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

Gefahrenanalyse selbstoptimierender Systeme

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

vorgelegt von

Dipl.-Wirt.-Inf. Matthias Tichy

Promotionskommission:
Vorsitzender: Prof. Dr. Wilhelm Schäfer (Universität Paderborn)
Koreferat: Prof. Dr. Heike Wehrheim (Universität Paderborn)
Koreferat: Prof. Dr.-Ing. Peter Liggesmeyer (Technische Universität Kaiserslautern)
Beisitzer: Prof. Dr. Albert Zündorf (Universität Kassel)
Beisitzer: Prof. Dr. Stefan Böttcher (Universität Paderborn)

Die Dissertation wurde am 19. Dezember 2008 bei der Fakultät für Elektrotechnik, Informatik und Mathematik der Universität Paderborn eingereicht und am 6. Mai 2009 vor der Promotionskommission verteidigt und durch die Fakultät angenommen.

Zusammenfassung

Selbstoptimierende, mechatronische Systeme können ihr Verhalten in Reaktion auf Änderung der Umgebung oder der Ziele anpassen und somit diese Ziele besser als herkömmliche mechatronische Systeme erfüllen. Da selbstoptimierende Systeme oftmals in sicherheitskritischen Systemen eingesetzt werden, muss bei der Entwicklung ein besonderer Schwerpunkt auf die Sicherheit gelegt werden.

In dieser Arbeit wird eine Gefahrenanalyse für selbstoptimierende Systeme vorgestellt, die speziell auf die Eigenschaften dieser Systeme, im Besonderen die Verhaltensanpassung durch Strukturrekonfiguration, zugeschnitten ist.

Die Gefahrenanalyse basiert auf einer Spezifikation der verschiedenen Fehler in einzelnen Komponenten des Systems sowie deren Propagierung durch die Komponentenstruktur. Dies ermöglicht dann in Kombination mit einer Gefahrenspezifikation qualitative und quantitative Analysen des Auftretens der Gefahr.

Diese Analysen betrachten nicht nur die verschiedenen Fehler im System, sondern explizit auch die sich aus der Selbstoptimierung ergebenden verschiedenen Strukturkonfigurationen. So können beispielsweise die beste und schlechteste Strukturkonfiguration in Bezug auf die Wahrscheinlichkeit der Gefahren aber auch alle Konfigurationen berechnet werden, welche geforderte Wahrscheinlichkeiten einhalten.

Grundlage dieser Gefahrenanalyse für selbstoptimierende Systeme ist eine geeignete Spezifikation der Struktur und deren Rekonfiguration. Die hierfür entwickelte Spezifikationssprache nutzt Graphtransformationen, um visuell die Transformation einer Strukturkonfiguration in eine andere Konfiguration zu modellieren. Die Sprache ist stark typisiert und formal so definiert, dass bei der Ausführung der Transformationen nur typkonforme Strukturkonfigurationen entstehen, die dann von der Gefahrenanalyse ausgewertet werden.

Die in der Arbeit beschriebenen Konzepte werden anhand eines realen selbstoptimierenden Systems, den RailCabs der Neuen Bahntechnik Paderborn, erläutert. Die Konzepte wurden prototypisch in einem Entwurfswerkzeug für die Software selbstoptimierender Systeme umgesetzt.

Abstract

Advanced embedded systems increasingly contain self-optimizing behavior to improve or maintain properties like quality of service, dependability, or performance in spite of failures or environmental changes. As embedded systems are often employed in a safety-critical context, the effects of self-optimization on the safety must be analyzed carefully. Self-optimization is usually implemented by software and results in the exchange of system components at runtime which means a change of the architecture.

This thesis presents a hazard analysis approach for such systems which is geared especially towards the properties of self-optimizing systems, e.g. architectural reconfiguration for behavioral adaptation.

The hazard analysis is based on a specification of the errors and failures of individual components as well as their propagation in the system's component structure. This specification enables qualitative and quantitative hazard analyses. For the special case of self-optimizing systems, the different configurations of the system's component structure are taken into account.

The hazard analysis approach contains a modeling language for the structure and structural reconfigurations. The specification of structure is based on UML component diagrams and composite structures. The accompanying language for structural reconfigurations combines graph transformations with the concrete syntax of component diagrams for a tight integration with the modeling language for component structures.

Examples from the RailCab-Project are used to illustrate the presented concepts. The hazard analysis has been implemented prototypically in the Fujaba4Eclipse case tool.

Danksagung

Ich bedanke mich bei meinem Doktorvater Prof. Dr. Wilhelm Schäfer für die wissenschaftliche Betreuung in den vergangenen Jahren, die Unterstützung bei der Verfassung dieser Arbeit und für die Möglichkeit in seiner Gruppe forschen zu dürfen. Prof. Dr. Heike Wehrheim und Prof. Dr.-Ing. Peter Liggesmeyer danke ich für die Übernahme der weiteren schriftlichen Gutachten. Für die Teilnahme an meiner Promotionskommission danke ich weiterhin Prof. Dr. Albert Zündorf und Prof. Dr. Gregor Engels.

Prof. Dr. Holger Giese hat mich durch die vielen interessanten Fachgespräche bei dieser Arbeit unterstützt. Die (ehemaligen) Mitarbeiter des Fachgebiets Softwaretechnik Dr. Jörg Niere, Dr. Jörg Wadsack, Ulrich Nickel, Dr. Matthias Gehrke, Ahmet Mehic, Robert Wagner, Dr. Lothar Wendehals, Björn Axenath, Matthias Meyer, Dr. Daniela Schilling, Dr. Sven Burmester, Prof. Dr. Ekkart Kindler, Florian Stallmann, Dr. Vladimir Rubin, Dr. Martin Hirsch, Stefan Henkler, Dietrich Travkin, Christian Bimmermann, Jan Meyer, Joel Greenyer, Oliver Sudmann, Markus von Detten, Claudia Priesterjahn und Jörg Holtmann haben mir mit der sehr guten Arbeitsatmosphäre und den vielen kreativen Diskussionen geholfen, diese Arbeit zu verfassen. Christian Henke, Carsten Rustemeier, Jens Geisler, Helene Waßmann und Tobias Schneider danke ich für die interessante Zusammenarbeit im RailCab-Projekt.

Darüberhinaus bedanke ich mich bei Frau Jutta Haupt für die Überwindung jeglicher organisatorischen Hürden und Herrn Jürgen Maniera für die technische Unterstützung.

Herzlicher Dank gilt meiner Mutter, Vera, Claudia, Martin, Matthias M., Dietrich und Jörg für das Korrekturlesen dieser Arbeit.

Diese Arbeit wäre ohne die Hilfe einer Vielzahl von Studenten nicht möglich gewesen. Im Besonderen danke ich Jörg Holtmann und Stefan Scharberth sowie den Studenten der Projektgruppe „Automotive Software Engineering“.

Ein großer Dank gilt auch meiner Mutter für die Unterstützung auf meinem ganzen Lebensweg. Schlussendlich wäre diese Arbeit nicht ohne die seelische Unterstützung meiner Frau Vera möglich gewesen. Ihr gilt mein besonderer Dank.

Inhaltsverzeichnis

1 Einleitung

In den letzten Jahren ist der Einsatz von Software in technischen Systemen stark gestiegen. Diese Systeme zeichnen sich durch die Kombination von Maschinenbau, Elektrotechnik und Informatik aus. Für diese Kombination wurde der Begriff *Mechatronik* geprägt [VDI04]. Derzeit sind zum Beispiel 70% der Innovationen in der Automobilindustrie getrieben durch die in technischen Systemen eingebettete Software und es wird erwartet, dass der Anteil der Software an den Entwicklungskosten von 20% auf bis zu 40 % steigt (vgl. [Gri03, HKK04]). Durch diesen steigenden Einfluss mechatronischer Systeme auf die Gesellschaft wird deren Verlässlichkeit [Lap92] (Kenngrößen: Sicherheit, Verfügbarkeit, Zuverlässigkeit und Vertraulichkeit) immer wichtiger.

Besondere Herausforderungen bei der Entwicklung verlässlicher, mechatronischer Systeme ergeben sich durch die oben skizzierte Kombination verschiedener Domänen und deren spezifischen Entwicklungstechniken. Vor allem die Software hat einen großen Einfluss auf die Verlässlichkeit. Für die Entwicklung verlässlicher, mechatronischer Systeme müssen also Techniken bereitgestellt werden, die explizit auf die domänenübergreifenden Eigenschaften mechatronischer Systeme eingehen.

Selbstoptimierende Systeme sind mechatronische Systeme mit einer inhärenten Teilintelligenz, die in der Lage sind autonom und flexibel auf Änderungen in der Umgebung zu reagieren [GKP08]. Diese Reaktion resultiert in einer Anpassung oder Neugewichtung der Ziele und nachfolgend einer Anpassung des Verhaltens z.B. durch Rekonfiguration der Struktur [FGK+04]. Selbstoptimierung ermöglicht damit Systeme, die je nach Situation im Vergleich zu herkömmlichen Systemen (z.B. adaptiven Systemen) ein besseres und durch die Zieländerung passenderes Verhalten zeigen.

Diese neuartigen Systeme stellen neue Herausforderungen für die Modellierung und Analyse der Verlässlichkeit dar [FGM+07]. So muss die Ziel- und nachfolgende Verhaltensanpassung geeignet modelliert werden. Die resultierenden Modelle müssen einer rigorosen Analyse der Verlässlichkeit unterworfen werden, die speziell auf die Besonderheiten selbstoptimierender Systeme, vor allem der Verhaltensanpassung, zugeschnitten ist.

1.1 RailCab – ein selbstoptimierendes System

Ein Beispiel für ein selbstoptimierendes System ist die Neue Bahntechnik Paderborn (NBP). Die Forschung im Rahmen der Neuen Bahntechnik Paderborn wurde 1998 mit dem Ziel begonnen, die Vorteile von Individualverkehr und öffentlichem Verkehr zu verbinden. Grundlegende Idee ist, den herkömmlichen Schienenverkehr durch autonome selbstfahrende Fahrzeuge – die so genannten RailCabs – zu ersetzen. RailCabs sind in verschiedenen Ausführungen für den Personenverkehr und den Güterverkehr geplant. Abbildung 1.1 zeigt zwei RailCab-Fahrzeuge in der Simulation und auf der Teststrecke der NBP.

RailCabs können einzeln fahren oder sich auf längeren Strecken mit einer höheren Verkehrsdichte zu autonomen Konvois berührungslos zusammenschließen. Die im Vergleich zu herkömmlichen Zügen kleineren autonom fahrenden Einheiten und die ohne mechanische Kupplungsvorgänge auskommende Bildung von Konvois ermöglicht eine flexiblere Nutzung der RailCabs, da auf Anforderung des Nutzers gefahren werden kann und keine starren Fahrpläne existieren.

Abbildung 1.1: Zwei RailCabs in der Simulation und auf der Teststrecke

Software ist ein integraler Bestandteil eines RailCabs, da viele Funktionen wie das autonome Fahren, die Bildung von Konvois, das Federneige-Modul und die aktive Steuerung nur mit Hilfe von Software realisiert werden können. Ein großer Teil der Software ist für die Regelung der physikalischen Aktoren zuständig. Weiterhin wird Software u.a. für die Echtzeitkoordination, die Reaktion im Notfall sowie die Auswahl und den Wechsel der Regler genutzt. Im Rahmen dieser Arbeit werden die folgenden Szenarien betrachtet.

Konvoibetrieb Die Konvoifahrt besteht aus mehreren Phasen. In einer ersten Phase wird zwischen den Fahrzeugen kommuniziert, ob ein Konvoi aufgebaut wird bzw. ein weiteres Fahrzeug dem Konvoi beitritt. Während der darauf folgenden Konvoifahrt tauschen die Fahrzeuge periodisch Daten aus. So

schickt ein ausgewähltes Fahrzeug, das so genannte Koordinator-Fahrzeug, periodisch Solldaten an die übrigen Fahrzeuge, um die Konvoistabilität zu garantieren. Die letzte Phase der Konvoifahrt besteht aus der Auflösung eines Konvois bzw. dem Austritt eines einzelnen Fahrzeugs aus einem Konvoi. Hierfür werden ebenfalls zwischen den Fahrzeugen Nachrichten ausgetauscht. Der Konvoibetrieb stellt insgesamt hohe Anforderungen an die Sicherheit des Gesamtsystems [HVB+05, HTS+08b, HTS+08a].

Komfort durch Federneige-Modul Beim Konvoibetrieb wird die Längsdynamik des Fahrzeugs sowie des ganzen Konvois geregelt. Der Komfort der Fahrgäste in einem Fahrzeug wird allerdings auch durch die oben/unten- und links/rechts-Dynamik des Fahrzeugs bestimmt. Die grundlegende Idee ist, dass ein Fahrzeug eine Referenztrajektorie über die Strecke für die aktive Federung erhält. Auf Basis dieser Daten kann dann die Regelung der aktiven Federung den Komfort optimieren [TMV06].

