1.1 Komponenten Struktur

In diesem Abschnitt wird beschrieben, wie sich die Architektur mittels MECHATRONIC UML beschreiben lässt. Abbildung 3.4 zeigt die Architektur. Die Monitor Komponente koordiniert die Einbettung der Komponenten BC, Sensor, und Storage. Außerdem kommuniziert sie über das Echtzeit-Koordinationsmuster MonitorRegistration mit einer Streckenabschnittskontrolle Registry. Die Streckenabschnittskontrolle sendet Informationen über die kommenden Streckenabschnitte an die Komponente Monitor, die diese daraufhin in der Unterkomponente Storage abspeichert. Die Unterkomponente Sensor liefert die benötigten Signale.

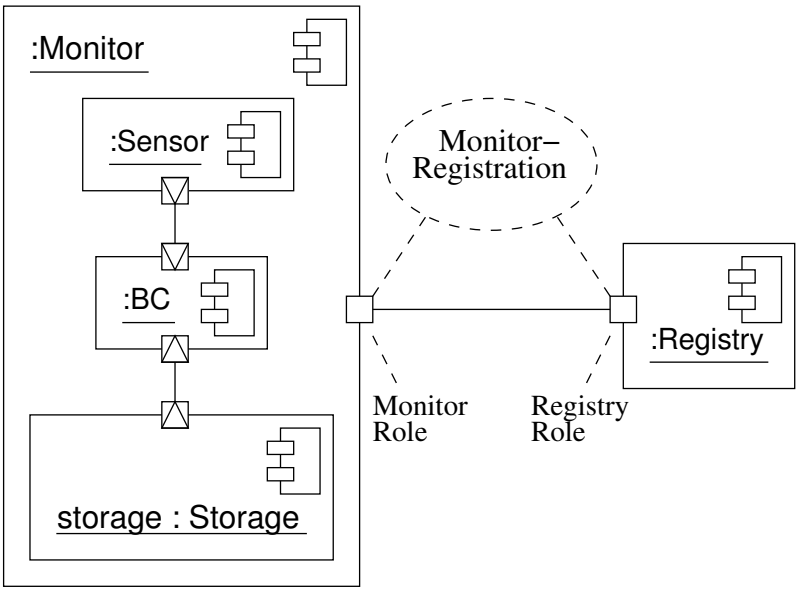


Abbildung 3.4: Die Architektur

3.1.2 Verhalten der Komponenten

Das Verhalten sowie die Einbettung von kontinuierlichem Verhalten einer Komponenten wird durch ein Hybrides Rekonfigurations Chart beschrieben (siehe Kapitel 2.4). Abbildung 3.5 zeigt das Hybride Rekonfiguration Chart der Komponente BC. Diese besteht aus den drei Zuständen Robust, Absolute und Reference. Jedem Zustand ist ein kontinuierlicher Regel mit verschiedenen Eingangs- und Ausgangssignalen zugeordnet. Die fett gezeichneten Transitionen zeigen an, das bei diesen Zustandswechseln umgeschaltet wird, es sich also um *nicht-atomares Umschalten* handelt. Die anderen Transitionen stellen *atomares Umschalten* dar.

3.1.3 Beschreibung des Interface

Für die Einbettung oder Verknüpfung von hybriden Komponenten werden nicht immer alle Details der Realisierung einer Komponente benötigt. Es reichen hier die Informationen über die externen Signale aus, so dass die Kompatibilität analysiert werden kann. Abbildung 3.6 zeigt das abgeleitete Interface Statechart (siehe Definition 17) der Komponente BC. Die BC Komponente hat drei mögliche verschiedene externe Zustände mit unterschiedlichen kontinuierlichen Eingabesignalen.

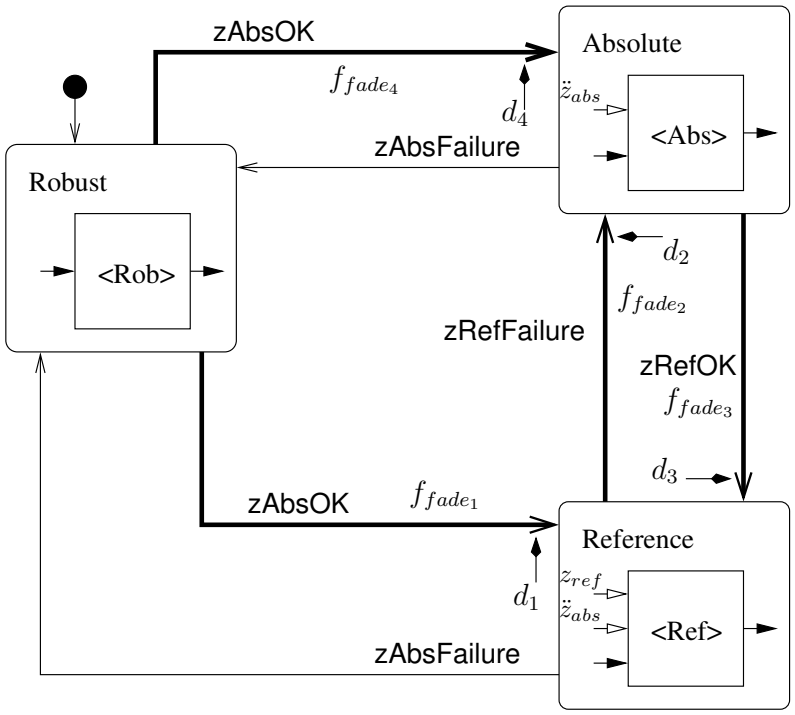


Abbildung 3.5: Verhalten der Body Komponente

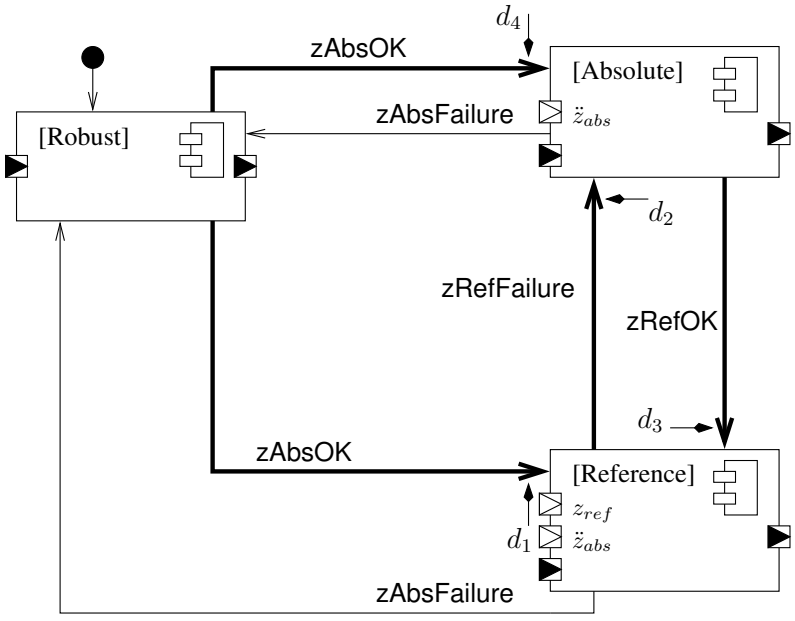


Abbildung 3.6: Interface Statechart der Komponente BC

Im Folgenden wird die Einbettung der Unterkomponenten innerhalb eines hybriden Rekonfigurations Charts beschrieben. Diese Einbettung erlaubt es später, die korrekte Einbettung rein syntaktisch zu überprüfen (siehe Abschnitt 3.2).

3.1.4 Einbettung

Durch Zuordnung von Konfigurationen der Untergeordneten Komponenten zu jedem Zustand eines hybriden Rekonfigurations Chart wird die Einbettung realisiert. (siehe Abbildung 3.7). Ein Wechsel zwischen den Zuständen in der Monitor Komponente impliziert einen Wechsel der Zustände im Interface Statechart der eingebetteten Komponenten.

Das Verhalten der Monitor Komponente ist wie folgt durch das hybride Rekonfigurations Chart beschrieben (siehe Abbildung 3.7). Der obere orthogonale Zustand besteht aus den Zuständen AbsAvailable, NoneAvailable, RefAvailable und AllAvailable. Die letzten beiden repräsentieren, ob die benötigte Referenztrajektorie für den aktuellen Schienenabschnitt zur Verfügung steht oder nicht.

Die Komponente BC ist jedem Zustand des oberen orthogonalen Zustandes zugeordnet. So ist z. B. die Komponenteninstanz BC im Zustand Reference dem Zustand AllAvailable der Komponente Monitor zugewiesen, in dem z_{ref} sowie as $\ddot{z}_{abs}$ verfügbar sind.

Die Kommunikation mit der Streckenabschnittskontrolle Registry ist im unteren orthogonalen Zustand in der Abbildung (Abbildung 3.7) modelliert. Der obere orthogonale Zustand synchronisiert sich mit dem unteren Zustand.

3.2 Verifikation der hierarchischen Rekonfiguration bedingt durch lokale Zeitbedingungen

Für die Verifikation eines OCMs werden die folgenden zwei Verifikationsverfahren bereitgestellt. Als erstes muss das reine Echtzeitkoordinationsverhalten der Software, modelliert durch Komponenten und Echtzeit-Koordinationsmuster, verifiziert werden. Hierfür wird der in [Gie03][GTB+03][Hir04][BGH+05b] beschriebene Ansatz verwendet und im Folgenden kurz skizziert. In diesem Zusammenhang wird eine Definition von Verfeinerung gegeben, welche die Eigenschaft der *deadlock-Freiheit* erhält. Weiterhin wird eine Menge von kompositionellen Bedingungen eingeführt. Diese Grundlagen bilden ein Framework, welches es erlaubt, komplexe Echtzeitsysteme auf high-level Ebene (siehe Abschnitt 2.4) zu spezifizieren und zu verifizieren. Der Ansatz bezieht sich auf die Notation von Komponenten und Echtzeit-Koordinationsmustern, wie sie in Kapitel 2.4 eingeführt wurden. Der Vorteil ist, dass der erörterte Ansatz es erlaubt, ein System zu verifizieren,

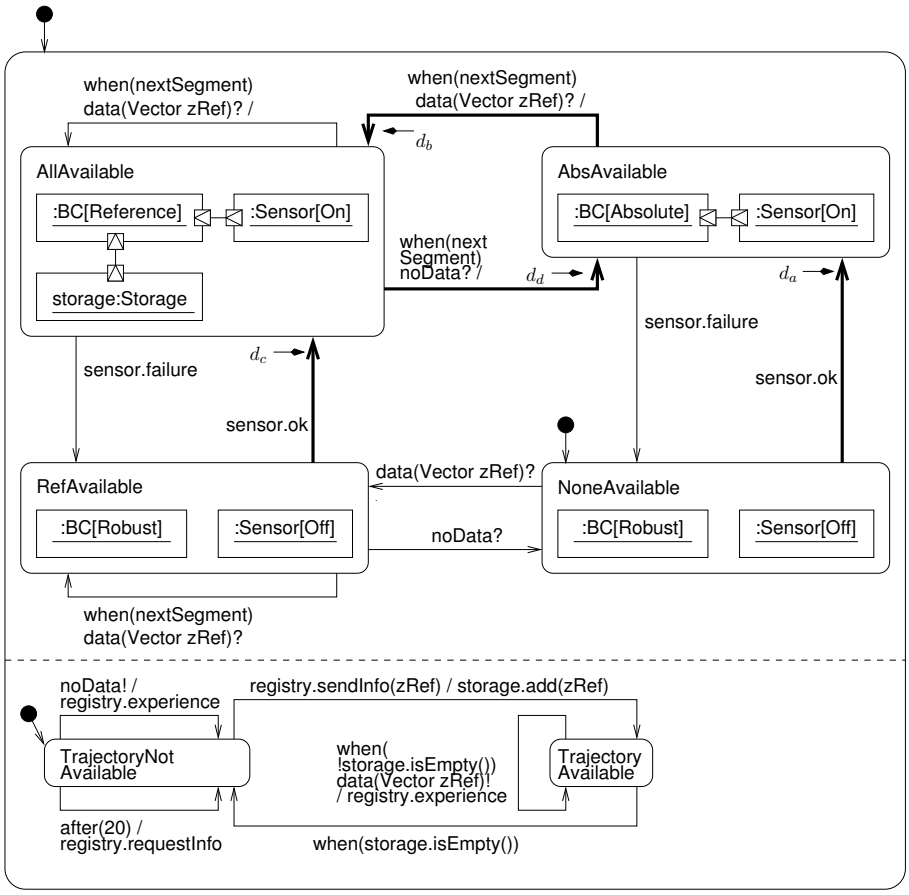


Abbildung 3.7: Einbettung der Untergeordneten Komponenten im Monitor

ohne jemals den gesamten Zustandsraum aufzubauen. Stattdessen kann jede Komponente und jedes Echtzeit-Koordinationsmuster einzeln durch einen Model Checker verifiziert werden. Die folgenden fünf Schritte skizzieren den Ablauf der Verifikation:

1. Spezifiziere alle Echtzeit-Koordinationsmuster und ihre Rollen.
2. Verifiziere jedes Echtzeit-Koordinationsmuster einzeln.
3. Spezifiziere die Komponenten durch Verfeinerung der Rollen zu Ports.
4. Verifiziere jede Komponente einzeln.
5. Konstruiere durch Komposition der Echtzeit-Koordinationsmuster und Komponenten das vollständige Modell.

Schritt 5 sichert die korrekte semantische Komposition bei einer korrekten syntaktischen Komposition zu. Ein zusätzlicher sechster Schritt, der die Verifikation des ganzen Sys-

tems durchführen würde, ist nicht erforderlich. Dieses Resultat folgt aus Theorem 1 in [Gie03]. Jedoch ist dieses Theorem nur unter der Annahme gültig, dass lokale Eigenschaften für Echtzeit-Koordinationsmuster und Komponenten vorliegen. Abbildung 3.8 zeigt die Überschneidung der Elemente Echtzeit-Koordinationsmuster und Komponenten. Diese ist immer durch ein wohl definiertes Protokoll, das von beiden Seiten eingehalten wird, gekennzeichnet. Der Echtzeitcharakter der Protokolle stellt sicher, dass unbegrenzte gegenseitige Sperreffekte ausgeschlossen werden. In nicht zeitbehafteten Systemen ist ein ähnlicher Ansatz nicht möglich, da maximale Sperrzeiten nicht explizit angegeben sind und deshalb zyklische Sperreffekte entstehen können (siehe hierzu [Gie00]).

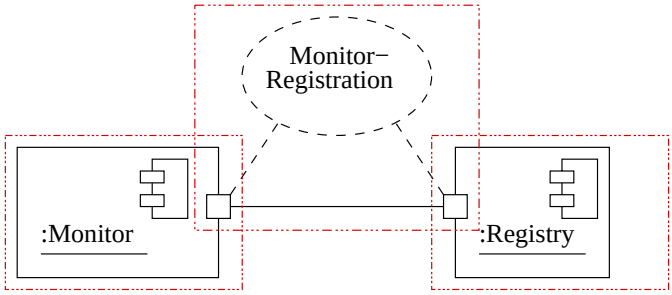


Abbildung 3.8: Verifikation des Echtzeitkoordinationsverhaltens der Software, modelliert durch Komponenten und Echtzeit-Koordinatiosmuster

Als zweites muss die im letzten Abschnitt beschriebene hierarchische Komponentenstruktur für die Modellierung von diskreten und kontinuierlichen regelungstechnischen Verhalten hinsichtlich der konsistenten Rekonfiguration und der korrekten Echtzeitsynchronisation hinsichtlich der Rekonfiguration verifiziert werden. Das zweite Verifikationsverfahren kann dabei in das erste integriert werden [GBSO04][GH05b][GH06].

Hierbei werden die in der Einleitung 1.3 beschriebenen Konzepte der Abstraktion und Verfeinerung eingesetzt. Wie schon angedeutet, ist es das Ziel, den bisherigen kompositionellen Ansatz weiter zu verwenden. Hierzu muss nun geeignet vom hybriden Verhalten abstrahiert werden. In Abbildung 3.9 ist eine Abstraktion skizziert, die vorgenommen werden muss. Um diese formal zu beschreiben, wird im Folgenden das Hierarchie- und Modularitätskonzept von hybriden Rekonfigurations Charts formal eingeführt.

3.2.1 Modularität

In diesem Abschnitt wird das modulare Konzept, welches von dem Verifikationsverfahren unterstützt wird, beschrieben. Als erstes werden notwendige Bezeichnungen eingeführt. Eine hierarchische Komponentenstruktur ist durch eine Menge von Komponenteninstanzen $C_1, \ldots, C_n$ und Funktionen $sub, sub^* : \{1, \ldots, n\} \rightarrow \wp(\{1, \ldots, n\})$ beschrieben.

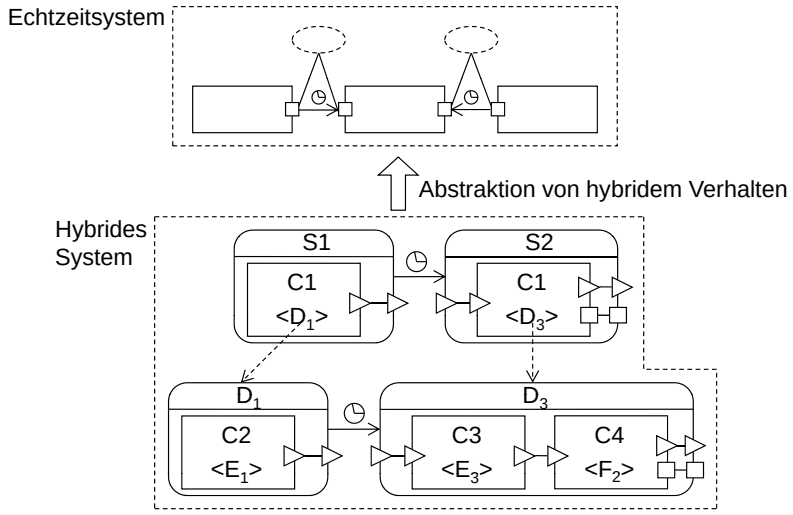


Abbildung 3.9: Abstraktion

Hierbei ist $sub(i)$ die Indexmenge von allen direkt angrenzenden Komponenten von C_i. $sub^*(i)$ bildet die transitive Hülle von sub einschließlich des Input-Index i. Das Verhalten jeder Komponenteninstanz C_i wird durch einen zugehörigen Automaten M_i, der ein hybrides Rekonfiguration Chart (siehe Definition 16, Kapitel 2.4.4) repräsentiert, beschrieben. Weiterhin gibt es einen Automaten M_i^I, der das zugehörige Interface Statechart (siehe Definition 17, Kapitel 2.4.4) repräsentiert. Das Interface eines Automaten M wird mit $I(M)$ bezeichnet. Es besteht aus Eingabe- und Ausgabevariablen. Die Parallelausführung zweier Automaten wird durch $\|$ beschrieben. Das Interface eines einzelnen Automaten M kann eingeschränkt werden, indem alle Signale zur Synchronisation sowie alle Variablen, die nicht im Interface vorkommen, versteckt werden. Dies wird mit $M|_I$ beschrieben. (siehe [GH05a] für die verwendete Definition der Verfeinerung)

Im Folgenden wird nun das Modularitätskonzept formal definiert. Für jede *Blatt-Komponente* C_j mit $sub(j) = \emptyset$ gilt die Verfeinerung $\sqsubseteq_{HY}$ zwischen dem Komponentenverhalten M_j, beschrieben durch das hybride Rekonfigurations Chart, und dem abstrakten Interface Statechart M_j^I

$$M_j \sqsubseteq_{HY} M_j^I. \tag{3.1}$$

Für jede nicht *Blatt-Komponente* C_k wird angenommen, dass für M_k^I und M_k inklusive aller Interface Automaten der untergeordneten Komponenten gilt:

$$(M_k \| \left(\|_{l \in sub(k)} M_l^I\right))|_{I(M_k^I)} \sqsubseteq_{HY} M_k^I. \tag{3.2}$$

Die Bedingung 3.2 gilt, da zwischen dem Verhalten einer Komponente und dem Verhalten eines Interface Automaten die *Verfeinerung*

$$M_k|_{I(M_k^I)} \sqsubseteq_{HY} M_k^I \tag{3.3}$$

gilt und dass der Einfluss von M_k auf das Verhalten seiner Unterkomponenten M_l mit $l \in sub(k)$ immer in einer korrekten *Einbettung*

$$(M_k \| \left(\|_{l \in sub(k)} M_l^I\right))|_{I(M_k)} \sqsubseteq_{HY} M_k \tag{3.4}$$

endet.

Die Bedingungen 3.1 und 3.2 stellen jeweils eine lokale Abstraktionsbedingung für jede Komponente C_k und dem zugehörigen Verhalten M_k sowie des Interface Automaten M_k^I dar. Per Induktion über die Baumstruktur kann nun modular gezeigt werden, dass jedes Komponentenverhalten sowie der Interface Automat durch den vollständigen Baum, bestehend aus den direkten und indirekten Unterkomponenten inklusive der Komponenten selber, abgebildet werden kann.

$$\left(\|_{l \in sub^*(k)} M_l\right)|_{I(M_k)} \sqsubseteq_{HY} M_k \qquad \wedge \qquad \left(\|_{l \in sub^*(k)} M_l\right)|_{I(M_k^I)} \sqsubseteq_{HY} M_k^I \tag{3.5}$$

Um das beschriebene Modularitätskonzept praktisch anwenden zu können, wird ein effektiver und effizienter Algorithmus, um die Verfeinerung und Einbettung zu überprüfen, benötigt. Für allgemeine hybride Systeme, die sich nicht mehr durch die Klasse der *rectangular automata*, bei denen die analogen Variablen Trajektorien mit teilweise-linearer Entwicklung und Sprüngen, bedingt durch Re-Initialisierungen, folgen, beschreiben lassen, ist die Erreichbarkeit nicht mehr entscheidbar [HKPV98]. Selbst bei der Verwendung der eingeschränkten Klasse ist die Verifikation durch Model Checking ebenfalls nur für kleine Beispiele anwendbar [Dor08].

Aufgrund dieser Tatsache werden die Analysen zuerst auf das reine Echtzeitverhalten und die Analyse, ob rein konsistente Konfigurationen mit wohl-definierten kontinuierlichen Gleichungen erreicht werden können, beschränkt.

In einem ersten Schritt wird hierfür von dem kontinuierlichen Verhalten eines Automaten M abstrahiert, indem nur die Uhren betrachtet werden, so dass das Hybride Rekonfigurations Chart sowie das zugehörige Interface Statechart auf ein Realtime Statechart abgebildet werden können. Als nächstes kann das Realtime Statechart nach den in [Hir04][BGHS04] beschriebenen Regeln auf Timed Automata abgebildet werden. Auf dem Modell der Timed Automata kann nun die Überprüfung der Verfeinerung sowie die Überprüfung der korrekten Einbettung durchgeführt werden. Für die Timed Automata müssen hier anstelle von $\sqsubseteq_{HY}$ nur $\sqsubseteq_{RT}$ sowie die Bedingung $D^e(M_R, c) \subseteq D^e(M_I, c'')$ für die Variablenabhängigkeiten überprüft werden.

3.2.2 Überprüfung der Verfeinerung

In vorangegangenen Arbeiten [GBSO04] wurden die Zeitbedingungen in den Interface Statecharts auf solche eingeschränkt, welche die reine Schaltdauer einer Transition angeben (die Verweildauer in einem Zustand in einem Timed Automaton (siehe [Hir04])).

Definition 18
Sei $M = (L, D, I, O, T, S^0)$ *ein hybrider Automat. Ein Zustand* l' *ist ein* Umschalt-*Zustand, falls eine Uhr* c *und Konstanten* a' *und* b' *existieren, so dass nur eine einzelne Transition* $t \in T$ *um* l' *zu verlassen existiert mit* $t = (l', a' \leq t \leq b', l''')$ *und für alle Transitionen* $(l'', g, S, R, l') \in T$ *gilt, dass* $c \in R$ *und* $C(l') = c \leq b'$*. Ein Zustand* l *ist* passiv *genau dann, wenn* $C^I(l) = true$ *und für alle Transitionen* $(l, g, S, R, l') \in T$*, wobei* l' *ein* Umschalt-*Zustand ist und* $g = true$ *gilt.*

Definition 19
Ein Automat $M = (L, D, I, O, T, S^0)$ *wird als* simpel *bezeichnet, falls Mengen von passiven Zuständen* L_p *und Umschalt-Zustände* L_f *existieren, mit* $L = L_p \uplus L_f$*.*

Aufgrund dieser Annahmen ist es möglich, die Verfeinerung zwischen einem simplen Interface Automaten $M^I = (L^I, D^I, I^I, O^I, T^I, S^{0^I})$ für ein gegebenes simples Interface Statechart und dem zugehörigen Komponentenverhalten $M = (L, D, I, O, T, S^0)$ durch reine syntaktische Regeln zu überprüfen. Für $L_f \subseteq L$ und $L^I = L_p^I \uplus L_f^I$, wobei L_p^I die Menge der passiven Zuständen von L^I, L_f^I und L_f die Mengen der Umschalt-Zustände, $map : L_p^I \to \wp(L)$ eine Abbildungsfunktion zwischen den passiven Zuständen des Interface Automaten und den zugehörigen Zuständen des Hybriden Rekonfigurations Charts der unterliegenden Komponente sind, kann einfach überprüft werden, ob für die Verfeinerung des Echtzeitverhaltens gilt:

1. Für alle Zustände $l_i \in L_p^I$, $l \in map(l_i)$ und $l' \in L_f$ wird überprüft, ob für jedes Paar von Transitionen zwischen passiven Zuständen und Umschalt-Zuständen gilt, dass bei der Verfeinerung nicht die externen Signale sowie Zeitrestriktionen, wie vom Interface Automaten vorgegeben, verändert werden:

$$\begin{gathered}\forall (l, g, S, R, l'), (l', g', S', R', l'') \in T, \exists (l_i, g'', S, R, l_i'), (l_i', g''', S', R', l_i'') \in T^I : \\ g' = \text{true} \wedge g'' = a \leq t \leq b \wedge g''' = a^I \leq t \leq b^I \wedge \\ c(l') = t \leq b \wedge c(l_i') = t \leq b^I \wedge \\ a^I \geq a \wedge b \geq b^I \wedge \\ l' \in L_f \wedge l_i' \in L_f^I \wedge \\ g = \text{true} \wedge R = R' = \{t\} \wedge l'' \in map(l_i'')\end{gathered}$$

$$\forall (l_i, g, S, R, l_i') \in T^I : g = \text{true} \wedge \exists (l, g', S, R', l') \in T \wedge \bigvee_{(l, g', S, R', l') \in T} g' = \text{true}.$$

2. Für alle Zustände $l, l' \in L \setminus L_f$ muss überprüft werden, dass die Transitionen zwischen ihnen zugehörige Transitionen im Interface Automaten haben oder die folgende Abbildung erfüllen:

$$\forall (l, g, S, R, l') \in T :$$
$$(S = \emptyset \wedge \forall l_i \in L^I : (l \in map(l_i) \Rightarrow l' \in map(l_i))) \vee$$
$$(g = \text{true} \wedge \exists (l_i, \text{true}, S, R', l'_i) \in T^I) : l' \in map(l'_i).$$

3. Für alle Zustände $l \in L_i$ muss überprüft werden, dass $l \notin L_f$ und dass sie durch die initialen Locations des zugehörigen Interface Automaten abgedeckt sind:

$$\exists l_i \in L_i^I : l \in map(l_i).$$

Zusätzlich muss für alle $l \in L$ und $l_i \in L_i$ mit $l \in map(l_i)$ gelten, dass jede Abhängigkeit zwischen den Eingabe und Ausgabe Variablen $D(l)$ ebenfalls im zugehörigen Interface Automaten als $D^I(l_i)$ vorkommen.

3.2.3 Überprüfung der korrekten Einbettung

Um die korrekte Einbettung sicherzustellen, muss als erstes die korrekte Echtzeitkoordination der Umschaltzeiten der Transitionen überprüft werden. Hierbei reicht es aus, die simplen Interface Statecharts zu betrachten, um zu zeigen, dass ein Hybrides Rekonfiguration Chart alleine eine Abstraktion von einem Hybriden Rekonfigurations Chart zusammen mit den Interface Statecharts der Unterkomponenten ist (siehe Bedingung 3.2). Da auch hier die Erreichbarkeitsfrage nicht entscheidbar und für die meisten praktischen Systeme auch nicht anwendbar ist, wird in [GBSO04] vorgeschlagen, anstelle der Analyse des kompletten Zustandsraums statische Analysen, die auf den Transitionsmengen und Zustandsmengen der Hybriden Rekonfigurations Automaten M_i und der Interface Automaten M_l^I mit $l \in sub(i)$ operieren, zu verwenden. Dieses wird im Folgenden formalisiert.

Gegeben sei eine Funktion $mode : L \times sub(i) \rightarrow \bigcup_{j \in sub(i)} L_j^I$, so dass für alle $l \in L$ und $j \in sub(i)$ gilt, dass $mode(l, j) \in L_j^I$. Weiterhin wird angenommen, dass alle lokalen Transitionen mit den Eingabe und Ausgabe Variablen $\emptyset$ markiert sind und dass $L_f \subseteq L$ und $L_j^I = L_{j,p}^I \uplus L_{j,f}^I$ mit $L_{j,p}^I$ und L_p alles passive Locations und $L_{j,f}^I$ und L_f alles Umschaltlocations sind.

1. Für alle Zustände $l \in L \setminus L_f$ und $l' \in L_f$ wird überprüft, dass für jedes Paar von Transitionen zwischen passiven Locations und Umschaltlocations gilt, dass für jede

zugehörige Komponente $j \in sub(i)$ gilt, dass ein Paar von Transitionen existiert, das in den vorgegebenen Zeitschranken von M arbeitet:

$$\forall(l, g, S, R, l'), (l', g', S', R', l'') \in T, \exists(l_j, g_j, S_j, R_j, l'_j), (l'_j, g'_j, S'_j, R'_j, l''_j) \in T_j^I :$$
$$g' = a \leq t \leq b \wedge g'_j = a_j^I \leq t \leq b_j^I \wedge$$
$$c(l') = t \leq b \wedge c(l'_j) = t \leq b_j^I \wedge$$
$$a_j^I \geq a \wedge b \geq b_j^I \wedge$$
$$l' \in L_f \wedge l'_j \in L_{i,f}^I \wedge$$
$$g' = \text{true} \wedge R' = \{t\} \subseteq R \wedge l_j = mode(l, j) \wedge l''_j\, mode(l'', j)$$

2. Für alle Zustände $l, l' \in L \setminus L_f$ jeder zugehörigen Komponente $j \in sub(i)$ und alle Transitionen $(l, g, S, R, l') \in T$ muss überprüft werden, das jede atomare Transition keine Rekonfiguration hervorruft oder von ihr abgedeckt ist:

$$(mode(l, j) = mode(l', j)) \vee (\exists(l_j, \text{true}, S_j, R_j, l'_j) \in T^I) : l'_j = mode(l', j).$$

3. Für alle initialen Zustände $l \in L_i$ und alle zugehörigen Komponenten $j \in sub(i)$ mit $l_j = mode(l, j)$, muss überprüft werden, dass sie alle durch alle initialen Locations abgedeckt werden:

$$l_j \in L_{j,i}^I.$$

In [GBSO04][BGO04] wurde gezeigt, dass die Überprüfungen ausreichen um für simple Interface Statecharts zu zeigen, dass die Bedingung 3.2 erfüllt ist.

In Abbildung 3.10 ist das Verhalten der Monitorkomponenten und der Teil der Interface Statecharts der eingebetteten BC Komponenten (siehe Abbildung 3.6 und 3.7) dargestellt. Die Semantik der hybriden Rekonfigurations Charts erfordert, dass eine Transition vom Zustand AbsAvailable zum Zustand AllAvailable einen Transitionswechsel in der BC Komponente vom Zustand Absolute zum Zustand Reference bedingt. Für die monitor Komponenten gilt, dass die Transition innerhalb des Zeitintervalls d_b abgeschlossen ist. Für den implizierten Zustandswechsel in der BC Komponente hingegen gilt, dass dieser innerhalb des Zeitintervalls d_3 abgeschlossen sein muss. Dies bedeutet eine konsistente parallele Ausführung beider Transitionen die erfordert, dass $d_3 \subseteq d_b$ erfüllt ist. Für die Transition zu AllAvailable und der Transition zum Zustand Reference in der BC Komponente gilt, dass $d_2 \subseteq d_d$ erfüllt sein muss.

Für einen spezifischen Zustandswechsel eines Hybriden Rekonfigurations Chart gilt, dass es nur dann in einer inkorrekten Rekonfiguration endet, falls die Abhängigkeit der zugehörigen Zustandskombinationen der Unterkomponenten einen Zyklus enthält. Da die verwendete Verfeinerungsbeziehung sicherstellt, dass für jede Abhängigkeit zwischen Eingabe- und Ausgabesignalen von M diese ebenfalls im zugehörigen Interface Automaten M^I vorkommen, ist es in diesem Falls ausreichend, den Interface Automaten zu

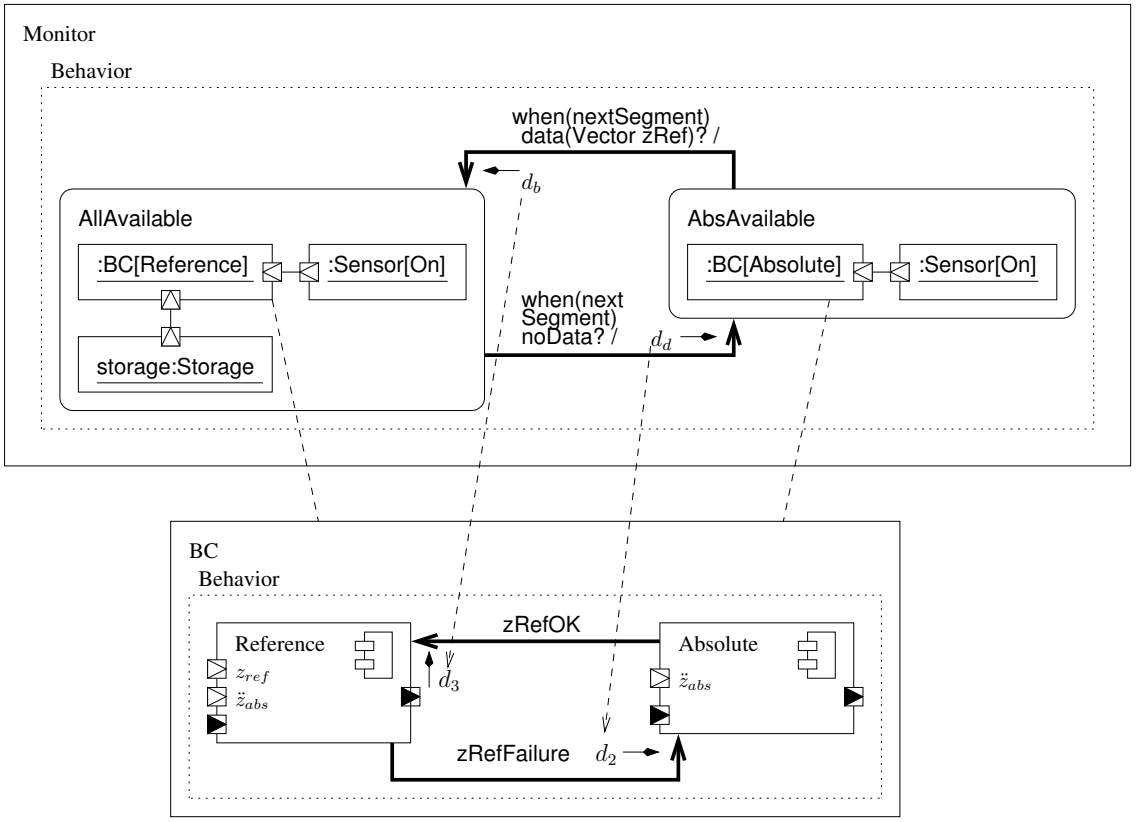


Abbildung 3.10: Schema für die syntaktische Überprüfung bei der korrekten Einbettung und korrekten Rekonfiguration

betrachten und alle Kombinationen von $mode$ zu betrachten um diese inkorrekten Zustandskonfigurationen auszuschließen.

3.2.4 Grenzen des Ansatzes

Der hier vorgestellte Ansatz erlaubt die systematische Entwicklung von mechatronischen Systemen mit sicheren Rekonfigurationseigenschaften, bei denen eine strikte Hierarchie mit einer top-down Rekonfiguration zugrunde liegt. Jedoch zeigt dieser Ansatz eine Reihe von Einschränkungen, welche oftmals bei komplexen mechatronischen Systemen so nicht mehr angenommen werden können.

Eine Haupteinschränkung ist, dass in einem Interface Statechart nur die Dauer einer Transition und nicht deren Ausführungszeitpunkt spezifiziert werden kann. Beispiele hierfür sind Interface Statecharts, bei denen z.B. die Frequenz zwischen Rekonfigurationen modelliert werden soll. In dem hier vorliegenden Beispiel könnte durch die Domäne der Re-

gelungstechnik vorgegeben werden, um die Stabilität des Systems zu beeinflussen, dass nach dem Umschalten von Zustand Reference zu Zustand Absolute eine Zeitspanne verstreichen muss, bevor die BC Komponente es erlaubt, zurück zu dem komfortableren Reference Zustand erneut zu wechseln.

Eine weitere Restriktion ist die strikte top-down Rekonfiguration. Wenn z.B. ein Sensor bereits einen Fehler feststellt, muss dieser in den Unterkomponenten behandelt werden. Der bisherige Ansatz von Interface Statecharts erlaubt dies nicht. Um also auch Warnungen, Fehler oder ähnliche Signale zu den eingebetteten Komponenten zu propagieren, müssen die Interface Statecharts um entsprechendes *proaktives Verhalten* [Woo00][Ge05] angereichert werden, so dass das Interface Statechart aufgrund dieser Ereignisse entsprechende Reaktionen innerhalb einer Zeitspanne anstoßen kann.

Als Beispiel hierfür kann der Fall betrachtet werden, dass die BC Komponente bemerkt, dass die Referenzdaten ein unerwartetes Problem hervorrufen und dieses an die Monitor Komponenten propagieren möchte. Dieses kann ebenfalls für das Interface Statechart für die BC Komponente bedeuten, dass hier eine Deadline spezifiziert werden muss, um den Zustand Reference zu verlassen um zu einen sicheren Zustand zu wechseln.

Formal kann man diese beiden Fälle wie folgt definieren, um die Interface Automaten zu erweitern:

Definition 20
Ein Interface Automat M ist komplex*, falls es nicht mehr simple, aber immer noch deterministisch ist. Ein Interface Automat M ist* proaktiv*, falls es autonom entscheiden kann, dass eine Rekonfiguration erforderlich ist.*

3.3 Modellierung hierarchischer Rekonfiguration bedingt durch proaktives Verhalten

In diesem Abschnitt wird ein Beispiel für die Modellierung von hybriden Systemen mit proaktivem Verhalten gegeben. Hierzu wird das Beispiel aus Abschnitt 3.1 erweitert. Als erstes wird das neue Verhalten informal beschrieben. Danach werden die Auswirkungen des neuen Verhaltens auf das gesamte System beschrieben.

3.3.1 Erweitertes Beispiel

In Abschnitt 3.1 wurde das Feder-Neige Modul beschrieben und die Software modelliert. Eine Besonderheit der Modellierung war hier die strikte top-down Hierarchie. So war es für die BC Komponente nicht möglich, die Monitor Komponente direkt über Signale zu

beeinflussen. Wenn z.B. ein Fehler in der BC Komponente passiert, während die Komponente im Reference Zustand ist, muss die BC direkt zum Robust wechseln und gleichzeitig die übergeordnete Monitor Komponente informieren, um entsprechend zu reagieren. Ebenfalls soll ein zu schnelles Hin- und Herschalten vermieden werden, um Stabilitätseigenschaften zwischen den Zuständen Absolute und Reference sicherzustellen.

3.3.2 Verhalten der Komponente

In Abbildung 3.11 ist das erweiterte Verhalten der BC Komponente dargestellt. Das Verhalten der alten BC Komponente ist nun um proaktives Verhalten erweitert worden. Befindet sich die BC Komponente in dem Zustand Reference, kann die Komponente nun autonom entscheiden, in den Zustand Robust zu wechseln. Dies ist durch eine non-urgent Transition (gestrichelte Linie), die Nicht-Determinismus modelliert, beschrieben. Im Detail ist dies wie folgt modelliert: Solange der Zustand Reference aktiv ist, sendet die BC Komponente eine Nachricht switchToRobust zu der übergeordneten Monitor Komponente. Nach Verschicken wechselt die BC Komponente in den Zustand Timeout Zustand. Ist der Timeout erreicht, wechselt die BC Komponente in den Robust Zustand.

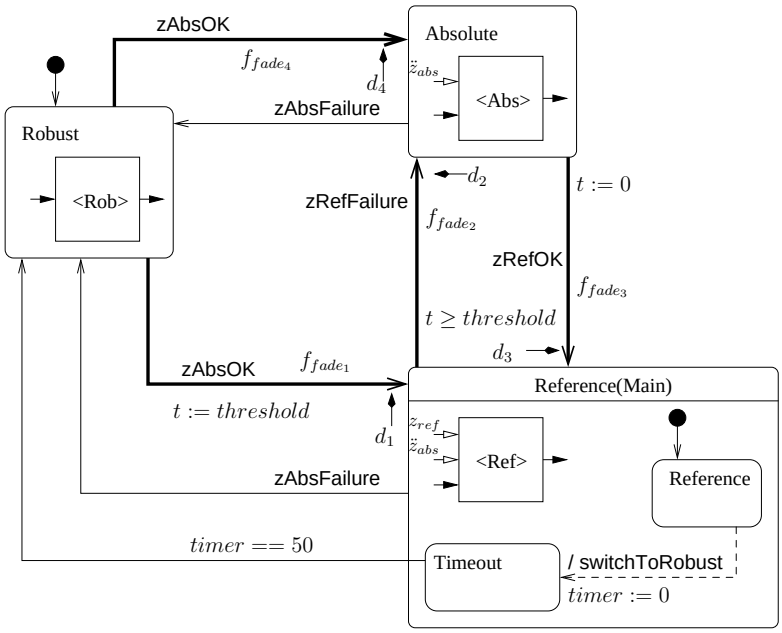


Abbildung 3.11: Verhalten der BC Komponente

Um das Schaltverhalten zwischen Absolute und Reference zu kontrollieren, wird ein Timer t verwendet. Jedesmal, wenn der Reference ausgehend vom Zustand Absolute betreten wird, wird die Uhr t auf den Wert 0 gesetzt. Um ein sofortiges Zurückspringen

und somit Instabilität zu vermeiden, wird eine Bedingung (Schwellwert) t>=threshold der Transition hinzugefügt. Alle anderen eingehenden Transitionen zum Reference Zustand bekommen als Zuweisung t:=threshold, was bedingt, dass der Schwellwert nicht berücksichtigt wird.

In Abbildung 3.12 ist das Interface Statechart der sensor Komponente dargestellt. Das Interface Statechart besteht aus zwei Zuständen, on und off.

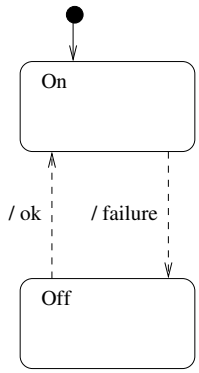


Abbildung 3.12: Interface Statechart der Komponente Sensor

3.3.3 Einbettung

Das erweiterte Verhalten der Monitor Komponente (ähnlich Abbildung 3.7) ist in Abbildung 3.13 dargestellt. Zusätzlich zum alten Verhalten der Monitor Komponente müssen nun das proaktive Verhalten und die Zeitangaben der eingebetteten Komponenten berücksichtigt werden. In diesem Fall muss das von der BC Komponente verschickte Signal switchToRobust entsprechend verarbeitet werden.

3.4 Verifikation der hierarchischen Rekonfiguration bedingt durch proaktives Verhalten

Um das Modularitätskonzept, wie im letzten Abschnitt beschrieben, für die Erweiterung der Interface Automaten anwenden zu können, müssen die Überprüfungen für die Verfeinerung sowie für die korrekte Einbettung erweitert werden. Die Erweiterungen werden im Folgenden beschrieben.

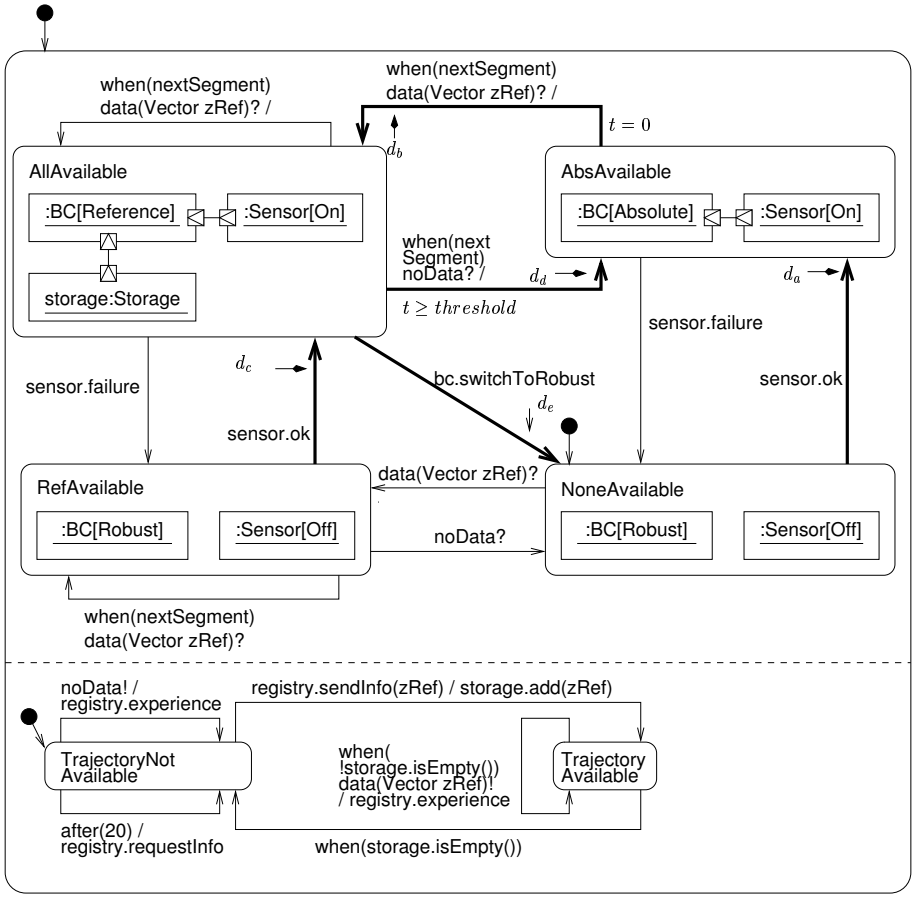


Abbildung 3.13: Einbettung von Verhalten in die Monitor Komponente

3.4.1 Überprüfung der Verfeinerung

Komplexe und proaktive Komponenten können nicht auf ein *simples* Interface Statechart abgebildet werden (siehe Definition 20). Deshalb können die bisher beschriebenen Verfahren zur Überprüfung, dass das Verhalten eines Interface Statecharts dem Verhalten einer Komponente entspricht (siehe Abschnitt 3.2), so nicht angewendet werden.

In [JGGS00] (siehe Verwandte Arbeiten 6.1.2) wird ein Ansatz vorgestellt, der die Verfeinerung $M \sqsubseteq_{RT} M^I$ überprüft. Jedoch wird hier gefordert, dass M^I deterministisch ist. Falls das Interface Statechart M^I komplex ist, jedoch nicht proaktiv, kann dieser Ansatz verwendet werden. For einen deterministischen Automaten M^I erhält man den entsprechenden *Test Automaten* M_t^I wie in [JGGS00] beschrieben und kann entsprechend $M \| M_t^I$ auf *time stopping deadlocks* überprüfen.

Für den Fall, dass das Interface Statechart M^I proaktiv ist und deshalb nichtdeterministisch, lässt sich das Verfahren aus [JGGS00] nicht anwenden. Um hier

einen deterministischen Automaten für einen nicht-deterministischen zu erhalten, lassen sich ähnliche Verfahren, wie in [Tri04] beschrieben, anwenden. Bei genauerer Betrachtung des Ansatzes [JGGS00] stellt man fest, dass bei der on-the-fly Bestimmung des Kreuzproduktes einfach eine eindeutige Abbildung von einem Zustand in das verfeinerte Modell existieren muss, die in [JGGS00] durch die deterministischen Eigenschaften von M^I garantiert wird.

Der hier vorgeschlagene Ansatz nutzt die Abbildung $map : L_p^I \rightarrow L$ zwischen den passiven Zuständen des Interface Automaten und den zugehörigen Zuständen in der Realisierung aus, um eine geeignete Lösung vorzuschlagen. Für die Abbildung map, die jedem Zustand in der Realisierung genau einen Zustand des Interface Automaten zuordnet ($\forall l \in L : |\{l' | l' \in map(l)\}| = 1$ und deshalb ist map^{-1} eine Funktion so dass $l' = map^{-1}(l)$) und für den Fall, dass keine zwei Transitionen mit der selben Quelle, Markierung und Ziellocation existieren ($\forall l, l' \in L, s \subseteq I \cup O : |\{(l, g, S, R, l') \in T\}| \leq 1$), kann das Kreuzprodukt syntaktisch erstellt werden $M' = M^I \times_{map} M$ mit $M' = (L', D', I', O', T', S'^0)$, $M^I = (L^I, D^I, I^I, O^I, T^I, S^{I,0})$ und $M = (L, D, I, O, T, S^0)$ und L_f^I und L_f sind Umschaltzustände von L^I oder L wie folgt:

- $L' = \{(l, l') \in L^I \times L\} \cup \texttt{error}$,
- $D'(l, l') = D^I(l) \| D(l')$ und $D'(\texttt{error})$ sind leer,
- $I' = I^I \cup I$, $O' = O^I \cup O$.
- $T' = \bigcup_{(l,l') \in L' \setminus \{\texttt{error}\}, S \subseteq I' \cup O'} T'_{l,l',S}$ mit $T'_{l,l',S} = T'^r_{l,l',S} \cup T'^{e1}_{l,l',S} \cup T'^{e2}_{l,l',S}$ ist die Vereinigung der zugehörigen normalen n und fehlerhaften Transitionsmengen ($e1$, $e2$).
- $S'^0 = S'^I \times S^0$

Die Menge der normalen Transitionen $T'^n_{l,l',S}$ bildet sich durch alle Paare von Transitionen die durch eine Übereinstimmung s in der Menge $\{((l, l'), g \wedge g', S, R \cup R', (l'', l''')) | (l, g, S, R, l'') \in T^I \wedge (l', g', S, R', l''') \in T \wedge l' = map^{-1}(l)\}$ landen.

Die initiale Menge von fehlerhaften Transitionen $T'^{e1}_{l,l',S}$ für $l \notin L_f^I$ oder $l' \notin L_f$ behandelt das Verhalten, bei dem das Interface Statechart schalten kann, aber nicht die Realisierung: $T'^{e1}_{l,l',S} = \{((l, l'), g \wedge \neg g'', S, \emptyset, \texttt{error}) | (l, g, S, R, l'') \in T^I \wedge g'' = \bigvee_{(l',g',S,R',l''') \in T} g'\}$.

Analog wird die nächste Menge von fehlerhaften Transitionen bestimmt $T'^{e2}_{l,l',S}$ für $l \notin L_f^I$ or $l' \notin L_f$ wobei das Verhalten betrachtet wird, das in der Realisierung schalten kann, jedoch nicht im Interface Statechart: $T'^{e2}_{l,l',S} = \{((l, l'), g' \wedge \neg g'', S, \emptyset, \texttt{error}) | (l', g', S, R', l''') \in T \wedge g'' = \bigvee_{(l,g,S,R,l'') \in T^I} g\}$.

Für $l \in L_f^I$ und $l' \notin L_f$ im Gegensatz kann angenommen werden, dass die Bedingungen einer Transition immer die Form $a' \leq t \leq b'$ haben. Deshalb ist $T'^{e1}_{l,l',S}$ gleich $\{((l, l'), a \leq t \leq a^I, S, \emptyset, \texttt{error}) | (l, a^I \leq t \leq b^I, S, R, l'') \in T^I \wedge (l, a \leq t \leq b, S, R, l'') \in T\}$ und

$T'^{e2}_{l,l',S}$ ist gleich $\{((l,l'), b^I \leq t \leq b, S, \emptyset, \texttt{error}) | (l, a^I \leq t \leq b^I, S, R, l'') \in T^I \wedge (l, a \leq t \leq b, S, R, l'') \in T\}$ um inkompatible Zeitbedingungen darzustellen. Im ersten Fall wird eine zu frühe Terminierung des Umschaltvorgangs überprüft, im zweiten Fall wird eine zu späte Terminierung überprüft.

Danach kann einfach überprüft werden, ob ein *time stopping deadlock* besteht oder der Zustand `error` in M' erreicht wird. Daraus lässt sich schließen, ob die Verfeinerung gilt oder verletzt ist. Aufgrund der Einschränkung, dass niemals zwei Transitionen mit derselben Quelle, Markierung und Ziellocation in T^I existieren, gilt, dass für jede $t' = (l', g', S, R', l''') \in T$ mindestens eine zugehörige Transition $t = (l, g, S, R, l'') \in T^I$ existiert, die in $T'^{n}_{l,l',S}$ ist und die Eigenschaft $l' = map^{-1}(l)$ erfüllt.

Eine Abbildung map, die jedem realisierten Zustand mehr als einen Zustand des Interface Automaten zuweist, resultiert nicht in der beabsichtigten Zustandsraumreduzierung für M^I bezüglich M und deshalb ist diese Restriktion anwendbar für die hier verwendete Idee. Im Falle, dass zwei Transitionen $(l, g, S, R, l') \in T^I$ und $(l, g', S, R', l') \in T^I$ mit derselben Start- und Ziellocation sowie Markierung existieren, müssen zwei Fälle unterschieden werden. (1) Falls $R = R'$ können diese einfach vereint werden in einer Regel $(l, g \vee g', S, R, l')$ ohne Verhaltensänderung. (2) Andernfalls muss ein zusätzlicher Zustand l'' und eine zusätzliche Uhr hinzugefügt werden und die erste Regel $(l, g, S, R, l') \in T^I$ durch die folgenden beiden Regeln ersetzen: $(l, g, S, R \cup \{t\}, l'')$ und $(l'', \text{true}, \emptyset, \emptyset, l')$ und die Angabe der Invarianten $C(l'')$ mit $t \leq 0$ um die Annahme zu erfüllen.

Um zu überprüfen, dass der hybride Rekonfigurations Automat von der Komponente BC eine korrekte Verfeinerung des Interface Statecharts der BC Komponente ist, muss ein Timed Automata Modell wie oben beschrieben, erstellt werden. Auf diesem Modell kann dann mittels Model Checking durch den Model Checker UPPAAL [BDL04] die korrekte Verfeinerung verifiziert werden. Hierzu muss die Eigenschaft A[] not deadlock sowie E<> BodyControl.Error überprüft werden.

3.4.2 Dynamische Überprüfung der Einbettung

Die dynamische Überprüfung der korrekten Einbettung der Komponenteninstanzen hat das gleiche Ziel wie die syntaktische Überprüfung. Sie berücksichtigt - im Gegensatz zur syntaktischen Überprüfung - auch komplexeres Zeitverhalten, das z.B. durch Timeguards ausgedrückt wird. Die dynamische Überprüfung ist damit für nicht simple, proaktive Interface Statecharts geeignet.

In Abbildung 3.14 sind ein Echtzeitsystem und ein hybrides System exemplarisch dargestellt. Die Verifikation des Echtzeitverhaltens jeder einzelnen Komponente wird mit Hilfe von Model Checking realisiert. Um zu überprüfen, ob die Rekonfiguration nicht zu Inkonsistenzen führt, müssen weitere Verifikationsschritte durchgeführt werden. Verlässt

bzw. betritt die übergeordnete Komponente ihren Zustand, müssen zeitgleich die eingebetteten Komponenteninstanzen ihren internen Zustand verlassen bzw. betreten. Dies ist jedoch nur möglich, wenn sich die spezifizierten Echtzeitangaben, wie Deadlines, Timeguards und Uhren-Resets nicht gegenseitig ausschließen. Dies wird mittels der dynamischen Überprüfung verifiziert.

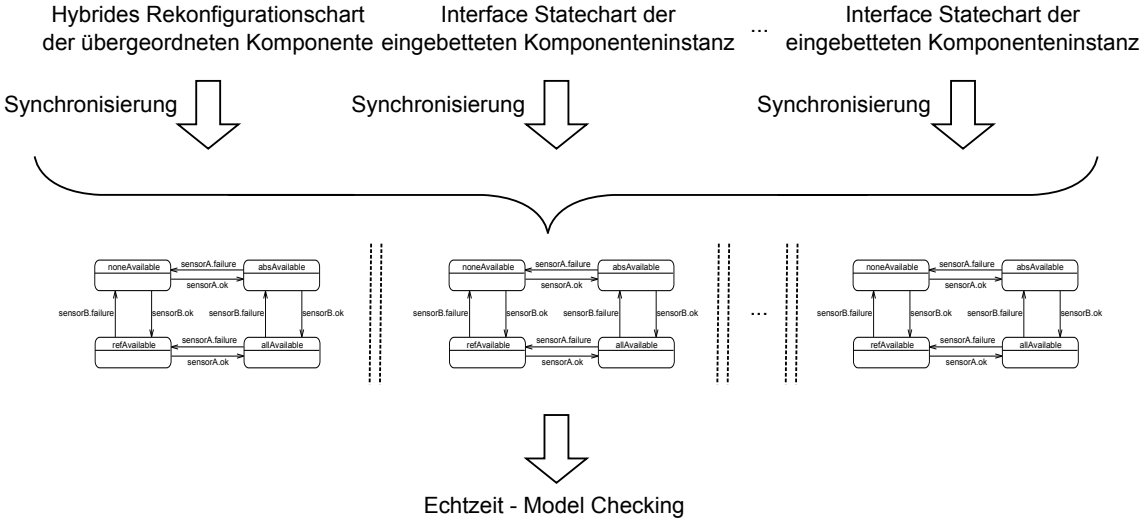


Abbildung 3.14: Dynamische Überprüfung der korrekten Einbettung der Komponenteninstanzen

Um zu verifizieren, ob eine Komponente korrekt in eine weitere Komponente eingebettet ist, darf sich das Echtzeitverhalten beider Komponenten nicht gegenseitig ausschließen. Dies ist der Fall, wenn das spezifizierte Echtzeitverhalten des gesamten Systems keinen Deadlock enthält. Eine Überprüfung auf Deadlockfreiheit kann erfolgen, wenn das hybride Rekonfigurationschart der übergeordneten Komponente und die Interface Statecharts der eingebetteten Komponenteninstanzen parallel initialisiert werden und das Verfahren des Model Checkings mit der Bedingung *A[] not deadlock* angewandt wird.

Um die dynamische Überprüfung für die korrekte Rekonfiguration zu realisieren, müssen sowohl das hybride Rekonfigurations Chart sowie das Interface Statechart in ein geeignetes Eingabemodell für einen Model Checker transformiert werden. In [Hir04][BGHS04] wurde eine Transformation, die Realtime Statecharts auf Timed Automata abbildet, formal beschrieben. Im Folgenden werden diese Transformationsregeln wiederverwendet und erweitert.

In hybriden Rekonfigurations Charts sind Komponenteninstanzen in Zustände eingebettet. Während der Transformation können die eingebetteten Instanzen vernachlässigt werden, da diese durch das Interface Statechart, welches ebenfalls transformiert wird, abgedeckt

werden. Aufgrund dieser Tatsache können für die Zustände genau dieselben Regeln wie in [Hir04][BGHS04] angewendet werden.

Neben den Zuständen müssen auch noch die Transitionen abgebildet werden. Im Gegensatz zu der Transformation in [Hir04][BGHS04] ist hier eine Transition mit einer Umschaltfunktion markiert. Da die Umschaltfunktion jedoch nicht das Echtzeitverhalten beeinflusst, kann sie ebenfalls vernachlässigt werden. So kann auch hier die Transformation aus [BGHS04] verwendet werden.

Bei der Modellierung des hybriden Rekonfiguration Charts der übergeordneten Komponente wird explizit angenommen, dass die eingebetteten Komponenteninstanzen ihren internen Zustand zeitgleich mit dem Zustand der übergeordneten Komponente verlassen bzw. betreten. Da die Aktivierung der Transitionen und der Schaltvorgang innerhalb derselben Perioden stattfinden, wird bei der Ausführung des Systems das zeitgleiche Verlassen und Betreten von Zuständen erzwungen. Verlässt zum Beispiel die Komponente Monitor den Zustand NoneAvailable, muss die eingebettete Komponenteninstanz BC den internen Zustand Robust verlassen.

Um das zeitgleiche Verlassen und Betreten von Zuständen auch beim Model Checking zu realisieren, ist eine Synchronisierung der Schaltvorgänge notwendig. Die Synchronisierung kann mit Hilfe von Synchronisationskanälen umgesetzt werden. Die übergeordnete Komponente ist dabei Sender, während die eingebetteten Komponenteninstanzen Empfänger sind. Die Synchronisierung muss genau dann erfolgen, wenn ein Zustand verlassen und betreten wird. Da hybride Rekonfiguration Charts und Interface Statecharts sowie auch Realtime Statecharts diese Möglichkeit nicht bieten, muss eine Transformation der Modelle erfolgen. Im Fall, dass mehrere eingebettete Komponenteninstanzen reagieren müssen, kann eine Kette von *committed locations* (siehe [BDL04]) verwendet werden. In Abbildung 3.15 ist ein Ausschnitt aus dem Mapping dargestellt.

3.5 Evaluierung

In [Kud05] wurden die Funktionalitäten zur Modellierung und die Verifikation der rein syntaktischen Verifikationsverfahren aus [GBSO04] in der Fujaba Realtime Tool Suite[1] implementiert. Der in diesem Kapitel vorgestellte Ansatz wurde anhand des eingeführten Beispiel des Feder-Neige-Moduls evaluiert. Für die Evaluierung wurde der Model Checker UPPAAL [BDL04] verwendet, da dieser bereits in der Fujaba Realtime Tool Suite integriert wurde [Hir04][BGH+05b].

In Abbildung 3.16 und 3.18 sind die transformierten Timed Automata der Interface Statecharts BodyControl und des Rekonfigurations Charts Monitor, wie sie von UPPAAL

[1] http://www.fujaba.de/realtime

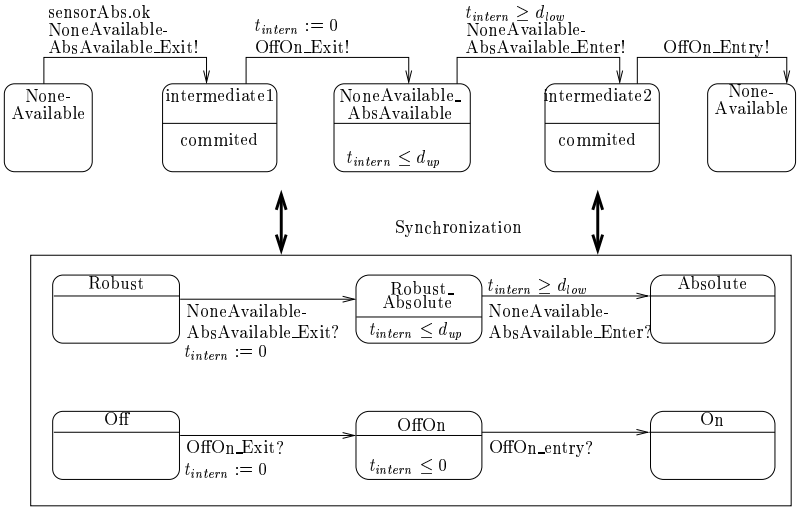


Abbildung 3.15: Synchronisation zwischen Monitor, Sensor und BodyControl

als Eingabe verwendet werden, dargestellt. Der Timed Automaton des Sensors ist in Abbildung 3.17 dargestellt. Für die Verifikation wurde die parallele Ausführung betrachtet und die Eigenschaft A[] not deadlock überprüft Das Ergebnis der Verifikation war, dass

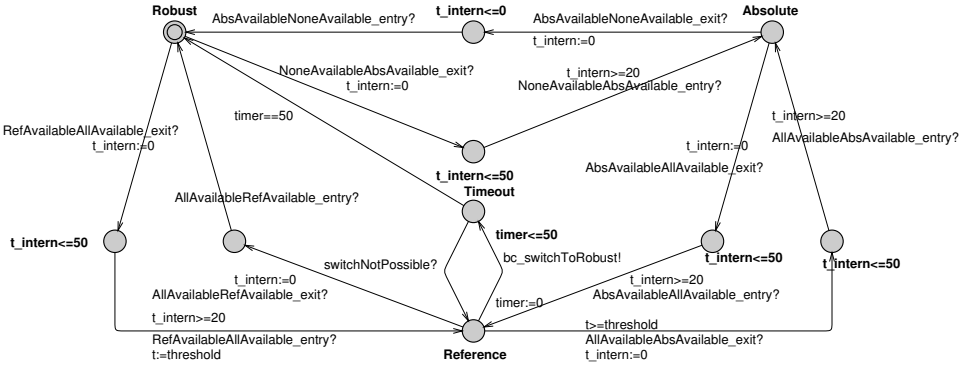


Abbildung 3.16: Timed Automaton des Interface Statecharts der BC Komponente

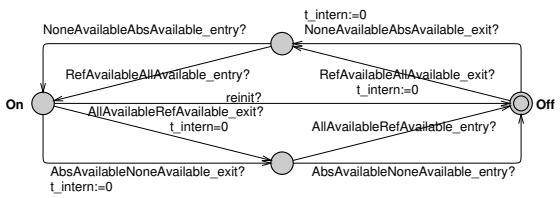


Abbildung 3.17: Timed Automaton des Interface Statecharts der Sensor Komponente

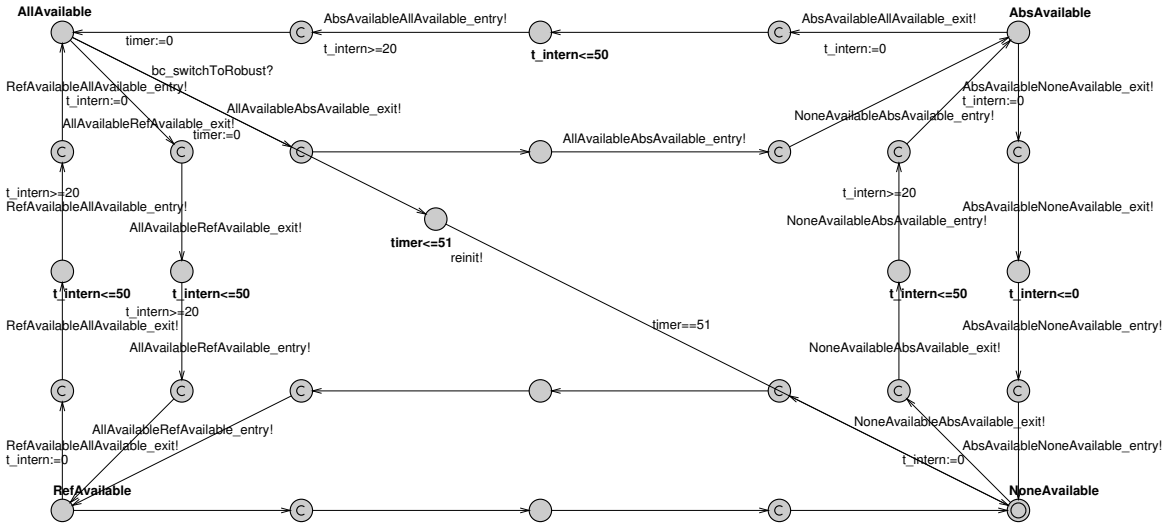


Abbildung 3.18: Timed Automaton des Monitor Verhaltens

das System deadlock frei ist. Daraus ergibt sich, dass die Einbettung aller Komponenten korrekt ist. Die Verifikation hat ca. 0.31 Sekunden bei einem Speicherverbrauch von 2092 KB benötigt.[2].

3.6 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie sich ein OCM verifizieren lässt. Dabei wurde ein bereits vorhandener Ansatz [GBSO04] zur Verifikation der Rekonfiguration von MECHATRONIC UML Modellen erweitert, der durch zahlreiche Einschränkungen in der Ausdrucksstärke für die Interface Statecharts für die komplexen mechatronischen Systeme nicht anwendbar war. Der neue Ansatz erlaubt nun auch die Modellierung und Verifikation von komplexen Zeitbedingungen (nicht lokale Zeitbedingungen) und proaktivem Verhalten. Die Konzepte wurden formal definiert und eine experimentelle Evaluierung hat die Ergebnisse bestätigt.

[2]Wurde auf einem Pentium 4, 2.4 GHz, 1 GB memory, OS Linux Redhat durchgeführt.

Kapitel 4

Verifikation des Verhaltens eines OCM in der Umwelt

Im vorangegangenen Kapitel wurde beschrieben, wie sich ein einzelnes OCM modelliert durch statische Konstrukte hinsichtlich Sicherheitseigenschaften verifizieren lässt. In komplexen, vernetzten mechatronischen Systemen stehen allerdings nur begrenzte Rechen- und Speicherkapazitäten zur Verfügung. Zusätzlich unterliegt das System zur Laufzeit einer Evolution abhängig vom gegebenem Kontext bestimmt durch die Umwelt. Anforderungen an komplexe, mechatronische Systeme sehen deshalb Dynamik vor, d.h. Steuerungssoftware muss zu Laufzeit ausgetauscht werden können. In Schilling [Sch06] wurde bereits beschrieben, wie Graphtransformationssysteme zur Beschreibung von dynamischen Veränderungen im Kontext von mechatronischen Systemen eingesetzt werden können. So wurde das in Abbildung 4.1 dargestellte Szenario auf der Basis von Graphtransformationssystemen beschrieben. Die obere Hälfte des Bildes zeigt einen aktuellen Systemzustand. Hier fahren zwei Shuttles hintereinander auf zwei verschiedenen Streckenabschnitten. Dabei wird die Diskretisierung vorgenommen, dass ein Shuttle immer genau auf einem Streckenabschnitt steht. Die untere Hälfte des Bildes zeigt die koordinierte Bewegung des Shuttles auf den Streckenabschnitten, modelliert durch eine graphbasierte Regel. Hier wird beschrieben, welche Objekte existieren und wie miteinander verbunden sind. Im vorliegenden Beispiel existiert eine Instanz des *DistanceCoordinationPattern*, welches die Verhaltenskoordination zweier verbundener Shuttles realisiert.

4.1 Grenzen des bisherigen Ansatzes

Der von Schilling vorgeschlagene Ansatz [Sch06] zeigt eine Schwäche in der Modellierung und Verifikation auf: es wird keine Zeit berücksichtigt. Daraus ergeben sich die folgenden zwei Probleme:

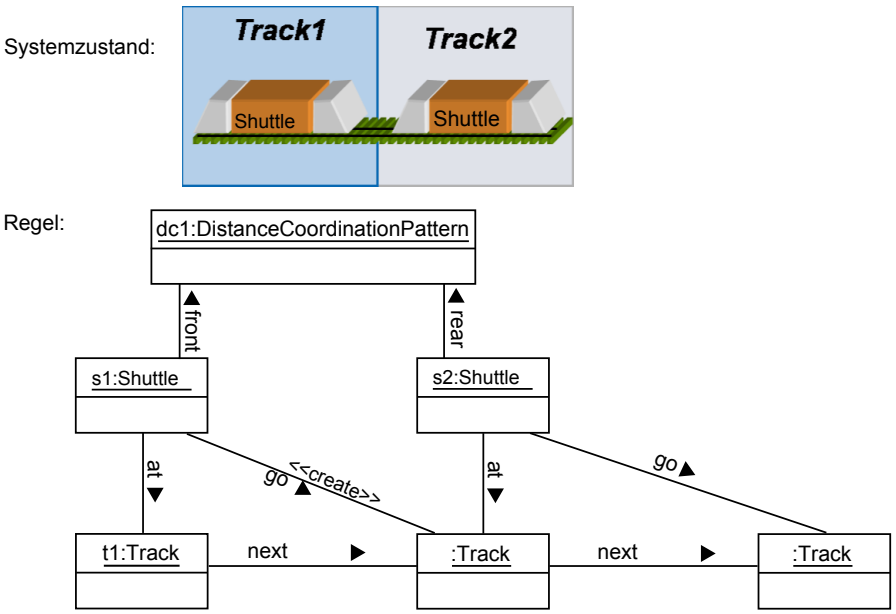


Abbildung 4.1: Beispiel für ein Graphtransformationssystem

Instanziierungsdauer von Echtzeit-Koordinationsmustern: Es ist nicht möglich, die Instanziierungsdauer eines Echtzeit-Koordinationsmusters zu beschreiben. In Abbildung 4.1 ist durch die Regel lediglich beschrieben, dass zwei aufeinander folgende Shuttles das *DistanceCoordinationPattern* instanziieren müssen. Im Detail aber bedeutet die Instanziierung eines Echtzeit-Koordinationsmusters das Aktivieren und Deaktivieren von Softwarekomponenten, welches, ähnlich wie das Umschalten zwischen Reglern, Zeit benötigt. Neben diesem Zeitaspekt muss auch das aktuelle kontinuierliche Verhalten, wie z.B. die Geschwindigkeit, des Shuttles berücksichtigt werden. Es ist offensichtlich, dass ein Shuttle, das $160\frac{km}{h}$ fährt, ebenfalls rechtzeitig die Aktivierung und Deaktivierung von Softwarekomponenten vornehmen können muss wie ein Shuttle, das nur $40\frac{km}{h}$ fährt. D.h. die Abhängigkeit zwischen der Dauer der Instanziierung einer Softwarekomponente zur aktuellen Geschwindigkeit muss ebenfalls berücksichtigt werden.

Beschreibung von kontinuierlichen Bewegungen: Bei der in Abbildung 4.2 dargestellten Situation bewegt sich ein Shuttle über aufeinander folgende Streckenabschnitte. Hierbei entspricht die Größe eines Streckenabschnitts nun nicht mehr genau der Größe eines Shuttles, sondern kann beliebig endlich lang sein, um das Modell realitätsgetreu abzubilden. Falls nun aber mehrere Shuttles hintereinander in die gleiche Richtung fahren, wäre es möglich, dass diese kollidieren, da sich die Position der Shuttles nicht diskret bestimmen lässt.

Um eine derartige Situation im Modell auszuschließen, bestünde eine Möglichkeit darin, dafür zu sorgen, dass alle Shuttles mit der gleichen Geschwindigkeit die Streckenabschnitte passieren. Wenn es im zugrunde liegenden Modell also möglich ist, Aussagen über die Zeit zu formulieren, so kann über den Zusammenhang, dass sich die Geschwindigkeit aus der Strecke pro Zeit ermitteln lässt, hiermit auch die Eigenschaft, dass die Shuttles alle die gleiche Geschwindigkeit fahren, formuliert werden. Hiermit ist im Beispiel eine derartige Kollision ausgeschlossen. Um diese Eigenschaften umzusetzen ist die Idee, die Regeln mit Zeitbedingungen zu versehen sowie Zeitinvarianten für Graphsituationen zu beschreiben, um kontinuierliche Zeitvorgänge zu modellieren. In Abbildung 4.2 ist die entsprechende Regel mit Zeitbedingung (unten links) sowie die Invariante (unten rechts) für die Fortbewegung eines Shuttles modelliert.

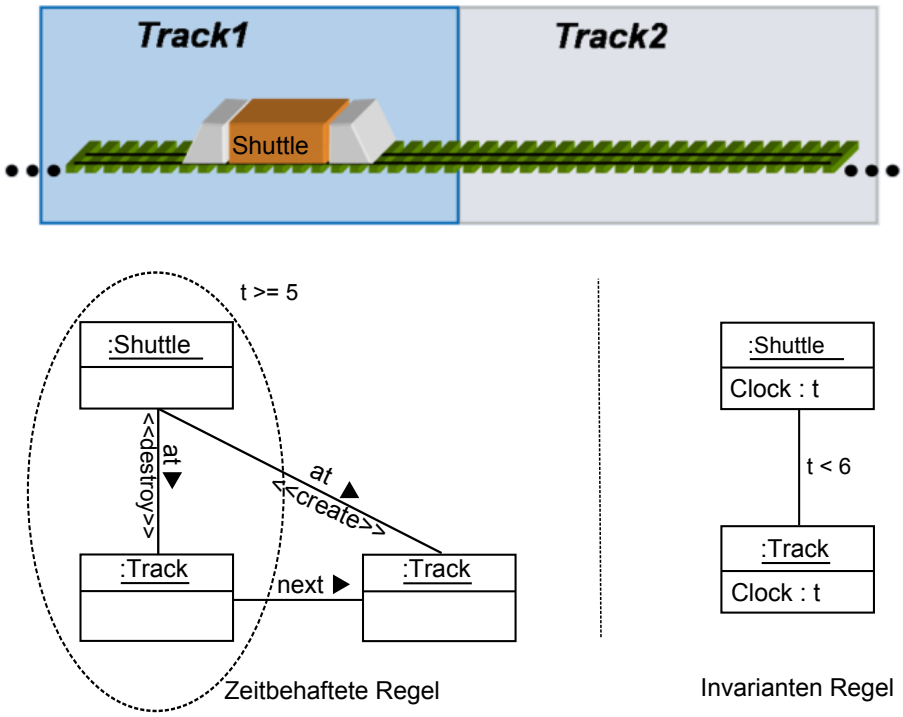


Abbildung 4.2: Beispiel für ein zeitbehaftetes Graphtransformationssystem

Um eine derartige Situation zu modellieren ist es möglich, auch auf andere Modellarten zurück zu greifen. So könnte etwa ein Konvoi durch Timed Automata oder Petrinetze abgebildet werden, bei welchem die Eigenschaft berücksichtigt wird, dass alle Shuttles die gleiche Geschwindigkeit fahren. Allerdings verfügen diese Modelle nicht direkt über eine dynamische Struktur, was es schwierig macht, ein entsprechendes Modell für eine beliebige Anzahl an Shuttles zu erstellen und den zur Laufzeit benötigten Austausch von Softwarekomponenten zu modellieren.

Im Folgenden werden nun zuerst die Erweiterungen hinsichtlich der Modellierung in Abschnitt 4.2 beschrieben und anschließend in Abschnitt 4.3 formalisiert. Die Verifikation des Verhaltens von OCMs in der Umwelt wird in Abschnitt 4.4 beschrieben.

4.2 Modellierung

In diesem Abschnitt wird beschrieben, in welcher Art und Weise Graphtransformationssysteme erweitert werden, um zeitliche Bestandteile in das bestehende Modell zu integrieren. Als erstes wird das Shuttlebeispiel aufgegriffen, anhand dessen die neuen Konzepte durchgängig erklärt werden. Ein Vergleich der Modelle der Graphtransformationssysteme und der Timed Automata zeigt Gemeinsamkeiten und Probleme auf, die sich durch die dynamische Struktur von Graphtransformationssystemen ergeben und darauf, wie diese Eigenschaften Einfluss auf die Modellierung von Zeit haben. Die hier gewonnenen Erkenntnisse gehen in die Erweiterungen für die Definition eines zeitbehaftetes Graphtransformationssystem ein.

4.2.1 Beispiel

Als Beispiel für ein entsprechendes Modell, welches sowohl über eine dynamische Struktur als auch über eine zeitliche Komponente verfügt, dient das in der Einleitung (siehe Abschnitt 1.2) beschriebene Shuttlesystem. Dabei wird betrachtet, wie die Fortbewegung eines Shuttles von Schienenabschnitt zu Schienenabschnitt modelliert werden kann.

Alle diese Komponenten werden im Modell berücksichtigt. Dabei wird die Darstellung dieser Komponenten innerhalb des Beispiels in Form von speziellen UML-Klassen-Diagrammen und UML-Objekt-Diagrammen vorgenommen. Die hierbei verwendete Notation orientiert sich an den von Zündorf [Zün01] vorgestellten Story-Pattern.

Die Bestandteile, z.B. einzelne Shuttle oder die Schienenabschnitte sind in Form von Knoten beschrieben. Verbindungen oder Assoziationen zwischen den Knoten werden durch gerichtete Kanten dargestellt. Eine solche Assoziation existiert z.B., wenn sich ein Shuttle auf einem Schienenabschnitt befindet. Dieses wird durch eine gerichtete Kante zwischen den entsprechenden Knoten abgebildet. Die Abbildung 4.3 zeigt einen Ausschnitt des Schienensystems mit Shuttlen. Zusätzlich zu den Strukturbeschreibungen sollen auch zeitliche Abhängigkeiten und Eigenschaften im Modell berücksichtigt werden. Bei einem realen System ist es wie bereits erwähnt entscheidend, wie lange ein Shuttle zum Überfahren eines Schienenabschnitts benötigt, um hierdurch die aktuelle Geschwindigkeit des Shuttles zu modellieren.

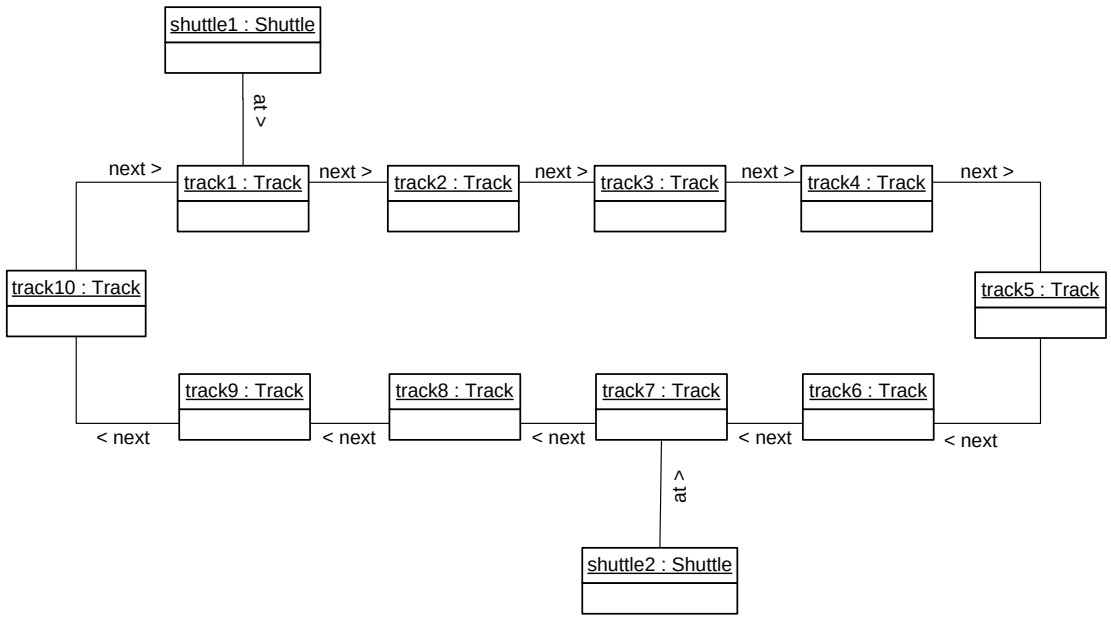


Abbildung 4.3: Das durch einen Graphen beschriebene Shuttlesystem

4.2.2 Zeit und Graphtransformationssysteme

In diesem Abschnitt wird beschrieben, wie Graphtransformationssysteme mit zeitlichen Bedingungen modelliert werden können. Als Referenzmodell für die Beschreibung von Echtzeit-Verhalten wird das Modell der Timed Automata (siehe Abschnitt 2.3.2) verwendet.

Es gibt grundsätzlich verschiedene Möglichkeiten, Zeit innerhalb eines Modells abzubilden, so etwa, wie bei [CGP00] beschrieben, in Form von diskreter oder kontinuierlicher Echtzeit. Die in dieser Arbeit gewählte Form, wie sie bei den Graphtransformationssystemen ergänzt wird, orientiert sich an der Art wie sie auch beim Timed Automaton Verwendung findet. Dort wird kontinuierliches Echtzeitverhalten modelliert. Hierzu werden, wie in Kapitel 2.3.2 beschrieben, einzelne Clocks $x_i, x_j : i, j \in \mathbb{N}^+$ verwendet, die beliebige Werte aus den positiven reellen Zahlen annehmen können. Über diese Clocks werden dann, wie beim Timed Automaton, einzelne Bedingungen in der Form $x_i - x_j \sim c$ formuliert (siehe Definition 3, Abschnitt 2.3.2).

4.2.2.1 Vergleich: Timed Automaton - GTS

Anhand der Überführung eines Graphtransformationssystems in das entsprechende Modell eines Timed Automaton wird hier ein Vergleich beider Modelle beschrieben. Die

hierbei auftretenden Probleme geben Hinweise darauf, wie die einzelnen zeitlichen Bestandteile des Timed Automaton angepasst werden müssen, um diese bei den Graphtransformationssystemen zu berücksichtigen. Ziel ist es, anhand der Ergebnisse dieses Vergleiches eine Lösung zu entwickeln, die es ermöglicht nach dem Vorbild des Timed Automaton zeitliche Bestandteile bei den Graphtransformationssystemen zu ergänzen. Bei diesen zu ergänzenden Bestandteilen handelt es sich um einzelne Clocks, Guards, Clockresets sowie Invarianten.

Bei der Überführung steht dabei nicht im Vordergrund, ein möglichst optimales oder vollständiges Verfahren zu entwickeln, vielmehr werden hierdurch Unterschiede zwischen den beiden Modellen erarbeitet. Das in der Abbildung 4.4 dargestellte System wird nun

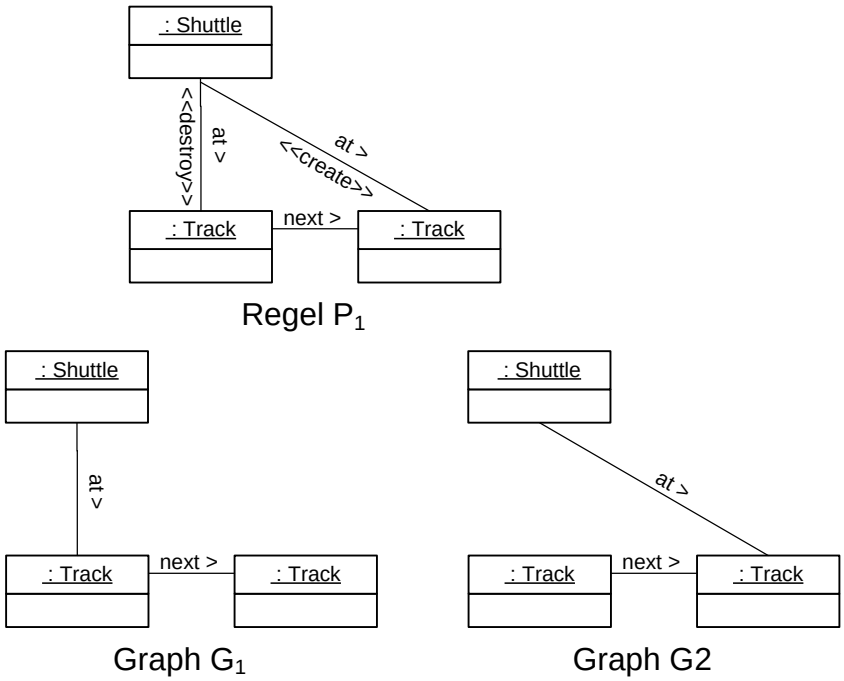


Abbildung 4.4: Ein Beispiel für ein Graphtransformationssystem mit zwei Graphen G_1, G_2 und einer Graphtransformationsregel P_1

in einen entsprechenden Automaten überführt (siehe Abbildung 4.5). Hierfür werden die einzelnen Graphen G_1 und G_2 in zwei entsprechende Zustände s_0 und s_1 eines endlichen Automaten überführt. Die Anwendung der Graphtransformationsregel P_1 wird durch eine Transition p_1 zwischen diesen beiden Zuständen ausgedrückt.

Nachfolgend wird der Algorithmus 4.1 vorgestellt, der eine entsprechende Zuordnungsvorschrift für einen endlichen Automaten M (siehe Kapitel 2.3.1, Abbildung 2.9), mit $M := (\Sigma, \mathcal{Q}, \Delta, \mathcal{Q}^0)$ beschreibt. Als Ausgang dient ein Graphtransformationssystem

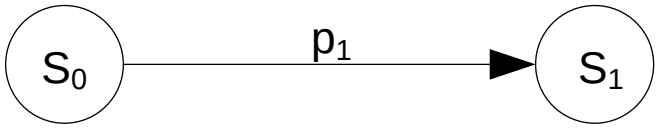


Abbildung 4.5: Ein dem Graphtransformationssystem in Abbildung 4.4 entsprechender Automat

$\mathcal{S} := (\mathcal{G}, \mathcal{G}^0, \mathcal{P})$, mit $\mathcal{G}$ der Menge der Zustände, $\mathcal{G}^0$ der Menge der Startzustände und $\mathcal{P}$ der Menge der Graphtransformationsregeln:

Algorithmus 4.1 procedure $M = transfer(\mathcal{S})$

procedure $M = transfer(\mathcal{S})$

1: $M := (\Sigma, \mathcal{Q}, \Delta, \mathcal{Q}^0)$
2: $\mathcal{Q}, \mathcal{Q}^0, \Sigma, \Delta = \emptyset$
3: **for all** $g_i \in \mathcal{G}$ **do**
4: $\quad f(g_i) = q_i$
5: $\quad \mathcal{Q} = \mathcal{Q} \cup q_i$
6: **end for**
7: **for all** $g_i^0 \in \mathcal{G}^0$ **do**
8: $\quad q^0$
9: $\quad f(g_i^0) = q_i^0$
10: $\quad \mathcal{Q}^0 = \mathcal{Q}^0 \cup q^0$
11: **end for**
12: **for all** $m : m := g_j \times P_i \times g_k$ **do**
13: $\quad \Delta = \Delta \cup (f(g_j) \times P_i \times f(g_k))$
14: **end for**

Σ bleibt bei dieser Abbildung leer, worauf nachfolgend noch eingegangen wird.