1.2 Motivation

Zur Modellierung und der Analyse der Verlässlichkeit selbstoptimierender Systeme wurde die Mechatronic UML [BGT05, GTB+03, GBSO04, BBG+06] als domänenspezifische Verfeinerung der UML [Obj07] entwickelt. Die Mechatronic UML unterstützt die komponentenbasierte [Szy98] Spezifikation der Systemarchitektur sowie die Modellierung des Verhaltens der einzelnen Komponenten. Zur qualitativen Analyse der Sicherheit eines selbstoptimierenden Systems wurden verschiedene formale Verfahren entwickelt, um die Korrektheit von Verhaltensmodellen gegenüber Sicherheitseigenschaften zu überprüfen.

Mit der Mechatronic UML kann die Software für viele Aspekte der obigen Szenarien spezifiziert und deren Korrektheit bzgl. Sicherheitseigenschaften formal überprüft werden [GTB+03, GBSO04, BBG+06]. Die Ansätze der Mechatronic UML zur formalen Verifikation verfolgen die Strategie Gefahren zu vermeiden, indem systematische Fehlerursachen in den Softwaremodellen während des Entwurfs erkannt werden und diese dann behoben werden, bevor die Systeme eingesetzt werden und Gefahrensituationen auftreten können.

Die Mechatronic UML enthält allerdings bisher keine Gefahrenanalyse, die die Fehlerpropagierung [Lap92] von zufälligen Fehlerursachen betrachtet. Zufällige Fehlerursachen, wie mögliche Ausfälle von Hardware und falsche Sensordaten, können nicht immer aus dem System entfernt werden. Wenn diese Fehlerursachen sich in Fehlerzuständen manifestieren, könnten sie zu Ausfällen einer Komponente führen, die sich dann über Verbindungen zwischen den Komponenten weiter

durch das System propagieren und zu Gefahrensituationen führen können.

Selbstoptimierende, mechatronische Systeme besitzen einige spezielle Eigenschaften, welche die Fehlerpropagierung und deren Auswirkungen auf den Eintritt von Gefahren beeinflussen. So sind diese Systeme typischerweise stark hierarchisch aufgebaut, einzelne Fehler können Auswirkungen auf viele Teile des Systems haben, das Verhalten bzgl. der Fehler ist oftmals vom Typ des Fehlers abhängig und durch den starken Einsatz von Reglern ergeben sich zyklische Fehlerpropagierungen.

Besonderer Faktor bei der Fehlerpropagierung in selbstoptimierenden Systemen ist die Rekonfiguration. Hierdurch ist die Systemstruktur, die bei der Propagierung betrachtet werden muss, nicht starr, sondern es existieren viele Strukturen, in denen unterschiedliche Fehlerursachen enthalten sind und damit unterschiedliche Fehlerzustände auftreten können, die zu verschiedenen Gefahren oder zu unterschiedlichen Gefahrenwahrscheinlichkeiten führen können.

Durch die starke Vernetzung selbstoptimierender Systeme [SW07] können Fehler Auswirkungen auf viele verschiedene Teilsysteme haben. Bei einer Gefahrenanalyse auf Grundlage der Fehlerpropagierung müssen daher große Systeme mit vielen Teilen betrachtet werden. Ein Ansatz sollte daher für große Systeme skalieren.

Manche dieser spezifischen Eigenschaften selbstoptimierender Systeme, wie zyklische Strukturen, Rekonfiguration und common-mode Fehler, werden von verwandten Ansätzen [FM93, Wal05, KLM03, Gru03b, GKP05, KGF07, PM99, ORS06, GOR06] zur Gefahrenanalyse betrachtet. Es existiert allerdings kein Ansatz, der alle dieser Eigenschaften unterstützt.

Grundlage einer Analyse der Fehlerpropagierung und der resultierenden Gefahren in selbstoptimierenden Systemen muss eine geeignete Modellierung der Fehlerpropagierung sowie der Komponentenstrukturen und deren Rekonfiguration sein.

Die Mechatronic UML unterscheidet zwischen Komponententypen und deren Instanzen analog zur UML sowie anderen Architekturbeschreibungssprachen wie Darwin [MDK94] und Acme [GMW00]. Die Mechatronic UML bietet bereits eine Möglichkeit Komponentenstrukturen hierarchisch zu beschreiben. Die Modellierung der Softwarearchitektur ist allerdings dahingehend eingeschränkt, dass jede Komponenteninstanz, die in einem Komponententyp eingebettet ist, implizit die Kardinalität $0 \ldots 1$ zugewiesen bekommt. Die Rekonfiguration zur Verhaltensanpassung in Hybriden Rekonfigurationscharts [GBSO04] beschränkt sich daher auf die Aktivierung und Deaktivierung einzelner Komponenteninstanzen.

Durch die immer weitergehende Vernetzung mechatronischer und selbstoptimierender Systeme und die zugehörigen sich ständig ändernden Systemstruktu-

ren mit einer variablen Anzahl an Teilnehmern [SW07], wie auch in [BSDB00] für mechatronische Systeme bereits beschrieben, kann die Struktur dieser Systeme und deren Rekonfiguration mit den derzeitigen Spezifikationssprachen der Mechatronic UML nicht modelliert werden.

Die Modellierungstechniken müssen also dahingehend erweitert werden, dass die Kommunikation eines Teilsystems mit einer variablen und zum Entwurfszeitpunkt unbekannten Anzahl an Partnern spezifiziert werden kann. Diese variable Anzahl an Kommunikationspartnern kann dazu führen, dass die interne Softwarearchitektur eines Teilsystems ebenfalls eine variable und beim Entwurf unbekannte Anzahl an Komponenteninstanzen und Portinstanzen enthält. Solch ein System muss als Erzeugendensystem modelliert werden, indem die Struktur je nach Situation auf Basis einer regelbasierten Spezifikation durch Erzeugen und Löschen von Komponenten- und Portinstanzen sowie der Änderung der Kommunikationsstruktur angepasst wird.

Eine Modellierungssprache zur regelbasierten Spezifikation der Rekonfiguration muss die in [BCDW04] beschriebenen Anforderungen für rekonfigurierbare Systeme erfüllen. Hier wird neben der Möglichkeit zur Spezifikation einzelner Rekonfigurationen gefordert, dass auch Sequenzen, Entscheidungen und Schleifen unterstützt werden.

Durch den Einsatz selbstoptimierender Systeme in sicherheitskritischen Umgebungen ist die Vermeidung von Fehlern ein wichtiger Aspekt bei der Spezifikation der Rekonfiguration. Ein möglicher Fehler ist eine fehlerhafte Spezifikation der Regeln hinsichtlich der Typisierung der Komponenten, Ports und Konnektoren. Solche Fehler sollten so früh wie möglich im Entwicklungsprozess erkannt und behoben werden und nicht erst bei der Ausführung der Regel zur Laufzeit. Dementsprechend sollte die Spezifikationssprache für die Rekonfigurationsregeln stark typisiert, semantisch geeignet definiert und möglichst durch einen kontextsensitiv syntaxgesteuerten Editor umgesetzt werden, so dass es nicht möglich ist Rekonfigurationsregeln zu spezifizieren, die bei der Ausführung nicht typkonforme Konfigurationen erzeugen.

Mechatronische Systeme müssen harte Echtzeitanforderungen einhalten. Dementsprechend muss für die Ausführung der Rekonfigurationsregeln deren maximale Ausführungszeit (WCET) berechnet werden können.

Schließlich muss eine solche Sprache für den Entwickler intuitiv zu nutzen sein. Eine Möglichkeit hierzu stellt eine graphische Sprache dar. Da mit dieser Sprache die Rekonfiguration von Komponentenstrukturen spezifiziert wird, sollte sie auch geeignet mit der Modellierungssprache für Komponentenstrukturen integriert sein.

Die oben dargestellten verwandten Arbeiten für die Analyse der Fehlerpropagierung unterstützen nicht die regelbasierte Spezifikation der Rekonfiguration.

Verwandte Ansätze im Bereich der Modellierung rekonfigurierbarer Architekturen [TGM00, KKJD06, Gru03c, Mét98, WLF01] wie auch die bereits in der MECHATRONIC UML enthaltenen Formalismen [Krä06, Sch06, Hir08] erfüllen nur einige der dargestellten Anforderungen und sind nicht mit Ansätzen zur Analyse der Fehlerpropagierung integriert.

1.3 Ziel und Lösungsansatz

Ziel dieser Arbeit ist eine Gefahrenanalyse selbstoptimierender Systeme auf Basis der Fehlerpropagierung zufälliger Fehler. Die Gefahrenanalyse unterstützt im Gegensatz zu den vorgestellten verwandten Ansätzen die im vorherigen Abschnitt beschriebenen speziellen Eigenschaften selbstoptimierender Systeme.

Voraussetzung für die Gefahrenanalyse ist eine geeignete Modellierung der Komponentenstrukturen und deren Rekonfiguration, welche die im vorherigen Abschnitt aufgestellten Anforderungen erfüllt, da auf diesen Modellen die Algorithmen der Gefahrenanalyse arbeiten.

Im Folgenden werden die Lösungsansätze für die Modellierungssprache sowie die darauf aufbauende Gefahrenanalyse genauer vorgestellt.

1.3.1 Modellierung der Komponentenstrukturen und deren Rekonfiguration

Die Modellierung der Komponententypen in der MECHATRONIC UML wird um das UML 2 [Obj07] Konzept der Kompositionsstrukturen (Composite Structures) erweitert. Dies ermöglicht, die in einem Komponententyp eingebetteten Komponenten- und Portinstanzen mit Kardinalitäten zu versehen. Das gleiche gilt auch für Konnektoren zwischen den Komponenteninstanzen, die eine Quell- und Zielkardinalität erhalten.

Abbildung 1.2 zeigt die Spezifikation des Komponententyps eines RailCabs für das Szenario des Konvoibetriebs des Anwendungsbeispiels. Instanzen dieses Komponententyps können sowohl als einzelnes Fahrzeug unter Nutzung des Geschwindigkeitsreglers als auch in einem Konvoi fahren. Bei der Konvoifahrt schickt das Koordinator-Fahrzeug an die folgenden Fahrzeuge Positionssolldaten. Diese werden von passend verbundenen Instanzen des Komponententyps RefGen berechnet. Die nachfolgenden Fahrzeuge nutzen diese Daten für den Positionsregler. Die Komponenten zur Berechnung der Positionssolldaten sind als Kette verbunden.

Mit dieser erweiterten Komponententypdefinition werden *deklarativ* viele verschiedene Konfigurationen als Ausprägungen der zugehörigen Instanzen zur Lauf-

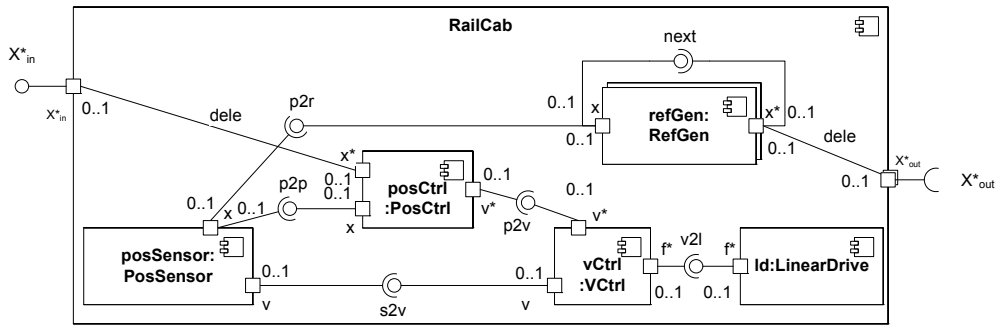


Abbildung 1.2: Hierarchischer RailCab Komponententyp

zeit beschrieben.

Die Rekonfiguration zwischen diesen Konfigurationen wird *operational* mit Hilfe von Graphtransformationen modelliert. Dies schränkt die Menge der möglichen Konfigurationen auf die durch die Ausführung der Graphtransformationen Erreichbaren ein. Die operationale Spezifikation ermöglicht eine direkte Ausführung der Rekonfigurationen durch eine automatische Quelltextsynthese sowie die Berechnung der maximalen Ausführungszeit (WCET), die in mechatronischen Systemen mit harten Echtzeiteigenschaften bekannt sein muss.

Die Graphtransformationen werden nicht wie Storydiagramme [FNTZ98, Zün02] auf dem Metamodell der Komponenteninstanzstrukturen spezifiziert, da dies zu komplexen Modellen führt, die zum Verständnis die Kenntnis des Metamodells voraussetzen. Stattdessen werden *Komponentenstorydiagramme* als spezielle Anpassung von Storydiagrammen für hierarchische Komponentenstrukturen vorgestellt, die auf Basis der annotierten konkreten Syntax der Komponentenstrukturen Rekonfigurationen beschreiben.

Die resultierende Transformationssprache erfüllt die in [BCDW04] vorgestellten Anforderungen an die Spezifikation von Rekonfiguration vor allem hinsichtlich einzelner Operationen sowie der Spezifikation von Sequenzen, Entscheidungen und Schleifen. Ebenfalls ermöglicht die Transformationssprache die Spezifikation typsicherer Rekonfigurationen und unterstützt hierarchische Strukturen.

Abbildung 1.3 zeigt das Komponentenstorydiagramm für die Rekonfiguration der Komponenteninstanzstruktur des Koordinatorfahrzeugs, wenn ein zusätzliches Fahrzeug dem Konvoi beitritt. Parameter sind u.a. die Position des neuen Fahrzeugs im Konvoi sowie die Portinstanz, über die die Solldaten verschickt werden. Das Komponentenstorydiagramm traversiert die Kette an Komponenteninstanzen für die Berechnung der Solldaten und fügt an der korrekten Stelle eine neue Komponenteninstanz ein.

Diese Arbeit betrachtet nur die strukturellen Aspekte der Rekonfiguration.

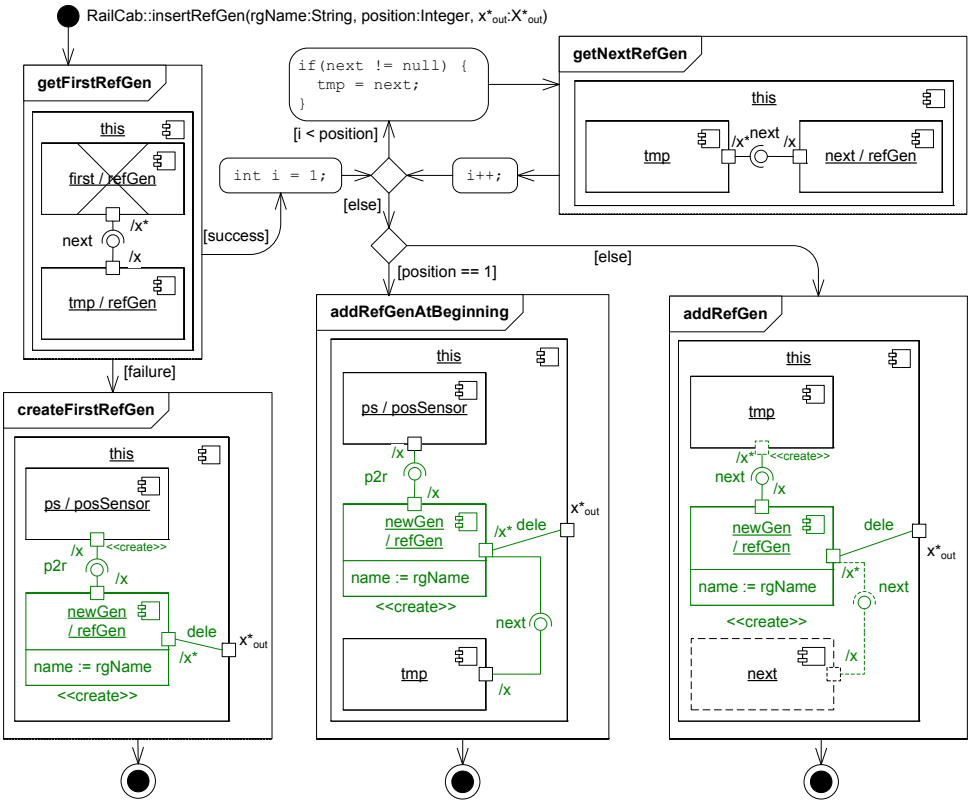


Abbildung 1.3: Komponentenstorydiagramm für die Rekonfiguration des Koordinatorfahrzeugs, wenn ein neues Fahrzeuges in den Konvoi eintritt

Die resultierenden Auswirkungen auf die zustandsbasierten Verhaltensmodelle und deren Verifikation werden für ähnliche Ansätze zur regelbasierten Rekonfiguration in [Krä06, HHG08, Hir08] betrachtet.

1.3.2 Gefahrenanalyse

Die Sicherheit wird maßgeblich durch Fehler beeinflusst. Man unterscheidet zufällige und systematische Fehler [Sto96]. Zufällige Fehler entstehen durch physikalische Effekte wie Abnutzung von Bauteilen, während der Entwickler systematische Fehler, wie z.B. Fehler im Entwurf oder im Quelltext der Software, verursacht. In dieser Arbeit werden zufällige Fehler und deren Auswirkungen auf die Sicherheit betrachtet, indem überprüft wird, ob sie zu einer Gefahr führen können. Systematische Fehler können durch formale Verifikationsverfahren wie Model Checking oder auch systematische Testverfahren erkannt [Lap92] und

nachfolgend behoben werden (z.B. [Pel01, BGH+05]).

Verschiedene Techniken, die auch zufällige Fehler betrachten, werden in der Praxis zur Analyse der Kenngrößen der Verlässlichkeit eingesetzt. Dies sind zum Beispiel Fehlerbaumanalyse (FTA), Hazard and Operability Study (HAZOP), Failure Mode Effect Analysis (FMEA), und Zuverlässigkeitsblockdiagramme (vgl. [Sto96]). Die dabei entwickelten Modelle werden typischerweise unabhängig von der Systemarchitektur erstellt und können daher von dieser abweichen. Dies kann zu ungenauen oder falschen Analyseergebnissen führen, da z.B. Fehlerpropagierungen vergessen werden können. In dieser Arbeit wird daher die Systemarchitektur als Basis für eine Gefahrenanalyse genutzt. Eine Gefahr ist hierbei eine Situation, die zu einem Unfall führen kann [Lev95].

Für die vorgeschlagene Gefahrenanalyse muss die MECHATRONIC UML um Möglichkeiten zur Beschreibung des abstrakten Fehlerverhaltens einzelner Komponenten sowie der Konnektoren zwischen Komponenten erweitert werden. Das Fehlerverhalten der Komponenten beschreibt im Wesentlichen, welche Fehlerzustände innerhalb der Komponenten eintreten können, wie die Komponenten auf die Fehlerzustände durch Ausfälle reagiert und wie Komponenten auf Ausfälle anderer Komponenten reagieren. Dieses wird im Weiteren Fehlerpropagierungsverhalten genannt. Abbildung 1.4 zeigt die Fehlerpropagierung verschiedener Fehlerzustände (e) über Ausfälle (f) innerhalb einer Komponentenstruktur.

Auf Basis der Komponenten und der spezifizierten Fehlerpropagierung sowie der Komponentenstruktur wird in dieser Arbeit eine komponentenbasierte Gefahrenanalyse [GTS04, GT06] als Erweiterung der FTA vorgestellt, welche die Wahrscheinlichkeiten von Gefahren bzw. deren Risiken berechnet. Die Fehlerpropagierung der Komponenten und die Gefahrensituationen werden mit booleschen Formeln beschrieben.

Selbstoptimierende Systeme verändern zur Laufzeit die Komponentenstruktur, auf der die Gefahrenanalyse beruht. Dies bedeutet, es gibt für jede Komponente zur Laufzeit verschiedene Konfigurationen mit unterschiedlichen Strukturen der unterlagerten Komponenteninstanzen.

Eine Gefahrenanalyse, die auf der Komponentenstruktur basiert, muss natürlich die durch die Selbstoptimierung entstehenden unterschiedlichen Konfigurationen mitbetrachten. Es müssen so zum Beispiel die Konfigurationen identifiziert werden, die die Sicherheitsanforderungen bzgl. der Gefahrenwahrscheinlichkeit bzw. des Risikos verletzen. Zusätzlich ist es oft sinnvoll, nur die schlechteste Konfiguration aus der möglicherweise großen Menge an Konfigurationen zu identifizieren, da diese dann eine obere Schranke für die Gefahrenwahrscheinlichkeit aller Konfigurationen darstellt.

Die komponentenbasierte Gefahrenanalyse bietet daher spezielle Unterstützung für die Rekonfiguration des Systems. Basis dafür ist die oben dargestellte

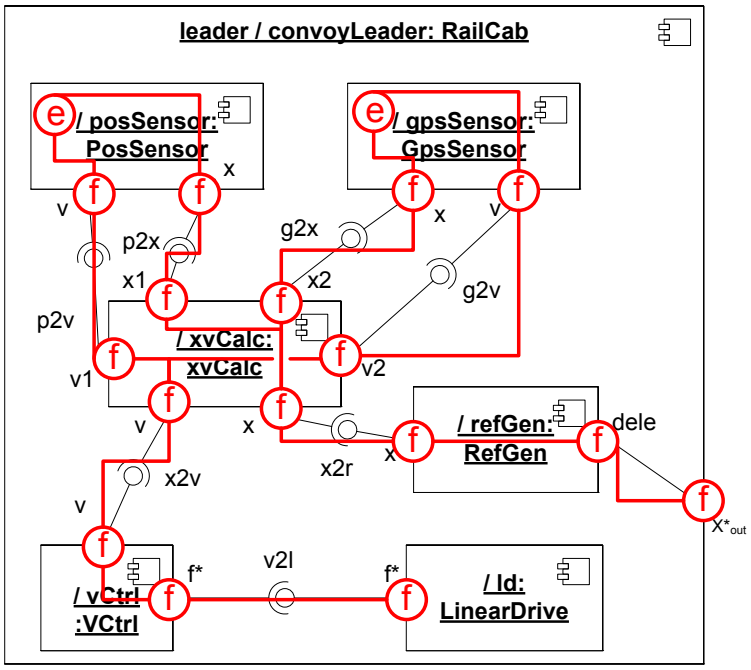


Abbildung 1.4: Fehlerpropagierung für eine Komponentenstruktur des Anwendungsbeispiels

Sprache zur Spezifikation der Rekonfiguration.

1.3.3 Einbettung in den Entwicklungsprozess

Die in dieser Arbeit vorgestellten Modellierungs- und Analysetechniken müssen in Prozesse für die Entwicklung sicherheitskritischer und mechatronischer Systeme, wie die VDI 2206 [VDI04] und die IEC 1508 [IEC], eingebettet werden.

Abbildung 1.5 zeigt den Entwicklungsprozess der VDI-Richtlinie 2206 für mechatronische Systeme. Die Richtlinie unterscheidet im Entwicklungsprozess, der durchaus iterativ durchlaufen werden kann, grundlegend drei Phasen. Die Prinziplösung wird von den Experten der verschiedenen Domänen gemeinsam in der ersten Phase entworfen. Die Prinziplösung besteht aus verschiedenen Partialmodellen z.B. der Wirkstruktur und verschiedenen Verhaltensmodellen, die u.a. eine Rekonfiguration der Wirkstruktur beschreiben. In der nächsten Phase wird die Prinziplösung durch die Experten domänenspezifisch verfeinert. Abstrakt dargestellt sind dies getrennte Aktivitäten; im Detail können die Aktivitäten durchaus von Experten verschiedener Domänen gemeinsam durchgeführt werden, wenn

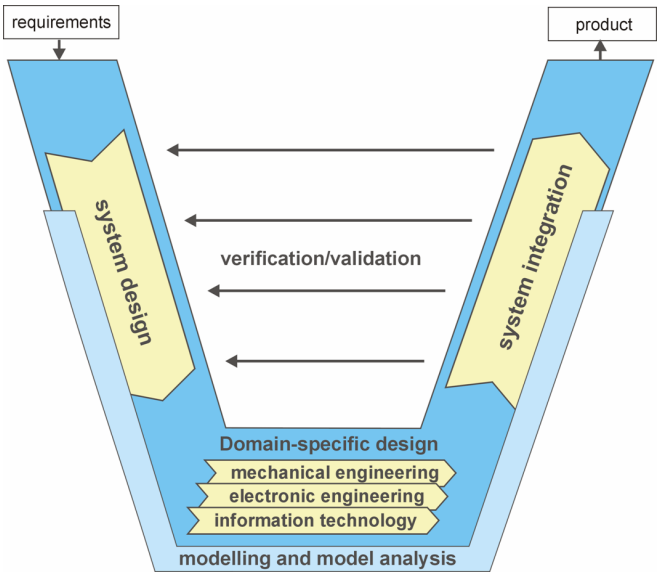


Abbildung 1.5: V-Entwicklungsmodell der VDI-Richtlinie 2206

zum Beispiel Regler in einen Softwareentwurf eingebettet werden. Als letztes werden die Systeme wieder integriert und bzgl. der Anforderungen und der Prinziplösung verifiziert und validiert.