An dieser Stelle wird auf zwei der Eigenschaften eingegangen, die aufzeigen, weshalb die Abbildung schwierig, bzw. unvollständig ist. Dabei handelt es sich zum einen um die Kantenbeschriftungen, die bei den Modellen eine unterschiedliche Bedeutung haben, zum anderen um den Zeitpunkt, zu dem eine Abbildung wie oben beschrieben vorgenommen werden kann.

Bei endlichen Automaten hat die Kantenbeschriftungen Σ eine andere Bedeutung als die Transitionen über Graphtransformationsregeln innerhalb von Graphtransformationssystemen. Bei einem Automaten entspricht die Beschriftung einer möglichen Eingabe, bei welcher der Automat seinen Zustand über die entsprechende Kante wechselt. So kann etwa bei den Kanten des Beispielautomaten aus dem Kapitel 2.3.1, von dem Zustand s_0 nach s_1 gewechselt werden, wenn die Eingabe a erfolgt. Bei Graphtransformationssys-

temen bezeichnet diese Beschriftung der Kante die Graphtransformation, welche angewendet wird, um vom Wirtsgraphen zum Tochtergraphen zu gelangen. Dabei ist die Voraussetzung für die entsprechende Transition keine Eingabe sondern eine Bedingung, die im aktuellen Zustand zutreffen muss. Diese Bedingung ist, dass die linke Seite der jeweiligen Graphtransformationsregel im Wirtsgraphen auffindbar sein muss. Der Unterschied liegt hier also darin, dass beim endlichen Automaten die Kanten mit der Eingabe beschriftet werden und bei den Graphtransformationssystemen mit dem Namen der Graphtransformationsregel, über die zwischen den einzelnen Graphen gewechselt wird. Beim Modell des Automaten wird diese Bedingung in Form einer Eingabe vorgegeben, während bei den Graphtransformationssystemen diese Bedingung einer Struktur entspricht, deren Vorkommen bei der Analyse überprüft werden muss.

Ein weiterer Unterschied zwischen den beiden Modellen besteht darin, zu welchem Zeitpunkt die Zustände und Transitionen festgelegt werden. Bei den Automaten geschieht dies bereits bei der Aufstellung des Modells. Im Gegensatz dazu entstehen die Knoten und Kanten des Graphtransformationssystems, mit Ausnahme der initialen Startzustände G^0, nicht bei der Erstellung des Modells. Das erstellte Modell eines Graphtransformationssystems verfügt vor der Erreichbarkeitsanalyse nur über die initialen Graphen und eine Anzahl an Graphtransformationsregeln. Um daraus den vollständigen Zustandsraum des Graphtransformationssystems zu erzeugen, ist es notwendig, eine Erreichbarkeitsanalyse durchzuführen. Vorher wäre es nicht möglich, nach dem oben beschriebenen Vorgehen eine Abbildung des Graphtransformationssystems auf einen Automaten vorzunehmen. Im Gegensatz hierzu ist die gesamte Struktur eines Automaten bereits vollständig vorgegeben, sobald das Modell formuliert ist. Bei der dort durchgeführten Erreichbarkeitsanalyse wird dann die Reihenfolge festgelegt, in der die einzelnen Zustände erreicht werden können; die Struktur des endlichen Automaten wird hierdurch aber nicht verändert.

Die hier aufgezeigte Eigenschaft, dass die Struktur des endlichen Automaten fest vorgegeben ist, lässt sich auf die einzelnen Locations bei dem erweiterten Modell des Timed Automaton übertragen. Die hinzukommenden Bestandteile des Timed Automaton, welche die zeitlichen Eigenschaften in Form von einzelnen Clocks und deren Bedingungen repräsentieren, sind den fest vorgegebenen Strukturen des Modells des Automaten zugeordnet, wie in Abbildung 4.6 dargestellt. Dabei sind den einzelnen Locations die Invarianten und den Kanten, bzw. Transitionen die Guards und Clock-Resets zugeordnet. Die Clocks selbst existieren zu jedem Zeitpunkt und Zustand, in dem sich der Timed-Automaton gerade befindet. Somit sind diese immer dem gesamten Modell des Timed Automaton zugeordnet.

Im Folgenden werden Zuordnungsvorschriften der Clocks, Guards, Resets und Invarianten beschrieben, die dem Modell der Graphtransformationssysteme die Clocks, Guards, Resets und Invarianten zuordnen, ohne bereits alle erreichbaren Zustände zu kennen. Dabei wird zuerst grundlegend behandelt, wie eine Zuordnung von einzelnen zeitlichen Bestandteilen, wie diese beim Timed Automaton vorhanden sind, im Fall der Graphtransfor-

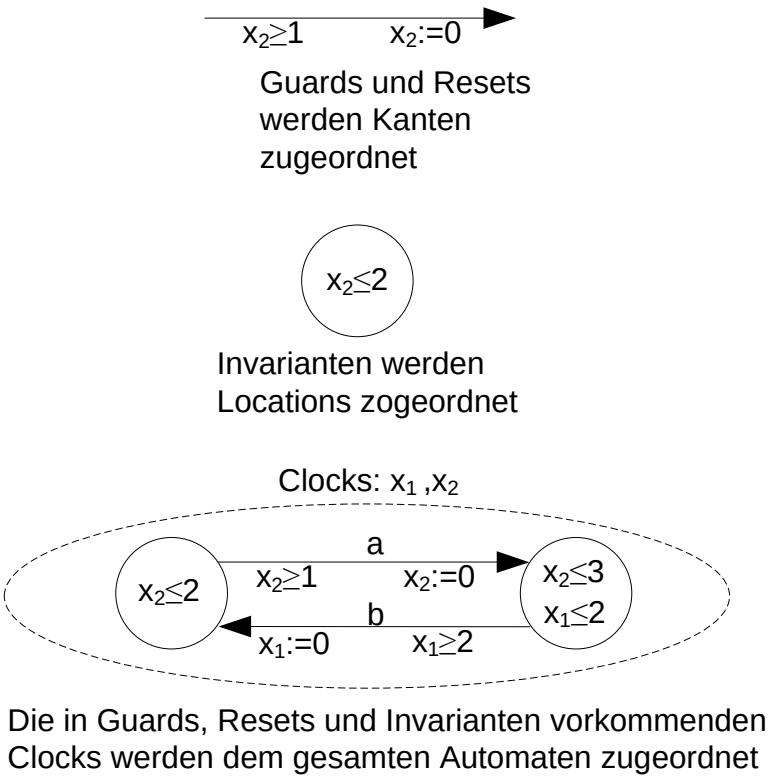


Abbildung 4.6: Zuordnung der zeitlichen Bestandteile zu den festen Strukturen des Timed Automaton.

mationssysteme vorgenommen wird. Hierbei gibt es verschiedene Aspekte, die betrachtet werden müssen, um eine solche Zuordnung zu ermöglichen. Allerdings ist neben der Frage, wie dies im Modell technisch möglich ist, auch entscheidend für welchen Zweck entsprechende Eigenschaften formulierbar sind.

Es wird hier mit einer bestimmten Eigenschaft von Graphtransformationssystemen gearbeitet, nämlich, dass es bei diesen hauptsächlich um Strukturen geht, welche innerhalb der initialen Graphen und den daraus resultierenden Folgegraphen vorhanden sind. Nach diesen Strukturen wird eben durch die Graphtransformationsregeln gesucht. Diese Strukturen beschreiben Situationen und Zustände, bzw. einen Teil dieser, welche innerhalb des aktuellen Graphen gelten.

Mit diesen Strukturen werden einzelne zeitliche Eigenschaften verbunden. Dies bedeutet, dass Graphtransformationsregeln dahingehend erweitert werden, bzw. neue Regeln hinzukommen. Diese sollten in der Lage sein, durch ihre Anwendung zeitliche Eigenschaften zu einem Graphtransformationssystem hinzuzufügen. Zunächst wird hier an einem Beispiel darauf eingegangen, warum diese Lösung gewählt wurde und welche Eigenschaften hierdurch modellierbar sind.

Als Eigenschaft soll formuliert werden, dass ein Shuttle mindestens 10 Zeiteinheiten zum Überfahren eines Schienenabschnittes benötigt. Abbildung 4.7 zeigt diese Regel. Entscheidend für die Information, wie lange das Shuttle auf dem Schienenabschnitt sein soll, ist die mit einem Kreis markierte Struktur im Wirtsgraphen.

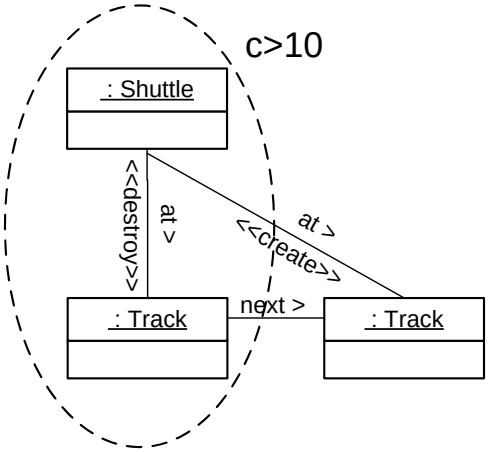


Abbildung 4.7: Zuordnung einer Clock c zu einer Graphtransformationsregel

Um allerdings eine solche Bedingung formulieren zu können ist es notwendig über zugehörige Clocks zu verfügen. Im Beispiel wird die Clock c benötigt, um über diese die Bedingung $c > 10$ formulieren zu können.

4.2.2.2 Clockinstanzen

Die benötigten Clocks stellen im Vergleich zum Timed Automaton eine Besonderheit dar, denn bei Graphtransformationssystemen ergibt sich die Struktur erst bei der Erreichbarkeitsanalyse. Ohne eine bereits vorgegebene Struktur ist es nicht möglich, alle Clocks festzulegen, die innerhalb des Graphtransformationssystems benötigt werden. Warum dies der Fall ist, wird nachfolgend an einem Beispiel illustriert. Dabei handelt es sich um den Wirtsgraphen G in Abbildung 4.8, bei welchem die dort abgebildete Regel an den zwei markierten Stellen angewendet werden kann. Da es möglich ist, dass eine der beiden Stellen im Wirtsgraphen bereits länger existiert als die andere, werden für die Berücksichtigung der zeitlichen Bedingungen $c \geq 3$ auch entsprechend zwei einzelne Clocks benötigt.

Um mehrere Clocks innerhalb eines Wirtsgraphen zu ermöglichen, werden von einer Clock mehrere Instanzen erzeugt. Dabei wird der Name der jeweiligen Clock um die IDs der einzelnen Knoten des Wirtsgraphen erweitert, auf welchen die Regel angewendet wird. Entsprechend werden in dem Beispiel aus Abbildung 4.8 zwei Clocks c erzeugt, die

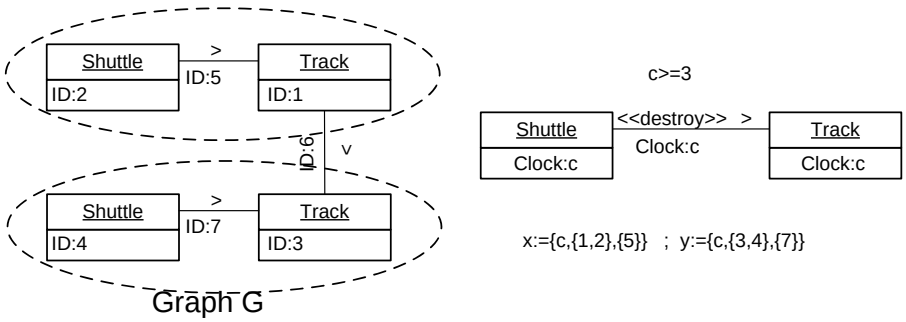


Abbildung 4.8: Der Graph G verfügt über zwei Stellen, bei denen die Graphtransformationsregel mit der zeitlichen Bedingung $c \geq 3$ angewendet werden kann.

sich hinsichtlich der IDs der Knoten des Wirtsgraphen unterscheiden. Dem Graphen des Beispiels werden die beiden Clocks x und y zugeordnet, wobei diese über einen Namen identifiziert werden, sowie über einen Schlüssel, der sich aus den IDs des Wirtsgraphen zusammen setzt. Um die Zuordnung von einzelnen Clocks mit gleichem Namen und unterschiedlichen IDs eindeutig zu machen, werden zusätzlich die Kanten des Wirtsgraphen bei dem erzeugten Schlüssel mit berücksichtigt:

$$x := c[1, 5, 2], y := c[3, 7, 4]$$

Die so erzeugten Clocks werden nachfolgend als Clockinstanzen bezeichnet, um eine Differenzierung zu den üblichen Clocks zu erlauben.

Weiterhin soll es möglich sein, nur einen Teil der linken Seite einer Anwendungsregel mit einer zugehörigen Clockinstanz zu verbinden. So etwa, wie es im Beispiel der Abbildung 4.7 dargestellt ist. Dort wird die Clockinstanz c, über welche eine Bedingung formuliert ist, nur mit zwei der drei Knoten sowie einer der drei Kanten der Regel assoziiert. Um dies zu ermöglichen werden die Anwendungsregeln erweitert, so dass den einzelnen Elementen ein entsprechendes Attribut hinzugefügt werden kann, welches angibt, ob ein Knoten oder eine Kante zu einer Clockinstanz zugehörig ist. Dabei wird wie in Abbildung 4.8 an jedes Element der Anwendungsregel geschrieben, mit welchen Clockinstanz-Namen das Element verbunden ist (gelb markiert). In Abbildung 4.8 ist die gesamte linke Seite mit der Clockinstanz c verbunden.

Aus diesen erweiterten Anwendungsregeln werden dann für die Clockinstanzen weitere Regeln abgeleitet. Diese sorgen dafür, dass den einzelnen Graphen des Graphtransformationssystems die entsprechenden Clockinstanzen hinzugefügt werden. Für die Regel aus dem Beispiel der Abbildung 4.7, zeigt die Darstellung 4.9 auf der rechten Seite die abgeleitete Clockinstanzregel.

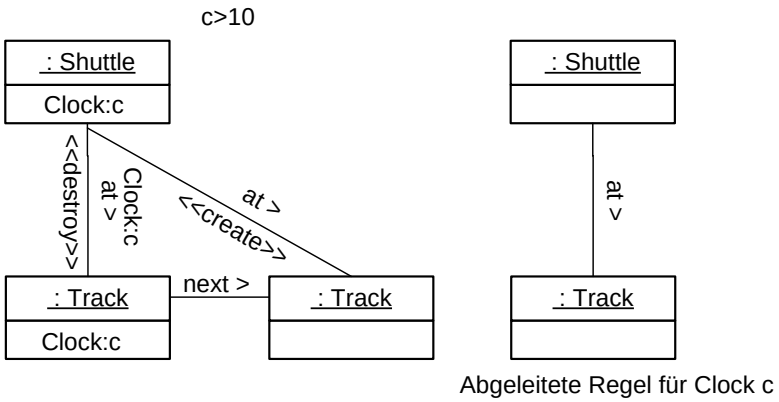


Abbildung 4.9: Die Abbildung zeigt auf der linken Seite eine Anwendungsregel mit den attributierten Elementen, welche der Clock c zugehörig sind. Rechts ist die daraus abgeleitete Clockinstanzregel dargestellt.

Mit Hilfe dieser abgeleiteten Regeln werden die einzelnen Clockinstanzen einem Wirtsgraphen hinzugefügt. Dabei muss noch berücksichtigt werden, dass die Ausführung der Clockinstanzregel vor den zugehörigen Anwendungsregeln, aus welchen diese abgeleitet wurden, geschehen muss. Dies ist der Fall, da es für einzelne Clockinstanzen, welche durch eine Clockinstanzregel erzeugt werden, entscheidend ist, seit wann diese existieren. Um sicher zu stellen, dass Clockinstanzregeln vor den Anwendungsregeln ausgeführt werden, wird den Graphtransformationsregeln eine Priorität hinzugefügt. Dabei erhalten die Clockinstanzregeln eine höhere Priorität als alle anderen Arten von Regeln. Der Mechanismus von priorisierten Anwendungsregeln wurde bereits in Definition 10 beschrieben und kann von den bestehenden Modellen der Graphtransformationssysteme direkt übernommen werden.

4.2.2.3 Clockresets

Eine weiterer Bestandteil, der von dem Modell des Timed Automaton übernommen wird, sind die so genannten Clockresets. Diese werden bei den Graphtransformationssystemen ebenfalls mit der Anwendungsregel verknüpft, wobei angegeben wird, welche Clockinstanzen durch die Anwendung der erweiterten Regel auf den Wert 0 zurück gesetzt werden. Das Vorgehen orientiert sich dabei an der Art und Weise, wie Guards mit Hilfe der Graphtransformationsregeln hinzugefügt werden. Ein Beispiel zeigt die Abbildung 4.10. Dort ist die Clockinstanz c mit der Graphtransformationsregel verknüpft und wird durch einen entsprechenden Reset bei Anwendung auf den Wert 0 zurück gesetzt. Dabei werden für die verwendete Clockinstanz innerhalb des Resets nach dem gleichen Prinzip wie für die zeitlichen Bedingungen einzelne Clockinstanzregeln abgeleitet.

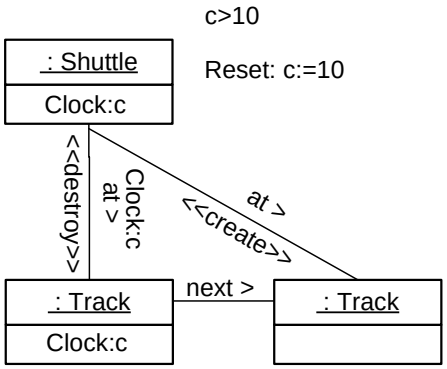


Abbildung 4.10: Eine um Clockreset erweiterte Graphtransformationsregel. Dabei wird die Clock c nach Anwendung der Regel auf den Wert 0 zurück gesetzt.

4.2.2.4 Invarianten

Hier wird beschrieben, wie die Invarianten des Timed Automaton bei dem Modell des Graphtransformationssystems übernommen werden. Über diese werden beim Timed-Automaton zeitliche Bedingungen zu einer Location hinzugefügt. Hier wird ein Ansatz vorgestellt, der es ermöglicht, diese Invarianten auf die einzelnen Graphen eines Graphtransformationssystems zu übertragen. Da vor einer Erreichbarkeitsanalyse des Graphtransformationssystems nicht klar ist, welche Folge-Graphen erreichbar sind, ist eine Zuordnung von Invarianten direkt zu einzelnen Graphen nicht sinnvoll möglich. Um aber eine Umsetzung von Invarianten zu ermöglichen, wird hierfür wiederum auf Strukturen zurück gegriffen, nach denen innerhalb eines Graphen durch eine Graphtransformationsregel gesucht wird. Es werden Anwendungsregeln eingeführt, die nur dem Zweck dienen, einzelne zeitliche Bedingungen zu einzelnen Graphen hinzuzufügen. Hierzu werden so genannte Invariantenregeln eingeführt. Diese Regeln verändern die Struktur des Wirtsgraphen, auf den sie angewendet werden, nicht, genauso wie die Clockinstanzregeln. Es wird lediglich ein Matching mit der linken Seite der Anwendungsregel durchgeführt, d.h. die Regel verfügt über keine rechte Anwendungsseite. Falls eine Invariantenregel anwendbar ist, wird die mit dieser verbundene zeitliche Bedingung dem Wirtsgraphen hinzugefügt. Eine mögliche Eigenschaft, die mit Hilfe einer solchen Invariantenregel formuliert wird, wäre, dass die Verweildauer eines Shuttles auf einem Schienenabschnitt max. 11 Zeiteinheiten dauern darf (siehe Abbildung 4.11)

Ähnlich wie die Anwendungsregeln mit zeitlicher Bedingung benötigen die Invariantenregeln ebenfalls zugehörige Clockinstanzen. Beim Beispiel der Abbildung 4.11 ist dies die Clock d, von der es innerhalb eines Wirtsgraphen mehrere Instanzen geben kann. Dies ist der Fall, da es wie bei den normalen Anwendungsregeln mehrere Stellen innerhalb des Wirtsgraphen geben kann, an denen die Invariantenregeln anwendbar sind.

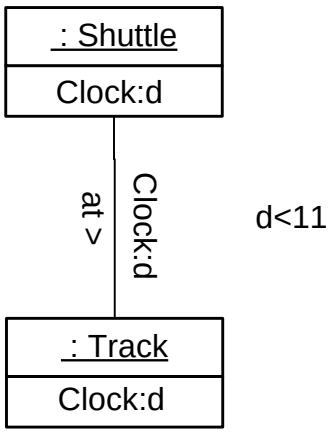


Abbildung 4.11: Die Regel fügt einem Graphen die Invariante $d < 11$ hinzu.

Das Vorgehen bei den Invarianten ist äquivalent zu dem der zeitbehafteten Graphtransformationsregeln. Bei den Invarianten werden ebenfalls einzelne Elemente der linken Seite der Regel mit den zugehörigen Clockinstanzen der Ungleichungen verknüpft. In der Abbildung 4.11 sind alle Elemente mit der Clockinstanz d verbunden.

4.2.2.5 Zeitbehafteter Graph

Bei den hier betrachteten Erweiterungen spielen bei den Graphen des Graphtransformationssystems auch die zeitlichen Erreichbarkeitsräume eine Rolle. Dabei werden einzelne Graphen mit Bedingungen über einzelne Clockinstanzen versehen. Ein Beispiel hierfür ist die Ungleichung der Form $d < 11$, die über die Invariantenregel in Abbildung 4.11 den einzelnen Graphen hinzugefügt wird. Somit muss das hier verwendete Modell eines Graphen erweitert werden, um diesen Graphen einzelne zeitliche Erreichbarkeitsräume hinzuzufügen.

Dabei ist der Fall zu berücksichtigen, dass ein einzelner Graph nicht nur über einen, sondern auch über mehrere, eventuell disjunkte, zeitliche Erreichbarkeitsräume verfügen kann. Dies ist der Fall, wenn innerhalb eines solchen erweiterten Graphtransformationssystems ein Zyklus aus Transitionen zwischen den einzelnen Graphen existiert und durch den Clockreset einer Graphtransformationsregel zu einem bereits existierenden Graphen ein weiterer zeitlicher Erreichbarkeitsraum hinzugefügt wird. In Abbildung 4.12 ist ein Graphtransformationssystem dargestellt, bei dem zwei einzelne Graphen über jeweils zwei unterschiedliche Erreichbarkeitsräume verfügen.

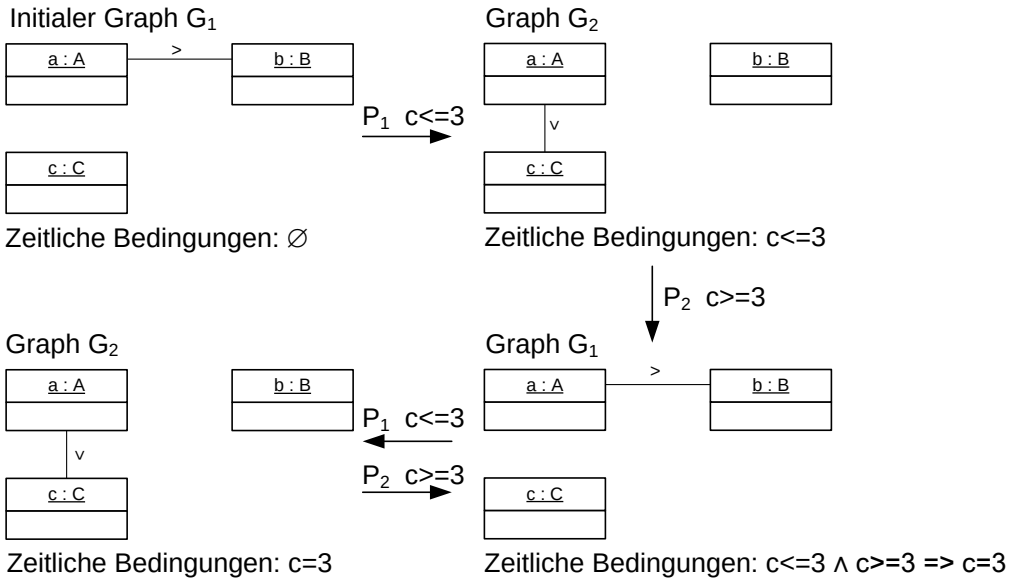


Abbildung 4.12: Die beiden Graphen G_1 und G_2, welche über jeweils zwei unterschiedliche zeitliche Erreichbarkeitsräume verfügen.

4.3 Semantik

Nachfolgend wird die Semantik der gerade vorgestellten Konzepte beschrieben. Dabei wird das Modell der Graphtransformationssysteme aus dem Grundlagenkapitel 2.3.4 erweitert. Der Abschnitt schließt mit der Definition eines zeitbehafteten Graphen und eines zeitbehafteten Graphtransformationssystems.

4.3.1 Clockinstanzen

Um Zeit in Graphtransformationssystemen berücksichtigen zu können, müssen zeitliche Bedingungen und die dort verwendeten Clockinstanzen definiert werden. Die bei den erweiterten Graphtransformationssystemen verwendeten zeitlichen Bedingungen setzen sich aus Ungleichungen der Form $x_i - x_j \sim d$ zusammen. x_i, x_j sind Clockinstanzen, wobei $x_i, x_j \in \mathbb{R}^+$, sowie $\sim \in \{<, \leq\}$ und $d \in \mathbb{Z}$. Jede Clockinstanz ist immer mit mindestens einem Graphen G verbunden. Dies bedeutet, dass eine Clockinstanz einen Namen M und eine Menge an Knoten und Kanten aus G zugeordnet hat. Diese Kanten und Knoten müssen dabei anhand einer ID eindeutig in G identifizierbar sein. Die Identität einer solchen Clockinstanz ergibt sich also aus einem Namen und den IDs der zugeordneten Elemente aus G. Ein Beispiel für einen Graphen sowie den zugehörigen Clockinstanzen ist in Abbildung 4.13 dargestellt.

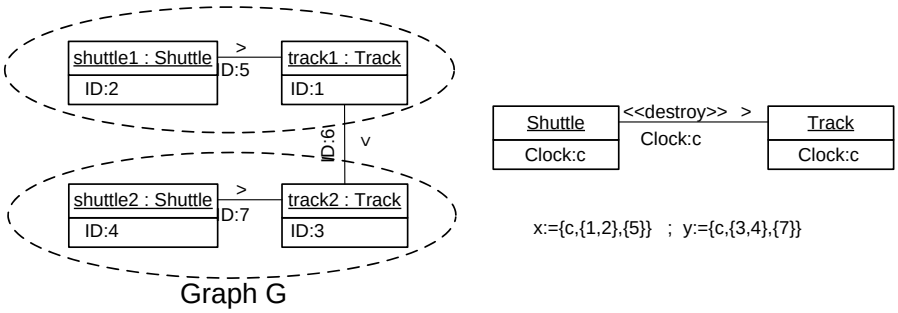


Abbildung 4.13: Zum Wirtsgraphen G werden zwei Instanzen x, y der Clockinstanz c erzeugt. Dabei ergibt sich $x := (c, \{1, 2\}, \{5\})$ und $y := (c, \{3, 4\}, \{7\})$.

Definition 21
Eine Graph-Clockinstanz ist eine Clock $x := (M, \mathcal{N}, \mathcal{E})$, wobei M der Name der Clock ist, $\mathcal{N}$ die Menge der Knoten-IDs und $\mathcal{E}$ die Menge der Kanten-IDs darstellt. x kann dabei, über die Zuweisungsfunktion $V(x)$, ein Wert aus den reellen positiven Zahlen zugewiesen werden.

Nachfolgend wird in dieser Arbeit die Zuweisungsfunktion $V(x)$ weg gelassen, um die Notation abzukürzen und die Clockinstanzen bezüglich der Notation, wie die Clocks der Clockzones aus Kapitel 2.3.5.3, behandeln zu können. Ähnlich wie bei den Clockzones existiert auch hier eine Referenzclock x_0, die keinem Knoten und keiner Kante zugeordnet ist. Die hier definierten Clockinstanzen werden bei den zeitlichen Bedingungen zur Darstellung der Erreichbarkeitsräume verwendet. Innerhalb der Ungleichungen werden die entsprechenden Clockinstanzen eingesetzt.

Definition 22
Eine zeitliche Bedingung $t := x_i - x_j \sim d$ mit Clockinstanzen setzt sich zusammen aus x_i, x_j, wobei entweder x_i oder x_j der Referenz-Clock x_0 entsprechen kann. Falls x_i, bzw. $x_j \neq x_0$, so ist x_i, bzw. x_j eine Clockinstanz wie in Definition 21 beschrieben. Weiterhin gilt $\sim \in (<, \leq)$, $i, j \in \{\mathbb{N}^+ \setminus 0\}$ und $d \in \mathbb{Z}$.

Aus einer Menge dieser zeitlichen Bedingungen wird ein zeitlicher Erreichbarkeitsraum aufgebaut. Hierzu wird eine Anzahl an t_i von zeitlichen Bedingungen mit Clockinstanzen zu einer Menge T zusammen gefasst. Der eigentliche zeitliche Erreichbarkeitsraum ergibt sich, indem ähnlich wie bei den in Kapitel 2.3.5.3 beschriebenen Clockzones die Konjunktion über alle $t_i \in T$ gebildet wird.

Definition 23
Ein zeitlicher Erreichbarkeitsraum $T := (t_l, .., t_n)$ mit $l, n \in \mathbb{N}^+$ besteht aus einer Menge an einzelnen zeitlichen Bedingungen t_i, wie diese in Definition 22 definiert sind. Dabei ist es möglich das $T := \emptyset$ ist.

4.3.2 Zeitbehaftete Anwendungsregeln

In diesem Abschnitt werden die neuen zeitbehafteten Anwendungsregeln definiert.

Definition 24
Eine Graphtransformationsregel $P_t : (P_l, P_r, h, T, V_i, E_i)$ mit zeitlichen Bedingungen besteht aus einem linken Anwendungsteil P_l und einem rechten Anwendungsteil P_r. Zusätzlich zu einer Graphtransformationsregel, wie in Definition 9 beschrieben, verfügt P_t über eine Menge an zeitlichen Bedingungen T mit Clockinstanzen, wie in Definition 22 definiert. Weiterhin existieren die Zuordnungsfunktionen V_i und E_i, welche den Elementen der linken Seite P_l von P_t IDs zuordnen. V_i ordnet allen Knoten von $P_l \setminus d(P_t)$ ein innerhalb von P_t eindeutiges n_k zu und E_i ordnet allen Kanten in $P_l \setminus d(P_t)$ ein eindeutiges e_k zu. Dabei gilt $n_k, e_k \in \mathbb{N}^+$. Für jede Clockinstanz $x_j \in X$, mit $X := (x_1, .., x_n)$ und $x_j := (M, \mathcal{N}, \mathcal{E})$, die in einem $t \in T$ verwendet wird (siehe Definition 22), ergeben sich $\mathcal{N}$ und $\mathcal{E}$ wie folgt: $\mathcal{N}$ werden alle Elemente aus V_i und $\mathcal{E}$ alle Elemente aus E_i zugewiesen.

In der obigen Definition sind alle Clockinstanzen immer mit dem gesamten Teil der linken Seite $P_l \setminus d(P_t)$ einer Anwendungsregel verbunden. Diese wird nun im Folgenden, angelehnt an Abschnitt 4.2.2.2, derart erweitert, dass es auch möglich ist, Clockinstanzen nur mit einem Teil einer Anwendungsregel zu verknüpfen.

Definition 25
Eine Graphtransformationsregel $P_t := (P_l, P_r, h, T, V_i, E_i, f)$ mit zeitlichen Bedingungen und einer Anzahl an zugeordneten Clock-Elementen verfügt zusätzlich zu Definition 24 über eine Funktion f, welche den in T vorkommenden Clockinstanzen die Menge v und e zuordnet, wobei $v \subseteq V_i$ und $e \subseteq E_i$.

4.3.2.1 Resets

Bei den bis hier vorgestellten erweiterten Graphtransformationsregeln fehlen noch die Resets, bei deren Anwendung einzelne Clockinstanzen zurück gesetzt werden.

Definition 26
Ein Clockinstanz-Reset $r := (x_r)$ besteht aus der zugehörigen Clockinstanz x_r, welche bei Anwendung von r auf den Wert 0 zurück gesetzt wird.

Die Graphtransformationsregel aus Definition 25 wird entsprechend um die Clockinstanz-Resets erweitert.

Definition 27
Eine Graphtransformationsregel $P_t := (P_l, P_r, h, T, V_i, E_i, R, f)$, welche um Clockinstanz-Resets erweitert ist, besteht zusätzlich aus der Menge R. Für alle $r \in R$ gilt, dass

r ein Clockinstanz-Reset ist. Für die Zuordnungsfunktion f gilt zusätzlich, dass diese den Clockinstanzen x_k, welche in $r \in R$ vorkommen, ebenfalls einzelne Elemente aus V_i und E_i zuordnet.

Dabei gilt der bei einer Anwendung ausgeführte Reset als eine Nachbedingung, welche mit dem rechten Teil P_r der Graphtransformationsregel verbunden wird.

4.3.2.2 Invarianten

Neben den Clockinstanz-Regeln gibt es eine weitere Art von Graphtransformationsregeln, nämlich die in Abschnitt 4.2.2.4 beschriebenen Invarianten.

Definition 28
Eine Invarianten-Graphtransformationsregel $I_t := (I_l, h, t, V_i, E_i, f)$ besteht aus einer linken Seite I_l, welche die Vorbedingung darstellt, einem Graphmorphismus h sowie aus einer zeitlichen Bediungung $t := (x_1 - x_2 \sim d)$ mit Clockinstanzen, wie bei Definition 22 definiert. Falls x_1, bzw. x_2 nicht der Referenz-Clock x_0 entspricht, so wird den Mengen $\mathcal{N}$ und $\mathcal{E}$ (der Clockinstanzen x_1 und x_2) das Ergebnis der Funktion $f(n)$ zugewiesen, wobei gilt, n stammt aus den Knoten und Kanten von I_l.

Da der Graphmorphismus h I_l immer auf I_l selbst abbildet, werden bei der Anwendung entsprechend keine Kanten oder Knoten entfernt oder hinzugefügt. Eine so definierte Invariantenregel verfügt über keine rechte Seite. Bei Anwendung dieser Regel wird den zeitlichen Erreichbarkeitsräumen eines Graphen als Nachbedingung die Ungleichung t hinzugefügt.

4.3.2.3 Ableitung von Clockinstanzregeln

Um die Clockinstanzen zu einem Graphen G hinzuzufügen, werden aus den erweiterten Graphtransformationsregeln aus Definition 27 weitere Regeln abgeleitet. Dabei werden für jede Clockinstanz x_k, die in den Bedingungen T einer zeitbehafteten Graphtransformationsregel, in den Clockinstanz-Resets R oder in der jeweiligen Bedingung t einer Invariantenregel vorkommt, einzelne Graphtransformationsregeln C_t abgeleitet. Diese werden nachfolgend als Clockinstanzregeln bezeichnet und ergeben sich aus der linken Seite P_l der Graphtransformationsregel P_t, bzw. aus der linken Seite I_l der Invariantenregel. Hierbei werden für eine Clockinstanz x_k, die durch die Funktion f aus P_t oder I_t über die IDs zugeordneten Knoten und Kanten verwendet, um die linke Seite der Clockinstanzregel C_t herzuleiten. Die einzige Nachbedingung von C_t ist, dass dem Graphen G eine Clockinstanz x'_k hinzugefügt wird, falls diese innerhalb von G noch nicht existiert. Dabei werden x'_k nicht die Knoten und Kanten IDs zugeordnet, welche die Funktion f

liefert, sondern die Knoten- und Kanten-IDs, welche innerhalb des Wirtsgraphen G beim Anwenden der Clockinstanz-Regel im Wirtsgraphen aufgefunden werden (siehe Abbildung 4.13).

Definition 29
Eine Clockinstanzregel $C_t := (M, C_l, h, C_i)$, welche auf einen Wirtsgraphen G angewendet wird, besteht aus dem Namen M der Clockinstanz, sowie der linken Seite C_l der Graphtransformationsregel. C_l entspricht dabei einer linken Seite P_l einer Graphtransformationsregel, wie in Definition 9 angegeben, mit dem Unterschied, dass die Funktion $d(C_t)$ des Graphmorphismus h die leere Menge $\emptyset$ liefert. C_t verfügt über eine Nachbedingung $C_i := (M, \mathcal{N}, \mathcal{E})$, wobei C_i eine Clockinstanz ist und $\mathcal{N}$ eine Teilmenge der Kanten-IDs $V_i(v)$ mit $v \in V$ und V aus C_l und $\mathcal{E} \subseteq E_i(e)$ mit $e \in E$ und E aus C_l.

Im Folgenden wird darauf eingegangen, wie aus einer Graphtransformationsregel oder Invariantenregel mit zeitlicher Bedingung die einzelnen Clockinstanzregeln hergeleitet werden. Das hier beschriebene Vorgehen findet nach dem Erstellen eines entsprechenden initialen Graphtransformationssystems und vor der Durchführung einer Erreichbarkeitsanalyse statt. Auf das entsprechende Verfahren, um die Clockinstanzregeln auf einen Wirtsgraphen anzuwenden, wird im nächsten Abschnitt eingegangen.

Eine entsprechende Graphtransformationsregel $P_t := (P_l, P_r, h, T, V_i, E_i, R, f)$ verfügt über eine Menge an zeitlichen Bedingungen T und Clockinstanz-Resets R. Für alle $t \in T$ gilt $t := (x_1 - x_2 \sim d)$ und für alle $r \in R$ gilt $r := (x_r)$. $X_{t,r}$ ist die Menge, welche sich aus der Vereinigung der Clockinstanzen x_1, x_2 und aus allen $t \in T$ sowie allen x_r, die in den $r \in R$ vorhanden sind, ergibt. Für jedes $x := (M, \mathcal{N}, \mathcal{E})$ mit $x \in X_{t,r}$ wird dann eine Clockinstanz-Regel $c_j := (M_j, C_{l,j}, h, C_{i,j})$ mit $j \in \mathbb{N}^+$ nach dem im Algorithmus 4.2 beschriebenen Schema hergeleitet, wobei $C_{l,j} := (V_c, E_c)$. Die Funktion f der Invariantenregel $I_t := (I_l, h, t, E_i, V_i, f)$ kann mit Graph I_l anstelle des Graphen P_l aus P_t verwendet werden, um die Clockinstanzregeln der Invarianten analog herzuleiten.

Die bei der Nachbedingung $C_{i,j} := (M_j, \mathcal{N}_j, \mathcal{E}_j)$ vorhandenen Mengen $\mathcal{N}_j$ und $\mathcal{E}_j$, welche die Knoten- und Kanten-IDs der durch die Regel erzeugten Clockinstanz darstellen, werden erst bei der Anwendung auf einen Wirtsgraphen G mit den entsprechenden Elementen gefüllt. Somit sind diese bei der Erstellung der Clockinstanz-Regel immer leer.

4.3.3 Zeitbehafteter Graph & Graphtransformationssystem

4.3.3.1 Zeitbehafteter Graph

Um ein erweitertes Graphtransformationssystem zu analysieren, muss ein zeitbehafteter Graph definiert werden. Basierend auf diesem können dann Nachfolger- und Vorgängerzustände hergeleitet werden. Ein gerichteter und benannter Graph G wird um die ent-

Algorithmus 4.2 Schema zur Herleitung einer Clockinstanz-Regel $c_j := (M_j, C_{l,j}, h, C_{i,j})$, $j \in \mathbb{N}^+$

```
1: M_j = M
2: for all n ∈ P_l do
3:     if f(n) ∈ 𝒩 then
4:         V_c = V_c ∪ n
5:     end if
6: end for
7: for all e ∈ P_l do
8:     if f(e) ∈ ℰ then
9:         E_c = E_c ∪ e
10:    end if
11: end for
12: C_{i,j} = (M, ∅, ∅)
```

sprechenden Bestandteile erweitert, so dass zeitliche Erreichbarkeitsräume mit diesem verknüpft werden. Hierzu werden zu G Ungleichungen wie in Definition 22 sowie die dort vorhandenen Clockinstanzen hinzugefügt.

Definition 30
Ein zeitbehafteter Graph $G_t := (G, C, T)$ ist ein Tripel mit einem gerichteten Graph G, einer Anzahl an Clockinstanzen C und einer Menge an zeitlichen Bedingungen T über einzelnen Elementen aus C. Jeder Knoten v aus G besitzt eine eindeutige ID n_i und jede Kante $e \in E$ eine entsprechende ID e_j mit $i, j \in \mathbb{N}^+$.

Mit Hilfe der oben stehenden Definition 30 wird einem einzelnen Graphen g aus der Menge aller Graphen G eines Graphtransformationssystems, wie in Definition 11 beschrieben, ein zeitlicher Erreichbarkeitsraum zugewiesen und dieser gleichzeitig über die verwendeten IDs der Clockinstanzen mit g verknüpft.

Bei dem im weiteren Verlauf verwendeten Modell ist es möglich, dass ein Graph G_t mehrere zeitliche Erreichbarkeitsräume haben kann. Ein Beispiel hierfür ist das Graphtransformationssystem, welches in Abbildung 4.14 aufgebaut wird. Dort gibt es einen initialen Graphen G_1 und zwei zeitbehaftete Graphtransformationsregeln, P_1 und P_2. Ohne bereits genau darauf einzugehen, wie ein entsprechender Algorithmus zum Aufbau des Graphtransformationssystems aussieht, ist es möglich, wie in Abbildung 4.12 dargestellt, die Folgegraphen sowie die zu den Graphen zugehörigen zeitlichen Bedingungen in diesem einfachen Fall zu erstellen. Dabei ergeben sich, wie in Abbildung 4.12 zu sehen, mehrere identische Folgegraphen, welche sich nur in den ihnen zugeordneten zeitlichen Bedingungen unterscheiden. So existieren dort die Graphen G_1 und G_2 jeweils mit zwei unterschiedlichen zeitlichen Bedingungen.

Somit muss es möglich sein, einem zeitbehafteten Graphen nicht nur eine sondern mehrere der zeitlichen Erreichbarkeitsräume zuzuordnen, die durch Bedingungen wie in Definition 22 beschrieben aufgebaut werden.

Definition 31
Ein zeitbehafteter Graph $G_t := (G, C, \mathcal{T})$ mit mehreren zeitlichen Erreichbarkeitsräumen ist ein Tripel mit einem gerichteten Graph G, einer Anzahl Clockinstanzen C und einer Menge $\mathcal{T}$. Dabei ist $\mathcal{T} := \{T_l, ..., T_n\}$ mit $l, n \in \mathbb{N}^+$. Jedes $T_i \in \mathcal{T}$ besteht dabei aus einer Menge an zeitlichen Bedingungen über einzelnen Elementen aus C. Jeder Knoten v aus G besitzt eine eindeutige ID n_i und jede Kante $e \in E$ eine entsprechende ID e_j mit $i, j \in \mathbb{N}^+$.

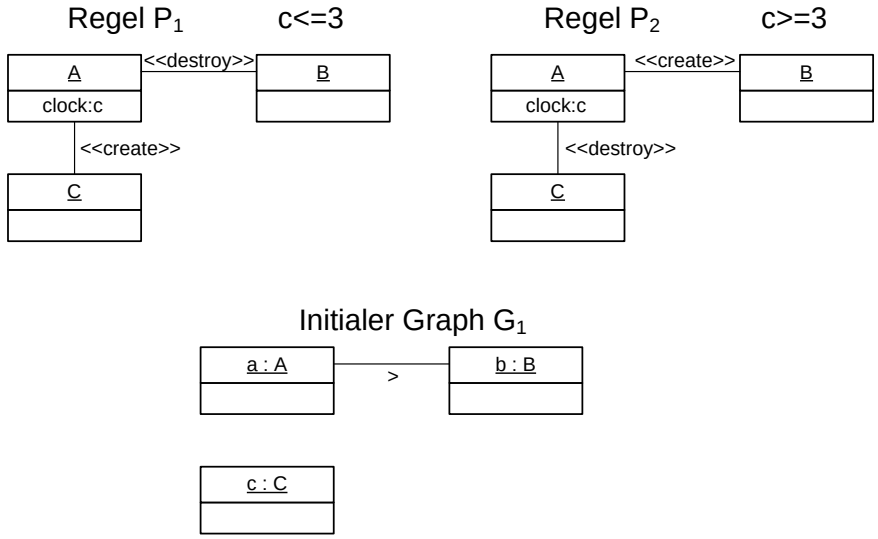


Abbildung 4.14: Ein Graphtransformationssystem mit einem initialen Startgraphen G_1, sowie zwei zeitbehaftete Graphtransformationsregeln P_1 und P_2.