Die in dieser Arbeit vorgestellten Modellierungs- und Analysetechniken können in verschiedenen Phasen dieses Entwicklungsprozesses eingesetzt werden. Die Sprache zur Modellierung von Komponentenstrukturen und deren Rekonfiguration kann sowohl in der Prinziplösung genutzt werden, um die Struktur des Gesamtsystems und deren Rekonfiguration (auch in den frühen Phasen) zu beschreiben, als auch in der domänenspezifischen Phase für die Modellierung der Software.

Die Sicherheit eines Systems muss während der gesamten Entwicklung und auch während des Betriebs betrachtet werden [IEC]. Dementsprechend kann die vorgestellte Gefahrenanalyse auch in allen Phasen durchgeführt werden. Die Gefahrenanalyse basiert auf der Systemstruktur. Sie kann daher auch in der Prinziplösung auf der Wirkstruktur inkl. Rekonfigurationen durchgeführt werden. Wenn die Modelle dann domänenspezifisch verfeinert und nachfolgend mit den übrigen Modellen integriert werden, wird die Gefahrenanalyse auch auf diesen Modellen ausgeführt.

1.4 Übersicht über die Arbeit

Im nächsten Kapitel werden die für die in der Arbeit präsentierten Inhalte benötigten Grundlagen vorgestellt. Dies sind vor allem die im Bereich der Entwicklung sicherheitskritischer Systeme relevanten Begriffe sowie Techniken zur modellbasierten Entwicklung selbstoptimierender Systeme.

Kapitel 3 beschreibt eine Erweiterung der Mechatronic UML für die Strukturmodellierung um Komponententypen mit einer variablen Anzahl an unterlagerten Komponenteninstanzen sowie eine zugehörige Transformationssprache, die basierend auf Graphtransformationen die Modellierung komplexer hierarchischer Transformationen auf der unterlagerten Komponentenstruktur ermöglicht.

Die Gefahrenanalyse auf Basis der Komponentenstrukturen aus Kapitel 3 ist Inhalt von Kapitel 4. Im ersten Teil des Kapitels werden die Modelle für das abstrakte Fehlerverhalten der einzelnen Bestandteile der Systemarchitektur vorgestellt. Darauf folgend werden verschiedene Techniken der Gefahrenanalyse auf Basis der Struktur- und Fehlerverhaltensmodelle beschrieben.

Kapitel 5 erweitert die Gefahrenanalyse des vorhergehenden Kapitels um die spezielle Behandlung von selbstoptimierenden Systemen. Die vorgestellte Gefahrenanalyse basiert auf der Komponentenstruktur. Wenn sich diese Komponentenstruktur bei selbstoptimierenden Systemen zur Laufzeit ändert, wird dies bei der Gefahrenanalyse geeignet mitbetrachtet.

Eine Beschreibung des Werkzeugs, welches die in den Kapiteln 3 bis 5 vorgestellten Techniken umsetzt, ist Inhalt von Kapitel 6. Auf dessen Grundlage wird des Weiteren die Skalierung der Gefahrenanalyse evaluiert.

Die relevanten verwandten Arbeiten für die Modellierung von hierarchischen Komponentenstrukturen und deren Transformation sowie die Gefahrenanalyse werden in Kapitel 7 vorgestellt.

Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick auf weitere und zukünftige Arbeiten in Kapitel 8.

2 Grundlagen

2.1 Selbstoptimierende Systeme

Intelligente, mechatronische Systeme beruhen auf dem Zusammenwirken der Domänen Maschinenbau, Elektrotechnik und Informatik. Vor allem der steigende Anteil eingebetteter Software ermöglicht es, Systeme mit einer Teilintelligenz zu entwickeln, die auf Änderungen der Umgebung reagieren:

> „Unter Selbstoptimierung eines technischen Systems wird die endogene Änderung der Ziele des Systems auf veränderte Umfeldbedingungen und die daraus resultierende zielkonforme autonome Anpassung der Parameter und ggf. der Struktur und somit des Verhaltens dieses Systems verstanden.“ [FGK+04]

In einem System findet nach [FGK+04] genau dann Selbstoptimierung statt, wenn durch das Zusammenwirken der enthaltenen Elemente die folgenden drei Aktionen wiederkehrend ausgeführt werden:

1. **Analyse der Ist-Situation** Die in der ersten Aktion betrachtete Ist-Situation umfasst den Zustand des Systems selbst sowie alle möglichen Beobachtungen über seine Umgebung. Dabei können Beobachtungen auch indirekt durch Kommunikation mit anderen Systemen gewonnen werden. Der Zustand eines Systems beinhaltet weiterhin eventuell gespeicherte zurückliegende Beobachtungen. Ein wesentlicher Aspekt der Analyse ist die Prüfung des Erfüllungsgrades der verfolgten Ziele.

2. **Bestimmung der Systemziele** Bei der Zielbestimmung in der zweiten Aktion können die neuen Ziele des Systems durch Auswahl, Anpassung oder Generierung dieser gewonnen werden. Hierbei wird unter Auswahl die Selektion einer Alternative aus einer fest vorgegebenen, diskreten, endlichen Menge von möglichen Zielen verstanden. Eine Anpassung von Zielen dagegen beschreibt die graduelle Veränderung bestehender Ziele. Von Generierung der Ziele wird dagegen dann gesprochen, wenn diese unabhängig von den bisher bekannten neu erzeugt werden.

3. **Anpassung des Systemverhaltens** Die Anpassung des Systemverhaltens ist durch die drei Aspekte Parameter, Struktur und Verhalten bestimmt. In der dritten Aktion wird die abschließende Rückwirkung des Selbstoptimierungskreislaufes durch Anpassung des Systemverhaltens beschrieben. Die einzelnen Fälle der Anpassung können je nach der Ebene eines mechatronischen Systems sehr unterschiedlich ausfallen. Auch die Domäne, in der die Anpassung umgesetzt wird, spielt hierbei eine wesentliche Rolle.

Im Rahmen dieser Arbeit wird vor allem der Aspekt der Verhaltensanpassung betrachtet. Verhalten kann durch Änderung von Parametern oder durch Änderung der Systemstruktur, was im Fokus dieser Arbeit steht, angepasst werden. Die Änderung der Struktur wird, in Bezug auf komponentenbasierte Systeme, als Änderung der Komponentenstruktur durch Anpassung der Verbindungen zwischen den Ports der Komponenten sowie dem Hinzufügen bzw. Entfernen der Komponenten und Ports verstanden.

2.2 Verlässlichkeit

Mechatronische Systeme und damit auch selbstoptimierende Systeme werden oft für sicherheitskritische Aufgaben eingesetzt, wie in Autos, Flugzeugen und Industrieanlagen. Diese Systeme müssen daher verlässlich funktionieren. Der Begriff der *Verlässlichkeit* (dependability) [Lap92, ALRL04] ist ein Grundkonzept, um solche Systeme hinsichtlich verschiedener Aspekte zu charakterisieren.

Im Folgenden wird Verlässlichkeit in den unterschiedlichen für diese Arbeit relevanten Aspekten und Begriffen erläutert. Bezüglich der deutschen Begriffe gibt es verschiedene Übersetzungen; für diese Arbeit werden die Übersetzungen aus [DHK+08] genutzt.

Laprie klassifiziert das Konzept der Verlässlichkeit, wie in Abbildung 2.1 dargestellt, in drei Aspekte: Kenngrößen, Beeinträchtigungen und Mittel, um Verlässlichkeit zu erreichen.

Fehlerursachen, Fehlerzustände und Ausfälle beeinträchtigen die Verlässlichkeit des Systems. Sie sind oftmals unerwünscht, aber nicht unerwartet. Mittel, wie Fehlertoleranz, werden genutzt, um die Auswirkungen der Beeinträchtigungen auf die Verlässlichkeit zu vermeiden. Schlussendlich werden die Kenngrößen genutzt, um erwartete Eigenschaften eines verlässlichen Systems zu formulieren. Die Verlässlichkeit hat die folgenden Kenngrößen:

- Verfügbarkeit (availability): Verlässlichkeit in Bezug auf das Bereit sein zur Benutzung [Lap92].

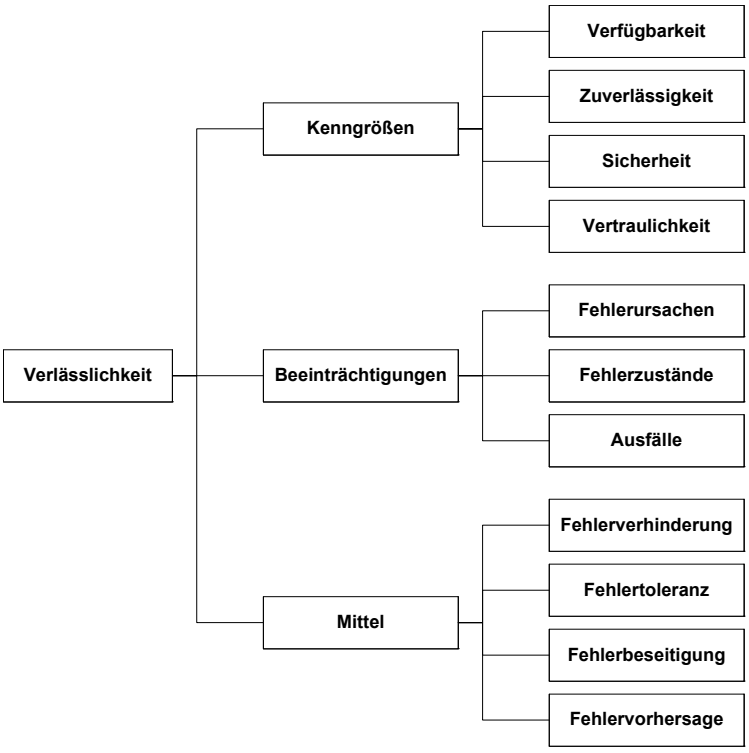


Abbildung 2.1: Verlässlichkeitsbaum nach Laprie

- Zuverlässigkeit (reliability): Verlässlichkeit mit Bezug auf die Kontinuität der Leistung – Überlebenswahrscheinlichkeit [Lap92].

- Sicherheit (safety): Verlässlichkeit mit Bezug auf das Nichtauftreten von kritischen Ausfällen [Lap92]; Sicherheit ist die Eigenschaft eines Systems, menschliches Leben oder die Umwelt nicht zu gefährden [Sto96]; Sicherheit ist die Freiheit von Unfällen oder Verlusten [Lev95].

- Vertraulichkeit (security): Verlässlichkeit mit Bezug auf die Verhinderung von nicht autorisiertem Zugriff und/oder Behandlung von Informationen [Lap92].

2.2.1 Sicherheit

Diese Arbeit beschäftigt sich vor allem mit sicheren Systemen. Bei der Entwicklung sicherer Systeme wird versucht, weitestgehend *Unfälle* (accidents) und damit deren Folgen auszuschließen.

Ein Unfall ist ein unerwünschtes und ungeplantes (aber nicht notwendigerweise unerwartetes) Ereignis, welches in einem definierten Verlust resultiert [Lev95].

Um Unfälle zu verstehen und nachvollziehen zu können, müssen der Zustand oder die Eigenschaften des Systems in Zusammenhang mit Eigenschaften der Umgebung betrachtet werden, die zwangsläufig zu diesem Unfall führen. Dieser Zustand und die Eigenschaften des Systems und dessen Umgebung wird *Gefahr* (hazard) genannt [Lev95]. Eine *Gefahrenanalyse* bestimmt also die Zustände und Eigenschaften, die zum Unfall führen. Dies beinhaltet auch die *Wahrscheinlichkeit* für das Eintreten dieser Gefahr.

Neben der Gefahrenwahrscheinlichkeit ist auch der zugehörige Schaden wichtig für die korrekte Einschätzung der Gefahrensituation. Diese Kombination von Gefahrenwahrscheinlichkeit und Schaden durch den Unfall wird mit *Risiko* bezeichnet.

2.2.2 Beeinträchtigungen

Beeinträchtigungen gefährden die Verlässlichkeit der Systeme. Laprie [Lap92] unterscheidet Fehlerursachen (faults), Fehlerzustände (errors) und Ausfälle eines Systems (failures):

- Fehlerursache (fault): Anerkannte oder hypothetische Ursache für einen Fehler.
- Fehlerzustand (error): Teil des Systemzustandes, der dafür verantwortlich ist, dass ein Ausfall auftritt. Offenbarung einer Fehlerursache im System.
- Ausfall (failure): Abweichung der erbrachten Leistung von der in der Systemspezifikation geforderten Leistung.