4.3.3.2 Zeitbehaftetes Graphtransformationssystem

Bisher wurden alle notwendigen Definitionen eines zeitbehafteten Graphen gegeben. Im Folgenden wird nun darauf aufbauend ein zeitbehaftetes Graphtransformationssystem formal definiert. Zu diesem erweiterten Graphtransformationssystem gehören die in den vorherigen Abschnitten beschriebenen Bestandteile, welche sich aus einzelnen zeitbehafteten Graphen $G_t := (G, C, T)$, aus erweiterten Graphtransformationsregeln $P_t := (P_l, P_r, h, T, V_i, E_i, R, f)$ und aus Invariantenregeln $I_t := (G_l, t)$ zusammensetzen.

Definition 32
Ein Graphtransformationssystem $S_t := (\mathcal{G}_t, \mathcal{G}_t^0, \mathcal{P}_t, \mathcal{I}_t)$ mit zeitlichen Bestandteilen besteht aus einer potentiell unendlichen Menge an zeitbehafteten Graphen $\mathcal{G}_t$, einer endlichen Menge an initialen zeitbehafteten Graphen $\mathcal{G}_t^0$, einer endlichen Menge an zeitbehafteten Graphtransformationsregeln $\mathcal{P}_t$, wie diese in Definition 27 definiert sind, sowie einer endlichen Menge an Invarianten-Regeln $\mathcal{I}_t$.

Um alle erreichbaren Zustände für das Graphtransformationssystem S_t zu berechnen, muss nun noch beschrieben werden, wie sich aus den Graphtransformationsregeln $\mathcal{P}_t$ ableitbare Clockinstanzregeln erzeugen lassen.

4.3.3.3 Clockinstanzregeln

Bei dem oben vorgestellten Graphtransformationssystem $S_t := (\mathcal{G}_t, \mathcal{G}_t^0, \mathcal{P}_t, \mathcal{I}_t)$ fehlen für eine Erreichbarkeitsanalyse noch die zugehörigen Clockinstanzregeln, welche innerhalb der Graphtransformationsregeln $\mathcal{P}_t$ sowie den Invariantenregeln $\mathcal{I}_t$ vorkommen. Diese können, wie im Abschnitt 4.3.2, Definition 29, dargestellt und aus den einzelnen Graphtransformationsregeln $P_t \in \mathcal{P}_t$ abgeleitet werden.

Um die entsprechenden Clockinstanzregeln $\mathcal{C}_t$ zu erhalten, wird für jede Graphtransformationsregel $P_t \in \mathcal{P}_t$ das in Abschnitt 4.3.2 (siehe Definition 29) beschriebene Verfahren zur Ableitung von Clockinstanzregeln angewendet, wobei jede erstellte Clockinstanzregel c_i der Menge $\mathcal{C}_t$ hinzugefügt wird. Es ist möglich, dass identische Regeln mehrfach abgeleitet und zu $\mathcal{C}_t$ hinzugefügt werden. Da $\mathcal{C}_t$ eine Menge ist, fallen doppelte Vorkommen weg.

4.4 Erreichbarkeitsanalyse

Auf der Basis des im vorherigen Abschnitt erstellten und erweiterten Modells eines Graphtransformationssystems der Form $S_t := (\mathcal{G}_t, \mathcal{G}_t^0, \mathcal{P}_t, \mathcal{I}_t)$ wird hier ein Algorithmus vorgestellt, mit dem eine Erreichbarkeitsanalyse durchgeführt werden kann. Dabei werden die hierzu verwendeten Operationen schrittweise erarbeitet.

4.4.1 Darstellung durch Clockzones

Der zeitliche Erreichbarkeitsraum eines zeitbehafteten Graphen G_t (siehe Abschnitt 4.3.3) lässt sich mit Hilfe der Datenstruktur der Clockzones beschreiben. Natürlich ist es auch

möglich, eine andere Datenstruktur zu wählen, um die zeitlichen Bedingungen aus Kapitel 4.3.3 zu beschreiben. Die Entscheidung, bereits an dieser Stelle die Datenstruktur der Clockzones zu verwenden, begründet sich darin, dass hierdurch die im Folgenden beschriebenen Algorithmen besser veranschaulicht werden können. Zusätzlich sind die hier verwendeten zeitlichen Bedingungen, welche in Form von Ungleichungen in Kapitel 4.3.3 definiert wurden, denen sehr ähnlich, die bei der Datenstruktur der Clockzones verwendet werden.

Aus den erwähnten Gründen erfolgt die Darstellung der zeitlichen Erreichbarkeitsräume in Form der in Definition 2.3.5.3 vorgestellten Clockzones. Die Definition eines zeitbehafteten Graphen wird entsprechend abgeändert. Somit ergibt sich der hier verwendete zeitbehaftete Graph $G_t := (G, C, \mathcal{Z})$, indem die in Definition 31 verwendete Menge $\mathcal{T}$ durch die Menge $\mathcal{Z}$ ersetzt wird. Bei $\mathcal{T}$ handelt es sich dabei um eine Menge dem Graphen zugeordneter zeitlicher Erreichbarkeitsräume T, die wiederum aus einzelnen zeitlichen Bedingungen t bestehen. Dabei wird jedes T in eine einzelne Clockzone Z überführt.

Diese Überführung ist problemlos möglich, da die zeitlichen Ungleichungen $t := (x_i - x_j \sim d)$ sehr stark denen der Clockzones ähneln. Der einzige Unterschied besteht darin, dass bei den Ungleichungen $t \in T$ anstelle der einfachen Clockvariablen der Clockzones die Clockinstanzen aus der Definition 21 verwendet werden. An dieser Stelle wird das Modell der Clockzones dahingehend angepasst, dass die dort verwendeten Clockvariablen durch Clockinstanzen ersetzt werden. Die daraus leicht abgeänderte Form der Clockzones ergibt sich zu:

Definition 33
Eine Clockzone Z mit Clockinstanzen hat eine Anzahl von Clockinstanzen x_i, wie diese in Definition 21 definiert sind. Die einzelnen x_i können Werte aus $\mathbb{R}^+ \cup 0$ annehmen, wobei $i \in \mathbb{N}^+$ und $i > 0$. Zusätzlich existiert eine Referenz-Clock x_0, die immer den Wert 0 besitzt sowie eine Anzahl von Bedingungen $c \in C$ in Form von Ungleichungen der Art $x_j \prec d, d \prec x_j, x_i - x_j \prec d$, mit $i, j \in \mathbb{N}^+$, $d \in \mathbb{Z}$ und $\prec \in \{<, \leq\}$. Die Clockzone ergibt sich aus der Konjunktion über Bedingungen aus C.

Die Eigenschaften und die Operationen auf den Clockzones ändern sich durch diese Erweiterung nicht. Die restlichen Bestandteile lassen sich direkt übernehmen. So entspricht die Referenzclock x_0 bei der Datenstruktur der Clockzones der Referenzclock, wie diese bei den zeitlichen Bedingungen in Kapitel 4.3.3 definiert wurde. Die Ungleichungen lassen sich direkt übernehmen. Ein zeitbehafteter Graph mit mehreren zeitlichen Erreichbarkeitsräumen, wie in Definition 31 dargestellt, wird dann zu einem entsprechenden Graphen $G_t := (G, C, \mathcal{Z})$ mit mehreren Clockzones $\mathcal{Z}$.

4.4.2 Zeitbehafteter Folgegraph

Die einzelnen Schritte, um einen Folgegraphen für einen zeitbehafteten Graphen abzuleiten, werden im Folgenden beschrieben. Ausgangspunkt sind dabei ein zeitbehafteter Graph $G_t := (G, C, \mathcal{Z})$, eine zeitbehaftete Graphtransformationsregel $P_t := (P_l, P_r, h, T, V_i, E_i, R, f, r)$, die Clockinstanzregeln $\mathcal{C}$ und Invariantenregeln $\mathcal{I}$. Dabei wird durch den zeitbehafteten Graphen G_t eine Anzahl von Zuständen abgebildet, die sich durch die Kombination des Graphen G aus G_t mit den einzelnen Clockzones $Z \in \mathcal{Z}$ ergeben. Ein solcher Zustand ist somit ein Tupel $\langle G, Z \rangle$. Weiterhin wird davon ausgegangen, dass die von der Graphtransformationsregel P_t abgeleiteten Clockinstanzregeln in der Menge $\mathcal{C}$ enthalten sind sowie, dass die durch diese Regeln erzeugbaren Clockinstanzen der Menge C des zeitbehafteten Graphen G_t bereits hinzugefügt wurden. Die sich aus der Anwendung der Graphtransformationsregel P_t auf den Graphen G_t ergebenden Folgezustände können durch die Funktion $prod$ (Kapitel 2.3.5.4) berechnet werden. Die Funktion liefert die Menge M der Graphmorphismen $m := (m_v, m_e)$ zurück, welche P_l aus P_t auf einen Teilgraphen g aus G abbilden, wobei wiederum G aus G_t stammt.

Nach Anwendung der Funktion $prod$ auf einem Graphen G müssen dann im Unterschied zu den ursprünglichen Graphtransformationssystemen weitere Schritte durchgeführt werden, bevor mit Hilfe der einzelnen Graphmorphismen $m \in M$ die Tochtergraphen hergeleitet werden. Zu diesen Schritten gehört die Überprüfung der zeitlichen Bedingungen $t \in T$ von P_t. Bevor allerdings diese Überprüfung stattfinden kann ist es notwendig, die innerhalb der einzelnen t verwendeten Clockinstanzen zuzuordnen. Warum und in welcher Art dies geschehen muss, ist nachfolgend beschrieben.

4.4.2.1 Zuordnen der Clockinstanzen zu den Regelanwendungen

Bei Clockinstanzen wird unterschieden zwischen Clockinstanzen, die innerhalb von zeitbehafteten Graphtransformationsregeln, Clockinstanzregeln sowie Invariantenregeln vorkommen und denen, wie diese innerhalb der zeitlichen Erreichbarkeitsräume vorhanden sind. Innerhalb der Graphtransformationsregeln, Clockinstanzregeln sowie Invariantenregeln handelt es sich um Clockinstanzen, die mit einzelnen Element-IDs des Graphen der linken Seite der jeweiligen Graphtransformationsregel P_t, bzw. der Clockinstanzregel C_t oder Invariantenregel I_t verbunden sind. Im Gegensatz dazu sind die Clockinstanzen der zeitlichen Erreichbarkeitsräume mit den IDs der Elemente des Graphen G aus G_t verbunden, auf den die einzelnen Regeln und Graphroduktionen angewendet werden. Wie diese Verknüpfung zu den IDs der Elemente aus G mit Hilfe der Clockinstanzregeln vorgenommen wird, ist im Kapitel 4.2.2.2 beschrieben.

Damit die mit einer Graphtransformationsregel verbundenen Guards T und Clockresets R innerhalb einer Graphtransformationsregel P_t verwendet werden können ist es notwendig, die dort vorhandenen Clockinstanzen bei der Anwendung von P_t auszutauschen. Dies bedeutet für jeden Morphismus m, der aus $prod(P_t, G)$ resultiert, eigene Guards T_m und Clockresets R_m herzuleiten. Hierzu wird die Funktion $assign(m, T, R)$ verwendet, welche nach dem folgenden Schema arbeitet und das Tupel $\langle T_m, R_m \rangle$ zurückliefert.

Dabei müssen die Clockinstanzen aus P_t mit denen dem Graphen G durch die Clockinstanzregeln hinzugefügten Clockinstanzen ausgetauscht werden. Hierzu muss beim Aufsuchen der linken Seite P_l von P_t innerhalb des Wirtsgraphen G eine Zuordnung der Element-IDs von der linken Seite P_l zu den Element-IDs der Stelle m in G vorgenommen werden, an der P_l innerhalb von G bei der aktuellen Anwendung erfüllt ist. Ein Beispiel hierfür zeigt die Abbildung 4.15.

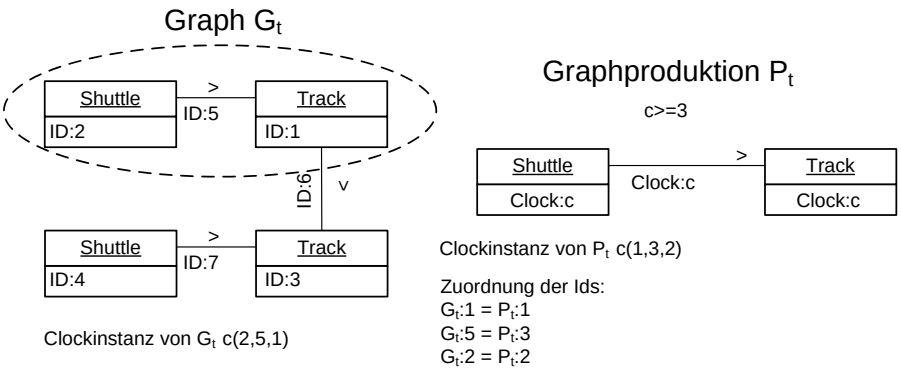


Abbildung 4.15: Im Wirtsgraphen G_t wird die Graphtransformationsregel P_t an der rot umrandeten Stelle angewendet

Diese Zuordnung geschieht, indem T und R wie folgt überführt werden. Dabei gilt für alle $t \in T$, $t := (x_i - x_j \sim d)$ ist eine zeitliche Bedingung mit Clockinstanzen x_i und x_j. x_0 beschreibt die Referenz-Clock, welche zu jedem Zeitpunkt den Wert 0 besitzt.

Zunächst wird die Funktion $graphID(m, x, G, P_l)$ eingeführt (siehe Anhang A, Algorithmus A.1). Diese nimmt eine Zuordnung der Knoten- und Kanten-IDs zu dem Graphmorphismus $m := (m_v, m_e)$, einer Clockinstanz $x := (M, \mathcal{N}, \mathcal{E})$, dem Wirtsgraphen $G := (V_G, E_g, E_{(s,G)}, E_{(t,G)}, V_{(i,G)}, E_{(i,G)})$ und der linken Seite $P_l := (V_P, E_P, E_{(s,P)}, E_{(t,P)}, V_{(i,P)}, E_{(i,P)})$ aus P_t vor. Dies geschieht nach dem Vorgehen wie im Beispiel der Abbildung 4.15 aufgezeigt. Der Rückgabewert sind die Knoten- und Kanten-IDs $\mathcal{N}', \mathcal{E}'$ aus dem Wirtsgraphen G.

Mit Hilfe der Funktion $graphID$ wird nachfolgend die Funktion $assign$ formuliert (siehe Anhang A, Abbildung A.2), mit der die Guards T und Resets R zu T_m und R_m überführt werden.

Die hierdurch entstandenen Guards T_m und Clockresets R_m gelten ausschließlich für die momentane Anwendung der Graphtransformationsregel P_t bezüglich m innerhalb des Wirtsgraphen G.

4.4.2.2 Erzeugen einer Folge-Clockzone

Bevor für die durch $prod$ hergeleiteten Graphmorphismen m angewendet werden, muss an dieser Stelle überprüft werden, von welchen Graphzuständen $\langle G, \phi \rangle$ aus zusammen mit den Guards T' eine Clockzone ϕ' mit Hilfe der Funktion $succ_\phi(\phi, I, \varphi)$ aus Kapitel 2.3.5.3 hergeleitet werden kann. Um diese Funktion anwenden zu können, sind die dem aktuellen Zustand zugehörigen Invarianten I notwendig.

Diese Invarianten werden hergeleitet, indem die einzelnen Invariantenregeln $I_t \in \mathcal{I}$ auf den Wirtsgraphen G angewendet werden. Dabei kann die Funktion $prod$ aus Kapitel 2.3.5.4 verwendet werden, um die Menge der Morphismen M herzuleiten. Die Funktion benötigt eine linke und rechte Anwendungsseite L und R, sowie einen Morphismus m. Eine Invariantenregel $I_t := (I_l, h, t, V_i, E_i, f)$ besitzt keine rechte Anwendungsseite R. Dabei ist bei I_t diese identisch mit I_l, also ergibt sich $R, L = I_l$ und $m = h$. Somit kann die Funktion $prod$ verwendet werden um die Menge der Graphmorphismen M herzuleiten.

Aus diesen Graphmorphismen M können nach dem gleichen Schema wie in Kapitel 4.4.2.1 beschrieben die einzelnen Ungleichungen hergeleitet werden. Die hierdurch entstehende Menge an Invarianten I, welche aus einzelnen zeitlichen Bedingungen i mit Clockinstanzen besteht, wird der Funktion $succ_\phi(\phi, I, T')$ zusammen mit der Clockzone ϕ und den Guards T' übergeben. Als Ergebnis liefert $succ_\phi$ die Clockzone ϕ_{succ}, bei der es zu überprüfen gilt, ob diese leer ist[1].

Falls ϕ_{succ} einer leeren Clockzone entspricht, wird das Ergebnis verworfen. Ein Folge-Clockzone kann entsprechend über die Funktion $succ_\phi$ nicht hergeleitet werden und somit die Transition von dem Wirtsgraphen G aus G_t zu einem Folgegraphen G' nicht über die Clockzone ϕ an der zu m zugehörigen Stelle vorgenommen werden. Andernfalls wird damit fortgefahren, die Clockresets R_m mit Hilfe der Funktion $succ'_\phi$ aus Kapitel 2.3.5.3 auf ϕ_{succ} anzuwenden, um die Clockzone ϕ' zu erhalten. Falls eine Clockzone ϕ_{succ} nicht leer ist, wird der aus dem Graphmorphismus m herleitbare Graph G' erzeugt. Welche Knoten $v_{G'}$ und Kanten $E_{G'}$ sich dabei aus m zu G' ergeben, ist in Abschnitt 2.3.4 beschrieben.

Bevor mit Hilfe des so erzeugten Graphen G' und der Clockzone ϕ' ein Folgezustand $\langle G', \phi' \rangle$ zu dem zeitbehafteten Folgegraphen G'_t hinzugefügt wird, müssen weitere Schritte durchgeführt werden. Hierzu gehört die Erzeugung der Clockinstanzen des Graphen G' durch die Clockinstanzregeln $\mathcal{C}$, sowie die weitere Überarbeitung der Clockzone ϕ'. Bei

[1] Zu leeren Clockzones siehe Kapitel 2.3.5.3.

dieser müssen die Invarianten hinzugefügt werden, welche über die Anwendung der Invariantenregeln auf G' erzeugbar sind. Zusätzlich müssen alle zeitlichen Bedingungen t entfernt werden, für die keine Clockinstanzen durch die einzelnen $c \in \mathcal{C}$ innerhalb von G' erzeugt wurden.

4.4.2.3 Erzeugen der Clockinstanzen des Folgegraphen

Zur Erzeugung der Clockinstanzen C durch die Clockinstanzregeln aus $\mathcal{C}$ innerhalb des Graphen G' wird eine Funktion $C = prod_{clock}(\mathcal{C}, G')$ eingeführt (siehe Anhang A, Algorithmus A.3).

4.4.2.4 Erzeugen des Folgezustands

Mit Hilfe der so erzeugten Clockinstanzen C, sowie den Invarianten I des Folgegraphen G', wird der Folgezustand hergeleitet. Innerhalb der Menge C können Clockinstanzen vorhanden sein, die beim Folgegraphen G' durch die Funktion $prod_{clock}$ neu erzeugt wurden. Diese neuen Clockinstanzen ergeben sich, indem alle Clockinstanzen $c \in C_G$ des Graphen G aus der Menge C der Clockinstanzen des Graphen G' entfernt werden. Die resultierende Menge wird an dieser Stelle mit $C_{new} = C \setminus C_G$ bezeichnet. Diese hinzugekommenen Clockinstanzen wurden gerade erst erzeugt und müssen entsprechend zu den bereits existierenden in Relation gesetzt werden. Für die neu hinzugekommenen Clockinstanzen c_{new} gilt, dass diese zu den bereits existierenden wie folgt in Relation gesetzt werden müssen:

Für die Menge der Clockinstanzen $C_{old} = C \setminus C_{new}$, welche bereits bei dem Vorgänger Graphen G vorhanden waren, existiert eine Anzahl an Bedingungen innerhalb der Clockzone ϕ'. Jede Clockinstanz $c_{old} \in C_{old}$ hat dort eine obere Schranke o und untere Schranke u, für die gilt, dass $o \in \mathbb{Z}^+ \cup \infty$ und $u \in \mathbb{R}^+$. Diese Schranken lassen sich aus den zeitlichen Bedingungen t aus ϕ' ermitteln. Für die hinzugekommenen Clockinstanzen $c_{new} \in C_{new}$ müssen für jede Clockinstanz aus $c_{old} \in C_{old}$ Ungleichungen der Art $c_{new} - c_{old} \sim -u$ und $c_{old} - c_{new} \sim o$ hinzugefügt werden. o und u müssen dabei zu jedem c_{old} ermittelt werden. $\sim$ entspricht $<$ oder $\leq$, je nachdem, welche Art der Ungleichung bei dem entsprechenden t aus ϕ' ermittelt wurde.

Nachfolgend werden die Invarianten I auf $\phi' := \phi' \cup I$ angewendet und anschließend alle Bedingungen $t = (x_i - x_j \sim d)$ aus ϕ' entfernt, für die gilt, dass mindestens eine Clockinstanz x_i oder x_j nicht in C enthalten ist (siehe Anhang A, Algorithmus A.4).

Zum Abschluss werden mit der Funktion $\phi' = succ'_{\phi}(\phi', R')$, wie in Kapitel 2.3.5.3 beschrieben, die Clockresets R' auf ϕ' angewendet. Die hieraus resultierende Clockzone

bildet zusammen mit dem Graphen G' in Form des Tupels $\langle G', \phi' \rangle$ einen Folgezustand des zeitbehafteten Folgegraphen G'_t.

4.4.2.5 Zeitbehafteter Folgegraph

Um den gesamten zeitbehafteten Folgegraphen G'_t zu berechnen, werden die hier vorgestellten Operationen bzw. Funktionen wie im Algorithmus $production_m$ angewendet (siehe Anhang A, Algorithmus A.5). Um alle durch P_t erzeugbaren Folgegraphen $\mathcal{G}'_t$ zu berechnen, wird die Funktion $production_m$ zum Algorithmus $production$ erweitert (siehe Anhang A, Algorithmus A.6).

4.4.3 Erreichbares Graphtransformationssystem

Abschließend kann nun beschrieben werden, wie sich mit den angegebenen Funktionen der gesamte Erreichbarkeitsraum eines zeitbehafteten Graphtransformationssystems erzeugen lässt. Ausgangspunkt ist hierfür das zeitbehaftete Graphtransformationssystem $S_t := (\mathcal{G}_t, \mathcal{G}_t^0, \mathcal{P}_t, \mathcal{I}_t)$, wobei $\mathcal{G}_t$ eine Menge an zeitbehafteten Graphen G_t, $\mathcal{G}_t^0$ die Menge der initialen zeitbehafteten Graphen G_t^0, $\mathcal{P}_t$ die Menge an zeitbehafteten Graphtransformationsregeln P_t und $\mathcal{I}$ die Menge der Invariantenregeln darstellt.

4.4.3.1 Erreichbarkeitsanalyse

Nach der Initialisierung und Erzeugung der Clockinstanzregeln $\mathcal{C}$ aus den einzelnen Graphtransformationsregeln P_t wird die folgende Funktion $reachGTS_t$ aufgestellt (siehe Algorithmus 4.3), mit welcher der Zustandsraum des Graphtransformationssystems aufgebaut wird. Dabei gibt es zwei Mengen $Open$ und $Closed$, wobei die vorhandenen Graphen aus S_t initial der Menge $Open$ zugewiesen werden. Der Menge $Closed$ sind alle Graphen aus $\mathcal{G}_t$ zugewiesen, die nicht in der Menge der initialen Graphen $\mathcal{G}_t^0$ vorhanden sind.

Dabei arbeitet der Algorithmus wie folgt: Solange noch ein zeitbehafteter Graph G_t innerhalb der Menge $Open$ enthalten ist (Zeile 3), durchlaufe alle Graphen der Menge $Open$ (Zeile 4) und wende auf jeden Graphen die einzelnen zeitbehafteten Graphtransformationsregeln P_t mit Hilfe der Funktion $production$ an (Zeile 7). Für jeden daraus resultierenden zeitbehafteten Folgegraphen $G'_t \in \mathcal{G}'_t$ (Zeile 8) überprüfe, ob der in G'_t enthaltene einfache Graph G' bereits in einem zeitbehafteten Graphen $G_{tmp} := (G_{tmp}, \mathcal{C}_{tmp}, \mathcal{Z}_{tmp})$ der Mengen $Open$ oder $Closed$ vorhanden ist (Zeile 11-15). Falls dies der Fall ist überprüfe, ob die Menge $\mathcal{Z}'$ der Folge-Clockzones des Graphen G'_t nicht identisch mit der Menge $\mathcal{Z}_f$ der Clockzones des Graphen G_f ist (Zeile 16). Falls dies zutrifft,

Algorithmus 4.3 procedure $\mathcal{S}'_t = reachGTS_t(S_t, C)$

procedure $\mathcal{S}'_t = reachGTS_t(S_t, C)$

1: $S_t := (\mathcal{G}_t, \mathcal{G}_0, \mathcal{P}_t, \mathcal{I}_t)$
2: $Open = \mathcal{G}_t^0, Closed = \mathcal{G}_t \setminus \mathcal{G}_t^0$
3: **while** $Open \neq \emptyset$ **do**
4: **for all** $G_t \in Open : G_t := (G, \mathcal{C}, \mathcal{Z})$ **do**
5: **for all** $P_t \in \mathcal{P}_t$ **do**
6: $selfedge = false$
7: $\mathcal{G}'_t = production(G_t, P_t, C, \mathcal{I}_t)$ ▷ *
8: **for all** $G'_t \in \mathcal{G}'_t : G'_t := (G', \mathcal{C}', \mathcal{Z}')$ **do**
9: $found := (G_f, \mathcal{C}_f, \mathcal{Z}_f)$
10: $found = NULL$
11: **for all** $G_{tmp} \in Open \cup Closed : G_{tmp} := (G_{tmp}, \mathcal{C}_{tmp}, \mathcal{Z}_{tmp})$ **do**
12: **if** $G' = G_{tmp}$ **then**
13: $found = G_{tmp}$, break
14: **end if**
15: **end for**
16: **if** $found \neq NULL \wedge \mathcal{Z}_f \neq \mathcal{Z}'$ **then** ▷ *
17: $\mathcal{Z}_f = \mathcal{Z}' \cup \mathcal{Z}_f$ ▷ *
18: $Open = Open \cup found$ ▷ *
19: $Closed = Closed \setminus found$ ▷ *
20: **if** $G_f = G$ **then**
21: $selfedge = true$
22: **end if**
23: **end if**
24: **if** $found == NULL$ **then**
25: $Open = Open \cup G'_t$
26: **end if**
27: **end for**
28: **end for**
29: **if** $\neg selfedge$ **then**
30: $Open = Open \setminus G_t$
31: **end if**
32: **end for**
33: **end while**

existierte bereits ein zeitbehafteter Graph innerhalb des Graphtransformationssystems, der sich nur bezüglich der diesem zugeordneten zeitlichen Erreichbarkeitsraum unterscheidet. Somit wird dem bereits existierenden Graphen G_f die Vereinigung beider zeitlichen Erreichbarkeitsräume zugewiesen (Zeile 17). Anschließend wird G_f in die Menge $Open$ verschoben und aus $Closed$ entfernt (Zeile 18 und 19). Die in Zeile 21 zugewiesene

Variable behandelt einen Sonderfall, nämlich, dass der Mutter- und Tochtergraph identisch ist. In diesem Fall darf G_t nicht aus der Menge $Open$ in Zeile 30 entfernt werden, da zu G_t ein zeitlicher Erreichbarkeitsraum hinzugekommen ist. Falls kein entsprechender Graph G_t innerhalb der Mengen $Open \cup Closed$ aufgefunden wird, so handelt es sich bei G'_t um einen neuen Graphen, welcher der Menge der offenen Graphzustände $Open$ zugewiesen wird (Zeile 20-21). Der Algorithmus ist beendet, wenn die Menge $Open$ aller offenen Graphen leer ist.

Im Vergleich mit dem Algorithmus zur Berechnung der erreichbaren Zustände eines Graphtransformationssystems ohne zeitliche Bestandteile (siehe Kapitel 2.3.5.4) sind im Wesentlichen die Zeilen, welche mit einem * am Ende versehen sind, hinzugekommen.

4.4.3.2 Prioritäten

Um die in Kapitel 2.3.4 beschriebenen Prioritäten zu berücksichtigen muss der oben angegebene Algorithmus angepasst werden. Dabei wird davon ausgegangen, dass die zeitbehafteten Graphtransformationsregeln $\mathcal{P}_t$ des zeitbehafteten Graphtransformationssystems $S_t := (\mathcal{G}_t, \mathcal{G}_t^0, \mathcal{P}_t, \mathcal{I}_t)$ zusätzlich über eine Priorität $r \in \mathbb{N}^+$ verfügen.

Hierbei muss die for-Schleife in Zeile 5 der Funktion $reachGTS_t$ so erweitert werden, dass alle Graphtransformationsregeln P_t mit gleicher Priorität r in jeweils einer Menge Q_r zusammengefasst werden. Die einzelnen Mengen werden dann mit der for-Schleife der Zeile 5 abgearbeitet, wobei die Menge mit den Regeln, welche die höchste Priorität r haben, zuerst abgearbeitet wird. Vor jeder Abarbeitung wird überprüft, ob durch mindestens eine der Regeln P_r der vorhergehenden Menge bereits ein oder mehrere Folgegraphen hergeleitet wurden. Ist dies der Fall, so wird mit dem nächsten Graphen G_t in Zeile 4 fortgefahren.

4.4.3.3 Verifikationsverfahren

Um weitergehende Verifikationsverfahren wie etwa das Model Checking anwenden zu können ist es notwendig, neben den erreichbaren Zuständen auch die Reihenfolge zu kennen, in der diese erreicht werden. Eine entsprechende Erweiterung kann vorgenommen werden, indem zu den Zuständen des Graphtransformationssystems zusätzlich die Übergänge angegeben werden. Ein solcher Zustand $\langle G, Z \rangle$ setzt sich zusammen aus einem Graphen G sowie einer Clockzone Z. Entsprechend müssen für eine weitere Analyse die Übergänge $\langle G, Z \rangle \times \langle G', Z' \rangle$ dem Graphtransformationssystem hinzugefügt werden. Diese Übergänge können in dem Algorithmus $reachGTS_t$ beim Anwenden der Funktion $production$ in Zeile 7 erzeugt werden. Dabei entstehen aus dem Graphen $G_t := (G, \mathcal{C}, \mathcal{Z})$ zusammen mit den Folgegraphen $\mathcal{G}'_t$ die einzelnen Übergänge. $\mathcal{G}'_t$ setzt sich dabei wieder-

um aus einer Anzahl an Graphen $G'_t := (G', \mathcal{C}', \mathcal{Z}')$ zusammen. Die Übergänge ergeben sich aus dem Kreuzprodukt $\langle G, Z\rangle \times \langle G', Z'\rangle$ für alle G, Z aus G_t, und für alle G', Z' der G'_t aus $\mathcal{G}'_t$. Der Algorithmus $reachGTS_t$ kann hierzu in Zeile 7 um die folgenden Zeilen ergänzt werden:

for all $G'_t \in \mathcal{G}'_t : G'_t := (G', \mathcal{C}', \mathcal{Z}'$ **do**
$TR := TR \cup \langle G, Z\rangle \times \langle G', Z'\rangle$
end for

TR bildet dabei die Menge der Transition ab, die innerhalb des Graphtransformationssystems vorhanden sind.

4.4.3.4 Optimierung

Um den erzeugten Zustandsraum bei der Erreichbarkeitsanalyse möglichst klein zu halten, können bestimmte Teilzustände zusammengefasst werden. Dies betrifft die zeitlichen Erreichbarkeitsräume Z, welche zusammen mit den einzelnen Graphen G einen Zustand $\langle G, Z\rangle$ bilden. Dabei kann es vorkommen, dass bei unterschiedlichen Zuständen $\langle G_1, Z_1\rangle$ und $\langle G_2, Z_2\rangle$ für die beiden Clockzones Z_1 und Z_2 gilt, dass die eine Teilmenge der anderen ist. Dies ist der Fall, wenn beide die gleichen Clockinstanzen enthalten und zusätzlich für die aufgespannten Erreichbarkeitsräume gilt, dass Z_1 in Z_2 enthalten ist bzw. Z_2 in Z_1 enthalten.

Falls dies der Fall ist und zusätzlich gilt, dass die beiden Graphen G_1 und G_2 isomorph sind, kann der Zustand verworfen werden, bei dem die zugehörige Clockzone eine Teilmenge der Clockzone des anderen Zustandes darstellt.

Um effizient feststellen zu können, ob eine Clockzone Teilmenge einer anderen ist, kann mit der Datenstruktur der Difference-Bound-Matrice gearbeitet werden. Jede Clockzone kann, wie in Kapitel 2.3.5.3 beschrieben, in Form einer Difference-Bound-Matrice dargestellt werden. Jede Difference-Bound-Matrice kann in eine kanonische Form überführt werden, womit entsprechende Vergleiche effizient möglich sind.

Eine derartige Optimierung macht nur Sinn, wenn keine erweiterten Verifikationsverfahren angewendet werden sollen, für die gilt, dass diese die genaue Abfolge von erreichten Zuständen kennen müssen. Dieser Fall ist etwa beim Model Checking gegeben. Dort ist von Interesse, in welcher Reihenfolge die Zustände erreicht werden. Da durch die hier aufgezeigte Optimierung einzelne Zustände wegfallen können, fallen damit entsprechende Informationen über die logische Abfolge, in der diese erreicht wurden, ebenfalls weg.

4.5 Evaluierung

Die hier vorstellten Konzepte wurden in [Neu07] prototypisch in dem Werkzeug GROOVE[2] [Ren04, RKS06] umgesetzt. GROOVE ist ein Werkzeug zur Modellierung und Analyse von Graphtransformationssystemen. GROOVE bietet die Möglichkeit, den kompletten Erreichbarkeitsraum eines Graphtransformationssystems zu erstellen. GROOVE besteht aus mehreren Teilwerkzeugen, einem Editor, Generator und Simulator. In dem Editor werden einzelne Graphtransformationsregeln erstellt, die später das Graphtransformationssystem aufspannen. Über den Simulator können später alle erreichbaren Zustände ermittelt werden. Der Generator arbeitet ähnlich dem Simulator, jedoch ohne grafische Oberfläche.

Das zu Beginn des Kapitels vorgestellte Beispiel (siehe Abschnitt 4.2.1) wurde in GROOVE modelliert. Dabei wurde das Beispiel erweitert derart, dass das Schienennetz nun auch aus Weichen besteht, die zwei Ovale miteinander verbinden. Neben der einfachen Regel, die eine Shuttlebewegung von einem Schienenabschnitt zum anderen beschreibt, muss nun auch die Überfahrt einer Weiche mit Zeit beschrieben werden (siehe Abbildung 4.16). Die vollständigen Regeln sind in Anhang B dargestellt.

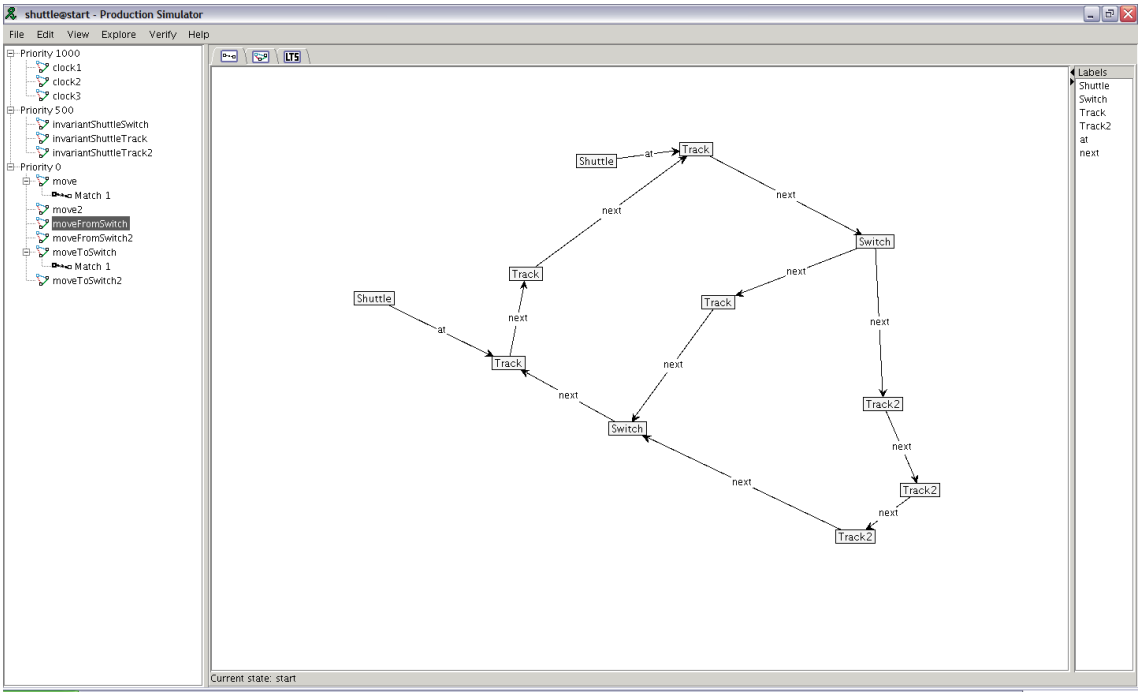


Abbildung 4.16: Schienennetz

[2] http://groove.cs.utwente.nl/groove-home/

In Abbildung 4.17 ist das aus der Erreichbarkeitsanalyse resultierende Graphtransformationssystem für das Beispiel dargestellt. Es besteht aus 28 Graphzuständen, 50 Transitionen zwischen den einzelnen Graphzuständen in Form von Kanten, sowie über 54 zeitliche Erreichbarkeitsräume. Die Analyse[3] hat 2 Sekunden gedauert. Wird noch ein weiteres Shuttle hinzugefügt, entstehen 165 Graphzustände, 537 Transitionen sowie 869 zeitliche Erreichbarkeitsräume. Bei diesem Szenario dauert die Analyse 7 Sekunden.

Eines der größten Probleme bei der Analyse von komplexeren Modellen ist die steigende Anzahl an Clockzones, welche zusammen mit den Graphzuständen einen Zustand des erweiterten Graphtransformationssystems abbilden. Dies liegt vor allem daran, dass eine Clockzone mit n Clockinstanzen die Größe n^2 hat. Hier greift die Optimierung, welche ebenfalls in GROOVE implementiert wurde. Ist diese aktiv, verringert sich die Anzahl der zeitlichen Erreichbarkeitsräume bei dem Szenario mit drei Shuttles auf 698. Die Analyse hat hierbei 8 Sekunden gedauert. Die erhöhte Analysezeit ist auf die Optimierung zurückzuführen. Für eine detaillierte Auswertung des Optimierungsverfahrens wird auf die Arbeit [Neu07] verwiesen.

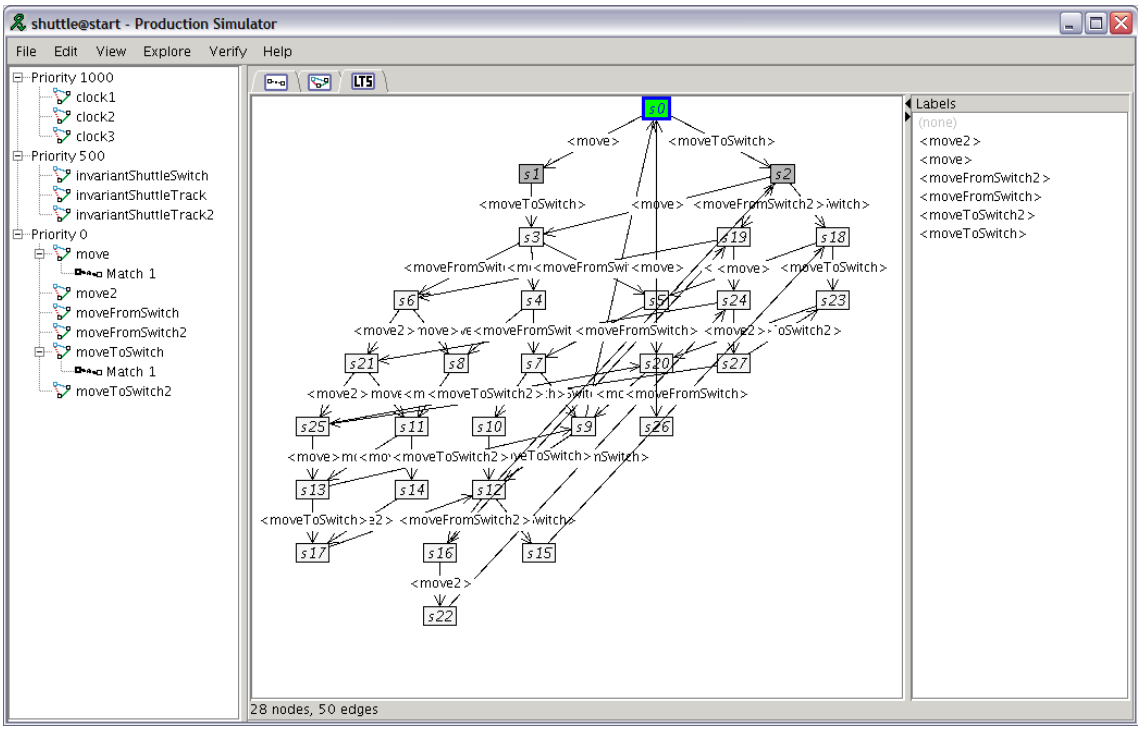


Abbildung 4.17: Das resultierende Graphtransformationssystem

[3]Wurde auf einem Pentium 4, 2.4 GHz, 1 GB memory, OS Linux Redhat durchgeführt.

4.6 Zusammenfassung

In diesem Kapitel wurden Modellierungs- und Verifikationstechniken für das äußere Verhalten eines OCMs in der Umwelt vorgestellt. Bei der Modellierung wurde auf den Formalismus der Graphtransformationssysteme zurückgegriffen, der hier durch Zeitannotationen angereichert wurde. Die Zeitannotationen basieren auf den Konzepten der Timed Automata. Da diese aber nicht so einfach übernommen werden können, wurde hierzu zuerst ein Vergleich beider Modelle diskutiert. Aufgrund dieser Erkenntnis wurden Konzepte zur Modellierung von zeitbehafteten Graphtransformationssystemen definiert. Nach der formalen Definition eines zeitbehafteten Graphtransformationssystems wurden anschließend die Erreichbarkeitsanalyse für ein solches System und Algorithmen hierfür beschrieben. Am Ende des Kapitels wurde eine Evaluierung des Ansatzes gezeigt.

Kapitel 5

Parametrisierte Koordinationsmuster

In den beiden vorangegangenen Kapiteln wurde beschrieben, wie sich ein OCM modellieren und verifizieren lässt. Der Fokus dieses Kapitels steht nun auf der Modellierung und Verifikation der Koordination von OCMs in vernetzten mechatronischen Systemen. Hierbei werden die bisher vorgestellten Techniken aus den vorangegangenen Kapiteln miteinander geschickt verknüpft.

Ein wichtiges Problem bei vernetzten mechatronischen Systemen ist, dass jedes Teilsystem eine potentiell unterschiedliche lokale Sicht haben kann, auf deren Basis jederzeit Entscheidungen autonom und lokal getroffen werden müssen. Die Logik aller Teilsysteme muss dabei auf Basis dieser lokalen Sicht bei einem Teilausfall im Gesamtsystem so koordiniert reagieren, dass Gefahren ausgeschlossen sind. Im Beispiel der „Neuen Bahntechnik Paderborn“ (siehe Abschnitt 1.2) müssen die Shuttles trotz möglicher Fehler immer ein sicheres Fahrmanöver garantieren. Diese Sicherheitseigenschaft muss für das modellierte Verhalten des Shuttles überprüft werden. Diese Überprüfung muss alle möglichen Situationen betrachten und den Ausschluss der Gefahren durch formale Verifikation absichern.

Der bisherige MECHATRONIC UML Ansatz stellt das Konzept der Echtzeit-Koordinationsmuster (siehe Grundlagen 2.4.2) zur Verfügung, um die Koordination verteilter Komponenten zu modellieren und formal zu verifizieren. Weiter unterstützt der Ansatz eine Integration der benötigten Steuer- und Regelungsalgorithmen (siehe Grundlagen 2.4.4).