Die Fehlerpathologie nach Laprie ordnet die verschiedenen Fehlerarten in eine kausale Reihenfolge (s. Abbildung 2.2) ein. Eine Fehlerursache ist die grundlegende Ursache für die folgenden Ereignisse. Sie kann z.B. ein Programmfehler oder ein Materialfehler sein. Eine Aktivierung der Fehlerursache resultiert in einem Fehlerzustand, einer Abweichung des Systemzustands von dem korrekten Zustand. Führt ein Fehlerzustand zu einer Abweichung des Verhaltens an der Schnittstelle des Systems zur Umgebung, wird von einem Ausfall des Systems gesprochen. Dieser Ausfall kann dann wieder zu einer Fehlerursache in verbundenen Systemen führen.

Man unterscheidet systematische und zufällige Fehlerursachen [Lap92]. Systematische Fehlerursachen führen immer zu einem Fehlerzustand, wenn eine bestimmte Situation eintritt. Bei einem Programmierfehler, tritt der Fehlerzustand

Abbildung 2.2: Fehlerpathologie

immer auf, wenn die betreffende Zeile ausgeführt wird. Im Unterschied dazu können für physikalische Fehler, zum Beispiel Fehler durch Abnutzung oder durch physikalische Effekte, oftmals Wahrscheinlichkeiten und zugehörige Verteilungen für das Auftreten des Fehlerzustands angegeben werden.

In [FMNP94, MP94] werden die verschiedenen Fehlerarten noch zusätzlich über ihre Eigenschaft typisiert. Diese Typisierung ermöglicht, die unterschiedlichen Auswirkungen der Fehler auf das System und weitere Systeme zu betrachten. In der Literatur werden die folgenden Typen unterschieden, die aber je nach Anwendung verfeinert werden können:

- Diensterbringung (service provision): Auslassung (omission), Unerwartet (commission)
- Zeit (timing): zu früh (early), zu spät (late)
- Wert (value): geringfügige Abweichung (subtle incorrect), große Abweichung (coars incorrect)

Die oben dargestellten Charakteristika von Fehlern sind für diese Arbeit relevant. Eine weitergehende Taxonomie von Fehlern aufbauend auf [Lap92] wird in [ALRL04] vorgestellt, deren zusätzliche Details für diese Arbeit allerdings nicht relevant sind.

2.2.3 Fehlerbaumanalyse

Die Fehlerbaumanalyse (Fault-Tree-Analysis FTA) [VGRH91] ist eine klassische, deduktive Technik zur Gefahrenanalyse sicherheitskritischer Systeme. Startend mit der Gefahr als Wurzel des Baums, dem so genannten Topereignis, werden die Ereignisse oder Zustände, die zur Gefahr führen rekursiv verfeinert und mit booleschen Operatoren wie Und und Oder verknüpft.

Der Baum wird verfeinert, bis an allen Blättern des Baumes Basisereignisse angegeben sind, die nicht mehr weiter verfeinert werden. Diese sind die Ursachen für die Gefahr. Die Blätter des Baumes ergeben zusammen mit den booleschen

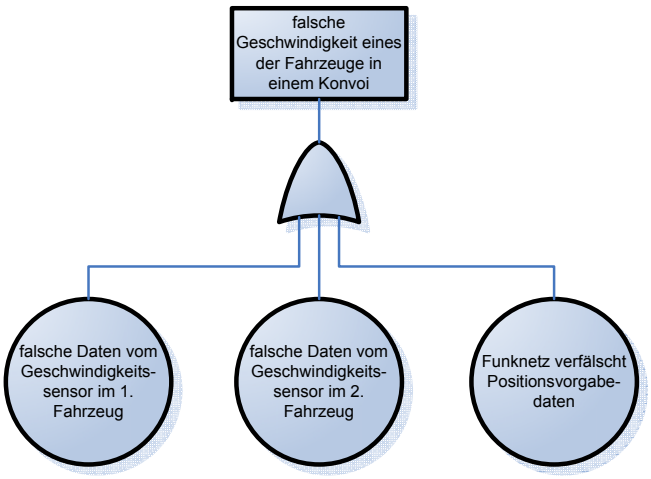


Abbildung 2.3: Einfacher Fehlerbaum für die Gefahrensituation, dass ein Fahrzeug in einem Konvoi eine falsche Geschwindigkeit hat, die zu einem Auffahrunfall führt.

Operatoren eine boolesche Formel. Wichtig ist, dass die Basisereignisse unabhängig voneinander sind.

Abbildung 2.3 zeigt einen stark vereinfachten Fehlerbaum für die Gefahr, dass ein Fahrzeug in einem Konvoi eine falsche Geschwindigkeit hat. Dies führt zu einem Auffahrunfall. Die Gefahr tritt auf, wenn der Geschwindigkeitssensor eines der Fahrzeuge falsche Daten liefert; das hintere Fahrzeug z.B. in Wahrheit schneller fährt, als durch eine Abstandsregelung vorgegeben. Wenn zum anderen durch die Funkverbindung zwischen den Fahrzeugen die Positionsvorgaben verfälscht werden, nimmt das hintere Fahrzeug durch die Positionsregelung falsche Positionen ein, was auch zu der Gefahr führt.

Bei der qualitativen Analyse eines Fehlerbaums ist der Begriff des *minimal cut set* ein wichtiges Hilfsmittel. Ein *cut set* ist eine Menge der Basisereignisse, deren gemeinsames Auftreten, dazu führt, dass das Topereignis, also die Gefahr, auftritt. Ein cut set wird als *minimal* bezeichnet, wenn keine Basisereignisse aus dem cut set entfernt werden können, ohne dass die Menge der Basisereignisse nicht mehr ein cut set ist. Für den oben dargestellten Fehlerbaum gibt es drei minimale cut sets, die jeweils aus einem der Basisereignisse bestehen.

Ein Fehlerbaum kann auch quantitativ analysiert werden. Hier wird die Wahrscheinlichkeit des Auftretens des Topereignisses auf Basis der Wahrscheinlichkeiten der Basisereignisse berechnet. Gegeben seien die Wahrscheinlichkeiten $p(q_i)$ für die Basisereignisse q_i.

Sind die Basisereignisse $q_i \in \{q_1, \ldots, q_n\}$ mit einem Und verbunden, ergibt

sich die Wahrscheinlichkeit des Topereignisses $p(Q)$ als:

$$p(Q) = \prod_{i=1}^{n} p(q_i) \tag{2.1}$$

Für mit Oder verbundene Basisereignisse ergibt sich die Wahrscheinlichkeit des Topereignisses als:

$$p(Q) = 1 - \prod_{i=1}^{n} (1 - p(q_i)) \tag{2.2}$$

Für ein einzelnes minimal cut set ist ebenfalls Formel 2.1 zu benutzen, da alle Basisereignisse eines minimal cut sets gemeinsam eintreten müssen, also durch Und verbunden sind.

Für alle minimal cut sets $\breve{Q}_i \in \{\breve{Q}_1, \ldots, \breve{Q}_m\}$ eines Fehlerbaums kann eine obere Abschätzung der Wahrscheinlichkeit des Topereignisses auf Basis der einzelnen Wahrscheinlichkeiten der minimal cut sets $p(\breve{Q})$ angegeben werden:

$$p(Q) \leq 1 - \prod_{i=1}^{m} (1 - p(\breve{Q}_i)) \tag{2.3}$$

2.3 Mechatronic UML

Beim Entwurf selbstoptimierender Systeme stellt die eingebettete Software einen großen Teil der Wertschöpfung dar. Sie ist daher zu einem großen Teil für die Verlässlichkeit des Gesamtsystems verantwortlich. Typischerweise werden Regelungen oder Steuerungen in Software umgesetzt. Durch die starke Vernetzung selbstoptimierender Systeme [SW07], wie auch in [BSDB00] für mechatronische Systeme bereits beschrieben, 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.

Um die Komplexität zu beherrschen, werden oftmals Techniken zur modellbasierten Entwicklung der Software eingesetzt [Fav05]. In Modellen werden Struktur und Verhalten der Software spezifiziert. Hierbei wird von Implementierungsdetails wie Betriebssystem, Hardwarearchitektur etc. abstrahiert, um das Modell

möglichst klein und übersichtlich zu halten. Beispiele für Strukturmodelle sind UML Klassendiagramme [Obj07], Bäume und Graphen [CSRL01]. Das Verhalten wird beispielsweise mit Automaten, Statecharts [Har87] und Petri-Netzen [Rei85, MBC+95] modelliert. Diese Modelle werden dann in Bezug auf funktionale und nicht-funktionale Anforderungen überprüft. Wenn die Anforderungen erfüllt sind, können wie im MDA-Ansatz [Obj01] die Modelle um Implementierungsdetails erweitert oder direkt Quelltext generiert werden. Der Quelltext muss dabei das Modell semantisch korrekt umsetzen.

Die Mechatronic UML ist eine Modellierungssprache mit zugehöriger Entwicklungsmethodik für die modellbasierte Entwicklung der Software selbstoptimierender Systeme. Die Struktur dieser Systeme wird mit domänenspezifisch verfeinerten UML Komponentendiagrammen auf Basis der OCM Architektur spezifiziert. Verhalten wird mit Real-Time Statecharts, einer domänenspezifischen Verfeinerung der UML state machines um Syntaxkonstrukte der Timed Automata zur Spezifikation von harten Echtzeiteigenschaften, modelliert.

Für den Architekturentwurf selbstoptimierender Systeme wurde das *OCM* entwickelt [OHG04, HOG04] (s. Abbildung 2.4). Den verschiedenen Aspekten der Software in selbstoptimierenden Systemen wurde durch die Aufteilung des OCM in den *Controller* für die kontinuierliche Regelung unter harten Echtzeitbedingungen, den *reflektorischen Operator* für die Konfigurationssteuerung und die Kommunikation mit anderen OCMs ebenfalls unter harten Echtzeitbedingungen und den *kognitiven Operator* für die Optimierung mit weichen Echtzeitanforderungen Rechnung getragen. Für das OCM wurde dabei ein Komponenten- und Schnittstellenbegriff als Verfeinerung der UML entwickelt, der den Eigenschaften selbstoptimierender Systeme sowie der Domäne entspricht.

Der Mechatronic UML liegen *Dekomposition* und *Abstraktion* als Konzepte zur Beherrschung der Komplexität zu Grunde. Beide Aspekte werden im musterbasierten Entwurfsansatz eingesetzt. *Echtzeitkoordinationsmuster* beschreiben die Koordination verschiedener Teilsysteme, *Rollen* genannt, auf einer abstrakten Ebene und werden bzgl. Sicherheitseigenschaften formal verifiziert. Der reflektorische Operator eines OCMs verfeinert die Rollen verschiedener Echtzeitkoordinationsmuster und fügt anwendungsspezifisches Verhalten hinzu. Wenn nun auch der reflektorische Operator die spezifizierten Sicherheitseigenschaften erfüllt, was ebenfalls durch eine formale Verifikation nachgewiesen wird, garantiert der entwickelte kompositionale Ansatz die Einhaltung der Sicherheitseigenschaften für das Gesamtsystem, ohne dass das Gesamtsystem formal verifiziert werden muss [GTB+03]. Die Echtzeitkoordinationsmuster ermöglichen daher die Dekomposition des Gesamtsystems in einzeln zu modellierende und zu verifizierende Teile.

Das Echtzeitverhalten der Rollen und des reflektorischen Operators eines OCMs wird mit Real-Time Statecharts spezifiziert. Die formale Semantik durch

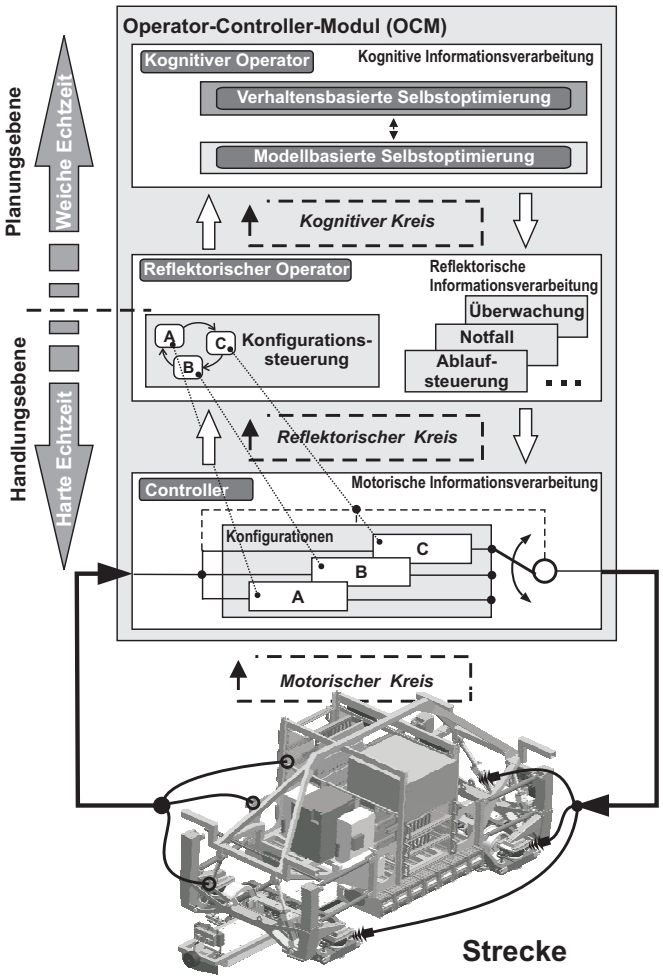


Abbildung 2.4: Das Operator-Controller-Modul (OCM) als Architektur für selbstoptimierende Systeme

eine Übersetzung auf Extended Hierarchical Timed Automata und nachfolgend auf Timed Automata [GB03] ermöglicht die formale Verifikation durch Modelchecking.

Des Weiteren wird die Integration der (kontinuierlichen) Modelle zur Regelung und Steuerung mit den diskreten Modellen unterstützt. Die Mechatronic UML ermöglicht die Modellierung von hierarchischen Rekonfigurationen zwischen verschiedenen Konfigurationen (z.B. Reglerstrukturen) der Software zur Laufzeit mit *Hybriden Rekonfigurationscharts* [GBSO04]. Auch die Einbettung und Rekonfiguration zwischen verschiedenen Konfigurationen wird formal verifiziert. Es existiert eine Quelltextsynthese für die automatische Generierung von Quelltext

aus den Modellen. Der erzeugte Quelltext setzt die spezifizierten Echtzeiteigenschaften korrekt um.

Im Folgenden werden die einzelnen Schritte und Modellierungssprachen anhand der Beispiele aus [GHH+08] erläutert.

2.3.1 Real-Time Statecharts

Das Verhalten mechatronischer Systeme besteht im reflektorischen Operator vor allem aus zustandsbasiertem harten Echtzeitverhalten. Zur Modellierung dieses Verhaltens sind die UML state machines nicht geeignet, da sie nur unzureichende Syntaxelemente für die Modellierung von Echtzeiteigenschaften besitzen [Bur02, GB03]. Real-Time Statecharts erweitern die UML state machines um Syntaxelemente der Timed Automata [AD94] wie Synchronisationskanäle, Clocks, Time Guards und Invarianten. Des Weiteren werden, im Unterschied zu anderen Ansätzen [Obj07, LQV01, BLM02, DMY02], mit Hilfe von Deadlines Zeitdauern für das Schalten einer Transition definiert. Somit wird nicht die Null-Zeit-Annahme getroffen, die in harten Echtzeitsystemen nicht umgesetzt werden kann.

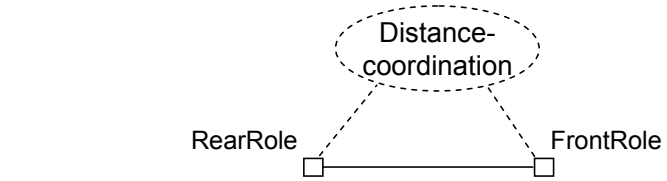
Die Semantik von Real-Time Statecharts wird durch eine Übersetzung auf Extended Hierarchical Timed Automata formal definiert, die wiederum in Timed Automata übersetzt werden [GB03]. Aus Real-Time Statecharts wird, wie in [BGS05b] beschrieben, automatisch Java-RT [Dib02] bzw. C++ Quelltext (auf Basis des IPANEMA-Frameworks [Hon98]) generiert, der die spezifizierten Echtzeiteigenschaften einhält.

2.3.2 Echtzeitkoordinationsmuster

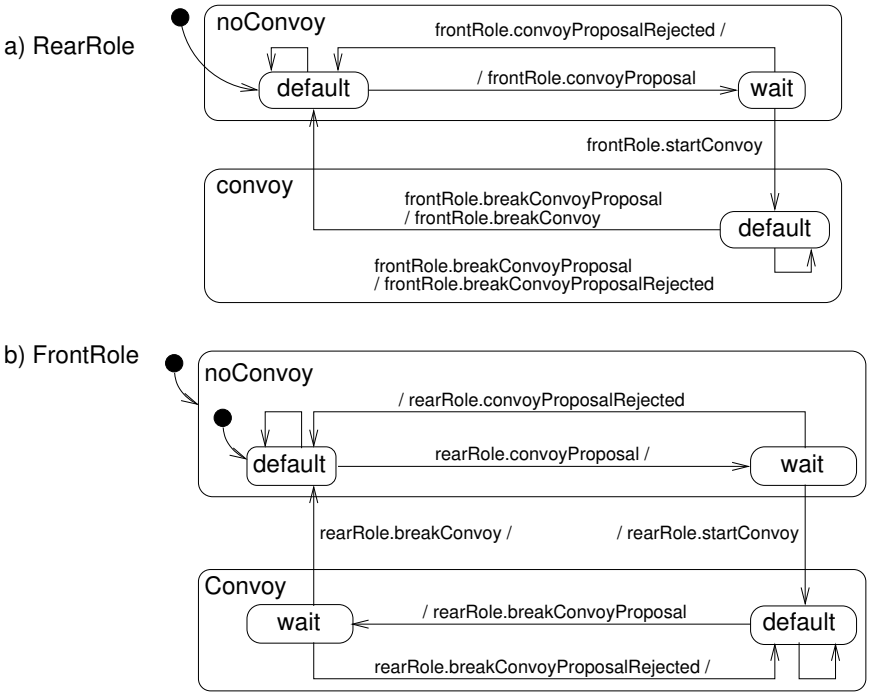
Echtzeitkoordinationsmuster bestehen aus Rollen und Konnektoren zwischen diesen. Das Verhalten der einzelnen Rollen wird mit Real-Time Statecharts beschrieben. Das Verhalten des Konnektors modelliert QoS-Eigenschaften wie z.B. Verzögerung, verlorene und veränderte Nachrichten; dies wird ebenfalls mit Real-Time Statecharts spezifiziert.

Das Verhalten der Rollen enthält typischerweise Nichtdeterminismus, damit das Muster in verschiedenen Situationen eingesetzt werden kann. Nichtdeterminismus wird vor allem hinsichtlich der Zeit sowie bzgl. des Schaltens mehrerer aktivierter Transitionen eingesetzt.

Abbildung 2.5 zeigt das Echtzeitkoordinationsmuster für die Konvoikommunikation zweier Fahrzeuge. Das Muster besteht strukturell aus zwei Rollen RearRole und FrontRole. Erstere steht für das hinten in einem Konvoi fahrende Fahrzeug, letztere für das vorne fahrende Fahrzeug.



(a) Struktur



(b) Verhalten

Abbildung 2.5: Echtzeitkoordinationsmuster für die Konvoikommunikation zweier Fahrzeuge

Die Verhaltensmodelle beschreiben die Kommunikation dieser beiden Rollen bzgl. der Bildung und der Auflösung eines Konvois. Wichtig ist, dass nur das Kommunikationsprotokoll beschrieben wird und nicht die Entscheidung, ob ein Konvoi gebildet wird. Grund dafür ist, dass diese Entscheidung applikationsspezifisch ist, so dass erst bei der Anwendung des Echtzeitkoordinationsmusters das Verhalten für diese Entscheidung durch eine entsprechende Verfeinerung des Echtzeitkoordinationsmusters modelliert wird.

Neben den Verhaltensmodellen werden für Echtzeitkoordinationsmuster sicherheitskritische Eigenschaften zum Beispiel in RT-OCL [GTB+03] oder ATCTL

[Hir04] spezifiziert. Es wird dann mittels Modelchecking (z.B. mit den Modelcheckern Raven [Ruf01] oder UPPAAL [BBD+02]) formal verifiziert, ob das spezifizierte Echtzeitverhalten der Rollen und Konnektoren die Eigenschaften erfüllt.

Für das dargestellte Muster muss gelten, dass niemals die Situation vorkommen darf, dass das vordere Fahrzeug nicht im Konvoimodus ist (Zustand noConvoy) während das hintere Fahrzeug im Konvoimodus (Zustand convoy) ist, um einen Auffahrunfall bei einer Konvoifahrt zu verhindern. In RT-OCL ist das folgendermaßen formuliert:

```
context DistanceCoordination inv:
    not (self.oclInState(RearRole::convoy) and
         self.oclInState(FrontRole::noConvoy))
```

Es wurde in [GTB+03, GST+03] gezeigt, dass für das oben dargestellte Muster diese Eigenschaft erfüllt ist.

Alle entworfenen Echtzeitkoordinationsmuster werden in einer Musterbibliothek zur weiteren Verwendung abgelegt.

2.3.3 Komponenten

Auf Basis der Echtzeitkoordinationsmuster werden die einzelnen Komponententypen[1] des Systems entwickelt. Dies geschieht, indem einzelne Rollen aus den (bereits verifizierten) Mustern jeweils als Ports für die Komponente verfeinert werden. Dies beinhaltet neben dem strukturellen Aspekt, dass auch das Echtzeitverhalten der Rollen übernommen wird.

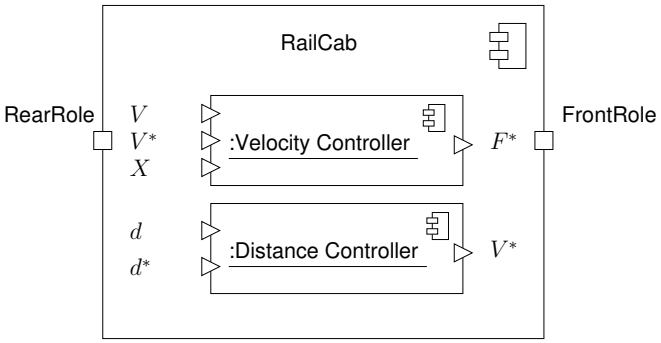


Abbildung 2.6: Komponententyp RailCab

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

Abbildung 2.6 zeigt den Komponententyp RailCab, der die Rollen RearRole und FrontRole des oben vorgestellten Echtzeitkoordinationsmusters als Ports übernimmt.

Kapselung durch die Einbettung von Komponenteninstanzen in andere Komponenten ist als Mittel zur Komplexitätsbeherrschung großer komponentenbasierter Systeme wichtig. Die eingebetteten Komponenten sind hierbei nur innerhalb der umgebenden Komponente bekannt; sie sind nicht von außerhalb erreichbar. Diese Einbettung kann mehrfach durchgeführt werden, so dass sich insgesamt eine hierarchische Komponentenstruktur ergibt.

Die Mechatronic UML erlaubt eine Modellierung dieser hierarchischen Komponenten, indem in einen Komponententyp eine Menge von Instanzen andere Komponententypen eingebettet werden können. Abbildung 2.6 zeigt, dass der Komponententyp RailCab zwei eingebettete Komponenteninstanzen zur Geschwindigkeitsregelung (:Velocity Controller) und Abstandskontrolle (:Distance Controller) enthält, die genutzt werden können, je nachdem ob ein RailCab im Konvoi fährt oder nicht.

Das Echtzeitverhalten der Ports kann verfeinert werden. Dies bedeutet im Wesentlichen, dass der oben angesprochene Nichtdeterminismus aufgelöst wird. Dies wird üblicherweise in Verbindung mit einem, so genannten, Synchronisationsstatechart durchgeführt. Dieses Statechart kommuniziert über Synchronisationskanäle mit den Statecharts der Ports. Dies ermöglicht, das aus den Rollen übernommene Echtzeitverhalten in Grenzen anzupassen. Wichtig ist hierbei, dass zwischen den Real-Time Statecharts der Ports und der Rollen die syntaktische und semantische Verfeinerungsbeziehung aus [GTB+03] erfüllt ist.

Abbildung 2.7 zeigt das Real-Time Statechart für den Komponententyp RailCab. Dieses Statechart besteht aus den zwei verfeinerten Rollen FrontRole und RearRole des oben dargestellten Echtzeitkoordinationsmusters sowie dem Synchronisationsstatechart. Die Verfeinerung der Rollen besteht darin, dass die Entscheidung, ob ein Konvoi gebildet bzw. aufgelöst wird, was im Muster noch offen gehalten wurde, nun durch interne Kommunikation mit dem Synchronisationsstatechart getroffen wird. Diese Entscheidung wird im Beispiel dahingehend getroffen, dass zum einen nur Konvois von zwei Fahrzeugen gebildet werden, also ein RailCab entweder vorne oder hinten fährt, und zum anderen eine boolesche Variable convoyUseful überprüft wird. Diese wird vom kognitiven Operator gesetzt.