Ziel dieses Kapitels ist es nun, für Verifikationszwecke eine abstrakte Betrachtung des relevanten Werte- und Zeit-kontinuierlichen Verhaltens der Koordinationslogik zu ermöglichen. Dabei werden entsprechend benötigte Eigenschaften der unterlagerten Regelung, die mit klassischen Techniken der Regelungstechnik und Mathematik verifiziert werden können, als Basis für weitere Betrachtungen verwendet. Darauf aufbauend lässt sich dann durch formale Verifikationstechniken für Echtzeitsysteme eine Verifikation der benötigten

Sicherheitseigenschaften der Echtzeitkoordination bzgl. der relevanten Fehlerszenarien erreichen.

Zuerst wird mit dem in den Grundlagen (siehe Kapitel 2.4.1) vorgestellten Ansatz zur Modellierung der bisherigen Echtzeit-Koordinationsmuster das Beispiel der Konvoikoordination noch einmal kurz beschrieben. Hieran werden anschließend die Grenzen des bisherigen Ansatzes aufgezeigt (siehe Abschnitt 5.2). Anhand eines erweiterten Beispiels wird in Abschnitt 5.3 die Idee eine Lösungsidee vorgestellt. Anschließend werden die in diesem Kapitel neu eingeführten parametrisierten Koordinationsmuster in Abschnitt 5.4 zuerst informal eingeführt und später formalisiert. Abschließend werden die nötigen Verifikationsschritte für ein parametrisiertes Koordinationsmuster beschrieben. Das Kapitel schließt mit einer Zusammenfassung in Abschnitt 5.5.

5.1 Beispiel

Mit dem kompositionellen Ansatz aus [GTB+03] ist es möglich, die Kommunikation zwischen Komponenten durch so genannte *Echtzeit-Koordinationsmuster* (siehe Kapitel 2.4) zu modellieren. Das Verhalten der Koordinationsmuster wird später bei der Anwendung zum Verhalten der Komponenten verfeinert.

In Abbildung 5.1 ist ein Echtzeit-Koordinationsmuster dargestellt. Es besteht aus mehreren Kommunikationspartnern, den so genannten *Rollen*. Rollen interagieren über einen Connector, durch den sie verbunden sind. Das Verhalten der Rollen und des Connectors wird durch Realtime Statecharts realisiert. Weiterhin besitzt eine Rolle Invarianten, welche eingehalten werden müssen. Das ganze Verhalten eines Echtzeit-Koordinationsmusters kann durch Constraints eingeschränkt werden.

In dem Beispiel wird die sichere Echtzeitkoordination in einem Konvoi für zwei hintereinander herfahrende Shuttles durch ein Echtzeit-Koordinationsmuster beschrieben. Das Echtzeit-Koordinationsmuster ConvoyCoordination besitzt zwei Rollen, die Rolle shuttle und die Rolle coordinator und einen Connector, der diese verbindet. Die Eigenschaft, die das Echtzeit-Koordinationsmuster zu erfüllen hat, ist, dass wenn das hinterherfahrende Shuttle im Konvoimodus ist, auch das vorherfahrende Shuttle im Konvoimodus sein muss (shuttle.Convoy implies coordinator.convoy). Wäre das hinterherfahrende Shuttle nämlich im Konvoimodus, das vorherfahrende jedoch nicht, würde in einer Notfallsituation das vorherfahrende Shuttle aufgrund der lokalen Information nicht richtig reagieren und einen Auffahrunfall verursachen.

In Abbildung 5.2 ist die Anwendung des Echtzeit-Koordinationsmusters gezeigt. Im Beispiel wendet das Shuttle *shuttle2* die Rolle *shuttle* an und das Shuttle *shuttle1* übernimmt die Rolle *coordinator*.

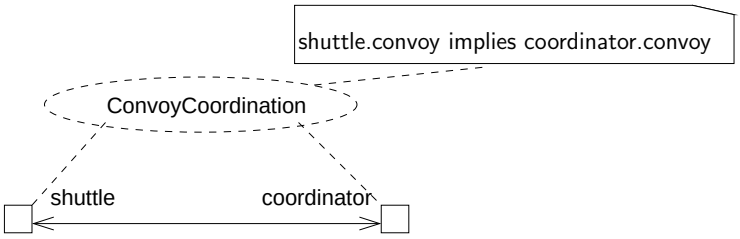


Abbildung 5.1: Echtzeit-Koordinationsmuster ConvoyCoordination

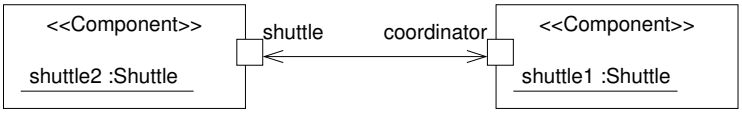


Abbildung 5.2: Anwendung des Echtzeit-Koordinationsmuster ConvoyCoordination

5.2 Grenzen des bisherigen Ansatzes

Der bisherige MECHATRONIC UML Ansatz stellt die fundamentalen Konzepte zur kompositionellen Modellierung und Verifikation zur Verfügung. Jedoch hat der bisherige MECHATRONIC UML Ansatz eine Reihe von Einschränkungen hinsichtlich der zur Beschreibung der Koordination von OCMs verwendeten Echtzeit-Koordinationsmuster. Dies lässt sich an den folgenden zwei Punkten manifestieren.

Dynamik: Ein Echtzeit-Koordinationsmuster besteht a priori immer aus einer festen Anzahl von Rollen. Anforderungen an komplexe, mechatronische Systeme sehen jedoch mehr Dynamik vor. Am Beispiel des Shuttlekonvois ist dies gut zu verdeutlichen. So ist nachzuvollziehen, dass ein Konvoi, um auch wirklich energieeffizient zu sein, aus mehr als zwei Shuttles bestehen muss. Dabei ist die Anzahl der Konvoiteilnehmer zum Zeitpunkt der Instanziierung des Koordinationsmusters unbekannt. Mal muss ein und dasselbe Koordinationsmuster einen Konvoi der Länge k und im nächsten Moment einen Konvoi der Länge $k + 1$ koordinieren, ohne die Stabilität eines Konvois dabei zu verletzten.

Stabilität: Bei den bisherigen Echtzeit-Koordinationsmustern steht das Koordinationsverhalten nicht in Verbindung mit dem Werte-kontinuierlichen Verhalten eines OCMs. Um jedoch die Stabilität eines Shuttlekonvois zu erreichen und damit das „Aufschaukeln“ und den so genannten Ziehharmonikaeffekt zu vermeiden, muss zusätzlich zur reinen Echtzeitkoordination das Werte-kontinuierliche Verhalten berücksichtigt werden. So müssen Brems- und Beschleunigungssituationen, die durch nicht-lineares Verhalten beschrieben werden, berücksichtigt werden. Eine alleinige Verbindung beider Verhalten durch das Synchronisationsstatechart innerhalb eines OCMs durch ein Hybrides Rekon-

figurations Chart (siehe Grundlagen 2.4.4) kann dies noch nicht garantieren. Alleine die Koordination durch einen Konvoiführer, der auch die Werte-kontinuierlichen Vorgaben hinsichtlich Brems- und Beschleunigungssituationen initial vorgibt sowie bei dynamischen Änderungen zur Laufzeit diese neu verteilt, kann dies garantieren.

Durch das im Folgenden erweiterte Konvoibeispiel wird die Idee, wie diese Probleme durch die Modellierung mit MECHATRONIC UML gefasst werden können, beschrieben.

5.3 Erweitertes Beispiel

Das erweiterte Beispiel setzt auf dem Beispiel aus Abschnitt 5.1 auf. Es wird nun ein Konvoi der Länge n betrachtet (siehe Abbildung 5.3). Bei der Verhaltensmodellie-

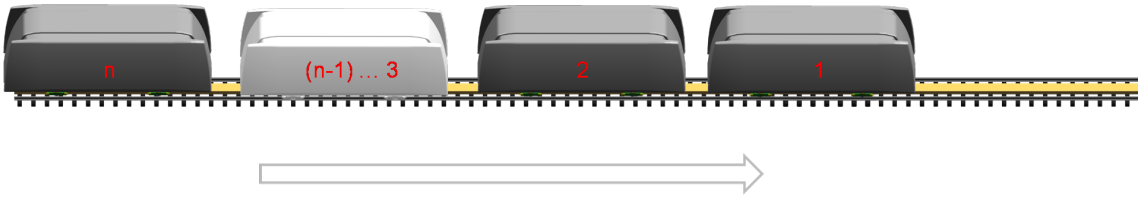


Abbildung 5.3: Konvoi der Länge n

rung für die Konvoibildung und -fahrt muss neben dem idealisierten fehlerfreien Fall auch der Ausfall einzelner Systemelemente betrachtet werden. Das modellierte Verhalten muss hierbei nicht tolerierbare Gefahren ausschließen. Durch Szenarien (siehe Echtzeit-Sequenzdiagramme, Abschnitt 2.4.2) wurden die folgenden in dem vorliegenden Anwendungsbeispiel identifizierten regulären Abläufe des Systems beschrieben:

(1) n-Shuttles fahren in einem Konvoi,

(2) Shuttle/Konvoi fährt auf ein weiteres Shuttle/Konvoi auf,

(3) n-Shuttles fahren unabhängig,

(4) Konvoi fährt mit Sicherheitsabstand hinter einem andern Konvoi und

(5) Auflösung eines Konvois in zwei unabhängige Konvois bzw. in n-Shuttles.

Anhand der Techniken zur Gefahrenanalyse, die von Tichy in seiner Arbeit vorgestellt werden [Tic08], konnten die in den Szenarien beobachteten Hazards:

(i) die Kollision von mehreren Shuttles,

(ii) die Kollision eines Shuttles mit einem Gegenstand oder

(iii) die Entgleisung eines Shuttles

genauer analysiert werden. In Abbildung 5.4 ist ein Ausschnitt eines Fehlerbaums dargestellt, der den Hazard *Kollision mehrerer Shuttles* genauer analysiert und beschreibt. Es ist zu sehen, dass entweder der *(a) Ausfall eines einzelnen Shuttles* oder der *(b) (partielle) Ausfall des Netzwerks* oder gar der *(c) Ausfall des Streckenstators* zu dem Hazard führen kann. Die primären Ereignisse, die jeweils die Ursache darstellen, werden hier nicht genauer beschrieben (grau dargestellt).

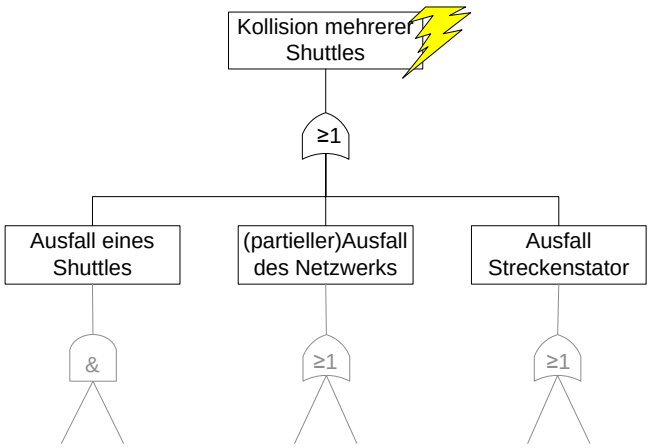


Abbildung 5.4: Fehlerbaum

Die Analyse sowie der Fehlerbaum sind nicht vollständig, sondern sollen hier nur andeuten, welches Fehlverhalten von dem Protokollverhalten bei einen Konvoi abgedeckt werden muss, um ein sicheres Konvoimanöver zu garantieren.

5.3.1 Lösungsidee

Um die in Abschnitt 5.2 aufgezählten und im Abschnitt 5.3 am Beispiel verdeutlichten Anforderungen adäquat für die Modellierung und Verifikation umzusetzen, wird im Folgenden eine Lösungsidee vorgestellt. Um die Komplexität auch hier zu beherrschen, wird das Zeit-kontinuierliche Verhalten getrennt vom Werte-kontinuierlichen Verhalten betrachtet. In Abbildung 5.5 ist die Idee der Dekomposition skizziert. Die obere Hälfte der Abbildung zeigt die Modellierung des Komponentenverbunds eines Shuttlekonvois. Jede Komponente kommuniziert mit ihrer Nachbarkomponente. In einer Komponente selber ist das interne, sowohl Zeit-kontinuierliche als auch Werte-kontinuierliche Verhalten, skizziert. Der untere Teil der Abbildung zeigt die Dekomposition des Modells. Der Komponentenverbund wurde in die Kommunikation und die Komponenten (siehe

Kompositioneller Ansatz), aufgeteilt. Weiterhin wurde auch das interne Verhalten dekomponiert. So ist zu erkennen, dass nun das Zeit-kontinuierliche Verhalten von dem Werte-kontinuierlichen Verhalten getrennt ist. Dies ermöglicht, wie schon beschrieben, eine getrennte Verifikation der einzelnen Verhalten, welches im Folgenden beschrieben wird.

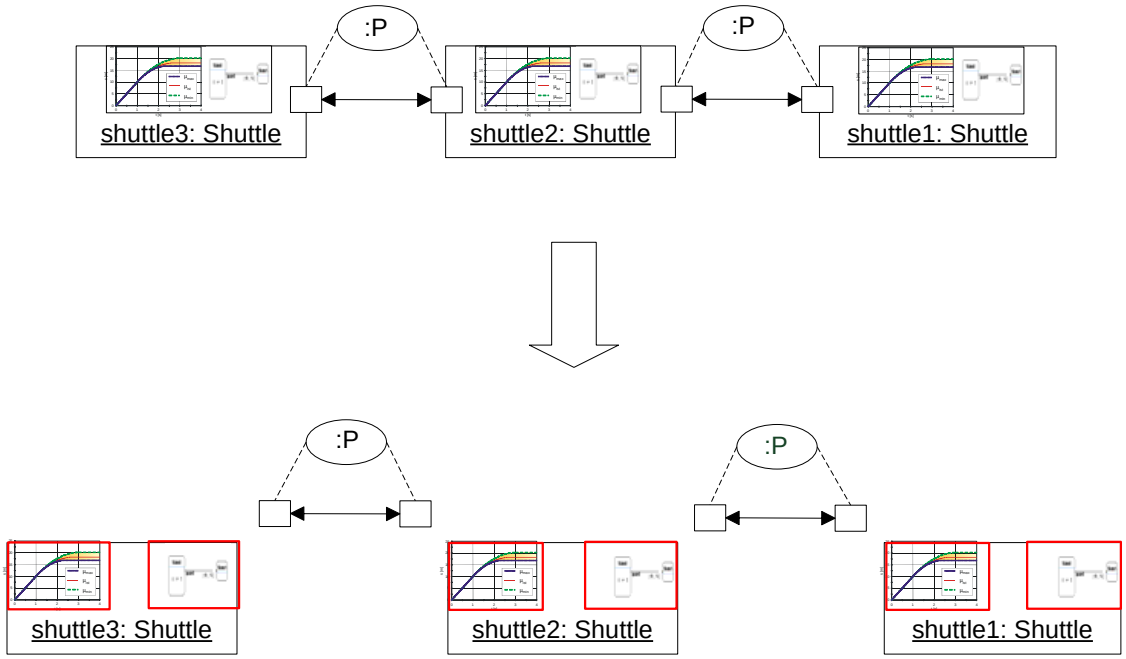


Abbildung 5.5: Dekomposition der Struktur

Zeit-kontinuierliche Verhalten: Um das Zeit-kontinuierliche Verhalten für eine beliebige Anzahl von gleichen Rollen zu modellieren, werden parametrisierte Rollen verwendet. Eine parametrisierte Rolle steht hierbei für eine Menge von Unterrollen, die sich untereinander synchronisieren können, um nach außen hin als eine Einheit aufzutreten. In Abbildung 5.6 ist dies schematisch dargestellt. M_{param} ist hierbei eine parametrisierte Rolle. Bei der Anwendung wird das Verhalten wie in der Abbildung dargestellt. Die parametrisierte Rolle wird quasi entfaltet. Die einzelnen Unterrollen koordinieren sich untereinander durch ausgezeichnete Signale. In dem Beispiel ist es das Signal next_i.

Eine Unterrolle kann als Struktur aufgefasst werden. Das Hinzufügen und das Löschen von einzelnen Unterrollen kann durch zeitbehaftete Graphtransformationssysteme beschrieben werden. Diese besitzen die Möglichkeit, strukturdynamische Änderungen mit Zeitbedingungen zu modellieren (siehe Kapitel 4). Die Integration der Graphtransformationssysteme geschieht nach dem von Klein [HHG08][Kle08] vorgestellten Ansatz.

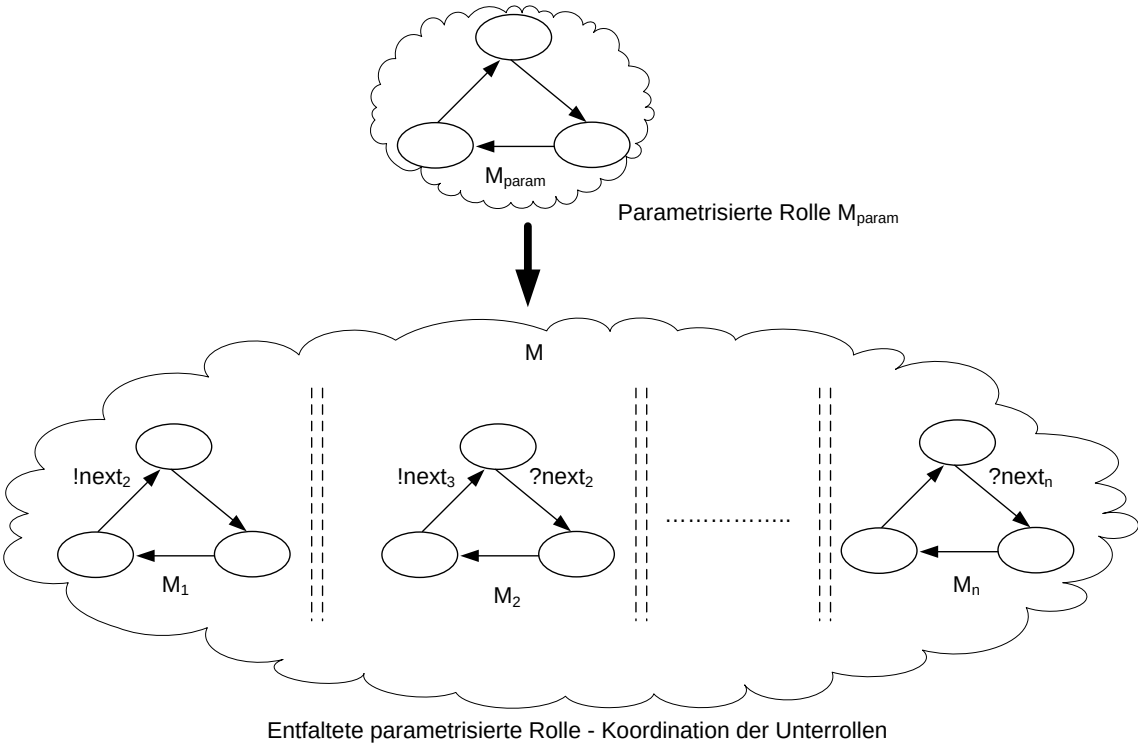


Abbildung 5.6: Parametrisierte Rolle mit zugehöriger Entfaltung und Koordination der Unterrollen

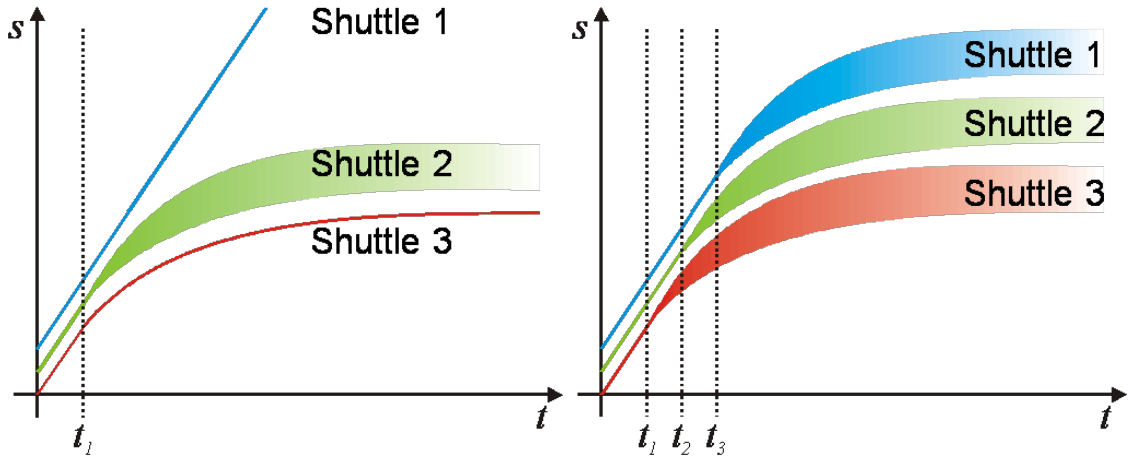
Hierbei wird ein gemeinsames Metamodell zur Integration beider Formalismen vorgeschlagen. Nachdem nun die Idee für die Modellierung des Zeit-kontinuierlichen Verhaltens skizziert wurde, wird die Lösungsidee für das Werte-kontinuierliche Verhalten vorgestellt.

Werte-kontinuierliche Verhalten: Basis der Lösungsidee sind Fahrprofile, die den Shuttles, die an einem Konvoi teilnehmen, von einem Leitfahrzeug zugeteilt werden. Im Normalbetrieb werden diese Fahrprofile ständig der aktuellen Situation angepasst und zwischen den Fahrzeugen kommuniziert. Ein solches Fahrprofil beinhaltet im Wesentlichen einen Bremskorridor, der dem jeweiligen Shuttle vorgibt, wie es sich in den verschiedenen betrachteten Gefahrensituationen zu verhalten hat. Maßgeblich wird ein Bremskorridor durch die physikalischen Eigenschaften eines Shuttles sowie durch die Position des Shuttles im Konvoi bestimmt. Um die Gefahr einer Kollision zu vermeiden, werden hier die diskutierten Ausfälle (a), (b) und (c) betrachtet.

Ein Ausfall eines Shuttles (Ausfallszenario (a)) und der partielle Ausfall des Netzwerks (Ausfallszenario (b)) ist Abbildung 5.7(a) zu entnehmen. In dem dort betrachteten Szena-

rio fällt Fahrzeug 2 aus. Die Fahrzeuge, die sich in dem Konvoi vor Fahrzeug 2 befinden, hier Fahrzeug 1, fahren weiter. Die Fahrzeuge, die sich hinter Fahrzeug 2 befinden, hier Fahrzeug 3, bremsen so stark wie möglich ab.

Abbildung 5.7(b) zeigt das Bremsverhalten bei einem Statorausfall. Um eine Kollision bei dem Bremsvorgang zu vermeiden, bremsen die Fahrzeuge zeitverzögert, wodurch die Bremskorridore disjunkt sind und damit keine Kollision auftreten kann.



(a) Bremsverhalten Netzwerkausfall Fahrzeug 2 (b) Bremsverhalten Statorausfall

Abbildung 5.7: Mögliches Bremsverhalten

Nachdem nun die Ideen skizziert wurden, werden im Folgenden detailliert der regelungstechnische Entwurf sowie der Softwaretechnische Entwurf zur Modellierung und Verifikation beschrieben.

5.3.2 Regelungstechnischer Entwurf

Ausgangspunkt für die hier betrachteten Überlegungen ist ein geregeltes Fahrzeug. Dabei müssen die zwei grundlegenden Fälle der Geschwindigkeitsregelung und der Abstands- bzw. Positionsregelung unterschieden werden. Ein einzelnes Fahrzeug bzw. das erste Fahrzeug im Konvoi soll sich mit einer vorgegebenen Geschwindigkeit v_{soll} bewegen. Die folgenden Fahrzeuge haben im Konvoi eine auf das Führungsfahrzeug bezogene Position einzunehmen und einen bestimmten Abstand zum direkt vorausfahrenden Fahrzeug einzuhalten. Um die Regelungen für die beiden beschriebenen Fälle auszulegen, wurde zunächst ein Fahrzeug als Starrkörper mit dem Entwicklungswerkzeug CAMeL-View [Ric96] modelliert, wobei nur die Längsdynamik berücksichtigt wird. Für die Geschwindigkeitsregelung genügt hier ein einfacher PI-Regler mit Anti-Windup (Abbildung 5.8),

um Probleme durch den Integratoranteil zu vermeiden, die sich aus der Begrenzung der Antriebskraft ergeben. Der Linearantrieb ist im Modell vereinfacht durch ein Verzögerungsglied erster Ordnung mit nachgeschaltetem Begrenzer abgebildet.

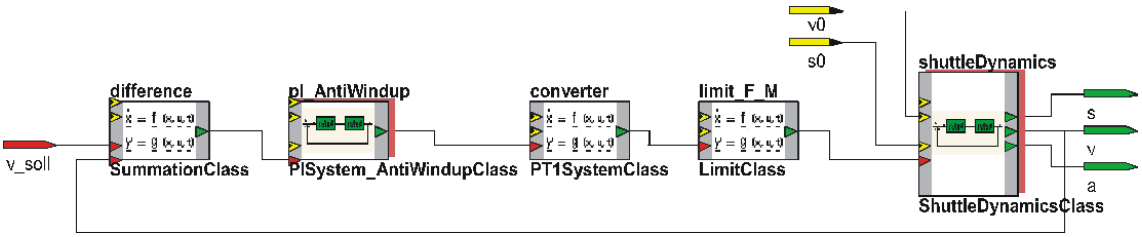


Abbildung 5.8: CAMeL-View-Modell eines geschwindigkeitsgeregelten Fahrzeugs

Dieser Geschwindigkeitsregelung ist für die Konvoifahrzeuge ein Abstands- bzw. Positionsregler überlagert, der einerseits die Position des Fahrzeugs bezogen auf die aktuelle Position des führenden Fahrzeugs regelt, andererseits den Abstand und die Differenzgeschwindigkeit zum direkt vorausfahrenden Fahrzeug berücksichtigt, um Kollisionen ausschließen zu können [HVB+05]. Durch diese Regelungsstrategie kann die Stabilität eines Konvois auch für längere Konvois garantiert werden. Allerdings ist dafür die Kommunikation jedes Fahrzeugs mit dem direkt vorausfahrenden Fahrzeug und dem Leitfahrzeug notwendig. Die erforderliche Kommunikationsstruktur ist in Abbildung 5.9 dargestellt. Sie gliedert die Informationsverarbeitung anhand der im Grundlagenkapitel vorgestellten Strukturierung mechatronischer Systeme (siehe Abschnitt 2.1) hier in Autonome Mechatronische Systeme (AMS), nämlich den einzelnen Fahrzeugen, und Vernetzte Mechatronische Systeme (VMS), den gesamten Konvoi. Die hier vorgeschlagene Regelung eines Konvois wurde bereits erfolgreich im RailCab-Projekt umgesetzt und in der Praxis erprobt [HTBS08]. Die vorgestellte Regelung geht von idealisierten Bedingungen unab-

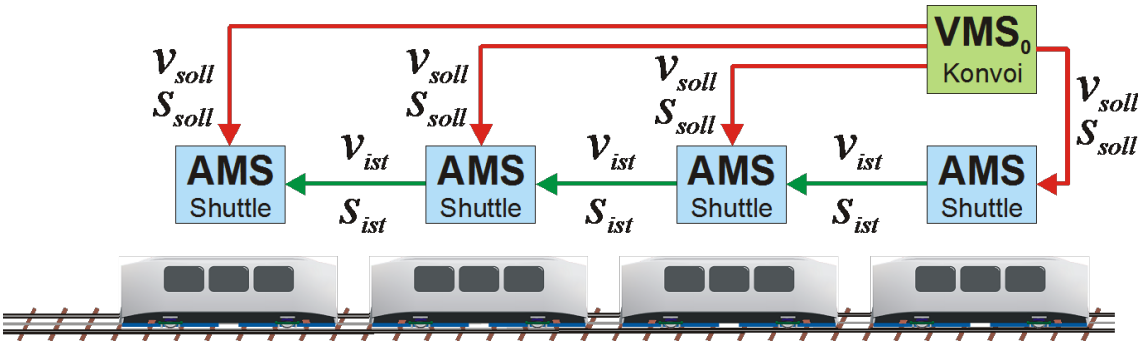


Abbildung 5.9: Struktur der Informationsverarbeitung im Konvoi

hängig von äußeren Einflüssen, Unterschieden in der Fahrzeugdynamik und den eingangs erwähnten möglichen Fehlern aus. Um den kollisionsfreien Betrieb des realen Systems

gewährleisten zu können, sind also zusätzliche Überlegungen erforderlich. Der hier vorgestellte Ansatz zielt darauf ab, gewisse Grenzen für das Verhalten eines einzelnen Fahrzeugs nachzuweisen, die das geregelte System mit Sicherheit nicht überschreitet.

Wir betrachten hier als mögliche Fehler (a) den Ausfall des Antriebsmotors eines Shuttles, (b) den Kommunikationsausfall und (c) den streckenseitigen Ausfall des Motors (Statorausfall). Tritt einer dieser Fehler auf, müssen die betroffenen und alle nachfolgenden Fahrzeuge bis zum Stillstand abgebremst werden[1]. Für das Bremsen stehen drei Möglichkeiten zur Verfügung. Die notwendige Bremskraft kann über den Linearantrieb erzeugt werden. In diesem Fall kann auf ein vorgegebenes Geschwindigkeitsprofil zurückgegriffen werden, das kontrolliertes Bremsen ermöglicht. Zusätzlich existiert eine mechanische Notbremse, die über Federn Bremsklötze direkt auf die Schienen drückt. Außerdem können beide Bremsen gleichzeitig eingesetzt werden. Je nach auftretendem Fehler stehen allerdings nicht alle drei Möglichkeiten zur Verfügung. Fällt der Linearantrieb strecken- oder fahrzeugseitig aus, kann nur die mechanische Notbremse eingesetzt werden. Dieser Fall soll beispielhaft für die Vorausberechnung von Grenzen betrachtet werden, die das System unter Berücksichtigung der Modellunsicherheiten einhält.

Der tatsächliche Bremsweg hängt bei den mechanischen Notbremsen im Wesentlichen von den Fahrzeugeigenschaften wie Masse und aktuelle Geschwindigkeit und dem Reibkoeffizienten μ ab. Da dieser nicht als exakt bekannt vorausgesetzt werden kann, muss man auf Minimal- bzw. Maximalwerte zurückgreifen. Ein mechanisch gebremstes Fahrzeug wird dann in einem Bereich zwischen dem minimal und maximal möglichen Bremsweg zum Stehen kommen. In Abbildung 5.10 ist das Ergebnis einer Simulation dieser Situation dargestellt, bei der der Reibkoeffizient während des Bremsvorgangs bei $t = 1,7s$ sprungförmig abnimmt. Das Fahrzeug bewegt sich innerhalb der für die Position und Geschwindigkeit vorausberechneten Korridore für μ_{min} und μ_{max}, die ohne exakte Kenntnis des tatsächlichen Reibkoeffizienten angegeben werden können. Für Bremsvorgänge mit dem Linearantrieb lassen sich diese Korridore noch genauer vorhersagen, da dann das Bremsen vom Geschwindigkeitsregler kontrolliert nach einem vorgegebenen Profil abläuft. Auf diese Weise kann jedes Fahrzeug vorausberechnen, in welchen Grenzen es sich im Fall einer der drei möglichen Notbremsungen bewegen wird. Darauf aufbauend lässt sich überlagert das Verhalten der Fahrzeuge im Konvoi und im Fehlerfall modellieren und mit den im Folgenden beschriebenen Verfahren verifizieren.

[1] Bei Kommunikationsausfall sind auch andere kontrollierte Manöver denkbar, z.B. das autonome Fahren mit vergrößertem Abstand mit Hilfe von Abstandssensoren. Diese Betrachtungen ändern nichts an dem vorgestellten Ansatz zur Verifikation des sicheren Verhaltens und werden deshalb nicht näher betrachtet.

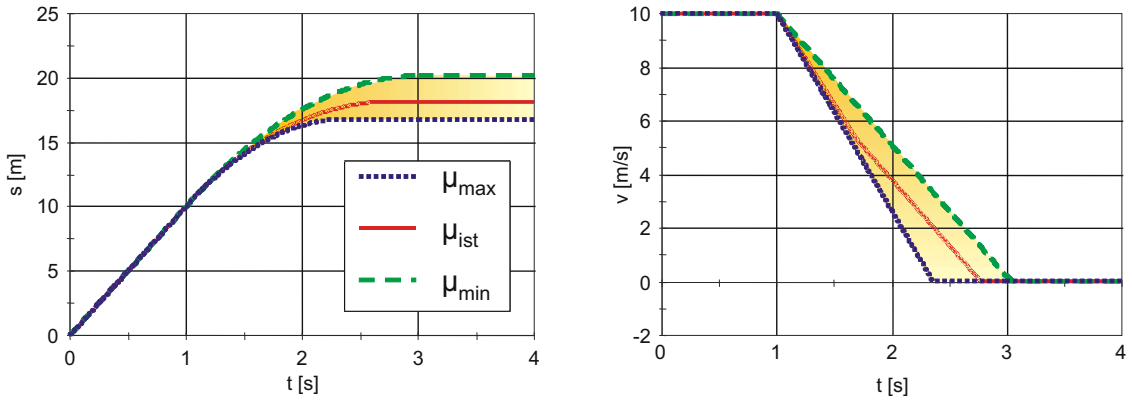


Abbildung 5.10: Bremskorridore bei einer mechanischen Notbremsung

5.3.3 Softwaretechnische Umsetzung

Um nun die neuen Anforderungen, beliebige Anzahl von Rollen sowie Werte-kontinuierliches Reglerverhalten zu integrieren, wird das Konzept der modell-basierten Entwicklung mit Echtzeit-Koordinationsmustern erweitert.

Hierzu werden zum einen Abstraktionstechniken eingesetzt, die auf den zugesicherten Eigenschaften der Regler aufbauen (siehe letzter Abschnitt). Um eine Verifikation durchführen zu können wird neben Model Checking das Verfahren der Induktion eingesetzt, um Verhaltenseigenschaften über die Struktur zu beweisen. Dies wird im Folgenden angedeutet. Kontinuierliche Systeme besitzen einen unendlichen Zustandsraum. Dieses macht eine formale Verifikation durch Model Checking alleine unmöglich, da hier ein endlicher Zustandsraum benötigt wird. Um dennoch Werte-kontinuierliches, regelungstechnische Verhalten zu verifizieren, im vorliegenden Fall kontinuierliches Beschleunigungsverhalten, wird dieses durch Abstraktionstechniken auf ein endliches, diskretes System abgebildet. Der Abstraktion liegen die zugesicherten Eigenschaften des Reglers zu Grunde. Dadurch ist es möglich, das Verhalten dieser einzuschränken. In dem Beispiel wird das Verhalten durch so genannte Bremskorridore beschrieben. Diese geben das minimale und maximale Beschleunigungsverhalten eines Shuttles an. Hierdurch ist es möglich, bei der Modellierung das Verhalten der Shuttles weiterhin durch diskrete Zustände zu beschreiben und die formale Verifikation durch Model Checking durchzuführen. Durch z.B. numerische Überprüfung [Pra05][PJ04][PP05] muss nun vorab verifiziert werden, ob sich die Bremskorridore schneiden oder nicht. Diese Überprüfung kann auch zur Laufzeit bei Neuverteilung der Profile effizient durchgeführt werden.

Durch geeignete Modellierung des Kommunikationsprotokolls werden diese Eigenschaften im System umgesetzt. Um mit der beliebigen Anzahl n zurecht zu kommen, werden hierfür parametrisierte Rollen verwendet. Jedem Shuttle wird ein so genanntes Fahrprofil

p_i zugeordnet. Dieses beinhaltet u.a. das Verhalten, wie es sich in Notfallsituationen zu verhalten hat. Insgesamt gibt es n Fahrprofile. Für die Fahrprofile gilt die Eigenschaft, dass p_i immer eine Notfallreaktion in einem Konvoi garantiert, die ein Shuttle mit Fahrprofil p_j nicht in Gefahr bringt, wobei $i <= j$, gelten muss. In Abbildung 5.7(b) hat das vorausfahrende Shuttle 1 das Fahrprofil p_2 und Shuttle 2 und Shuttle 3 das Fahrprofil p_1. Dieses Verhalten wird formal über die Struktur des Modells (Konvoiteilnehmer) mittels Induktion bewiesen. Da das Hinzufügen und Löschen von Unterrollen durch zeitbehaftete Graphtransformationssysteme beschrieben wird, kann zur Verifikation der Ansatz aus Kapitel 4 herangezogen und mit Model Checking geschickt verknüpft werden.

Abbildung 5.11 skizziert die Struktur des ConvoyCoordinator. Die gestrichelten Pfeile deuten die Abarbeitungsreihenfolge der beteiligten Rollen an, welche durch das Protokoll realisiert werden muss.

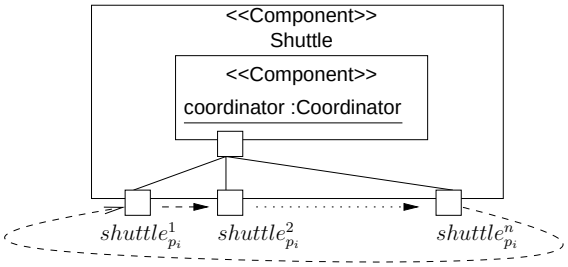


Abbildung 5.11: Modellierung und Koordination eines multi-Ports – jedem Port und damit Shuttle wird eine Eigenschaft p_i zugeordnet

In Abbildung 5.12 ist in einem Sequenzdiagramm beispielhaft die Konvoikommunikation für drei Shuttles modelliert. Hierbei ist der Coordinator als einzelne Komponente modelliert. Diese kann sich aber als Unterkomponente auf dem Führungsshuttle befinden. Es ist zu sehen, dass sich zuerst alle Shuttles bei dem Coordinator registrieren. Der Coordinator berechnet daraufhin eine gültige Profilreihenfolge für die Shuttles und weißt diese entsprechend zu. Periodisch wird nun die Erreichbarkeit aller Shuttles überprüft. Wenn die Erreichbarkeit ausbleibt, also ein potentieller Netzwerkausfall vorliegt, werden von den einzelnen Shuttles die entsprechenden Notfallroutinen gefahren, die durch das Profil, das hierfür das regelungstechnische Verhalten vorgibt, bestimmt ist.

5.4 Parametrisierte Koordinationsmuster

In diesem Abschnitt wird das Konzept der neuen *parametrisierten Koordinationsmuster* beschrieben. Hier werden die eben beschriebenen Modellierungskonzepte integriert.

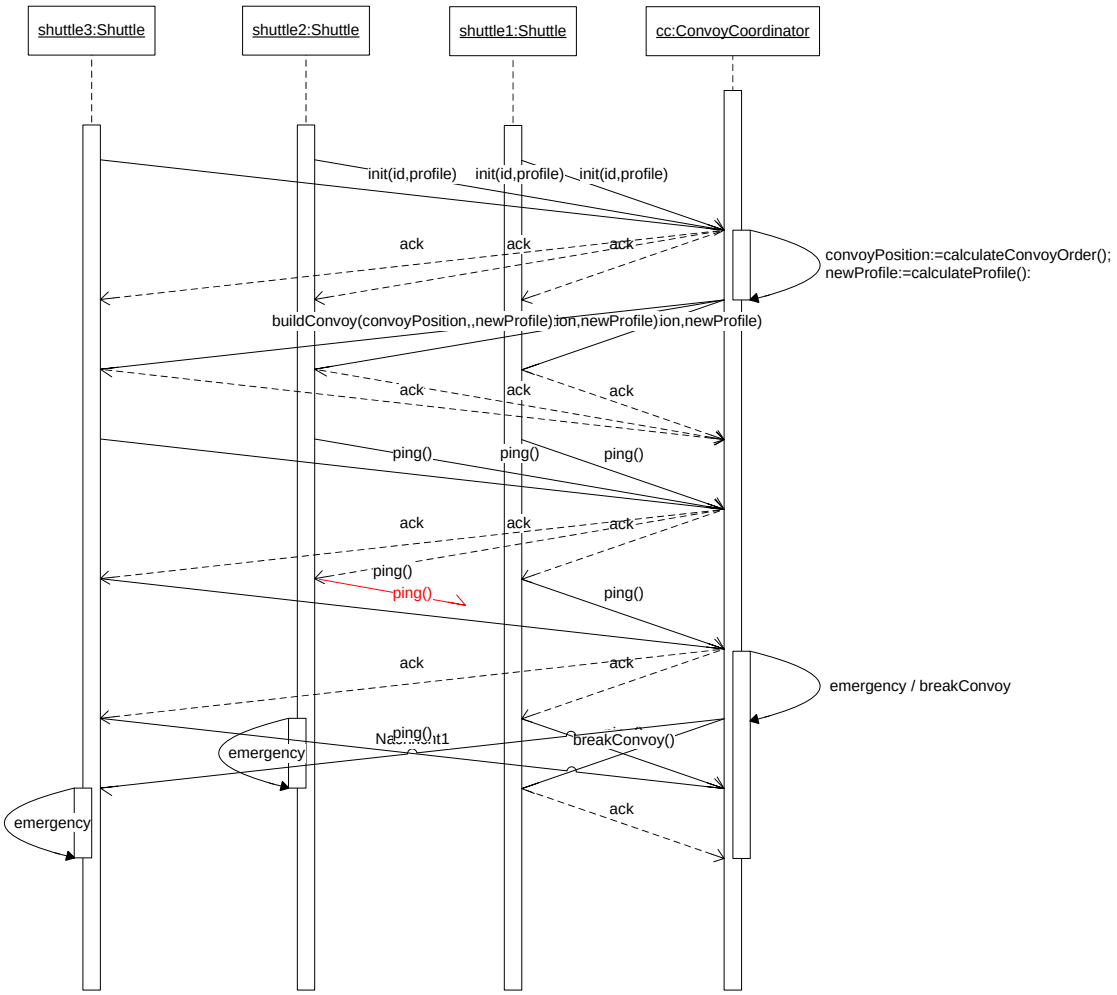


Abbildung 5.12: Beispielkommunikation für 3 Shuttles im Konvoi mit Netzwerkausfall

5.4.1 Informale Beschreibung

Genau wie die einfachen Echtzeit-Koordinationsmuster bestehen die *parametrisierten Koordinationsmuster* aus Rollen und Connectoren. Dabei richtet sich die grundsätzliche Idee der parametrisierten Koordinationsmuster nach den in der UML (siehe [OMG07], Seite 168ff) beschriebenen Collaborations. In der UML werden die Collaborations verwendet, um die dynamischen Beziehungen zwischen Rollen zu beschreiben. Jedoch ist dies auf die reine Software bezogen, so dass kontinuierliche Aspekte (Stabilität) sowohl auf der Ebene der Struktur als auch bei den Constraints, nicht berücksichtigt werden.

Abbildung 5.13 zeigt ein *parametrisiertes Koordinationsmuster* für die Konvoikoordination von n Shuttles. Ein optionales Modellelement sind die *multi-Rollen*. Multi-Rollen

werden durch überlappende Quadrate dargestellt und besitzen eine Kardinalität n. Multi-Rollen sind eine vereinfachte Darstellung für eine Menge von gleichen Rollen, die zum gleichen Typ einer Komponente gehören. Im vorliegenden Beispiel der Konvoikoordination besitzt die Rolle shuttle die Kardinalität 1 und die coordinator Rolle die Kardinalität n. Dieses bedingt eine eindeutige Zuordnung jeder „einfachen Rolle“ einer multi-Rolle zu einem eindeutigen Gegenpart, ähnlich wie bei einer 1 zu n Assoziation in Klassendiagrammen. Weiterhin ist es möglich, multi-Rollen mit Attributen wie {ordered} zu versehen. Hierdurch wird eine Reihenfolge für die Auswertung der Rollen festgelegt. Weiterhin hat jedes *parametrisierte Koordinationsmuster* Constraints, die das Verhalten, besonders das der Profileigenschaften, beschreiben. Das Constraint $\forall_{k,l \wedge k<l}(S^k_{p_i}, S^l_{p_j}) \Rightarrow |i-j| \leq 1$ beschreibt die Eigenschaft, dass die Profilreihenfolge zweier benachbarter Shuttles immer monoton ist und sich die Profile in ihrem Index niemals um mehr als 1 unterscheiden.