Das Verhalten des Komponententyps, bestehend aus den Statecharts der Ports und dem Synchronisationsstatechart, wird dann, analog zur Verifikation der Echtzeitkoordinationsmuster, hinsichtlich Sicherheitseigenschaften sowie Freiheit von Deadlocks formal verifiziert.

Wenn ein System, bestehend aus Instanzen der Komponententypen, syntak-

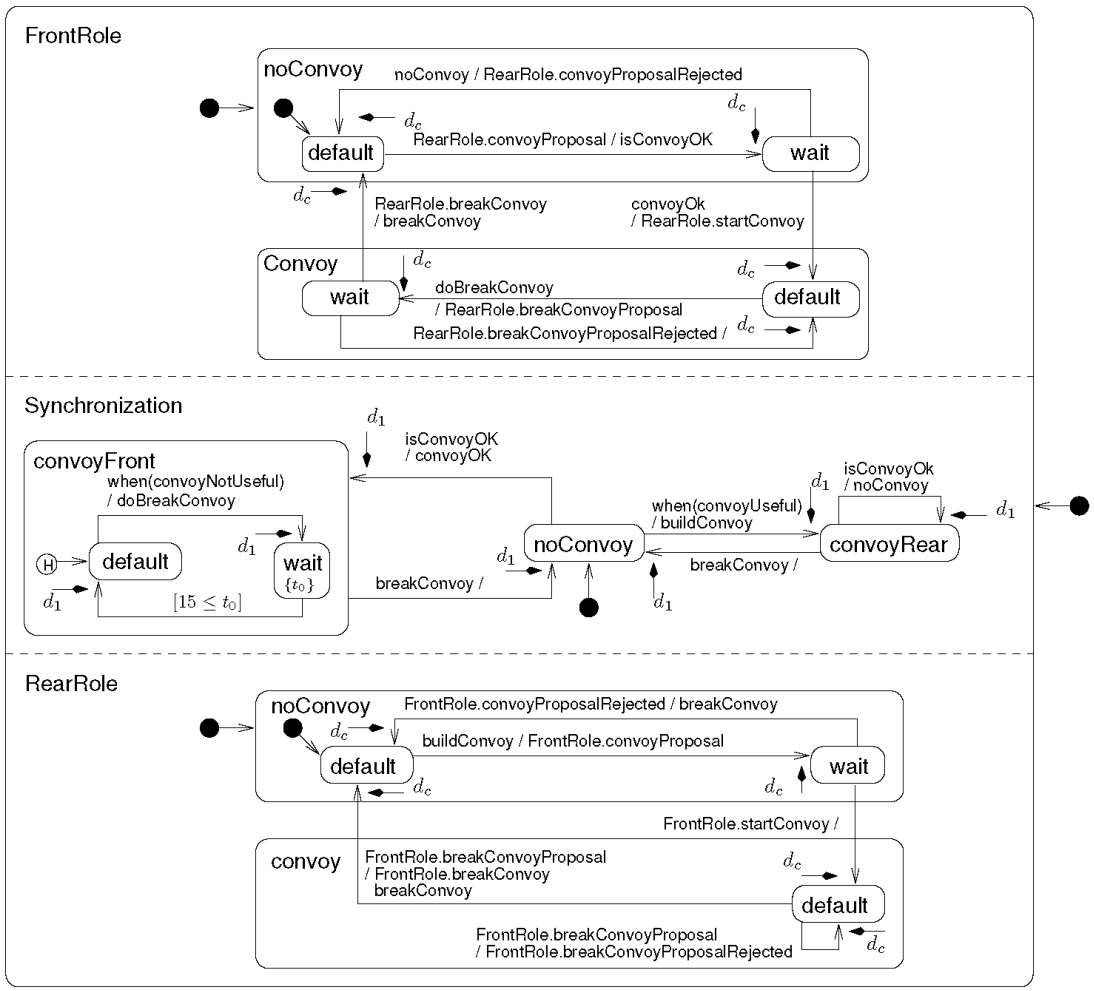


Abbildung 2.7: Real-Time Statechart Beispiel

tisch korrekt hinsichtlich der Ports erstellt wurde, alle Echtzeitkoordinationsmuster und alle Komponententypen die spezifizierten Eigenschaften erfüllen sowie die Verfeinerungsbeziehung zwischen den Ports und und den zugehörigen Rollen erfüllt ist, dann erfüllt das System die Eigenschaften ebenfalls [GTB+03, GST+03]. Dies bedeutet, dass nicht das ganze System auf einmal hinsichtlich der Eigenschaften verifiziert werden muss. Die Eigenschaften werden also kompositional verifiziert. Es ergibt sich somit eine bessere Skalierung bei der Verifikation großer Softwaresysteme [GST+03].

2.3.4 Hybride Rekonfigurationscharts

Selbstoptimierende Systeme passen ihr Verhalten in Reaktion auf Änderungen der Umgebung oder der Ziele an (vgl. Abschnitt 2.1). Eine Möglichkeit zur Verhaltensänderung ist die Strukturanpassung durch Rekonfiguration. In Bezug auf die in der Mechatronic UML genutzte komponentenbasierte Strukturmodellierung bedeutet dies, dass die Komponentenstruktur angepasst wird.

Zur Modellierung dieser Rekonfiguration wird das spezifizierte Echtzeitverhalten um Annotationen bzgl. der in den einzelnen Zuständen aktiven unterlagerten Komponenteninstanzstrukturen ergänzt. Dies betrifft typischerweise das Synchronisationsstatechart.

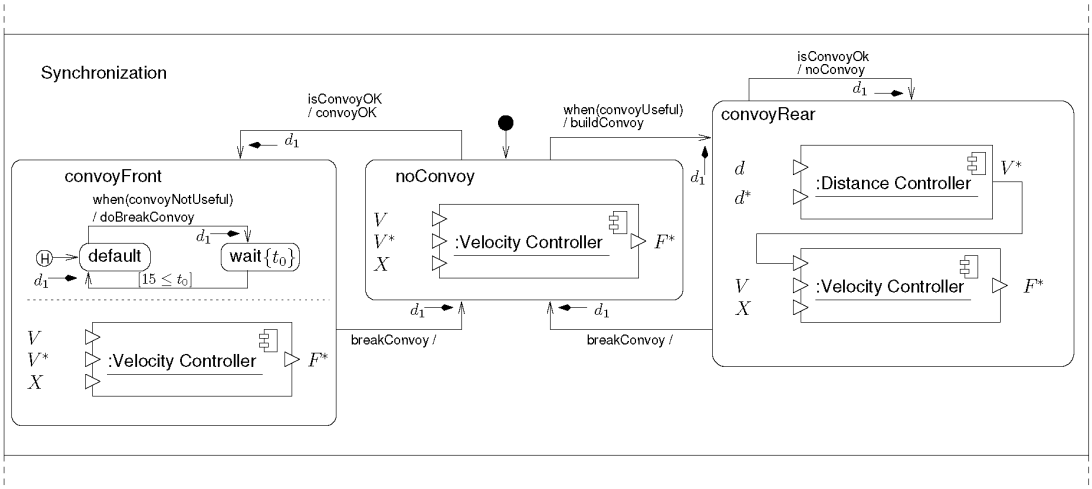


Abbildung 2.8: Ausschnitt der Erweiterung des Real-Time Statecharts um Rekonfiguration; es resultiert ein Hybrides Rekonfigurationschart

Abbildung 2.8 zeigt das Synchronisationsstatechart des bereits in Abbildung 2.7 dargestellten Echtzeitverhaltens des RailCab Komponententyps, welches um Informationen bzgl. der unterlagerten Komponenteninstanzen ergänzt wurde. So nutzt ein RailCab nur den Geschwindigkeitsregler, wenn es alleine (Zustand noConvoy) oder als vorderes Fahrzeug in einem Konvoi fährt (Zustand convoyFront). Wenn es allerdings hinten in einem Konvoi fährt (Zustand convoyRear), wird zusätzlich der Abstandsregler genutzt.

2.4 Story Driven Modeling

Ein großer Anteil des Verhaltens einer Software beschäftigt sich oftmals mit Datenstrukturen und deren Änderung. Bei selbstoptimierenden Systemen betrifft

dies zum Beispiel die Komponentenstrukturen und deren Rekonfiguration zur Verhaltensanpassung. Graphen und Graphtransformationen [Roz97] bieten sich als Formalismen dafür an.

Eine Graphtransformation besteht aus zwei Teilgraphen, der linken Regelseite (LHS) und der rechten Regelseite (RHS). Bei der Anwendung einer Graphtransformation wird die LHS durch einen Morphismus auf einen Wirtsgraph abgebildet. Wenn eine Abbildung möglich ist, wird der Wirtsgraph so geändert, dass danach die RHS durch einen Morphismus abgebildet werden kann (Knoten und Kanten, die sowohl in der LHS als auch der RHS enthalten sind, müssen in beiden Fällen auf die selben Knoten bzw. Kanten abgebildet werden). Die Anpassung des Wirtsgraphs geschieht dabei durch das Erzeugen sowie Löschen von Knoten und Kanten. Elemente, die nur in der LHS enthalten sind, werden bei der Anwendung gelöscht, während Elemente, die nur in der RHS enthalten sind, bei der Anwendung erzeugt werden.

Neben einfachen Graphen wurden Graphtransformationen auch für attributierte und typisierte Graphen definiert (z.B. in PROGRES [SWZ95]). Auf Basis dieser Arbeiten wurde durch eine Verfeinerung der UML *Story Driven Modeling* (SDM) [FNT98, Zün02] entwickelt, welches Graphtransformationen für objektorientierte Programme unterstützt. Story Driven Modeling nutzt UML Klassendiagramme zur Spezifikation von Struktur und Storydiagramme zur Spezifikation von Strukturänderungen.

Neben der Möglichkeit Änderungen der Objektstrukturen formal zu spezifizieren, ist ein weiteres Ziel von SDM aus diesen Spezifikationen automatisch Quelltext zu generieren [KNNZ99]. Dies ist derzeit sowohl für Java als auch für C++ sowie für die Ausführung des Quelltextes in Nichtechtzeitsystemen bzw. in harten Echtzeitsystemen möglich [FNT98, Sei05].

2.4.1 Klassendiagramme

Klassendiagramme werden zur Spezifikation der Klassen inkl. Attribute und Methoden sowie deren Beziehungen wie Generalisierung, Assoziation, Komposition, Referenz genutzt. In Bezug auf Graphtransformationen wird in SDM mit Klassendiagrammen der Typgraph für die Instanzgraphen und Graphtransformationen spezifiziert.

Abbildung 2.9 zeigt einen Ausschnitt des Metamodells der Komponenteninstanzstrukturen, welches im Detail in Kapitel 3 vorgestellt wird. Es zeigt die Metaklassen Instance und ComponentInstance, die durch Anwendung des *Composite* Entwurfsmusters [GHJV94] (Generalisierung und zu-* Assoziation) ermöglichen, hierarchische Komponenteninstanzstrukturen zu modellieren.

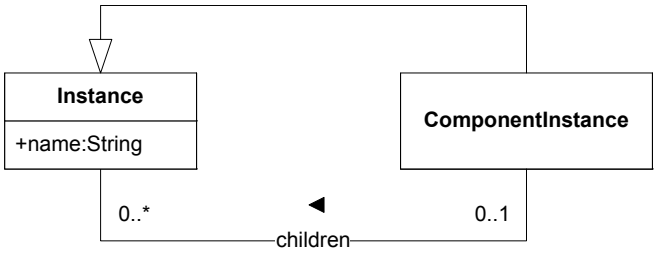


Abbildung 2.9: Ausschnitt aus dem Metamodell der Komponenteninstanzen (s. Abbildung 3.13)

2.4.2 Storypatterns

Mit einem Storypattern wird das Erzeugen und Anpassen von Objektstrukturen, also Instanzen der oben genannten Klassen sowie Links zwischen diesen, modelliert.

Storypatterns basieren auf UML 1.x Kollaborationsdiagrammen. UML Kollaborationsdiagramme werden genutzt, um Objektstrukturen, also Objekte und Links zwischen diesen, zu modellieren. Um Verhalten zu modellieren, wie zum Beispiel das Suchen nach der Objektstruktur und Änderungen wie Erzeugen und Löschen von Kanten, können in einem Kollaborationsdiagramm Methodenaufrufe annotiert werden. Für größere Objektstrukturen und mehrere Änderungen müssen viele Methodenaufrufe spezifiziert werden, so dass die Wartbarkeit der Modelle sinkt [Zün02]. Um diesem Problem entgegen zu wirken, werden Konzepte der Graphtransformationen auf Kollaborationsdiagramme angewendet, so dass sich eine klarere Verhaltensspezifikation ergibt.