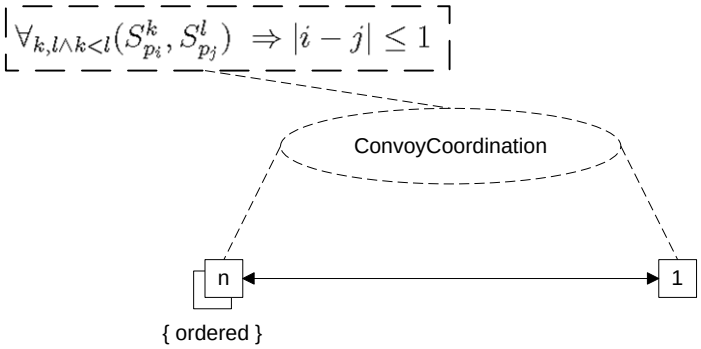


Abbildung 5.13: Parametrisiertes Koordinationsmuster

In Abbildung 5.14 ist der Typ Shuttle dargestellt. Die Komponente bettet zwei Unterkomponenten Coordinator und VelocityControl ein. Das Verhalten der Komponenten ist wie folgt definiert. Da die Komponente das parametrisierte Koordinationsmuster ConvoyCoordination anwendet, muss sie sowohl als coordinator als auch als shuttle fungieren. Der multi-Port ist durch eine Assembly mit der inneren Komponente Coordinator und VelocityControl verbunden, um das obige Verhalten zu realisieren. Die flache Komponente Coordinator realisiert die Berechnung sowie die Abspeicherung aller Profile im Führungsshuttle. Bei der Berechnung der Profile wird die aktuelle Geschwindigkeit benötigt. Die Komponente VelocityControl bettet eine kontinuierliche Reglerkomponente ein, welche diese Daten kontinuierlich zur Verfügung stellt.

In Abbildung 5.15 ist die Laufzeit-Instanz von zwei Shuttles dargestellt, welche das parametrisierte Koordinationsmuster ConvoyCoordination anwenden.

Das Verhalten bestimmt sich, wie in der Lösungsidee 5.3.1 beschrieben, sowohl durch zeitbehaftete Graphtransformationssysteme als auch durch parametrisierte Automaten. Dieses wird im Folgenden beschrieben.

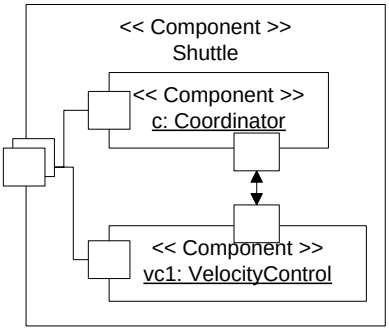


Abbildung 5.14: Der Typ Shuttle

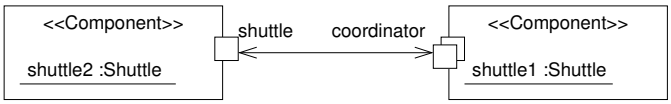


Abbildung 5.15: Laufzeit-Instanz zweier Shuttles, welche das parametrisierte Koordinationsmuster ConvoyCoordination anwenden

5.4.2 Modellierung des Verhaltens eines parametrisierten Koordinationsmusters

5.4.2.1 Verhalten der Rollen

Wie schon in der im Abschnitt 5.3.1 skizzierten Lösung beschrieben werden, um die multi-Rollen zu beschreiben, parametrisierte Automaten verwendet.

Die neue parametrisierte Rolle *coordinator* des parametrisierten Koordinationsmusters ist in Abbildung 5.16 dargestellt. Die Erweiterungen hinsichtlich der Parameter beziehen sich auf die Synchronisationskanäle next_k und $\mathsf{nextFailed}_k$. Über diese Kanäle synchronisieren sich die n Rollen untereinander. Die Rolle befindet sich initial im Zustand WaitForTrigger. Empfängt die Rolle das next_k Triggersignal (initial vom Synchronisationsstatechart und danach von der Vorgängerrolle), schaltet die Rolle in den Zustand Idle. Nun beginnt der Kommunikationsaustausch mit der shuttle Rolle. Es wird ein update Signal mit dem aktuellen Profil, welches das Shuttle annehmen soll, sowie der Bezugsgeschwindigkeit und der Bezugsposition verschickt. Danach wartet die Rolle in dem Zustand SentAcknowledge auf die Bestätigung acknowledge des Gegenparts, der Rolle shuttle. Wird diese empfangen, schaltet die Rolle wieder in den Zustand WaitForTrigger und sendet dabei das next_{k+1} Signal, um die nächste Rolle zu aktivieren. Empfängt die Rolle kein acknowledge, schaltet die Rolle in den Zustand NextFailed und signalisiert dies dem Synchronisationsstatechart, um geeignete Routinen zu aktivieren. Weiterhin be-

sitzt die Rolle den Zustand StatorFailure. Dieser wird vom Zustand Idle erreicht, falls die zugehörige shuttle Rolle dieses Signal propagiert.

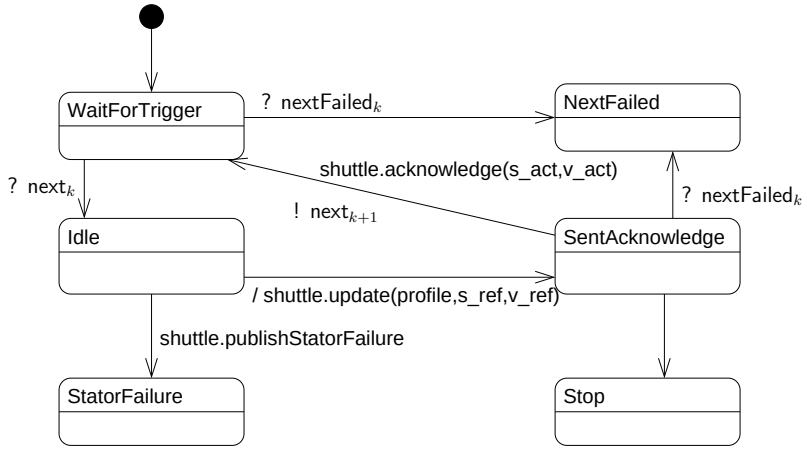


Abbildung 5.16: Das Verhalten einer parametrisierten Rolle coordinator

Die Rolle shuttle (siehe Abbildung 5.17) besteht aus drei Zuständen. Initial befindet sie sich im Zustand Normal. Spätestens alle 150 Zeiteinheiten muss die Rolle ein update Signal von der zugehörigen coordinator Rolle empfangen. Dies ist durch eine Selbsttransition am Zustand realisiert. Das Signal beinhaltet das aktuelle Profil profile, die Bezugsposition und die Bezugsgeschwindigkeit. Die Rolle bestätigt den Erhalt des Signals durch ein acknowledge Signal, welches die aktuelle Position und die aktuelle Geschwindigkeit enthält. Falls kein update Signal empfangen wird (z.B. Ausfall des Netzwerks), schaltet die Rolle nach 150 Zeiteinheiten in den Zustand NetworkFailure. Der Zustand StatorFailure wird nicht-deterministisch erreicht. Dabei wird der Fehler dann an die zugehörige coordinator Rolle propagiert.

Nachdem nun die Rollen modelliert sind ist die Frage, wie sich diese in die Architektur integrieren lassen. In Abbildung 5.18 ist hierfür eine hierarchische Architektur, die sich nach dem kompositionellen Ansatz aus [GTB+03] richtet, vorgeschlagen. Hierbei wird vorgeschlagen, eine zusätzliche Koordinationsschicht einzufügen. Diese Koordinationsschicht beinhaltet einen weiteren Automaten, der es ermöglicht, die multi-Ports mit dem eigentlichen Synchronisationsstatechart zu verbinden. Falls nur ein einfacher Port existiert, ist ein solcher Automat nicht notwendig.

Eine Shuttle Komponente, welche das parametrisierte Koordinationsmuster ConvoyCoordination anwendet, muss sowohl als coordinator als auch als shuttle agieren. Ein Ausschnitt des Synchronisationsstatechart, dass beide Rollen triggern kann, je nachdem, in welcher sich ein Shuttle befindet, ist in Abbildung 5.19 dargestellt. Die Entscheidung, ob ein Konvoi erzeugt werden soll und welche Rolle das Shuttle einnimmt, wird vorher durch den kognitiven Operator (siehe Kapitel 2.1) bestimmt, der Informationen z.B. von einer

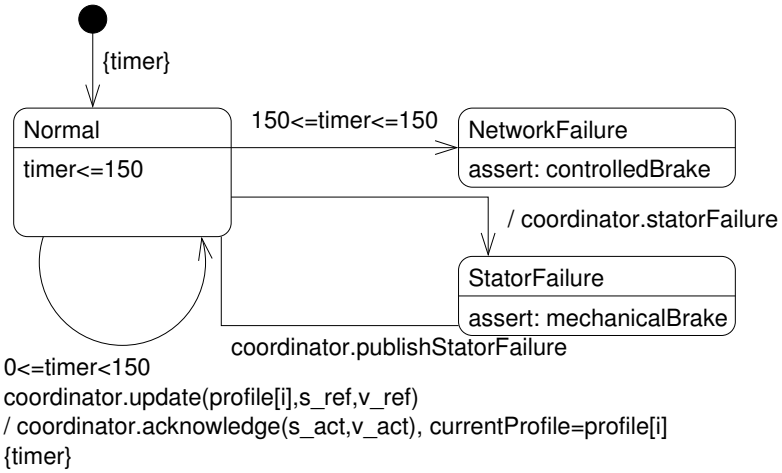


Abbildung 5.17: Das Verhalten der Rolle shuttle

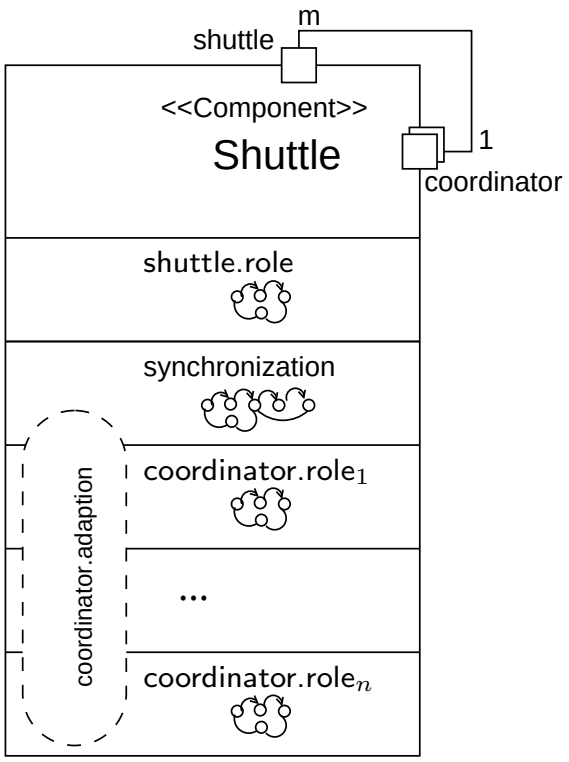


Abbildung 5.18: Hierarchische Architektur

Schienenabschnittskontrolle über die Position und Reihenfolge der Shuttles bekommen hat. Die Signale ?convoyUseful, ?shuttle und ?coordinator werden entsprechend getriggert. Anschließend initiiert das jeweilige Synchronisationsstatechart die entsprechenden

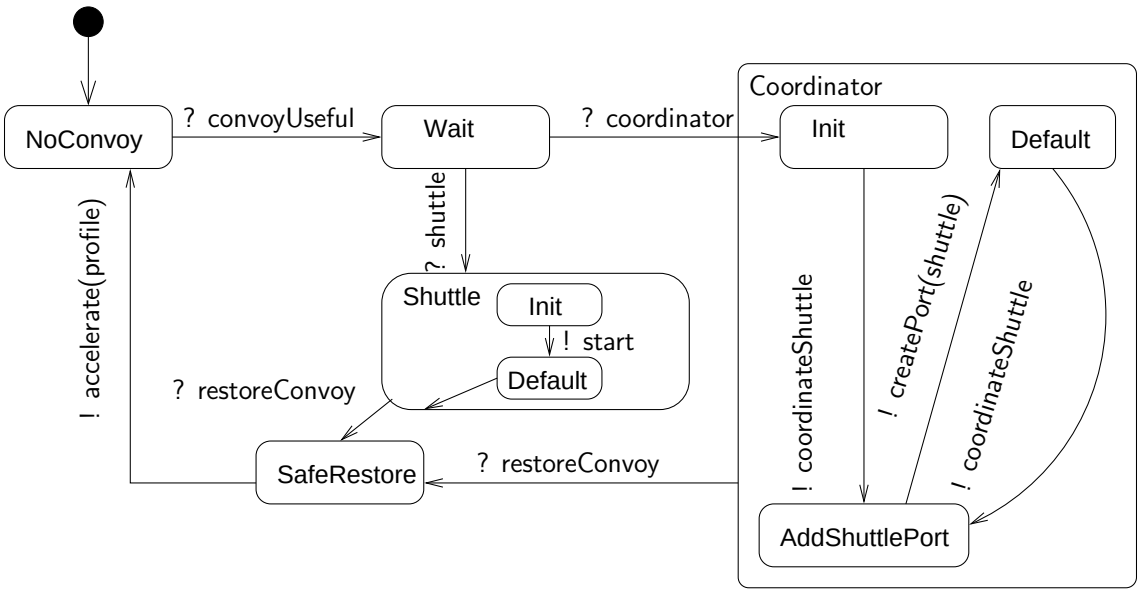


Abbildung 5.19: Synchronisationsstatecharts der Komponente Shuttle

Rollen, indem es das Signal start an die Rolle shuttle bzw. das Signal coordinateShuttle an die Rolle coordinator sendet.

Ein Konvoi wird aufgehoben, wenn ein Fehler festgestellt wurde. Diese werden von den Rollen festgestellt, wenn z.B. ein Signal nicht in der vorgegebenen Zeit angekommen ist. Das Synchronisationsstatechart beginnt dann, nach der vorgegebenen Profil zu fahren. Die Wiederherstellung und weitere Details werden hier nicht betrachtet.

Der neue Automat, in der Abbildung 5.18 als coordinator.adaptation bezeichnet, wird durch das Synchronisationsstatechart getriggert. Danach hat es die Möglichkeit, die Aktion CreatePort(shuttle), getriggert durch das Signal createPort, eine neue Rolle der multi-Rolle hinzuzufügen. Die neue Rolle wird entsprechend der Strukturregeln (siehe folgender Abschnitt 5.4.3) angelegt. Hierbei wird auch entsprechend der Parameter k inkrementiert. Dieser gibt Auskunft über die Anzahl der Shuttles und somit ist es möglich, immer die nächste Rolle eines multi-Ports, entsprechend der vorgegebenen Eigenschaft ordered, anzusprechen (siehe Abbildung 5.20).

In Abbildung 5.21 ist die verfeinerte shuttle Rolle dargestellt. Um dem Verhalten des Synchronisationsstatecharts zu genügen, konsumiert es das Signal start. Falls ein Fehler entdeckt wird, propagiert die Rolle ein entsprechendes Signal an das Synchronisationsstatechart.

Nachdem nun das Verhalten komplett modelliert ist, ist die Frage, wie sich die dynamischen Strukturänderungen zur Laufzeit modellieren lassen.

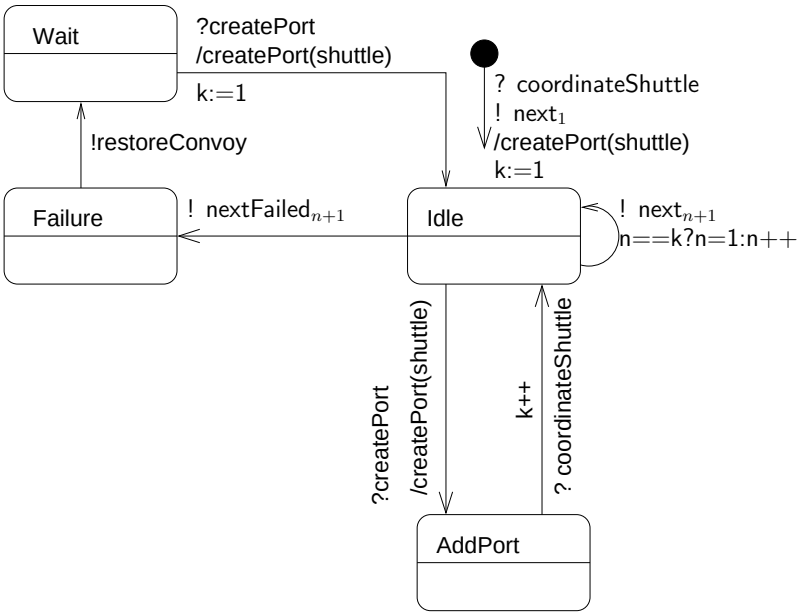


Abbildung 5.20: Koordinationsstatechart für die multi-Rolle

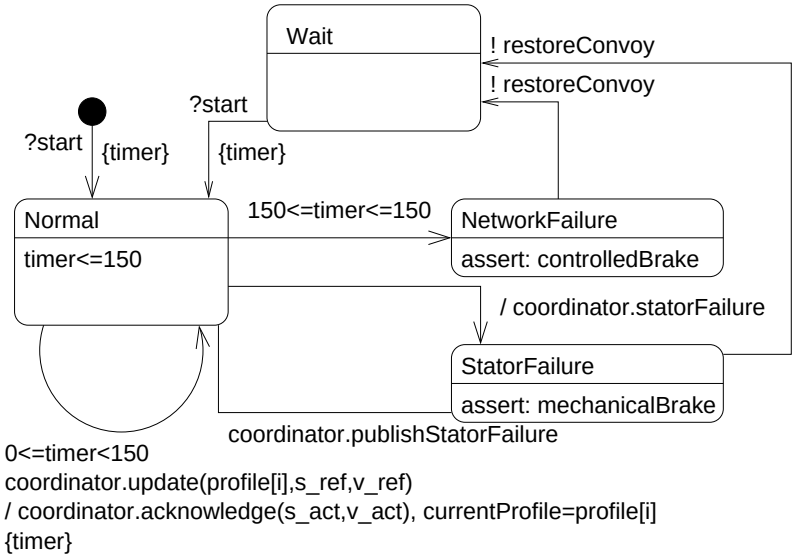


Abbildung 5.21: Verfeinerte shuttle Rolle

5.4.3 Modellierung der dynamischen Strukturänderungen

Im Folgenden werden die Regeln, welche die erlaubten Strukturänderungen des parametrisierten Koordinationsmusters angeben (siehe 5.3.1), beschrieben. Die erlaubten Strukturregeln werden in das Verhalten integriert, wie in [HHG08] beschrieben. Um die Re-

geln zu strukturieren, werden sie in *Erweiterungsregeln* und *Reduzierungsregeln* aufgeteilt. Alle möglichen strukturellen Rekonfigurationsschritte für ein parametrisiertes Koordinationsmuster werden durch zeitbehaftete Graphtransformationsregeln (siehe Kapitel 4) beschrieben.

5.4.3.1 Erweiterungsregeln

In Abbildung 5.22 ist die initiale Regel, die erste Erweiterungsregel, welche das parametrisierte Koordinationsmuster für zwei Shuttles anwendet, dargestellt. Damit ist ein eindeutiger Startgraph festgelegt. Wenn das parametrisierte Koordinationsmuster das erste Mal angewendet wird, wird bei dem Führungsshuttle ein multi-Port angelegt. Das hinterherfahrende Shuttle bekommt einen einfachen Port. Die Ports werden durch einen Connector verbunden. Die Zeitbedingung $t > 5$ (siehe Kapitel 4) gibt an, dass dies mindestens 5 Zeiteinheiten benötigt. Um auch eine Obergrenze für diese Aktion festzulegen, ist in Abbildung 5.23 eine Invariantenregel (siehe Kapitel 4) dargestellt, die dem Graphtransformationssystem hinzugefügt wird. Weiterhin wird dem Connector der Stereotyp ≪last≫ hinzugefügt. Dieser gibt an, dass das hinterherfahrende Shuttle das letzte im Konvoi ist. Damit ist markiert, an welcher Stelle neue Shuttles dem Konvoi beitreten können.

Natürlich muss bei der Anwendung des parametrisierten Koordinationsmusters auch das interne Verhalten sowie die interne Komponentenstruktur eines Shuttles rekonfiguriert werden. Dies ist in der initialen Regel 5.22 ebenfalls angedeutet. Hier wird bei der Erzeugung des parametrisierten Koordinationsmusters im Führungsshuttle entsprechend die Komponente Coordinator aktiviert. Wie die Erzeugung solcher Komponenten in das Verhalten von hybriden Rekonfiguration Charts integriert werden kann, ist in [Krä06] beschrieben.

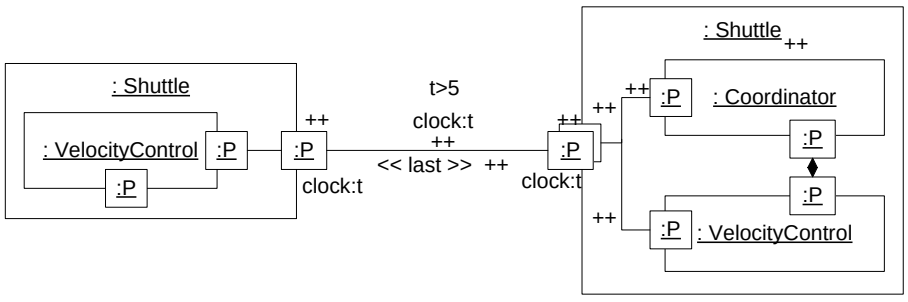


Abbildung 5.22: Initiale Regel zur Anwendung des parametrisierten Koordinationsmusters

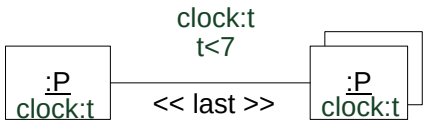


Abbildung 5.23: Regel zur Erzeugung einer Zeitinvariante

Bei dem parametrisierten Koordinationsmuster gibt es noch eine weitere Erweiterungsregel (siehe Abbildung 5.24). Die zweite Regel beschreibt das Auffahren eines Shuttles auf einen bereits existierenden Konvoi (siehe Abbildung 5.24). Hierbei wird bei dem neu hinzukommenden Shuttle ein shuttle-Port erzeugt. Weiterhin wird ein Connector zum Führungsfahrzeug erzeugt. Der Stereotyp ≪last≫ wird entsprechend vom alten Connector gelöscht und an den neu erzeugten Connector gebunden, um die letzte Position neu zu markieren. Die Instanzsituation ist in Abbildung 5.25 dargestellt.

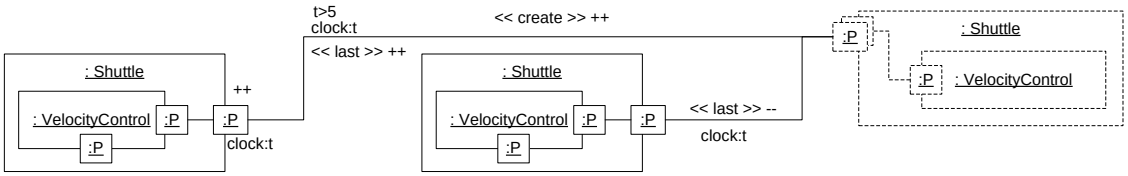


Abbildung 5.24: Ein Shuttle reiht sich hinten in den Konvoi ein

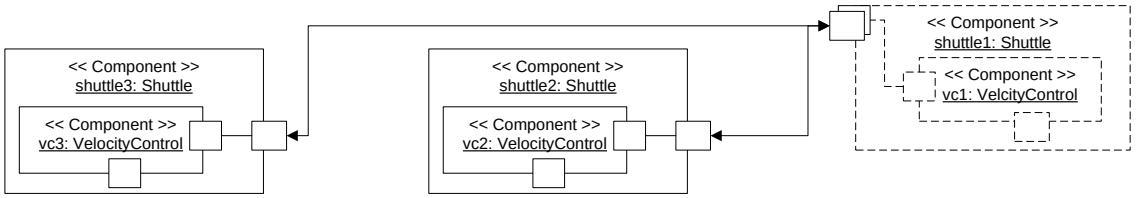


Abbildung 5.25: Instanzsicht nach der Anwendung von Regel aus Abbildung 5.24

5.4.3.2 Reduzierungsregeln

Um das Auflösen eines Konvois zu beschreiben, werden zusätzlich Reduzierungsregeln benötigt. In Abbildung 5.26 ist dargestellt, wie das letzte Shuttle eines Konvois diesen verlässt. Dabei werden die Ports vernichtet und der Connector ebenfalls. Um das neue letzte Shuttle zu markieren, wird nun der Stereotyp ≪last≫ an den Connector des vorherfahrenden Shuttles gebunden.

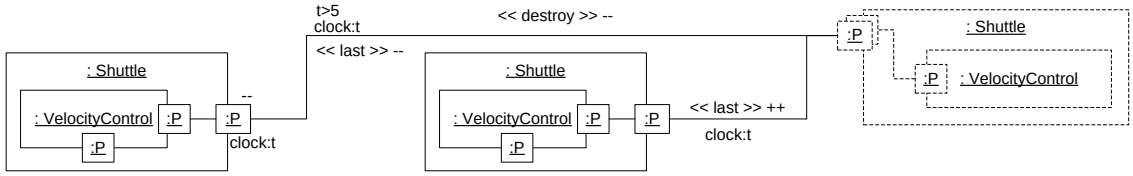


Abbildung 5.26: Letztes Shuttle verlässt den Konvoi

Falls der Konvoi nur noch aus zwei Teilnehmern besteht, muss das parametrisierte Koordinationsmuster entsprechend deinstanziiert werden. Bevor dies jedoch geschieht, muss die Regel aus Abbildung 5.27 angewendet werden. Hierbei wird die Komponente Coordinator im Führungsshuttle deaktiviert. Ebenfalls wird der Connector und die beteiligten Ports vernichtet.

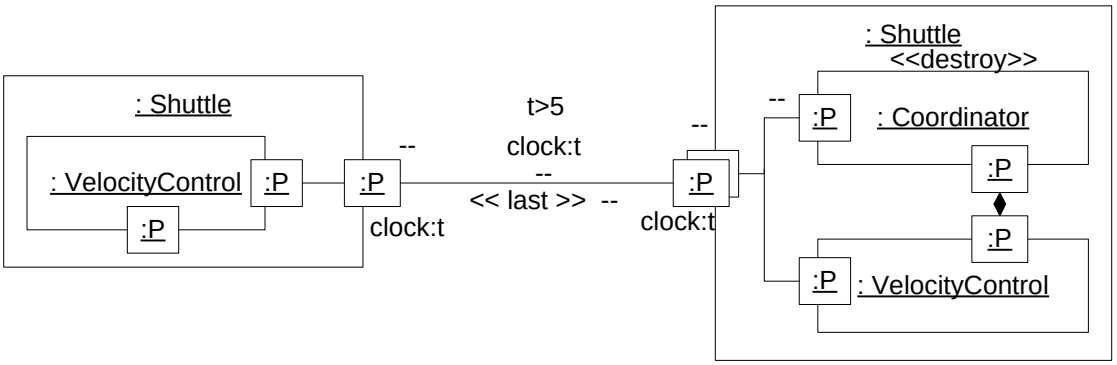


Abbildung 5.27: Konvoi der Länge 2 wird aufgelöst

Als letztes wird eine Regel dafür angegeben, dass das Führungsshuttle den Konvoi verlässt. Hierbei muss die Unterkomponente Coordinator des Führungsshuttles an das hinterherfahrende Shuttle übertragen werden. Ausserdem muss der multi-Port vernichtet werden. Das hinterherfahrende Shuttle hingegen muss seine einfache shuttle Rolle nun in einen multi-Port coordinator umwandeln. Die Regel ist in Abbildung 5.28 dargestellt.

Um die Verifikation eines parametrisierten Koordinationsmusters zu beschreiben, wird im nächsten Abschnitt zuerst eine formale Definition eines parametrisierten Koordinationsmusters vorgenommen.

5.4.4 Formalisierung

Als erstes wird der zur Beschreibung des Verhaltens einer Unterrolle verwendete Formalismus des Timed Automaton zu einem parametrisieren Timed Automaton erweitert.

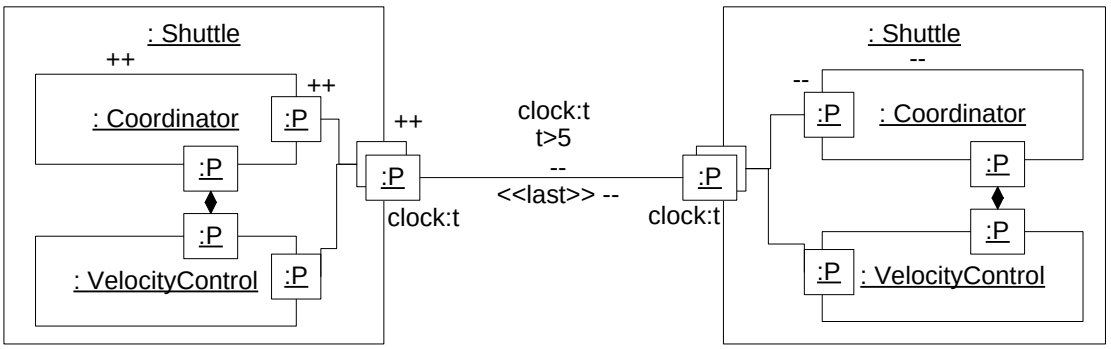


Abbildung 5.28: Führungsshuttle verlässt den Konvoi

Definition 34
Ein parametrisierter Timed-Automaton A *ist ein 7-Tupel* $A := (\Sigma, \mathcal{S}, \mathcal{S}^0, X, I, Sig(l, P), T)$, *wobei* Σ *ein endliches Eingabealphabet,* $\mathcal{S}$ *eine endliche Menge an Locations,* $\mathcal{S}^0 \subseteq \mathcal{S}$ *eine endliche Menge von Start-Locations,* $X := (x_1, .., x_n)$ *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 sind.* P *ist hierbei eine spezielle Eigenschaft 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 zurück gesetzt werden.* $sig \subseteq Sig(l, P)$ *ist ein durch einen Parameter* l *gekennzeichnetes Signal, dass den Wert* $p \in P$ *übermittelt.*

Die parallele Ausführung zweier *parametrisierter Automata* A^i *und* A^j ist wie folgt definiert:

Definition 35
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 34 definiert. Jeder Automat* A^i *und* A^j *verhält sich lokal wie in den Grundlagen (Abschnitt 2.3.2) beschrieben. 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* $i \leq j$

Mit dieser Beschreibung ist es nun möglich, eine Unterrolle zu definieren.

Definition 36
$R^l = (A, \Psi)$ ist eine Unterrolle mit dem Index l, wobei das Verhalten durch einen parametrisierte Timed Automaton A beschrieben wird. Ferner besitzt die Unterrolle eine Menge von lokalen Constraints Ψ.

Hiermit lässt sich nun der Begriff einer *multi-Rolle* definieren:

Definition 37
Eine multi-Rolle *eines parametrisierten Koordinationsmusters C ist definiert als ein* 3-*Tupel $MR^C = (n, \mathcal{R}^l, attr)$ wobei n die Multiplizität, $\mathcal{R}^l$ mit $l \in \{1 \dots n\}$ die Menge aller Unterrollen Rollen, $attr$ eine Ordnung auf $\mathcal{R}^l$ ist.*

Definition 38
Mit $I(MR^C) \subseteq \Sigma \times Sig(l, p)$ wird das Interfaceverhalten *einer multi-Rolle bezeichnet.*

Definition 39
Ein parametrisiertes Koordinationsmuster *$C = (\mathcal{MR}, \mathcal{P}, \Psi(P), \mathcal{P}_t^C, \mathcal{P}_t^R, \mathcal{P}_t^F)$ besteht aus einer Menge multi-Rollen $\mathcal{MR}$, einer Menge von Profilen $\mathcal{P} = p_1 \dots p_n$, Constraints über die Profile $\Psi(P)$, einer Menge von zeitbehafteten Erzeugungsregeln sowie Reduzierungsregeln $\mathcal{P}_t^C, \mathcal{P}_t^R$ sowie einer Menge von verbotenen Strukturregeln $\mathcal{P}_t^F$.*

5.4.5 Verifikation

Im Folgenden wird die Verifikation eines wie im letzten Abschnitt formal definierten parametrisierten Koordinationsmusters beschrieben. Wie eingangs im Kapitel beschrieben, kommen hier die Techniken des Model Checking, die Analyse von Graphtransformationssystem sowie der Induktion zum Tragen.

Die Korrektheit hinsichtlich der spezifizierten Profileigenschaften einer multi-Rolle wird induktiv bewiesen. Für die Verifikation eines parametrisierten Koordinationsmusters C sind die folgenden Schritte notwendig:

1. Verifiziere mittels Model Checking, dass $R^1_{MR^C} \models \Psi(R) \wedge \neg\delta$
2. Verifiziere $R^i_{MR^C} || R^{i+1}_{MR^C} \models \neg\delta \wedge val(R^i_{MR^C}) \leq val(R^{i+1}_{MR^C})$, wobei $val(R^i_{MR^C})$ die Werte von p liefert, die in der Rolle angenommen werden können
3. Nutze den Ansatz aus Kapitel 4 um mittels Erreichbarkeitsanalyse zu überprüfen, dass
   a) die Zeitbedingungen der Regeln $\mathcal{P}_t^C$ und $\mathcal{P}_t^R$ eingehalten werden und
   b) dass die durch die verbotenen Strukturregeln $\mathcal{P}_t^F$ beschriebenen Situationen nicht auftreten.

4. Verifiziere mittels Model Checking, dass $I(MR_1^C)|| \ldots ||I(MR_k^C) \models \neg\delta$ (k Anzahl der multi-Rollen von C) erfüllt ist.

Theorem 1
Ein parametrisiertes Koordinationsmuster erfüllt die hinsichtlich der Profile spezifizierten Constraints $\Psi(P)$, wenn jeder Verifikationsschritt $1-4$ erfüllt ist.

Beweis 1
Beweis durch Induktion (Beweisskizze): (1) *stellt sicher, das eine einzelne Rolle R^i einer multi-Rolle hinsichtlich $\Psi(R^i)$ korrekt ist und sie keinen Deadlock enthält (Induktionsanfang).* (2) *beweist die Induktionsannahme, dass für zwei benachbarte parametrisierte Rollen die Profileigenschaft erfüllt ist. Der Induktionsschritt wird durch* (3) *gezeigt. Das korrekte Zusammenspiel aller einzeln verifizierten multi-Rollen eines parametrisierten Koordinationsmusters wird durch die Verifikation der parallelen Komposition aller Interfaceverhalten der multi-Rollen gezeigt* (4)*. (q.e.d.)*

5.5 Zusammenfassung

In den beiden vorangegangenen Kapiteln wurde beschrieben, wie sich ein OCM modellieren und verifizieren lässt. Der Fokus dieses Kapitels steht nun auf der Modellierung und Verifikation der Koordination von OCMs in vernetzten mechatronischen Systemen. Hierbei werden die bisher vorgestellten Techniken aus den vorangegangenen Kapiteln miteinander geschickt verknüpft.

Hierzu wurde zuerst der bisherige kompositionelle Ansatz zur Modellierung und Verifikation der Echtzeit-Koordination vorgestellt. Bei der Anwendung für komplexe, vernetzte mechatronische Systeme zeigt dieser Ansatz jedoch einige Einschränkungen, die auch diskutiert wurden. Dies waren die Punkte Dynamik hinsichtlich der Struktur des Echtzeit-Koordinationsmusters, die statisch vorgegeben ist, sowie die Stabilität hinsichtlich Koordinationsverhalten bezüglich der Regelungstechnik. Basierend auf dem kompositionellen Ansatz wurde der neue Ansatz, die parametrisierten Koordinationsmuster, welche nun zusätzlich dynamische Strukturänderungen als auch Werte-kontinuierliches Reglerverhalten mit berücksichtigen, vorgestellt. Die parametrisierten Koordinationsmuster wurden zuerst informell eingeführt und danach formal definiert. Am Ende wurden die Verifikationsschritte, um ein parametrisiertes Koordinationsmuster formal zu verifizieren, beschrieben.

Kapitel 6

Verwandte Arbeiten

In diesem Kapitel werden nun verwandte Arbeiten betrachtet, die sich mit der modellbasierten Verifikation von komplexen, vernetzten mechatronischen Systemen befassen. Da es, wie in der Einleitung schon beschrieben, nicht *die* Verifikationsmethode für solche Systeme gibt, sondern immer einer geschickten Kombination mehrerer Techniken bedarf, wird im Folgenden zuerst die Verifikation von Echtzeitsystemen betrachtet (siehe Abschnitt 6.1). Hierbei werden verschiedene Modelle und Verifikationstechniken diskutiert, die eine effiziente Verifikation dieser Systeme ermöglichen, bzw. die aufzeigen, wo die Grenzen liegen. Daran anschließend wird im Abschnitt 6.2 die Verifikation von hybriden Systemen diskutiert. Im Abschnitt 6.3 wird diskutiert, wie sich Architekturen, beschrieben durch hybride Modelle, verifizieren lassen. Bevor das Kapitel in Abschnitt 6.5 mit einer Zusammenfassung schließt, wird im vorletzten Abschnitt 6.4 die Verifikation von adaptiven Systemen behandelt.

6.1 Verifikation von Echtzeitsystemen

Farn Wang stellt in einer Übersicht [Wan04] einen Katalog von Modellen, Techniken und Werkzeugen für die Verifikation von Echtzeitsystemen vor. Weiterhin wird von Giese und Henkler in [GH06] ein Überblick über Ansätze für die modell-basierte Entwicklung von software-intensiven Systemen gegeben. Es existiert eine Reihe von Modellierungs- und Verifikationstechniken für Echtzeitsysteme. Im Folgenden werden drei Projekte aus dieser Auswahl stellvertretend für die Verifikation von Echtzeitsystemen vorgestellt, welche die wesentlichen Techniken und Modelle aus der vorliegenden Arbeit abdecken.

6.1.1 Generelle Ansätze

UPPAAL. UPPAAL [BDL04] ist ein Werkzeug für die Modellierung, Validierung und Verifikation von Echtzeitsystemen, die durch ein Netzwerk von untereinander kommuni-

zierenden Timed Automata beschrieben sind. Die Modelle erlauben dabei die Verwendung von komplexen Datenstrukturen, wie Array usw. Zur Kodierung der Clocks wird die Datenstruktur der Difference-Bound-Matrices (DBM) [Dil90] verwendet, die auch in dieser Arbeit aufgegriffen wurde, um das Zeitmodell bei den zeitbehafteten Graphtransformationssystem zu verwalten (siehe Kapitel 4). Die Verifikation stellt verschiedene Optionen zur Optimierung des Zustandsraumes zur Verfügung, wie *clock-reduction*, *convex-hull approximation* usw.

HUGO/RT. In Knapp und andere [KMR02] wird das Werkzeug HUGO/RT vorgestellt. Mit diesem Werkzeug ist es möglich Modelle, beschrieben durch UML state machines, zu verifizieren. Die zu verifizierenden Eigenschaften werden durch Szenario Diagramme (Sequenzdiagramme) beschrieben. Um die Verifikation mit UPPAAL oder SPIN durchführen zu können, werden von HUGO/RT die UML state machines in Timed Automata transformiert. Ein UML Sequenzdiagramm wird auf einen Observer Timed Automaton abgebildet. Die Verifikation findet nun statt, indem Erreichbarkeitsanfragen über den Observer Timed Automaton gestellt werden. Balser und andere stellen in [BBK+04] einen Ansatz vor, der den interaktiven Verifizierer KIV [BRSS99] anstelle von UPPAAL und SPIN in die Werkzeugkette integriert. Die Modelle werden weiterhin mit HUGO/RT modelliert, die temporalen Eigenschaften der UML state machines werden nun allerdings mit KIV überprüft. Dieser Ansatz hat den Vorteil, dass prinzipiell auch UML state machines mit unendlichem Zustandsraum überprüft werden können, da bei der Verifikation mit KIV Techniken wie z.B. Induktion verwendet werden. Der Fokus dieses Ansatz liegt auf der Beschreibung von Verifikationsalgorithmen, hingegen verfolgt der in dieser Arbeit vorgestellte ansatz eine durchgängige modellbasierte Entwicklung durch die Integration von UML basierten Modell- und Verifikationstechniken unter Ausnutzung von vorgegebenen Architekturen.

IST OMEGA. Das Projekt IST OMEGA [GH04] hat sich zum Ziel gesetzt, den Korrektheitsnachweis für eine auf die speziellen Bedürfnisse eingebetteter Echtzeitsoftware abgestimmte Teilmenge der UML [DJVP03] durch Integration von Verifikationswerkzeugen wie Model Checker und Theorembeweiser (PVS) und UML CASE Werkzeugen zu ermöglichen. Dabei wurde die UML um eine Systemzeit (now) und Timerkonzepte ergänzt, die durch eine Abbildung auf Communication Extended Timed Automata semantisch fundiert wurde. Im Gegensatz zum MECHATRONIC UML Ansatz ermöglicht der OMEGA Ansatz nur eine kompositionelle Betrachtung bei semi-automatischen, interaktiven Beweisen mit dem Theorembeweiser.

Fazit. Alle hier vorgestellten Werkzeuge und Modelle basieren auf dem Grundmodell des Timed Automatons. Allerdings erreicht keines der Modelle die Ausdrucksstärke der

in dieser Arbeit vorgestellten Verhaltensmodelle der Realtime Statecharts bzw. der Hybriden Rekonfigurations Charts [GH06]. Auf Basis der hier vorgestellten Werkzeuge werden nun im Folgenden Techniken vorgestellt, die zum Einsatz kommen, um die Verifikation für Echtzeitsysteme, wie in der Einleitung dieser Arbeit beschrieben, überhaupt erst anwendbar bzw. effizient zu machen. Diese sind jedoch alleine nicht ausreichend, um dem domänenübergreifenden Charakter von mechatronischen Systeme gerecht zu werden und diesen entsprechend zu verifizieren.

6.1.2 Techniken

Abstraktion und Komposition. Es existieren viele Ansätze zur Abstraktion und Komposition von Echtzeitsystemen. Der von Jensen und anderen [JGGS00] vorgestellte Ansatz wird später in dieser Arbeit aufgegriffen und deshalb im Folgenden kurz vorgestellt. Der Ansatz beschäftigt sich damit, das bei der Verifikation von Echtzeitsystemen, die mit UPPAAL Timed Automata modelliert wurden, auftretende Problem der Zustandsraumexplosion durch eine Kombination von Abstraktion und Komposition zu beheben. Durch die *timed simulation* werden die für die Abstraktion notwendigen grundlegenden Eigenschaften zwischen Timed Automata erhalten. Das Problem ist, dass bei dem Timed Automata Konzept, wie es in UPPAAL verwendet wird, globale Variablen und *urgent* Kommunikationskanäle vorkommen. Diese führen dazu, dass eine reine Erhaltung der *timed simulation* nicht ausreichend für einen kompositionellen Ansatz ist (siehe Kapitel 3 in [JGGS00]). Es wird deshalb ein erweiterter Ansatz, *timed ready simulation*, vorgestellt, der Abstraktion und Komposition unterstützt. Der Test auf *timed ready simulation* zwischen Timed Automata wird in UPPAAL durch eine Erreichbarkeitsanalyse durchgeführt. Angenommen, $\leq$ ist eine Simulationsrelation und $M \leq M_{abs}$ gilt es zu überprüfen. Hierzu wird nun zuerst ein Test Timed Automaton T_{abs} für M_{abs} konstruiert. Im zweiten Schritt wird nun getestet, ob in $M \parallel T_{abs}$ ein so genannter *reject* Zustand erreicht werden kann. Ist dies der Fall, gilt $M \not\leq M_{abs}$, andernfalls $M \leq M_{abs}$. T_{abs} wird in der Form eines Komplementautomaten von M_{abs} konstruiert. Die Konstruktion eines solchen Komplementautomaten T_{abs} ist immer dann möglich, wenn M_{abs} ein deterministischer Timed Automaton ist. Ein Timed Automaton T ist deterministisch, wenn für alle Zustände s, s', s'' von T gilt: Falls $s \xrightarrow{v} s'$ und $s \xrightarrow{v} s''$, dann folgt $s' = s''$.

Das Problem bei Timed Automata ist, dass sich nicht-deterministische Timed Automata nicht einfach in deterministische umwandeln lassen. Dies liegt daran, dass die Klasse der nicht-deterministischen Timed Automata nicht gegen Komplement abgeschlossen ist. Nicht-deterministsiche *Event-Clock Automata*, die eine strenge Formulierung der Timed Automata sind, lassen sich hingegen in deterministische umwandeln [AFH97]. Allerdings ist diese Einschränkung für die bei mechatronischen Systemen verwendeten Modelle zu restriktiv, so dass die Idee aus Jensen und anderen [JGGS00] in dieser Arbeit aufgegriffen und erweitert wurde.

Counterexample basierte Abstraktion. Clarke und andere beschreiben in [CGJ+03] eine Technik, um eine obere Abstraktion eines original Modells zu erhalten. Ist ein Bedingung von dem abstrakten Modell erfüllt, so ist sie auch in dem konkreten Modell erfüllt. Ist die Eigenschaft in dem abstrakten Modell falsch, so resultiert ein Gegenbeispiel in einem Verhalten in der Approximation, dass nicht in dem original Modell vorhanden ist. In diesem Fall muss die Abstraktion verfeinert werden, so dass das Verhalten, welches durch das Gegenbeispiel beschrieben wird, nicht mehr berücksichtigt wird. Clarke und andere stellen in ihrem Beitrag eine effiziente, automatische Verfeinerungsmethode vor, um aus den Informationen der Gegenbeispiele dies zu erreichen.

Diese Methode wird von einigen Model Checkern in der Echtzeitdomäne eingesetzt. So z.B. auch in dem Werkzeug SAL [TK02]. Der Vorteil ist eine geschickte Abstraktion, jedoch werden das Werte-kontinuierliche Verhalten sowie das Zeit-kontinuierliche Verhalten nicht getrennt voneinander betrachtet, so dass hier die Komplexität nicht ausreichend reduziert wird. Außerdem ist die Abstraktion von der vorgegebenen Systemarchitektur abhängig, was zur Folge hat, dass das Verfahren nicht immer eine optimale und damit effiziente Abstraktion liefert.

6.1.3 Komplexe Ansätze

Nachdem nun grundlegende, verwandte Techniken und Methoden der Abstraktion von Echtzeitmodellen diskutiert wurden, befasst sich dieser Abschnitt mit komplexen Ansätzen zur modell-basierten Entwicklung von Echtzeitsystemen.

Zeitbehaftete Komponentenspezifikation. Metzler und Wehrheim stellen in [MW07] eine Spezifikationssprache für die Modellierung von zeitbehafteten Komponentenarchitekturen (timed CSP-OZ) vor. Der Ansatz baut auf vorhandenen Konzepten von CSP-OZ, bei denen die Schnittstellen der Komponenten bereits durch pre/post Bedingungen beschrieben werden können, auf. Um jedoch den Anforderungen von Echtzeitsystemen gerecht zu werden, wird hier der Ansatz um die Modellierung von Zeit in den Schnittstellen erweitert. Um die Integration von Zeit so einfach wie möglich zu gestalten, wird hier kein neuer Formalismus, sondern nur ein neuer Datentyp, der die Zeit repräsentiert, hinzugefügt. Die Semantik wird formal über Timed Automata definiert. Dies hat den Vorteil, dass zu Verifikationszwecken der Model Checker UPPAAL eingebunden werden kann. Weiterhin werden hierauf Bedingungen für *timed simulation* definiert, so dass es auch möglich ist, Kompositionalität o.ä. zu verwenden.

Service orientierte Modellierung und Verifikation. In [EHK+07] wird ein modell-basierter Ansatz für die Entwicklung von verteilten, eingebetteten Echtzeitsystemen beschrieben. Um die Komplexität solcher Systeme in den Griff zu bekommen, wird ein Service-orientierter Ansatz verfolgt. Dabei werden zu Beginn die Funktionalitäten des Systems unabhängig voneinander durch Interaktionsdiagramme modelliert und verifiziert. Ein vorgegebener Prozess, der einer wohl-definierten Verfeinerungsbeziehung unter den Modellen folgt, erlaubt die schrittweise Entwicklung solcher Systeme. Der Service-orientierte Ansatz unterstützt die Entwicklung verteilter, eingebetteter Systeme angefangen von der Anforderungsanalyse bis hin zur Implementierung, Verifikation und Validierung.

Aufbauend auf den Konzepten wird in [EMK+07][EFF+08] ein Failure Management Ansatz für eingebettete Systeme vorgestellt. Hierbei werden so genannte Interaktionsmuster für die Kommunikation zwischen Komponenten beschrieben, die entsprechend verifiziert werden können. Die Interaktionsmuster werden auf Automaten abgebildet, die vom Model Checker SPIN [AKPM05] verifiziert werden können.

Im nächsten Abschnitt wird nun die Verifikation von hybriden Modellen diskutiert. Die dabei vorgestellten Ansätze nutzen weiterhin die bereits vorgestellten Techniken aus und einwickeln entsprechende Lösungen für den hybriden Fall.

6.2 Verifikation von hybriden Systemen

6.2.1 Generelle Ansätze

MATLAB/Simulink[1] ist der Industriestandard beim Entwurf von Regelungssystemen. Basierend auf Blockdiagrammen, die via zeit-kontinuierlichen Signalen interagieren, wird der Entwurf komplexer Werte-kontinuierlicher Systeme ermöglicht. Die Integration von Stateflow ermöglicht zudem die Modellierung ereignis-diskreten Verhaltens. MATLAB/-Simulink Modelle sind im Vergleich zu den anderen betrachteten Verfahren nicht formal hinterlegt. Eine formale Verifikation wird durch zusätzliche Formalisierung, wie zum Beispiel bei dem CheckMate [SRKC00] Ansatz erreicht.

Neben MATLAB/Simulink und Stateflow exisitieren eine Vielzahl von Ansätzen und Werkzeugen für die Modellierung und Verifikation hybrider Systeme. Hybrid Statecharts [KP91], Charon [ADE+01], Masaccio [Hen00], HyCharts & HyRoom [GSB98][SPP01] und HyTech/PHAver [Fre05] adressieren die Modellierung in Form von hybriden State Charts und Verifikation von komplexen hybriden Systemen.

[1] `http://www.mathworks.com/`

Fazit. Alle existierenden Ansätze unterstützen jedoch keine adäquate Modellierung für die hier behandelten komplexen, vernetzten mechatronischen Systeme [Bur06][GH06]. Das Hautproblem ist, dass die hier geforderte Dynamik und Modularität der Architektur von keinem Ansatz geeignet unterstützt wird. Das hat zur Folge, das zur Verifikation immer das gesamte System betrachtet werden muss, da es nicht dekomponiert werden kann und deshalb die Verifikation trotz Techniken, wie sie im Folgenden vorgestellt werden, nicht anwendbar ist.

Ein anderer Nachteil der vorhandenen Ansätze ist, dass die Domänen der Regelungstechnik und der Softwaretechnik nicht sauber in der Modellierung getrennt sind. Hierbei werden sowohl das Koordinationsverhalten als auch das kontinuierliche Verhalten innerhalb einer einzigen hybriden Komponente modelliert. Dies erfordert eine enge Zusammenarbeit der Disziplinen und lässt sich auch manchmal so gar nicht realisieren. Der hier verfolgte Ansatz der MECHATRONIC UML erlaubt durch klar definierte Schnittstellen zwischen den Domänen die getrennte Modellierung des Echtzeit-Koordinationsverhaltens und der kontinuierlichen Regler und unterstützt durch einen klaren Modularitätsbegriff die Integration aller Domänen [GHH+08b].

Im Folgenden werden nun anhand dieser Ansätze und Werkzeuge Techniken vorgestellt, die bei der Verifikation von hybriden Systemen eingesetzt werden, um die Komplexität teilweise zu vermindern.

6.2.2 Techniken

Approximation. HyTech, 1995 entwickelt, ist ein symbolischer Model Checker für Hybride Systeme [HHWT95]. Um diese Systeme zu behandeln, benutzt HyTech Hybride Automaten, mit denen in einem einzigen Formalismus diskrete und kontinuierliche Zustandsänderungen formuliert werden können. Das besondere an HyTech ist, dass es auch eine parametrische Analyse vornimmt. Es wird also nicht nur überprüft ob ein Modell eine bestimmte Formel (bei HyTech eine CTL-Formel) erfüllt oder nicht, sondern es werden Bedingungen bzw. Begrenzungen für bestimmte Parameter des Modells berechnet, unter denen die Korrektheit des Modells garantiert werden kann. Neben dieser großen Stärke von HyTech gibt es allerdings auch eine große Schwäche [Fre05]: HyTech benutzt keine exakte Arithmetik. Dies führt dazu, dass die Zahldarstellung irgendwann ungenau wird und dies führt zu Overflow-Fehlern. Wegen dieser Ungenauigkeit kann HyTech nicht auf komplexe Systeme angewandt werden, welche eine große Genauigkeit fordern. Weiterhin ist HyTech eher für Systeme mit Variablen mit kleinen Änderungsraten geeignet, da ansonsten der Zustandsraum zu groß wird um ihn noch in vernünftiger Art und Weise zu behandeln.

HyTech wird seit Ende 1996 nicht mehr weiterentwickelt. PHAVer baut auf HyTech auf und hat es sich zum Ziel gesetzt, die größten Nachteile von HyTech zu eliminie-

ren [Fre05]. So benutzt PHAVer eine Arithmetik-Bibliothek, welche eine exakte Zahldarstellung ermöglicht und damit Overflow Fehler vermeidet. Auf der anderen Seite stellt PHAVer konservative Verfahren wie Approximationstechniken vor, um die Polyeder-Darstellung zu vereinfachen. Aufgrund der exakten Arithmetik können nämlich sowohl die Koeffizienten als auch die Anzahl der Constraints (Gleichungen oder (Ungleichungen), welche die konvexen Polyeder begrenzen, übermäßig groß werden. Daher werden hier Approximationstechniken eingesetzt, um einerseits die Bits (der Koeffizienten) als auch die Anzahl der Constraints zu begrenzen.

Prädikat Abstraktion. CHARON / R-CHARON stellt ein hierarchisches, hybrides Automatenmodell zur Verhaltensbeschreibung zur Verfügung [ADE+01] [ADI06] [ADI03] [Iva03] [KSPL06]. Daneben unterstützt CHARON auch das Strukturkonzept der Hierarchie, um die Komplexität zu beherrschen (ROOM actor diagrams) [AGLS01] und definiert dabei eine Verfeinerungsbeziehung zischen den eingebetteten Komponenten und damit Verhaltensmodellen.

Zur Verifikation wird das Programm d/dt verwendet, das auf Prädikatenabstraktion beruht. Das System wird über Prädikate definiert, die das zu untersuchende Verhalten des Systems wiederspiegeln und die nicht relevanten Systemeigenschaften für diesen Verifikationsschritt wegabstrahieren.

Der Model Checker d/dt untersützt zwei unterschiedliche Verifikationsformen [ADM02]. Es ist möglich, alle erreichbaren Zustände aus dem initialen Zustand zu berechnen. Bei der Auswertung des Ergebnisses wird festgestellt, ob ein oder mehrere kritische Systemzustände eintreten können. Ist man daran interessiert, ob ein bestimmter Zustand oder eine Menge von Zuständen erreicht werden kann, so definiert man zusätzlich das Schlüsselwort „bad set“. Anschließend führt d/dt eine gezielte Suche im Zustandsraum durch, ob diese Zustände erreicht werden können. Für jedes System in d/dt muss vorher festgelegt werden, welche Dimension es haben soll. Die Dimension ist die Anzahl der veränderlichen Variablen. Anschließend werden die Differentialgleichungen des Systems mit Hilfe von Matrizen angegeben.

Dekomposition. Komplexe Systeme, bei denen sowohl diskretes Verhalten als auch kontinuierliche Daten vorkommen, sind als Ganzes schwer bis gar nicht zu verifizieren. Metzler stellt in [Met07] einen Dekompositionsansatz vor, der es erlaubt, die komplexen Strukturen modelliert in CSP-OZ, in kleine, für die Verifikation handhabbare Teile zu dekomponieren. Die Dekomposition wird durch die Technik des *Slicing* [BDFW07] bestimmt. Dabei wird keine kompositionelle Modellierung vorausgesetzt, sondern anhand der globalen Eigenschaften wird durch das Slicing eine kompositionelle Aufteilung zur Verifikation bestimmt. Der Ansatz wurde anhand der Konvoifahrt aus der „Neuen Bahntechnik Paderborn“ beispielhaft gezeigt.

Fazit. Die letzten drei vorgestellten Techniken sind effizient anwendbar für kleine hybride Modelle. Allerdings werden durch die Ansätze keine komplexen, vernetzen mechatronischen Architekturen in ihrer Gänze wie dem in dieser Arbeit zu Grunde liegenden Ansatz unterstützt. In dieser Arbeit wurden die Ideen dieser Techniken aufgegriffen und in den in dieser Arbeit vorgestellten Ansatz zu Verifikation integriert.

Im Folgenden werden nun Techniken vorgestellt, die sich in der Basis mit der Stabilitätsanalyse von hybriden Systemen beschäftigen.

6.2.3 Stabilität

Zuerst wird die Lyapunov Stabilitätsanalyse betrachtet. Diese ist in der Regelungstechnik eine zentrale Technik zur Überprüfung von regelungstechnischen Stabilitätsverhalten. Daran anschließend werden spezielle Techniken zur Gewährleistung der Stabilität eines Konvois, wie sie bei der Modellierung der parametrisierten Koordinationsmuster betrachtet wurden, diskutiert.

Lyapunov Stabilitätsanalyse. Stabilität bezeichnet im Allgemeinen, dass ein System auch unter dem Einfluss von Störungen einen begrenzten Bereich nicht verlässt. In den einzelnen Domänen existieren häufig konkrete Definitionen, die auch eine feinere Unterteilung des jeweiligen Stabilitätsbegriffes zulassen. Beispielsweise wird in der Regelungstechnik ein lineares zeitinvariantes System dann als stabil bezeichnet, wenn die Sprungantwort $h(t)$ für $t \rightarrow \infty$ einem endlichen Wert zustrebt. Andernfalls wird es als instabil bezeichnet. Ein solches System wird als übertragungsstabil bezeichnet, wenn es auf eine beschränkte Eingangsgröße stets mit einer beschränkten Ausgangsgröße antwortet: $|u(t)| \leq M \rightarrow |y(t)| \leq N$ (BIBO-Stabilität: Bounded Input - Bounded Output). In der Stabilitätstheorie nach Lyapunov werden für derartige Systeme Techniken vorgeschlagen, die eine mathematische Analyse hinsichtlich bestimmter Kriterien ermöglicht [Föl05][Lud95].

Im Fall einer Reglerumschaltung in mechatronischen Systemen ergeben sich durch die Umschaltung jeweils neue Reglersysteme. Diese müssen einzeln domänenspezifisch stabil sein. Außerdem muss die Funktion, welche die Regler umschaltet, konvergieren [LM99][OMT+08]. In [SB03] werden für den Nachweis der Stabilität für die Umschaltfunktion Multiple Lyapunov Funktionen vorgeschlagen. Dabei wird davon ausgegangen, das die jeweilige Veränderung am System zusammen mit dem System als Hybrider Automat mit diskreten Zuständen für die Umschaltung und kontinuierlichen Teilzuständen für das jeweilige Systemverhalten in einem diskreten Zustand beschreibbar sind. Gemäß Lyapunov heißt ein System (bzw. dessen Ruhelage) genau dann stabil, wenn es eine verallgemeinerte Energiefunktion gibt, welche bezüglich der möglichen Zustandsverände-

rungen abnimmt. Dieses Prinzip muss auch bei Strukturvariablen, also schaltenden Systemen gelten. Ferner müssen die Einflüsse auf das System aus dem Umfeld, vom Benutzer und aus dem System selbst stets zu einem stabilen Zielsystem führen. Der in dieser Arbeit diskutierte MECHATRONIC UML Ansatz adressiert die Problematik der Stabilität beim Umschalten zwischen Strukturen durch die Erweiterung der klassischen Hybriden Automaten zu hybriden Rekonfigurations Charts [OMT+08]. Hier werden Umschaltfunktionen, die vorab verifiziert wurden, mit Transitionen, die einen Wechsel der Struktur beschreiben, assoziiert. Die Zeit, welche eine Umschaltfunktion benötigt, wird entsprechend als Deadline in das Modell der hybriden Rekonfigurations Charts übernommen und fließt damit in die in dieser Arbeit beschriebene Verifikation ein. Dies ermöglicht die formale Verifikation der Stabilität beim Umschalten zwischen Strukturen.

Konvoi Stabilität. In der Literatur werden verschiedene Konzepte zur Konvoiregelung vorgestellt. Diese Ansätze können grundlegend darin unterschieden werden, ob eine Kommunikation zwischen den einzelnen Fahrzeugen des Konvois möglich ist oder nicht. In [YEK98] wird eine mögliche Konvoiregelung beschrieben, die ohne Kommunikation arbeitet. Ein weiterer Ansatz zur Konvoiregelung ohne Kommunikation wird in [HWLL04] beschrieben. Dieser Ansatz zeichnet sich dadurch aus, dass neuronale Netze zur Regelung genutzt werden. Andere Ansätze wie [BG03] und [ZEA03] setzen explizit eine Kommunikation zwischen allen Fahrzeugen voraus, um eine bessere Konvoiregelung zu erreichen. Zlocki und Zambou nutzen WLAN-Standardkomponenten für die Kommunikation innerhalb eines Konvois aus zwei Fahrzeugen [ZZ05]. Die Kommunikationseigenschaften sind laut den Autoren hierbei ausreichend. Diesen Ansätzen ist gemein, dass sie Ausfälle der Kommunikation nicht umfassend betrachten. Vor allem eine geeignete Modellierung und formale Überprüfung der Kommunikation bzgl. Einhaltung der Sicherheit wird nicht durchgeführt.

6.2.4 Barrier certificates

Prajna und andere stellen in [PJ04][Pra05] eine Technik vor, um temporale Eigenschaften von hybriden Systemen zu verifizieren. Im Gegensatz zum Model Checking wird hier nicht der Zustandsraum aller erreichbaren Zustände aufgebaut. Um Sicherheitseigenschaften, Erreichbarkeitsfragen, Möglichkeiten oder deren Kombination zu verifizieren, wird das theoretische Konzept der *barrier certificates* und *density functions* verwendet. Ein *barrier certificate* ist eine Funktion oder eine Menge von Funktionen von Zuständen, die Ungleichungen der Funktion selber und deren Ableitungen über den Verlauf beschreiben. Im Detail kann durch *barrier certificates* berechnet werden, ob alle möglichen Trajektorien von einem definierten Startpunkt in einer sicheren Region enden oder nicht. Das Konzept der *barrier certificates* funktioniert ähnlich der Lyapunov Stabilitätsanalyse, jedoch lassen sich hier neben reinen Stabilitätseigenschaften auch die gerade beschriebe-

nen Eigenschaften verifizieren. Die Berechnung der *barrier certificates* basiert auf der Bestimmung der *sum of squares*[BKA+07][PP05], die sich effizient berechnen lassen.

6.3 Verifikation von Architekturen beschrieben durch hybride Modelle

Im Rahmen des Teilprojektbereichs H des AVACS Projektes „Automatic Verification and Analysis of Complex Systems“ (SFB/TR 14 AVACS[2]) wird die Verifikation von hybriden Systemen untersucht. In diesem Rahmen wird ein Großteil der bisher beschriebenen Techniken in neuen Techniken und Methoden eingesetzt, die anhand der Fallstudie ETCS (European Train Control System) evaluiert werden. [DMO+07][DHO04][PQ08][BBE+04][FH05].

Im Detail werden Verifikationstechniken für untereinander kooperierende Agenten beschrieben. Hierzu wurde eine drei Schichten Architektur mit den Ebenen *cooperation layer (kontinuierliche Zeit)*, *control layer (kontinuierliche Zeit)* und *design layer (diskrete Zeit)* beschrieben. Für jede einzelne Schicht werden eigens hierfür geeignete Verifikationstechniken zur Verfügung gestellt, die für das unterlagerte Zeitmodell geeignet sind. Die Verifikation umfasst vorverifizierte Entwurfsmuster, die automatische Synthese von Lyapunov Funktionen, die Erzeugung von Parametereigenschaften sowie definierte Verfeinerungsbeziehungen zwischen Modellen im Entwicklungsprozess.

Fazit. Die Techniken und Methoden aus dem AVACS Projekt kommen den in dieser Dissertation vorgestellten Ansätzen relativ nahe. Jedoch zeigt der AVACS Ansatz im Vergleich einige Schwächen. So ist die drei Schichtenarchitektur ähnlich der hier vorgestellten Architektur eines OCMs (siehe Kapitel 2.1), jedoch ist das Zusammenspiel der einzelnen Schichten nicht genauer definiert, so dass auch hier keine Vorgehensweise für eine Verifikation beschrieben ist. Im vorliegenden Ansatz wurden Schnittstellen und Abstraktionen zwischen den einzelnen Schichten definiert, die eine Verifikation ermöglichen. Weiterhin gibt es zwar die Möglichkeit, die Interaktion zwischen Agenten durch vorab verifizierte Muster zu beschreiben, jedoch ist diese auf eine feste, statische Struktur beschränkt. Durch den graphbasierten Ansatz aus dieser Arbeit ist es möglich, eine dynamische Struktur zu verifizieren.

[2] http://www.avacs.org/

6.4 Adaptive Systeme

Wie in der Einleitung motiviert, sind vernetzte mechatronische Systeme auch dadurch charakterisiert, dass sie zur Laufzeit ihre Struktur und ihr Verhalten ändern. Hierbei wird von adaptiven Systemen gesprochen.

Ein wichtiges Problem bei verteilten mechatronischen Systemen ist, dass bei vernetzten Systemen jedes Teilsystem aufgrund der zur Laufzeit erfolgten Adaption eine potentiell unterschiedliche lokale Sicht haben kann, auf deren Basis in Notfällen Entscheidungen autonom und lokal getroffen werden müssen. Deshalb müssen auch hier Techniken und Methoden entwickelt werden, die hier bei der Verifikation der Sicherheitseigenschaften verwendet werden können.

Ein weiterer Ansatz, welcher sich mit der Integration von Zeit in das Modell der Graphtransformationssysteme beschäftigt und somit die Dynamik von zeitlichen Strukturänderungen in adaptiven Systemen adressiert, wird von Heckel und anderen in [GVH03] aufgezeigt. Dort wird Zeit in Anlehnung an Time ER-Netze [GMMP91] modelliert. Ein maßgeblicher Unterschied liegt darin, dass es die in der vorliegenden Arbeit vorgestellten erweiterten und neu hinzugekommenen Graphtransformationsregeln erlauben, zeitliche Eigenschaften und Bedingungen gezielt mit einzelnen Teilgraphen zu verknüpfen. Dabei können die einzelnen zeitlichen Bedingungen einem Teil der linken Seite einer Graphtransformationsregel zugeordnet werden, wodurch es möglich ist, komplexe Bedingungen, wie sie in mechatronischen Systemen vorkommen, innerhalb einer Regel zu formulieren.

Fazit. Es gibt eine Reihe von Ansätzen für die Modellierung und Verifikation von strukturellen Aspekten von adaptiven Systemen [GVH03] [TGM00] [MÖ96] [HIM98] [OMT98] [KMS92] sowie für die Modellierung und Verifikation des Verhaltens [ZC06] [ADG98] [KM98] [CPT99]. Jedoch umfasst keiner dieser Ansätze allumfassend beide Aspekte [BCDW04]. Der in dieser Arbeit verfolge MECHATRONIC UML Ansatz kombiniert beide Aspekte und unterstützt hierfür ein Verifikationsverfahren und adressiert dabei die Komplexität von mechatronischen Systemen.

6.5 Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten zum Thema dieser Dissertation diskutiert. Zuerst wurden Werkzeuge für die Verifikation von Echtzeitsystemen vorgestellt. Daran anschließend wurden Techniken der Abstraktion, wie sie bei solchen Model Checkern eingesetzt werden, um die Komplexität von Echtzeitsystemen zu beherrschen, vorgestellt

und diskutiert. Abschließend wurden komplexe modell-basierte Ansätze zur Verifikation von Echtzeitsystemen vorgestellt. Die Diskussion dieses Abschnitts hat gezeigt, dass eine Reihe von einzelnen Techniken für die Verifikation von Echtzeitsystemen existieren, die in ihrer Theorie gut durchdacht, jedoch für praktische Verifikationsaufgaben im Bereich von mechatronischen Systemen alleine effizient nicht anwendbar sind. Gründe sind hierfür u.a. die fehlende Integration in eine vernetzte, strukturierte modulare und kompositionelle Architektur oder das nicht berücksichtigte inhärente domänenübergreifende Verhalten.

Daran anschließend wurden verwandte Arbeiten zur Verifikation von hybriden Systemen diskutiert, die sich mit domänenübergreifenden Verhalten beschäftigen. Hier wurden aufeinander aufbauende Techniken und Methoden vorgestellt, die es erlauben, bestimme Eigenschaften wie Erreichbarkeit oder Stabilität hinsichtlich spezifizierter Eigenschaften zu verifizieren. Allerdings hat die Diskussion gezeigt, dass die Ansätze keine komplexen, vernetzen mechatronischen Architekturen in ihrer Gänze, wie dem in dieser Arbeit zu Grunde liegenden Ansatz, unterstützen. In dieser Arbeit wurden die Ideen dieser Techniken aufgegriffen und in den in dieser Arbeit vorgestellten Ansatz zu Verifikation integriert.

Die Verifikation von komplexen Architekturen, beschrieben durch hybride Modelle, wurde anhand des AVACS Projekts vorgestellt, welches die vorgestellten Techniken integriert. Das AVACS Projekt kommt den Ansätzen dieser Arbeit sehr nahe, zeigt jedoch einige Einschränkungen hinsichtlich der zu verifizierenden Architektur und der Eigenschaften hinsichtlich der Dynamik.

Abschließend wurde noch auf die Verifikation von adaptiven Systemen eingegangen. Die Diskussion hat gezeigt, dass bisher keine adäquaten Ansätze, die sowohl Struktur als auch Verhalten in Verifikationsansätzen behandeln, existieren.

Kapitel 7

Zusammenfassung & Ausblick

Beim Entwurf selbstoptimierender, mechatronischer Systeme stellt die eingebettete Software einen großen Teil der Wertschöpfung dar. Typischerweise werden Regelungen oder Steuerungen in Software umgesetzt. Durch die starke Vernetzung selbstoptimierender Systeme wird Software auch zur nachrichtenbasierten Kommunikation und Koordination zwischen den einzelnen verteilten selbstoptimierenden Systemen eingesetzt. Diese Kommunikation geht über die Aufnahme von System- und Umweltdaten durch Sensorik hinaus. Hier werden ggf. komplexe Zustandsinformationen über entsprechende Protokolle und zugrunde liegende Kommunikationskanäle ausgetauscht, die dann wieder das Verhalten bzw. die zugrunde liegenden Berechnungen der einzelnen Komponenten massiv beeinflussen können. Diese Entwicklung führt zu äußerst komplexer hybrider (diskreter / kontinuierlicher) Software. Des Weiteren werden selbstoptimierende, mechatronische Systeme oftmals in sicherheitskritischen Umgebungen eingesetzt. Hierdurch müssen formale Verfahren zur Verifikation der Korrektheit des Systems gegenüber sicherheitskritischen Eigenschaften eingesetzt werden.

Ziel dieser Dissertation war es, Konzepte und Methoden zur Modellierung und Verifikation mechatronischer Systeme zu entwickeln und formal zu beschreiben. Ziel dabei war es, die besonders durch die Verwendung domänenübergreifender Modelle, wie sie bei der Modellierung von mechatronischen Systemen vorkommen, entstehenden inhärenten multi-Paradigmenwechsel [HH06] bei der Modellierung und Verifikation zu berücksichtigen. Der hier vorgeschlagene Ansatz zur modell-basierten Entwicklung mechatronischer Systeme zeichnet sich durch die Integration effizienter Verifikationstechniken, basierend auf Modellwissen, Abstraktionstechniken, regelbasierten und geschickten Modellierung aus.

7.1 Zusammenfassung

In dieser Arbeit wurden zuerst anhand des Vorgehens der modell-basierten Entwicklung Modelle und Verfahren zur Verifikation von mechatronischen Systemen vorgestellt. Diese sind jedoch für die komplexen, vernetzten mechatronischen Systeme, wie sie einleitend beschrieben wurden, nicht alleine anwendbar. Ziel der vorliegenden Arbeit war es, aufbauend auf dem vorhandenen Ansatz der MECHATRONIC UML, Erweiterungen und neue Ansätze zur Modellierung und Verifikation solcher Systeme vorzuschlagen.

In Kapitel 3 wurde zuerst beschrieben, wie sich ein einzelnes OCM verifizieren lässt. Hierbei wurde beschrieben, wie sich das Zusammenspiel des reflektorischen Operators mit dem Controller verifizieren lässt. Hierbei mussten die harten Echzeiteigenschaften des reflektorischen Operators sowie das kontinuierliche Verhalten des Controllers berücksichtigt werden. Um das hybride Verhalten verifizieren zu können, wurde ein Modularitätskonzept der Struktur beschrieben, welches auf einer wohl-definierten Verfeinerungsbeziehung aufbaut und die zur Verifikation nötigen Abstraktionen untersützt.

Im darauf folgenden Kapitel 4 wurde beschrieben, wie sich das äußere Verhalten von OCMs in der Umwelt modellieren und verifizieren lässt. Motivierend hierfür war eine möglichst realitätsnahe Beschreibung von Strukturveränderungen. In Schilling [Sch06] wurde bereits beschrieben, wie Graphtransformationssysteme zur Beschreibung von dynamischen Veränderungen im Kontext von mechatronischen Systemen eingesetzt werden können. Dieser Ansatz wurde derart erweitert, dass nun auch Zeitbedingungen bei der Modellierung und Verifikation berücksichtigt werden. So möchte man z.B. bei der Beschreibung der Fortbewegung eines Shuttles angeben können, wie lange ein Shuttle zum Durchfahren eines Schienenabschnitts benötigt oder wie lange die Instanziierung von Softwarekomponenten dauert, die bei der Anwendung von Echtzeit-Koordinationsmustern instanziiert werden müssen. Durch untere Schranken (guards) ist es möglich, eine Mindestzeitdauer festzulegen und diese nach oben hin durch eine Invariante zu begrenzen, die dadurch das Fortschreiten des Verhaltens garantiert. Hierbei wurde der Formalismus der Graphtransformationssysteme um Zeitannotationen erweitert.

In Kapitel 5 wurde die Koordination von mehreren OCMs auf der VMS Ebene betrachtet. Zwar bietet die MECHATRONIC UML hierfür schon den Ansatz der Echtzeit-Koordinationsmuster, diese jedoch weisen eine Reihe von Einschränkungen auf. So sind diese durch eine statische Struktur gekennzeichnet und berücksichtigen kein hybrides Verhalten. Die hier neu eingeführten parametrisierten Koordinationsmuster ermöglichen nun die Integration von kontinuierlichem Verhalten sowie von dynamischen Strukturänderungen.

7.2 Ausblick

Abschließend werden Ausblicke auf weitere Arbeiten gegeben. Im Fachgebiet Softwaretechnik[1] und im SFB 614 wird die MECHATRONIC UML stetig weiterentwickelt, um den neuen Forschungsergebnissen gerecht zu werden.

Wie anfangs in der Zusammenfassung und bei den Konzepten der modell-basierten Entwicklung erwähnt, steht neben der Modellierung und Verifikation auch die Codesynthese aus Modellen im Vordergrund. Um nun auch die neuen parametrisierten Koordinationsmuster zu unterstützen und die hier erreichten Verifikationsergebnisse zu verwenden, muss die bisherige Codesynthese [Bur06][BGS05] angepasst werden.

Eine nächste Erweiterung wäre, die bereits vorhandene Synthese aus [GHHK06] zur automatischen Synthese von dynamischen Collaborationen zu erweitern. In diesem Kontext kann auch untersucht werden, inwiefern sich TSSDs [Kle08][GHH+07] verwenden lassen, um die Constraints der neuen parametrisierten Koordinationsmuster zu formulieren. Hierbei kann auch untersucht werden, inwiefern sich der Ansatz aus der Automobilindustrie zur Beschreibung von komplexen Constraints integrieren lässt [GHS+07a][GHS+07b][GNN+06].

Als letzter Punkt steht nun die Integration von Testverfahren in den Ansatz. [GHHP07][HH07][GHH08a]. Oftmals ist die Überprüfung eines verifizierten Modells allein trotz automatischer Codegenerierung nicht immer ausreichend, um die Konformität des Systems zu seiner Spezifikation zu zeigen. Z.B. werden nachträglich Veränderungen am Code zur Optimierung vorgenommen oder Legacy Komponenten werden integriert. Um die Konformität der Software in den oben beschrieben Fällen dennoch sicherzustellen, ist es erforderlich, den Code zu „verifizieren“. Eine verbreitete Methode dazu ist der Software-Test. Aufbauend auf vorhandenen Verfahren müssen die Software-Tests jetzt auch auf komplexe, vernetze mechatronische Systeme erweitert werden.

[1] http://www.upb.de/cs/ag-schaefer

Kapitel 8

Literaturverzeichnis

Eigene Veröffentlichungen

[BGH05a] BURMESTER, Sven ; GIESE, Holger ; HIRSCH, Martin: Syntax and Semantics of Hybrid Components / University of Paderborn. Paderborn, Germany, October 2005 (tr-ri-05-264). – Forschungsbericht

[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: *Proc. of the 27th International Conference on Software Engineering (ICSE), St. Louis, Missouri, USA*, ACM Press, Mai 2005, S. 670–671

[BGH+07] BURMESTER, Sven ; GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; TICHY, Matthias ; GAMBUZZA, Alfonso ; MÜCH, Eckehard ; VÖCKING, Henner: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View. In: *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, IEEE Computer Society Press, Mai 2007, S. 801–804

[BGHS04] BURMESTER, Sven ; GIESE, Holger ; HIRSCH, Martin ; SCHILLING, Daniela: Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In: *Proc. of the International Workshop on Specification and Validation of UML Models for Real-Time and Embedded Systems, SVERTS2004, Satellite Event of the 7th International Conference on the Unified Modeling Language, UML2004*, 2004, S. 1–20

[GH05a] GIESE, Holger ; HIRSCH, Martin: Checking and Automatic Abstraction for Timed and Hybrid Refinement in Mechtronic UML / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Germany, December 2005 (tr-ri-03-266). – Forschungsbericht

[GH05b] GIESE, Holger ; HIRSCH, Martin: Modular Verificaton of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: *Proc. of the International Workshop on Modeling and Analysis of Real-Time and Embedded Systems (MARTES), Satellite Event of the 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML2005*, 2005, S. 7–26

[GH06] GIESE, Holger ; HIRSCH, Martin: Modular Verificaton of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML. In: BRUEL, Jean-Michel (Hrsg.): *Satellite Events at the MoDELS 2005 Conference, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers* Bd. 3844. Springer Verlag, January 2006, S. 67–78

[GHH06a] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin: Analysis and Modeling of Real-Time with Mechatronic UML taking Clock Drift into Account. In: *Proc. 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). – ISBN 82–7368–299–4, S. 41–60

[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: *Proc. of the Fourth Paderborner Workshop Entwurf mechatronischer Systeme* Bd. 189, 2006 (HNI-Verlagsschriftenreihe), S. 457–473

[GHH+07] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; KLEIN, Florian ; SPIJKERMAN, Michael: Monitoring of Structural and Temporal Properties. In: GEIGER, Leif (Hrsg.) ; GIESE, Holger (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proc. of the 5th International Fujaba Days 2007, Kassel, Germany*, 2007, S. 1–4

[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, Felicita D. (Hrsg.) ; GACEK, Cristina (Hrsg.) ; MUCCINI, Henry (Hrsg.) ; VIEIRA, Marlon (Hrsg.): *Architecting Dependable Systems V* Bd. 5135, Springer Verlag, Juni 2008 (Lecture Notes in Computer Science

(LNCS)), S. 248–272

[GHH+08b] 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*. Idea Group Publishing, Mai 2008, S. 21–57

[GHHK06] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; KLEIN, Florian: Nobody's perfect: Interactive Synthesis from Parametrized Real-Time Scenarios. In: *Proc. of the 5th ICSE 2006 Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'06),Shanghai, China*, ACM Press, Mai 2006, S. 67–74

[GHHP07] GIESE, Holger ; HENKLER, Stefan ; HIRSCH, Martin ; PRIESTERJAHN, Claudia: Model-Based Testing of Mechatronic Systems. In: GEIGER, Leif (Hrsg.) ; GIESE, Holger (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proc. of the 5th International Fujaba Days 2007, Kassel, Germany*, 2007, S. 1–4

[GHS+07a] GEHRKE, Matthias ; HIRSCH, Martin ; SCHÄFER, Wilhelm ; NIGGEMANN, Oliver ; STICHLING, Dirk ; NICKEL, Ulrich: Verifikation zeitlicher Anforderungen in automotiven komponentenbasierten Software Systemen. In: BLEEK, Wolf-Gideon (Hrsg.) ; SCHWENTNER, Henning (Hrsg.) ; ZÜLLIGHOVEN, Heinz (Hrsg.): *Proc. of the Software Engineering 2007 Conference, Hamburg, Germany, 27.-30.3.2007* Bd. P-105, Gesellschaft für Informatik, März 2007 (Lecture Notes in Informatics (LNI)), S. 251–252

[GHS+07b] GEHRKE, Matthias ; HIRSCH, Martin ; SCHÄFER, Wilhelm ; NIGGEMANN, Oliver ; STICHLING, Dirk ; NICKEL, Ulrich: Typisierung und Verifikation zeitlicher Anforderungen automotiver Software Systeme. In: CONRAD, Mirko (Hrsg.) ; GIESE, Holger (Hrsg.) ; RUMPE, Bernhard (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 15.-18.1.2007, Schloss Dagstuhl, Germany*. Technische Universität Braunschweig, January 2007 (Informatik-Bericht 2007-1), S. 73–82

[GNN+06] GEHRKE, Matthias ; NAWRATIL, Petra ; NIGGEMANN, Oliver ; SCHÄFER, Wilhelm ; HIRSCH, Martin: Scenario-Based Verification of Automotive Software Systems. In: GIESE, Holger (Hrsg.) ; RUMPE, Bernhard (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 9.-13.1.2005, Schloss Dagstuhl, Germany*. Technische Universität Braunschweig, January 2006 (Informatik-Bericht 2006-1), S. 35–42

[HG03] HIRSCH, Martin ; GIESE, Holger: Towards the Incremental Model Checking of Complex RealTime UML Models. In: GIESE, Holger (Hrsg.) ; ZÜNDORF, Albert (Hrsg.): *Proc. of the first International Fujaba Days 2003, Kassel, Germany* Bd. tr-ri-04-247, University of Paderborn, October 2003 (Technical Report), S. 9–12

[HH06] HENKLER, Stefan ; HIRSCH, Martin: A Multi-Paradigm Modeling Approach for Reconfigurable Mechatronic Systems. In: *Proc. 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

[HH07] HENKLER, Stefan ; HIRSCH, Martin: Compositional Validation of Distributed Real Time Systems. In: GEHRKE, Matthias (Hrsg.) ; GIESE, Holger (Hrsg.) ; STROOP, Joachim (Hrsg.): *Proc. 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

[HHG08] HIRSCH, Martin ; HENKLER, Stefan ; GIESE, Holger: Modeling Collaborations with Dynamic Structural Adaptation in Mechatronic UML. In: *Proc. of the ICSE 2008 Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08),Leipzig, Germany*, ACM Press, Mai 2008, S. 33–40

[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: *2008 ASME International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*. New York, NY, USA, 2008, S. 654–665

[Hir04] HIRSCH, Martin: *Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL.* Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, Diplomarbeit, Juni 2004

[OMT+08] OSMIC, Semir ; MÜNCH, Eckehard ; TRÄCHTLER, Ansgar ; HENKLER, Stefan ; SCHÄFER, Wilhelm ; GIESE, Holger ; HIRSCH, Martin: Safe Online-Reconfiguration of Self-Optimzing Mechatronic Systems. In: GAUSEMEIER, Jürgen (Hrsg.) ; RAMMIG, Franz (Hrsg.) ; SCHÄFER, Wilhelm (Hrsg.): *Selbstoptimierende mechatronische Systeme: Die Zukunft gestalten.*

7. Internationales Heinz Nixdorf Symposium für industrielle Informationstechnik, 2008, S. 411–426

Literatur

[ACD90] ALUR, Rajeev ; COURCOUBETIS, Costas ; DILL, David: Model-Checking for Real-Time Systems. In: *Proceedings of the 5th Annual Symposium on Logic in Computer Science*, IEEE Computer Society Press, 1990, S. 414–425

[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

[ADG98] ALLEN, Robert ; DOUENCE, Rémi ; GARLAN, David: Specifying and Analyzing Dynamic Software Architectures. In: *Lecture Notes in Computer Science (LNCS)* 1382 (1998), S. 21–36

[ADI03] ALUR, Rajeev ; DANG, Thao ; IVANCIC, Franjo: Progress on Reachability analysis of Hybrid Systems Using Predicate Abstraction. In: MALER, Oded (Hrsg.) ; PNUELI, Amir (Hrsg.): *HSCC '03: Proceedings of the 6th International Workshop on Hybrid Systems: Computation and Control, Prague, Czech Republic, April 3-5* Bd. 2623, Springer Verlag, 2003 (Lecture Notes in Computer Science (LNCS)). – ISBN 3–540–00913–2, S. 4–19

[ADI06] ALUR, Rajeev ; DANG, Thao ; IVANČIĆ, Franjo: Predicate abstraction for reachability analysis of hybrid systems. In: *Trans. on Embedded Computing Sys.* 5 (2006), Nr. 1, S. 152–199. – ISSN 1539–9087

[ADM02] ASARIN, Eugene ; DANG, Thao ; MALER, Oded: The d/dt Tool for Verification of Hybrid Systems. In: *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*. London, UK : Springer-Verlag, 2002. – ISBN 3–540–43997–8, S. 365–370

[AFH97] ALUR, Rajeev ; FIX, Limor ; HENZINGER, Thomas A.: Event-clock automata: a determinizable class of timed automata. In: *Theoretical Computer Science* 211 (1997), April, S. 253–273

[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

[AKPM05] AHLUWALIA, Jaswinder ; KRÜGER, Ingolf H. ; PHILLIPS, Walter ; MEISINGER, Michael: Model-based run-time monitoring of end-to-end deadlines. In: *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*. New York, NY, USA : ACM, 2005. – ISBN 1–59593–091–4, S. 100–109

[Bal98] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Heidelberg, Berlin, Oxford : Spektrum Akademischer Verlag, 1998

[BBE+04] BECKER, Bernd ; BEHLE, Markus ; EISENBRAND, Fritz ; FRÄNZLE, Martin ; HERBSTRITT, Marc ; HERDE, Christian ; HOFFMANN, Joerg ; KRÖNING, Daniel ; NEBEL, Bernhard ; POLIAN, Ilia ; WIMMER, Ralf: Bounded Model Checking and Inductive Verification of Hybrid Discrete-Continuous Systems. In: *GI/ITG/GMM Workshop: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*. Richard Petersens Plads, Building 321, DK-2800 Kgs. Lyngby : Informatics and Mathematical Modelling, Technical University of Denmark, DTU, February 2004

[BBF+01] BÉRARD, B. ; BIDOIT, M. ; FINKEL, A. ; LAROUSSINIE, F. ; PETIT, A. ; BETRUCCI, L. ; SCHNOEBELEN, Ph. ; MCKENZIE, P.: *Systems and Software Verification*. Springer, 2001

[BBG+06] BECKER, Basil ; BEYER, Dirk ; GIESE, Holger ; KLEIN, Florian ; SCHILLING, Daniela: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: *Proc. of the* 28^{th} *International Conference on Software Engineering (ICSE), Shanghai, China*, ACM Press, 2006, S. 72–81

[BBK+04] BALSER, Michael ; BÄUMLER, Simon ; KNAPP, Alexander ; REIF, Wolfgang ; THUMS, Andreas: Interactive Verification of UML State Machines. In: DAVIES, Jim (Hrsg.) ; SCHULTE, Wolfram (Hrsg.) ; BARNETT, Michael (Hrsg.): *ICFEM* Bd. 3308. Springer Verlag, 2004, S. 434–448

[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

[BDFW07] BRÜCKNER, I. ; DRÄGER, K. ; FINKBEINER, B. ; WEHRHEIM, H.: Slicing Abstractions. In: ARBAB, F. (Hrsg.) ; SIRJANI, M. (Hrsg.): *FSEN 2007: IPM International Symposium on Fundamentals of Software Engineering* Bd. 4767, Springer, April 2007 (Lecture Notes in Computer Science). – ISBN 978–3–540–75697–2, 17–32

[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* Bd. 3185, Springer Verlag, September 2004 (Lecture Notes in Computer Science (LNCS)), S. 200–236

[BG03] BAER, H. ; GERDES, j. C.: Parameter Estimation and Command Modification for Longitudinal Control of Heavy Vehicles / University of California. Berkeley, CA, USA, 2003 (PATH Research Report UCB-ITS-PRR-2003-16). – Forschungsbericht

[BGO04] BURMESTER, Sven ; GIESE, Holger ; OBERSCHELP, Oliver: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: ARAUJO, Helder (Hrsg.) ; VIEIRA, Alves (Hrsg.) ; BRAZ, Jose (Hrsg.) ; ENCARNACAO, Bruno (Hrsg.) ; CARVALHO, Marina (Hrsg.): *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, INSTICC Press, August 2004, S. 222–229

[BGO06] BURMESTER, Sven ; GIESE, Holger ; OBERSCHELP, Oliver: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: BRAZ, J. (Hrsg.) ; ARAÚJO, H. (Hrsg.) ; VIEIRA, A. (Hrsg.) ; ENCARNACAO, B. (Hrsg.): *Informatics in Control, Automation and Robotics I*. Springer Verlag, März 2006, S. 281–288

[BGS05] BURMESTER, Sven ; GIESE, Holger ; SCHÄFER, Wilhelm: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*, Springer Verlag, November 2005 (LNCS), S. 1–15

[BKA+07] BADAMCHIZADEH, Mohammad-Ali ; KHANMOHAMMADI, Sohrab ; ALIZADEH, Ghasem ; AGHAGOLZADEH, Ali ; KARIMIAN, Ghader: Using Sum of Squares Decomposition for Stability of Hybrid Systems. In: *IEICE Transactions* 90-A (2007), Nr. 11, S. 2478–2487

[BN03] BROEKMAN, Bart ; NOTENBOOM, Edwin: *Testing Embedded Software*. Addison-Wesley, 2003

[BRSS99] BALSER, Michael ; REIF, Wolfgang ; SCHELLHORN, Gerhard ; STENZEL, Kurt: KIV 3.0 for Provably Correct Systems. In: *FM-Trends 98: Proceedings of the International Workshop on Current Trends in Applied Formal Method*. London, UK : Springer Verlag, 1999, S. 330–337

[BS91] BRZOZOWSKI, J. A. ; SEGER, C. J.: *Advances in asynchornous circuit theory, Part II: Bounded inertial delay model, MOS circuits, design techniques*. Bd. 43. European Association for Theoretical Computer Science, 1991. – 199–263 S.

[Bur06] BURMESTER, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, PhD Dissertation, 2006

[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

[CGP00] CLARKE, E. M. ; GRUMBERG, O. ; PELED, D. A.: *Model Checking*. MIT Press, 2000

[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

[CW96] CLARKE, Edmund M. ; WING, Jeannette M.: Formal methods: state of the art and future directions. In: *ACM Comput. Surv.* 28 (1996), Nr. 4, S. 626–643. – ISSN 0360–0300

[DHO04] DAMM, W. ; HUNGAR, H. ; OLDEROG, E.-R.: On the Verification of Cooperating Traffic Agents. In: BOER, F.S. de (Hrsg.) ; BONSANGUE, M.M. (Hrsg.) ; GRAF, S. (Hrsg.) ; ROEVER, W.-P. de (Hrsg.): *Proc. FMCO '03: Formal Methods for Components and Objects* Bd. 3188, 2004 (Lecture Notes in Computer Science (LNCS)), 78–110

[Dil90] DILL, D. L.: Timing assumptions and verification of finite-state concurrent systems. In: *Proceedings of the international workshop on Automatic verification methods for finite state systems*. New York, NY, USA : Springer-Verlag New York, Inc., 1990. – ISBN 0–387–52148–8, S. 197–212

[DJVP03] DAMM, W. ; JOSKO, B. ; VOTINTSEVA, A. ; PNUELI, A.: A Formal Semantics for a UML Kernel Language / OMEGA: Correct Development of Real-Time Embedded Systems IST-2001-33522. 2003 (IST/33522/WP 1.1/D1.1.2-Part1). – Forschungsbericht. – Version 1.2

[DMO+07] DAMM, Werner ; MIKSCHL, Alfred ; OEHLERKING, Jens ; OLDEROG, Ernst-Rüdiger ; PANG, Jun ; PLATZER, André ; SEGELKEN, Marc ; WIRTZ, Boris: Automating Verification of Cooperation, Control, and Design in Traffic Applications. In: JONES, Cliff (Hrsg.) ; LIU, Zhiming (Hrsg.) ; WOODCOCK, Jim (Hrsg.): *Formal Methods and Hybrid Real-Time Systems* Bd. 4700, Springer Verlag, 2007 (Lecture Notes in Computer Science (LNCS)), S. 115–169

[Dor08] DOROCIAK, Rafal: *Hybride Verifikation von Mechatronic UML Modellen durch Integration des Modelcheckers PHAVer*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, Bachelorarbeit, Januar 2008

[DSBB00] DAWSON, D. ; SEWARD, D. ; BRADLEY, D.A. ; BURGE, S.: *Mechatronics and the Design of Intelligent Machines and Systems*. Nelson Thornes, 2000. – ISBN 0748754431

[Dwy02] DWYER, Matthew: *Software Model Checking Tutorial, FSE'02*. Charleston, South Carolina, USA, November 2002

[EFF+08] ERMAGAN, Vina ; FARCAS, Claudiu ; FARCAS, Emilia ; KRÜGER, Ingolf H. ; MENARINI, Massimilano: A Service-Oriented Approach to Failure Management. In: GIESE, Holger (Hrsg.) ; HUHN, Michaela (Hrsg.) ; NICKEL, Ulrich (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 7.4.-9.4.2008, Schloss Dagstuhl, Germany*. Technische Universität Braunschweig, April 2008 (Informatik-Bericht 2008-2), S. 102–116

[EHK+07] ERMAGAN, Vina ; HUANG, T.-J. ; KRÜGER, Ingolf ; MEISINGER, Michael ; MENARINI, Massimilano ; MOORTHY, P.: Towards Tool Support for Service-Oriented Development of Embedded Automotive Systems. In: CONRAD, Mirko (Hrsg.) ; GIESE, Holger (Hrsg.) ; RUMPE, Bernhard (Hrsg.) ; SCHÄTZ, Bernhard (Hrsg.): *Proc. of the Dagstuhl-Workshop: Model-Based Development of Embedded Systems (MBEES), 15.-18.1.2007, Schloss Dagstuhl, Germany*. Technische Universität Braunschweig, Januar 2007 (Informatik-Bericht 2007-1), S. 1–24

[EMK+07] ERMAGAN, Vina ; MENARINI, Massimilano ; KRÜGER, Ingolf ; MIZUTANI, Jun ichi ; OGUCHI, Kentaro ; WEIR, David: Towards Model-Based Failure-

Management for Automotive Software. In: *SEAS '07: Proceedings of the 4th International Workshop on Software Engineering for Automotive Systems*. Washington, DC, USA : IEEE Computer Society, 2007. – ISBN 0–7695–2968–2, S. 8

[Fav05] FAVRE, Jean-Marie: Foundations of Model (Driven) (Reverse) Engineering : Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In: BEZIVIN, Jean (Hrsg.) ; HECKEL, Reiko (Hrsg.): *Language Engineering for Model-Driven Software Development*. Dagstuhl, Germany : Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005 (Dagstuhl Seminar Proceedings 04101). – ISSN 1862–4405

[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*. 1. Auflage. Paderborn, Germany : Bonifatius GmbH, 2004 (HNI-Verlagsschriftenreihe Band 155)

[FH05] FRÄNZLE, Martin ; HERDE, Christian: Efficient Proof Engines for Bounded Model Checking of Hybrid Systems. In: *Electr. Notes Theor. Comput. Sci.* 133 (2005), S. 119–137

[Fre05] FREHSE, Goran: PHAVer: Algorithmic Verification of Hybrid Systems Past HyTech. In: *HSCC*, Springer Verlag, 2005 (Lecture Notes in Computer Science (LNCS)), S. 258–273

[Föl05] FÖLLINGER, Otto: *Regelungstechnik. Einführung in die Methoden und ihre Anwendung*. Hüthig, 2005

[GB04] GIESE, Holger ; BURMESTER, Sven: Analysis and Synthesis for Parameterized Timed Sequence Diagrams. In: GIESE, Holger (Hrsg.) ; KRÜGER, Ingolf (Hrsg.): *Proc. of the 3rd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools (ICSE 2003 Workshop W5S), Edinburgh, Scotland*, IEE, May 2004, S. 43–50

[GBSO04] GIESE, Holger ; BURMESTER, Sven ; SCHÄFER, Wilhelm ; OBERSCHELP, Oliver: Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In: *Proc. of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004), Newport Beach, USA*, ACM Press, November 2004, S. 179–188

[Ge05] GAUSEMEIER, Jürgen ; ET al.: *Sonderforschungsbereich 614 - Selbstoptimierende Systeme des Maschinenbaus, 2. Förderperiode, Finanzierungsantrag.*. Bd. 1. Universität Paderborn, 2005

[GH04] GRAF, Susanne ; HOOMAN, Jozef: Correct Development of Embedded Systems. In: OQUENDO, Flavio (Hrsg.) ; WARBOYS, Brian (Hrsg.) ; MORRISON, Ron (Hrsg.): *Proceedings of the First European Workshop on Software Architecture, EWSA2004* Bd. 3047. St Andrews, UK : Springer Verlag, May 21-22 2004 (Lecture Notes in Computer Science (LNCS)), S. 241–249

[GH06] 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

[Gie00] GIESE, Holger: Contract-based Component System Design. In: RALPH H. SPRAGUE, Jr. (Hrsg.): *Thirty-Third Annual Hawaii International Conference on System Sciences (HICSS-33), Maui, Hawaii, USA*, IEEE Computer Press, Januar 2000

[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

[GMMP91] GHEZZI, Carlo ; MANDRIOLI, Dino ; MORASCA, Sandro ; PEZZÈ, Mauro: A Unified High-Level Petri Net Formalism for Time-Critical Systems. In: *IEEE Trans. Softw. Eng.* 17 (1991), Nr. 2, S. 160–172. – ISSN 0098–5589

[GSB98] GROSU, Radu ; STAUNER, Thomas ; BROY, Manfred: A Modular Visual Model for Hybrid Systems. In: *FTRTFT '98: Proceedings of the 5th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* Bd. 1486. London, UK : Springer Verlag, 1998 (Lecture Notes in Computer Science (LNCS)). – ISBN 3–540–65003–2, S. 75–91

[GT06] GIESE, Holger ; TICHY, Matthias: Component-Based Hazard Analysis: Optimal Designs, Product Lines, and Online-Reconfiguration. In: *Proc. of the 25th International Conference on Computer Safety, Security and Reliability (SAFECOMP), Gdansk, Poland*, Springer Verlag, September 2006 (Lecture Notes in Computer Science (LNCS)), S. 156–169

[GTB+03] GIESE, Holger ; TICHY, Matthias ; BURMESTER, Sven ; SCHÄFER, Wilhelm ; FLAKE, Stephan: Towards the Compositional Verification of Real-Time UML Designs. In: *Proc. 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

[GVH03] GYAPAY, Szilvia ; VARRÓ, Dániel ; HECKEL, Reiko: Graph transformation with time. In: *Fundam. Inf.* 58 (2003), Nr. 1, S. 1–22. – ISSN 0169–2968

[Hen00] HENZINGER, Thomas A.: Masaccio: A Formal Model for Embedded Components. In: *Proceedings of the First IFIP International Conference on Theoretical Computer Science (TCS)* Bd. 1872, 2000 (Lecture Notes in Computer Science (LNCS)), 549-563

[Her99] HERRMANN, Debra S.: *Software Safety and Reliability : Techniques, Approaches, and Standards of Key Industrial Sectors*. IEEE Computer Press, 1999. – ISBN 0769502997

[HHT96] HABEL, Annegret ; HECKEL, Reiko ; TAENTZER, Gabriele: Graph Grammars with Negative Application Conditions. In: *Fundamenta Informaticae* 26 (1996), Nr. 3/4, S. 287–313

[HHWT95] HENZINGER, Thomas A. ; HO, P.-H. ; WONG-TOI, H.: HyTech: The Next Generation. In: *RTSS '95: Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS '95)*. Washington, DC, USA : IEEE Computer Press, December 1995. – ISBN 0–8186–7337–0, S. 55–65

[HIM98] HIRSCH, Dan ; INVERARDI, Paolo ; 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

[HKPV98] HENZINGER, Thomas A. ; KOPKE, Peter W. ; PURI, Anuj ; VARAIYA, Pravin: What's decidable about hybrid automata? In: *Journal of Computer and System Sciences* 57 (1998), S. 94–124

[Hof07] HOFFMANN, Thomas: *Spezifikation und Synthese von Konnektorverhalten für Mechatronic UML*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, Bachelorarbeit, Januar 2007

[HOG04] HESTERMEYER, Thorsten ; OBERSCHELP, Oliver ; GIESE, Holger: Structured Information Processing For Self-optimizing Mechatronic Systems. In: ARAUJO, Helder (Hrsg.) ; VIEIRA, Alves (Hrsg.) ; BRAZ, Jose (Hrsg.) ; ENCARNACAO, Bruno (Hrsg.) ; CARVALHO, Marina (Hrsg.): *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, INSTICC Press, August 2004, S. 230–237

[HPPS03] HAHN, Gabor ; PHILIPPS, Jan ; PRETSCHNER, Alexander ; STAUNER, Thomas: Prototype-based Tests for Hybrid Reactive Systems. In: *Proc. 14th*

IEEE Intl. Workshop on Rapid System Prototyping (RSP'03), 2003, S. 78–85

[HSE02] HESTERMEYER, Thorsten ; SCHLAUTMANN, Philipp ; ETTINGSHAUSEN, Clemens: Active suspension system for railway vehicles-system design and kinematics. In: *Proc. of the 2nd IFAC - Confecence on mechatronic systems*. Berkeley, California, USA, December 2002, S. 9–11

[HTBS08] HENKE, Christian ; TICHY, Matthias ; BÖCKER, Joachim ; SCHÄFER, Wilhelm: Organization and Control of Autonomous Railway Convoys. In: *Proceedings of the 9th International Symposium on Advanced Vehicle Control, Kobe, Japan*, 2008. – accepted

[HU79] HOPCRAFT, John E. ; ULLMAN, Jeffrey D.: *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979

[HVB+05] HENKE, Christian ; VÖCKING, Henner ; BÖCKER, Joachim ; FRÖHLEKE, Norbert ; TRÄCHTLER, Ansgar: Convoy Operation of Linear Motor Driven Railway Vehicles. In: *Proc. of the Fifth International Symposium on Linear Drives for Industry Applications - LDIA2005, Awaji Yumebutai, Hyogo, Japan*, 2005

[HWLL04] HSU, Chun-Fei ; WANG, Wen-June ; LEE, Tsu-Tian ; LIN, Chih-Min: Longitudinal control of vehicle platoon via wavelet neural network. In: *SMC (4)*, 2004, S. 3811–3816

[IML92] ISERMANN, Rolf ; MATKO, Drago ; LACHMANN, Karl-Heinz: *Adaptive Control Systems*. Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1992. – ISBN 0130054143

[Iva03] IVANCIC, Franjo: *Modeling and Analysis of Hybrid Systems*, University of Pennsylvania, Diss., 2003

[JGGS00] JENSEN, Henrik E. ; GULDSTR, Kim ; GULDSTR, Kim ; SKOU, Arne: Scaling up Uppaal Automatic Verification of Real-Time Systems using Compositionality and Abstraction. In: *Proceedings of the 6th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT 2000)* Bd. 1926. Pune, India : Springer Verlag, September 2000 (Lecture Notes in Computer Science (LNCS)), S. 19–30

[Kle08] KLEIN, Florian: *A Model-Driven Approach to Multi-Agent System Design*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, PhD Dissertation, 2008. – eingereicht

[KM98] KRAMER, J. ; MAGEE, J.: 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

[KMR02] KNAPP, A. ; MERZ, S. ; RAUH, C.: *Model Checking timed UML State Machines and Collaborations*. 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT 2002), Oldenburg, September 2002, Lecture Notes in Computer Science volume 2469 pages 395-414. Springer-Verlag, 2002

[KMS92] KRAMER, Jeff ; MAGEE, Jeff ; SLOMAN, Morris: Configuring distributed systems. In: *EW 5: Proceedings of the 5th workshop on ACM SIGOPS European workshop*. New York, NY, USA : ACM, 1992, S. 1–5

[Kop97] KOPETZ, Hermann: *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA, USA : Kluwer Academic Publishers, 1997. – ISBN 0792398947

[KP91] KESTEN, Yonit ; PNUELI, Amir: Timed and Hybrid Statecharts and Their Textual Representation. In: *Proceedings of the Second International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems* Bd. 571. London, UK : Springer Verlag, 1991 (Lecture Notes in Computer Science (LNCS)), S. 591–620

[Krä06] KRÄMER, Helmer: *Laufzeitanpassung von Softwarestrukturen für mechatronische Systeme*, University of Paderborn / Fakultät für Elektrotechnik – Informatik – Mathematik / Fachgebiet Softwaretechnik, Diplomarbeit, August 2006

[KSPL06] KRATZ, Fabian ; SOKOLSKY, Oleg ; PAPPAS, George J. ; LEE, Insup: R-Charon, a Modeling Language for Reconfigurable Hybrid Systems. In: *HSCC*, 2006, S. 392–406

[Kud05] KUDAK, Margarete: *Modulare Echtzeitverifikation hybrider UML-Komponenten*, University of Paderborn / Fakultät für Elektrotechnik – Informatik – Mathematik / Fachgebiet Softwaretechnik, Diplomarbeit, December 2005

[Lev95] LEVESON, Nancy G.: *Safeware : system safety and computers*. Addison-Wesley, 1995. – ISBN 0–201–11972–2

[LHLH01] LÜCKEL, Joachim ; HESTERMEYER, Thorsten ; LIU-HENKE, Xiaobo: Generalization of the Cascade Principle in View of a Structured Form of Mechatronic Systems. In: *IEEE/ASME International Conference on Advanced*

Intelligent Mechatronics (AIM 2001), Villa Olmo, Como, Italy Bd. 1, IEEE Service Center, Piscataway, 2001, S. 123–128

[LM99] LIBERZON, Daniel ; MORSE, A. S.: Basic problems in stability and design of switched systems. In: *IEEE Control Systems Magazine* 19 (1999), S. 59–70

[Lud95] LUDYK, G.: *Theoretische Regelungstechnik 1, Grundlagen, Synthese linearer Regelungssysteme*. Berlin / Heidelberg : Springer Verlag, 1995

[M96] MÉTAYER, Daniel L.: Software architecture styles as graph grammars. In: *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. New York, NY, USA : ACM, 1996. – ISBN 0–89791–797–9, S. 15–23

[Met07] METZLER, Björn: Decomposing Integrated Specifications for Verification. In: DAVIES, Jim (Hrsg.) ; GIBBONS, Jeremy (Hrsg.): *IFM* Bd. 4591, Springer Verlag, 2007 (Lecture Notes in Computer Science (LNCS)), S. 459–479

[MW07] METZLER, Björn ; WEHRHEIM, Heike: Extending a Component Specification Language with Time. In: *Electron. Notes Theor. Comput. Sci.* 176 (2007), Nr. 2, S. 47–67. – ISSN 1571–0661

[Neu07] NEUMANN, Stefan: *Modellierung und Verifikation von zeitbehafteten Graphtransformationssystemen mittels GROOVE*, University of Paderborn / Fakultät für Elektrotechnik – Informatik – Mathematik / Fachgebiet Softwaretechnik, Diplomarbeit, September 2007

[OHG04] OBERSCHELP, Oliver ; HESTERMEYER, Thorsten ; GIESE, Holger: Strukturierte Informationsverarbeitung für selbstoptimierende mechatronische Systeme. In: *Proc. of the Second Paderborner Workshop Intelligente Mechatronische Systeme* Bd. 145. Paderborn, Germany, 2004 (HNI-Verlagsschriftenreihe), S. 43–56

[OMG07] OMG ; OBJECT MANAGEMENT GROUP (Hrsg.): *UML 2.1.2 Superstructure Specification*. Document: formal/07-11-01. : Object Management Group, November 2007

[OMT98] OREIZY, Peyman ; MEDVIDOVIC, Nenad ; TAYLOR, Richard N.: Architecture-based runtime software evolution. In: *ICSE '98: Proceedings of the 20th international conference on Software engineering*. Washington, DC, USA : IEEE Computer Society, 1998. – ISBN 0–8186–8368–6, S. 177–186

[PJ04] PRAJNA, Stephen ; JADBABAIE, Ali: Safety Verification of Hybrid Systems Using Barrier Certificates. In: ALUR, Rajeev (Hrsg.) ; PAPPAS, George J. (Hrsg.): *HSCC* Bd. 2993, Springer Verlag, 2004 (Lecture Notes in Computer Science (LNCS)), 477-492

[PP05] PAPACHRISTODOULOU, Antonis ; PRAJNA, Stephen: A Tutorial on Sum of Squares Techniques for Systems Analysis. In: *Proceedings of the American Control Conference (ACC). Portland, OR, 2005.* Bd. 4, IEEE Computer Press, 2005, S. 2686 – 2700

[PQ08] PLATZER, André ; QUESEL, Jan-David: Logical Verification and Systematic Parametric Analysis in Train Control. In: EGERSTEDT, Magnus (Hrsg.) ; MISHRA, Bud (Hrsg.): *Hybrid Systems: Computation and Control, 10th International Conference, HSCC 2008, St. Louis, USA, Proceedings*, Springer Verlag, 2008 (Lecture Notes in Computer Science (LNCS))

[Pra05] PRAJNA, Stephen: *Optimization-Based Methods for Nonlinear and Hybrid Systems Verification*. California Institute of Technology, Pasadena, Calofornia, California Institute of Technology, PhD Dissertation, 2005

[Ren04] RENSINK, Arend: The GROOVE Simulator: A Tool for State Space Generation. In: PFALZ, J. (Hrsg.) ; NAGL, M. (Hrsg.) ; BÖHLEN, B. (Hrsg.): *Applications of Graph Transformations with Industrial Relevance (AGTIVE)* Bd. 3062, Springer Verlag, 2004 (Lecture Notes in Computer Science (LNCS)), S. 479–485

[Ric96] RICHERT, Jobst: Integration of Mechatronic Design Tools with CAMeL, Exemplified by Vehicle Convoy Control Design. In: *Proc. of the IEEE International Symposium on Computer Aided Control System Design*. Dearborn, Michigan, USA : IEEE Computer Press, 1996, S. 516–523

[RKS06] RENSINK, Arend ; KASTENBERG, Harmen ; STAIJEN, Tom: *User Manual for the GROOVE Tool Set*. Department of Computer Science, University of Twente, 2006

[Roz97] ROZENBERG, Grzegorz: *HANDBOOK of GRAPH GRAMMARS and COMPUTING by GRAPH TRANSFORMATION, Volume 1: Foundations*. World Scientific, 1997. – ISBN 9810228848

[SB03] SAMAD, T. (Hrsg.) ; BALAS, G. (Hrsg.): *Software-Enabled Control: Information Technology for Dynamical Systems*. IEEE Press and Wiley-Interscience, 2003. – 419 S.

[Sch06] SCHILLING, Daniela: *Kompositionale Softwareverifikation mechatronischer Systeme*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut

für Informatik, Universität Paderborn, PhD Dissertation, 2006

[SPP01] STAUNER, T. ; PRETSCHNER, A. ; PÉTER, I.: Approaching a Discrete-Continuous UML: Tool Support and Formalization. In: *Proc. UML'2001 workshop on Practical UML-Based Rigorous Development Methods – Countering or Integrating the eXtremists*. Toronto, Canada : Gesellschaft für Informatik, October 2001, S. 242–257

[SRKC00] SILVA, B. ; RICHESON, K. ; KROGH, B. ; CHUTINAN, A.: Modeling and verification of hybrid dynamical system using CheckMate. In: *ADPM 2000*, Shaker, 09 2000

[STF96] SHARASHIMA, F. ; TOMIZUKA, M. ; FUKUDA, T.: Mechatronics - „What Is It, Why, and How?" An Editorial. In: *Transactions on Mechatronics* Bd. 1. IEEE/ASME Transactions on Mechatronics, 1996, S. 1–4

[SW07] SCHÄFER, Wilhelm ; WEHRHEIM, Heike: The Challenges of Building Advanced Mechatronic Systems. In: *FOSE '07: 2007 Future of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2007, S. 72–84

[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

[Tic08] TICHY, Matthias: *Analyse und Verbesserung der Verlässlichkeit mechatronischer Systeme*. Fakultät für Elektrotechnik, Informatik und Mathematik / Institut für Informatik, Universität Paderborn, PhD Dissertation, 2008. – eingereicht

[TK02] TIWARI, Ashish ; KHANNA, Gaurav: Series of Abstractions for Hybrid Automata. In: TOMLIN, C.J. (Hrsg.) ; GREENSTREET, M.R. (Hrsg.): *Proceedings of the 5th International Workshop on Hybrid Systems: Computation and Control (HSCC 2002)* Bd. 2289. Stanford, CA, USA : Springer Verlag, März 2002 (Lecture Notes in Computer Science (LNCS)), S. 465ff

[TMV06] TRÄCHTLER, Ansgar ; MÜNCH, Eckehard ; VÖCKING, Henner: Iterative Learning and Self-Optimization Techniques for the Innovative Railcab-System. In: *Proceedings of the 32nd Annual Conference of the IEEE Industrial Electronics Society (IECON'06)*, IEEE Computer Press, 2006. – ISBN 1–4244–0391–X, S. 4683–4688

[Tri04] TRIPAKIS, Stavros: Folk Theorems on the Determinization and Minimization of Timed Automata. In: *Formal Modeling and Analysis of Timed Systems: First International Workshop, FORMATS 2003, Marseille, France, September 6-7, 2003. Revised Papers*, 2004. – ISBN 3–540–21671–5, S. 182 – 188

[Voe03] VOECKING, Henner: *Multirate-Verfahren und Umschaltstrategien für verteilte Reglersysteme*, Universität Paderborn, Diplomarbeit, 2003

[Wan04] WANG, Farn: Formal Verification of Timed Systems: A Survey and Perspective. In: *Proceedings of the IEEE* Bd. 92, IEEE Computer Press, August 2004, S. 1283–1307

[Win90] WING, Jeannette M.: A Specifier's Introduction to Formal Methods. In: *Computer* 23 (1990), Nr. 9, S. 8–23. – ISSN 0018–9162

[Woo00] WOOLDRIDGE, Michael: *Reasoning about Rational Agents*. 1st. The MIT Press, 2000. – 241 S. – ISBN 0262232138

[YEK98] YANAKIEV, Diana ; EYRE, Jennifer ; KANELLAKOPOULOS, Ioannis: Longitudinal Control of Heavy Duty Vehicles: Experimental Evaluation / University of California. Berkeley, CA, USA, 1998 (California PATH Research Report UCB-ITS-PRR-98-15). – Forschungsbericht

[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

[ZEA03] ZAMBOU, N. ; ENNING, M. ; ABEL, D.: Längsdynamikregelung eines Fahrzeugkonvois mit Hilfe der Modellgestützten Prädikativen Regelung. In: *Telematik 2003* Bd. VDI-Berichte 1785. Düsseldorf, Deutschland : VDI-Verlag, 2003

[Zün01] ZÜNDORF, Albert: *Rigorous Object Oriented Software Development*. Habilitation Thesis, University of Paderborn, 2001

[ZZ05] ZLOCKI, A. ; ZAMBOU, N.: Application of WLAN vehicle-to-vehicle communication for automatic guidance of a vehicle driven in platoon. In: *Proceedings of the 2nd International Workshop on Intelligent Transportation (WIT), März 2005, Hamburg*, 2005

Anhang A

Algorithmen zu zeitbehaften Graphtransformationssystemen

In diesem Anhang werden die Algorithmen, die bei der Erreichbarkeitsanalyse für zeitbehaftete Graphtransformationssysteme (siehe Abschnitt 4.4) verwendet werden, aufgelistet. Die informelle Beschreibung und Anwendung findet sich in Abschnitt 4.4.

Algorithmus A.1 Der Algorithmus $\mathcal{N}', \mathcal{E}' = graphID(m, x, G, P_l)$

procedure $\mathcal{N}', \mathcal{E}' = graphID(m, x, G, P_l)$

1: $\mathcal{N}', \mathcal{E}' = \emptyset$
2: **for all** $n \in \mathcal{N}$ **do**
3: $\quad v_g = m_v(V^{-1}_{(i,P)}(n))$
4: $\quad n' = V_{(i,G)}(v_g)$
5: $\quad \mathcal{N}' = \mathcal{N}' \cup n'$
6: **end for**
7: **for all** $e \in \mathcal{E}$ **do**
8: $\quad e_g = m_e(E^{-1}_{(i,P)}(e))$
9: $\quad e' = E_{(i,G)}(e_g)$
10: $\quad \mathcal{E}' = \mathcal{E}' \cup e'$
11: **end for**

Algorithmus A.2 Der Algorithmus $\langle T_m, R_m \rangle = assign(m, T, R, G, P_l)$

procedure $\langle T_m, R_m \rangle = assign(m, T, R, G, P_l)$

1: $T_m, R_m = \emptyset$
2: **for all** $t \in T : t := x_1 - x_2 \sim d, x_i = (M_1, \mathcal{N}_1, \mathcal{E}_1), x_2 := (M_2, \mathcal{N}_2, \mathcal{E}_2)$ **do**
3: $x'_1, x'_2 := x_0$
4: $t' := x'_1 - x'_2 \sim d$
5: **if** $x_1 \neq x_0$ **then**
6: $x'_1 := (M_1, \mathcal{N}'_1, \mathcal{E}'_1)$ mit $\mathcal{N}'_1, \mathcal{E}'_1 := \emptyset$
7: $\mathcal{N}'_1, \mathcal{E}'_1 = graphID(m, x_1, G, P_l)$
8: **end if**
9: **if** $x_2 \neq x_0$ **then**
10: $x'_2 := (M_2, \mathcal{N}'_2, \mathcal{E}'_2)$ mit $\mathcal{N}'_2, \mathcal{E}'_2 := \emptyset$
11: $\mathcal{N}'_2, \mathcal{E}'_2 = gaphID(m, x_2, G, P_l)$
12: **end if**
13: $T_m = T_m \cup t'$
14: **end for**
15: **for all** $r \in R : r := x, x = (M, \mathcal{N}, \mathcal{E})$ **do**
16: $\mathcal{N}', \mathcal{E}' = \emptyset$
17: $x' := (M, \mathcal{N}', \mathcal{E}')$
18: $r' := x'$
19: Erzeuge $x' := (M, \mathcal{N}', \mathcal{E}')$ mit $\mathcal{N}', \mathcal{E}' = \emptyset$
20: $\mathcal{N}', \mathcal{E}' := graphID(m, x, G, P_l)$
21: $R_m := R_m \cup r'$
22: **end for**

Algorithmus A.3 Der Algorithmus $C = prod_{clock}(\mathcal{C}, G)$

procedure $C = prod_{clock}(\mathcal{C}, G)$

1: $C = \emptyset$
2: **for all** $c \in \mathcal{C} : c = (M, C_l, C_i)$ **do**
3: **for all** $m = prod(C_l, G) : m = (m_v, m_e)$ **do**
4: $\mathcal{N}, \mathcal{E} = \emptyset$
5: $x = (M, \mathcal{N}, \mathcal{E})$
6: **for all** $v, e \in C_l$ **do**
7: $\mathcal{N} := N_{(i,G)}(m_v(v))$
8: $\mathcal{E} := E_{(i,G)}(m_e(e))$
9: **end for**
10: $C = C \cup x$
11: **end for**
12: **end for**

Algorithmus A.4 Der Algorithmus $\phi = clear(C, \phi)$

procedure $\phi = clear(C, \phi)$

1: **for all** $t \in \phi : t = (x_i - x_j \sim d)$ **do**
2: **if** $x_i \notin C \vee x_j \notin C$ **then**
3: $\phi := \phi \setminus t$
4: **end if**
5: **end for**

Algorithmus A.5 Der Algorithmus $G'_t = production_m(G_t, P_t, C, \mathcal{I}, m)$

procedure $G'_t = production_m(G_t, P_t, C, \mathcal{I}, m)$

1: $G_t := (G, \mathcal{C}, \mathcal{Z}); P_t := (P_l, P_r, T, V_i, E_i, R, f, r), m := (m_v, m_e)$
2: $\langle T', R' \rangle = assign(m, T, R)$
3: $\mathcal{Z}' = \emptyset$
4: **for all** $\phi \in \mathcal{Z}$ **do**
5: $\phi' = succ_\phi(\phi, I, T')$
6: **if** $\phi' notempty$ **then**
7: $\phi' = succ'_\phi(\phi', R')$
8: $\mathcal{Z}' = \mathcal{Z} \cup \phi'$
9: **end if**
10: **end for**
11: **if** $\mathcal{Z}' \neq \emptyset$ **then**
12: $G' = prod_{post}(m, P_l, P_r)$
13: $\mathcal{C}' = prod_{clock}(C, G')$
14: **for all** $\phi' \in \mathcal{Z}'$ **do**
15: $\phi' = \phi' \cap I(G')$
16: $\phi' = clear(\mathcal{C}', \phi')$
17: $\phi' = succ'_\phi(\phi', R')$
18: **end for**
19: $G'_t = (G', \mathcal{C}', \mathcal{Z}')$
20: **end if**

Algorithmus A.6 Der Algorithmus $\mathcal{G}'_t = production(G_t, P_t, C, \mathcal{I})$

procedure $\mathcal{G}'_t = production(G_t, P_t, C, \mathcal{I})$

1: $G_t := (G, \mathcal{C}, \mathcal{Z}); P_t := (P_l, P_r, T, V_i, E_i, R, f, r)$
2: $\mathcal{G}'_t = \emptyset$
3: **for all** $m = prod_{pre}(P_l, G) : m := (vertex, edge, ID)$ **do**
4: $\mathcal{G}'_t = \mathcal{G}'_t \cup production_m(G_t, P_t, C, \mathcal{I}, m)$
5: **end for**

Anhang B

Regeln zum Shuttlebeispiel aus Kapitel 4

In diesem Anhang werden die zeitbehafteten Graphtransformationsregeln für das Evaluierungsbeispiel aus Kapitel 4.5 beschrieben. Abbildungen B.1,B.2 und B.3 beschreiben jeweils die Clockinstanzen, die im System vorkommen. Um das Fortschreiten eines

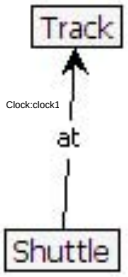


Abbildung B.1: Clockinstanz clock1

Shuttles zu gewährleisten, müssen noch Invarianten spezifiziert werden. Diese sind in den Abbildungen B.4, B.5 und B.5 dargestellt.

Abschließend werden noch die Regeln benötigt, die das Forschreiten eines Shuttles sowohl auf einem normalen Schienenabschnitt, als auch auf einer Weiche beschreiben. Diese Regeln besitzen die angegebenen Zeitguards:

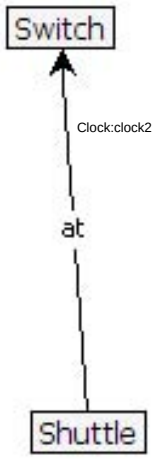


Abbildung B.2: Clockinstanz clock2

Abbildung B.3: Clockinstanz clock3

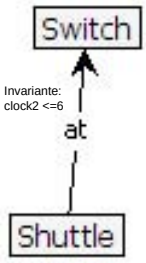


Abbildung B.4: Clockinvariante zum Überfahren einer Weiche

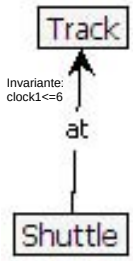


Abbildung B.5: Clockinvariante zum Überfahren eines Schienenabschnittes

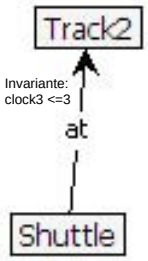


Abbildung B.6: Clockinvariante zum Überfahren eines Schienenabschnittes

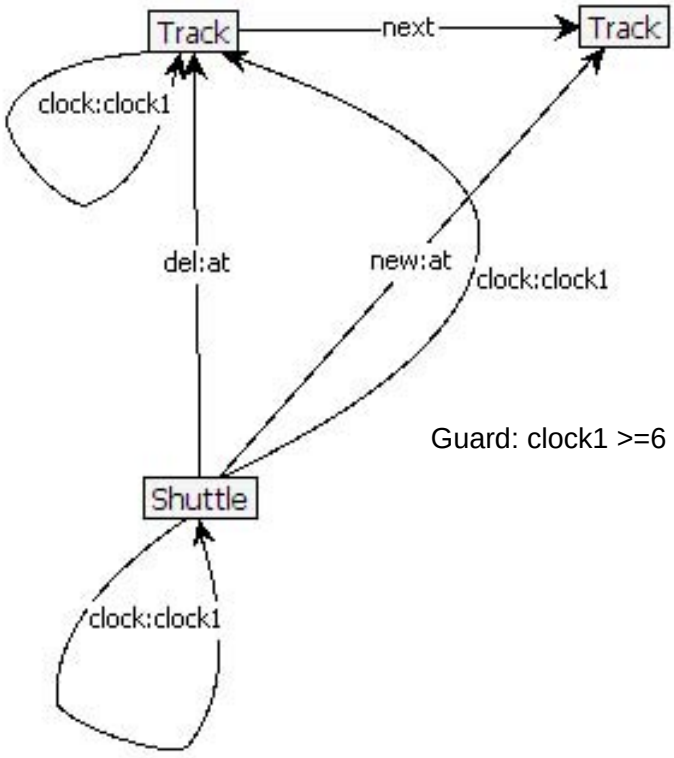


Abbildung B.7: Zeitbehaftete Regel zum Forbewegen von einem Schienenabschnitt zum nächsten

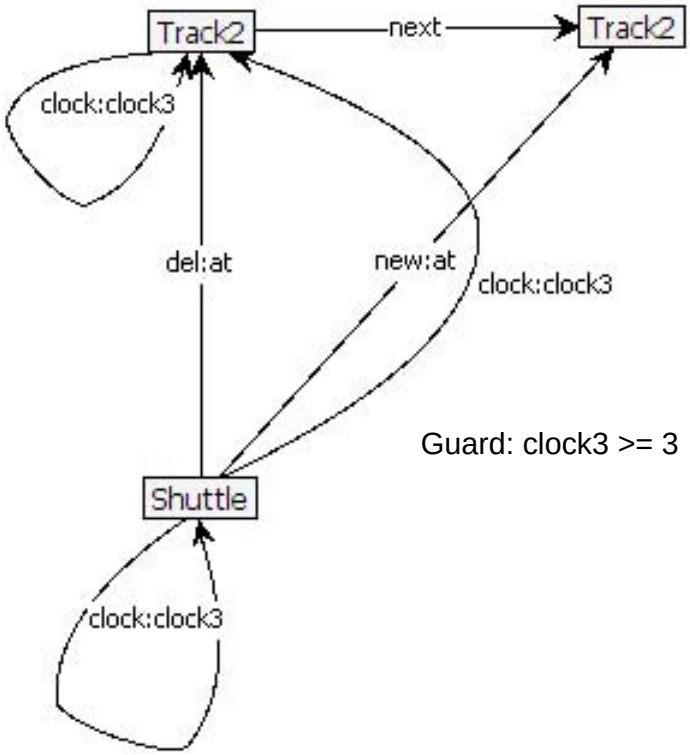


Abbildung B.8: Zeitbehaftete Regel zum Forbewegen von einem Schienenabschnitt zum nächsten

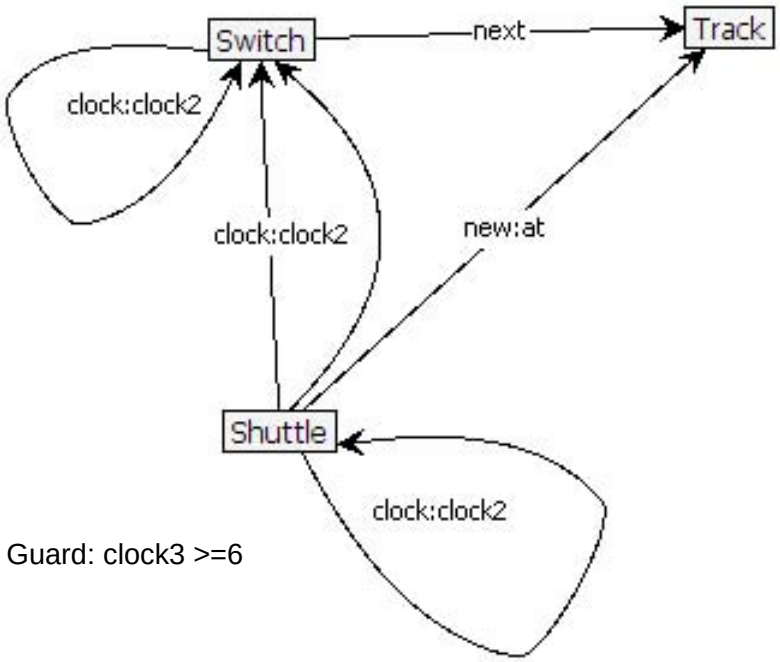


Abbildung B.9: Zeitbehaftete Regel, um von einer Weiche zu fahren

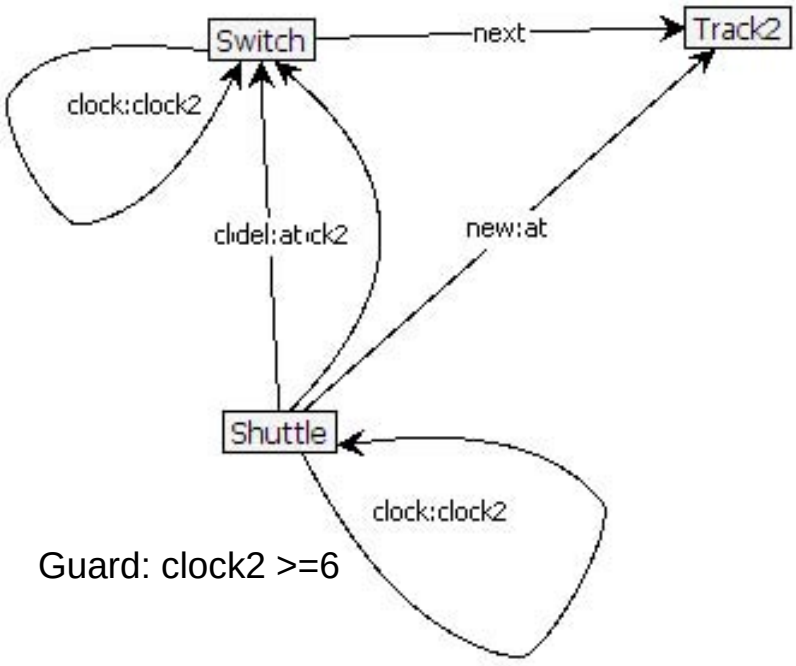


Abbildung B.10: Zeitbehaftete Regel, um von einer Weiche zu fahren

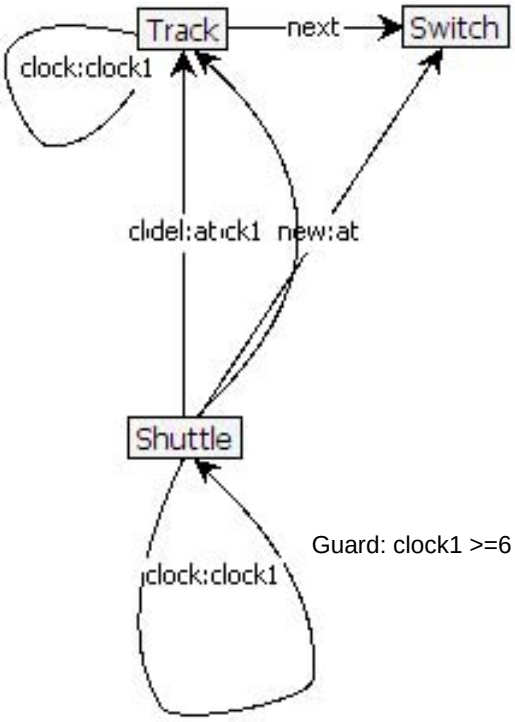


Abbildung B.11: Zeitbehaftete Regel, um auf eine Weiche zu fahren

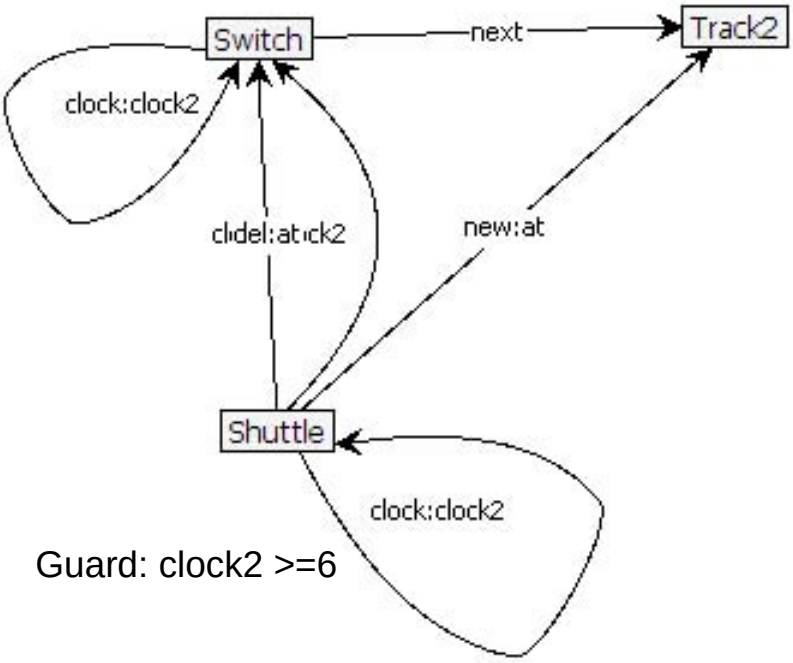


Abbildung B.12: Zeitbehaftete Regel, um von einer Weiche zu fahren

Abbildungsverzeichnis