Die Elemente des Kollaborationsdiagramms werden für Storypatterns mit Stereotypen versehen, je nachdem, ob sie nur in der LHS (≪destroy≫) oder nur in der RHS enthalten sind (≪create≫). Ist ein Element sowohl in LHS als auch in RHS enthalten, erhält es keinen Stereotyp. Hierdurch kombiniert ein Storypattern die LHS und die RHS einer Graphtransformation in einem einzelnen Graphen.

Abbildung 2.10 zeigt im oberen Teil ein Storypattern, welches eine neue Komponenteninstanz newGen erzeugt und in eine bereits existierende Komponenteninstanz railcab als Kind einbettet. In der Mitte der Abbildung wird die Aufteilung des Storypatterns in LHS und RHS dargestellt. Während das Objekt railcab sowohl Teil der LHS als auch der RHS ist, ist das zu erzeugende Objekt newGen und der Link zum Objekt railcab nur Teil der RHS. Im unteren Teil der Abbildung wird die Bindung der Objekte aus der LHS sowie der RHS auf die Objektstruktur vor und nach der Ausführung des Storypatterns dargestellt.

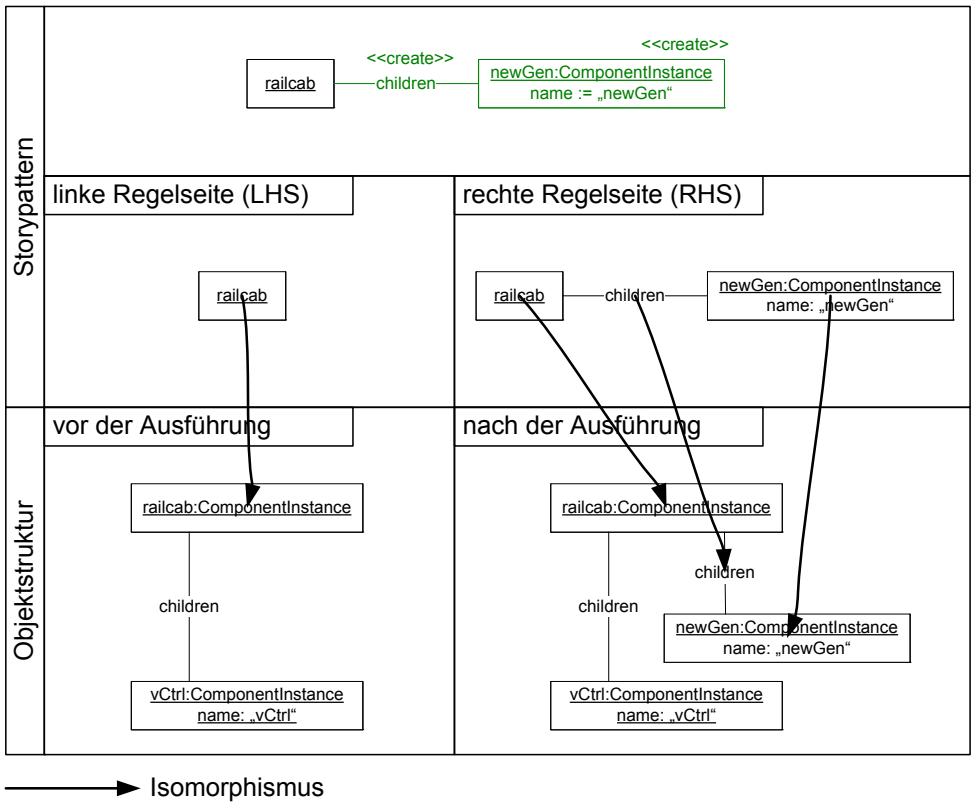


Abbildung 2.10: Storypattern für die Erstellung einer neuen Komponenteninstanz

2.4.3 Storydiagramme

Einzelne Storypattern werden als Aktivitäten in verfeinerte UML 1.x Aktivitätsdiagramme eingebettet. Dies führt dazu, dass Sequenzen von Storypatterns inkl. Schleifen sowie Entscheidungen modelliert werden können. UML Aktivitätsdiagramme bestehen aus einzelnen Aktivitäten, die durch Kontrollflusskanten verbunden sind. Da aus Storydiagrammen automatisch Quelltext generiert wird, werden Einschränkungen für den Kontrollfluss getroffen, der garantiert, dass Storydiagramme nur mit Hilfe von Schleifen (while) und Bedingungen (if) in Quelltext umgesetzt werden können. Der resultierende Quelltext ist dadurch wartbarer [Zün02].

Abbildung 2.11 zeigt die Graphgrammatik aus [Zün02] für den wohlgeformten Kontrollfluss eines Storydiagramms. Beginnend mit dem oben links dargestellten Startgraph werden die drei Graphersetzungsregeln für (a) Schleifen, (b) Sequenz von Aktivitäten und (c) Entscheidungen bzw. Abzweigungen ausgeführt, um das

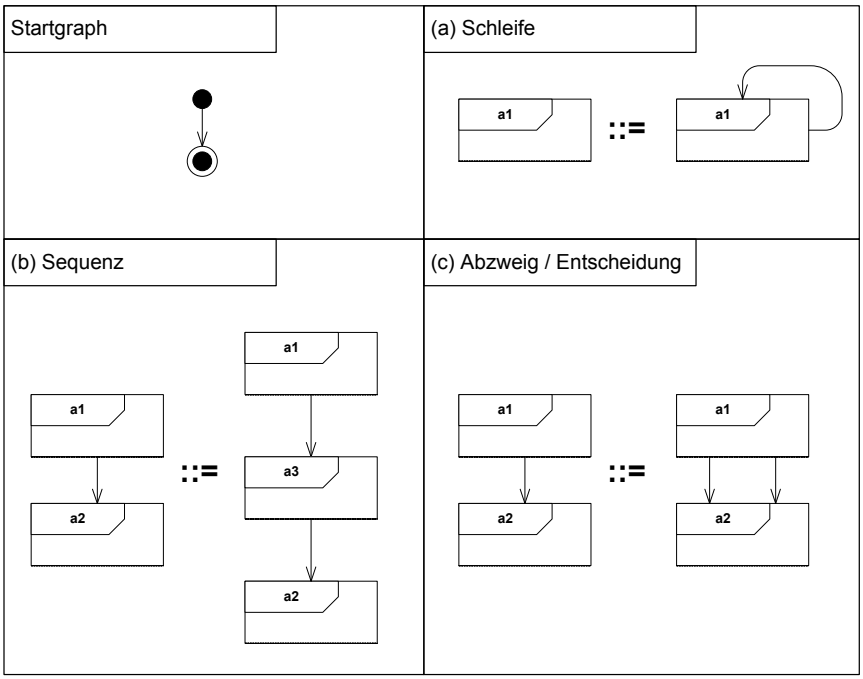


Abbildung 2.11: Graphgrammatik aus [Zün02] eines wohlgeformten Kontrollflusses

gewünschte Storydiagramm zu erzeugen. [Kle99] enthält eine detaillierte Version der oben dargestellten Graphgrammatik.

Für die komplette Syntax und formale Semantik von Story Driven Modeling und insbesondere für Storydiagramme und Storypatterns wird auf [Zün02] verwiesen.

3 Modellierung rekonfigurierbarer Architekturen

In den Grundlagen wurde die MECHATRONIC UML zur Spezifikation der Struktur und des Verhaltens mechatronischer Systeme vorgestellt. Die Architektur mechatronischer Systeme wird typischerweise komponentenbasiert entworfen [SZ04, Szy98, Gru03c, MDK94, GMW00]. Daher werden Komponenten in der MECHATRONIC UML als grundlegende Elemente zur Spezifikation der Struktur und damit der Architektur mechatronischer Systeme genutzt.

Die MECHATRONIC UML unterscheidet zwischen Komponententypen und deren Instanzen. Im Besonderen ermöglicht die MECHATRONIC UML die Definition der Struktur von Komponententypen als eine Komposition von unterlagerten Komponenteninstanzen [Bur06, BGT05]. Welche dieser Komponenteninstanzen in verschiedenen Situationen aktiv sind, wird mit Hybriden Rekonfigurationscharts [Bur06, BGT05] spezifiziert. Die Menge aktiver Komponenteninstanzen und deren Verbindungen wird als Konfiguration bezeichnet.

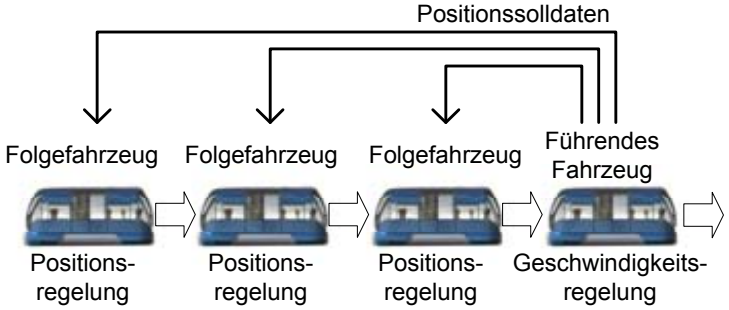


Abbildung 3.1: Regelung eines RailCab Konvois

In [BGT05] wurde die Echtzeitkoordination und die Regelung für Konvois aus RailCabs (s. Abbildung 3.1) mit der MECHATRONIC UML modelliert. Das Modell enthält die Komponenten eines RailCabs für die Regelung der Geschwindigkeit sowie der Position. In einem Hybriden Rekonfigurationschart wurde modelliert, welche Regelungskomponenten aktiv sind, je nachdem ob ein Fahrzeug vorne oder hinten in einem Konvoi beziehungsweise alleine fährt.

Die Regelung eines RailCab-Konvois beinhaltet neben der beschriebenen Geschwindigkeitsregelung eine Positionsregelung, um eine stabile Regelung des Konvois zu erreichen [HVB+05, HTS+08b, HTS+08a]. Ein Koordinator eines Konvois (im Folgenden füllt das führende Fahrzeug diese Rolle aus) versendet dabei kontinuierlich Sollpositionen an jedes einzelne Fahrzeug [GHH+06].

Für die Softwarearchitektur bedeutet dies, dass die Berechnung und das Versenden dieser Sollpositionen durch zusätzliche Komponenten im führenden Fahrzeug realisiert werden müssen. Die Berechnung der Sollposition kann, wie in [BG05, TH07, THHO08] beschrieben, für jedes nachfolgende Fahrzeug durch eine eigene Komponente im Koordinatorfahrzeug durchgeführt werden. Um dies für Konvois mit bis zu n Fahrzeugen mit der bisherigen Modellierung mit der Mechatronic UML umzusetzen, müssten prinzipiell $n + 1$ Zustände im Hybriden Rekonfigurationschart eines RailCabs modelliert werden: $n - 1$ Zustände um als führendes Fahrzeug die Sollpositionen für die folgenden Fahrzeuge zu berechnen und zu versenden (jeweils einen Zustand für einen Konvoi der Länge $2 \dots n$) sowie einen Zustand als folgendes Fahrzeug in einem Konvoi beliebiger Länge und einen Zustand für die Alleinfahrt. Dies führt dazu, dass die benötigten Modelle sehr groß und unübersichtlich werden.

Um dieses Problem zu beheben, wurde in [BG05] skizziert, wie Graphtransformationen genutzt werden können, um in den Hybriden Rekonfigurationscharts nicht alle Konfigurationen einzeln aufzuzählen, sondern stattdessen zu beschreiben, wie die unterlagerte Komponentenstruktur regelbasiert transformiert werden muss, wenn ein zusätzliches Fahrzeug zum Konvoi hinzustößt. Im Rahmen von [TH07, THHO08] wurde diese Idee aufgegriffen und mit den *Komponentenstorydiagrammen* eine Transformationssprache inkl. einer erweiterten Typdefinition für Komponenten in der Mechatronic UML auf Basis der UML Kompositionsstrukturen [Obj07] skizziert.

Die hierarchische Spezifikation der Komponententypen definiert *deklarativ* eine Menge an möglichen Komponenteninstanzstrukturen. Mit Komponentenstorydiagrammen wird dann *operational* beschrieben, wie ausgehend von einem Startgraph die Konfigurationen durch Ausführung einzelner Rekonfigurationen erreicht werden. Dies schränkt zum einen die Menge der möglichen Konfigurationen auf die Erreichbaren ein. Zum anderen ermöglicht das eine Quelltextsynthese aus den Komponentenstorydiagrammen zur Ausführung der Rekonfiguration zur Laufzeit. Da für den Quelltext eine maximale Ausführungszeit berechnet wird, sind Komponentenstorydiagramme auch in harten Echtzeitsystemen nutzbar.

Im Vergleich zu verwandten Arbeiten [TGM00, Mét98, Gru03c, KKJD06, WF02, Krä06, BGS05a, Sch06] erfüllen Komponentenstorydiagramme alle der in [BCDW04] beschriebenen Anforderungen zur Ausdrucksmächtigkeit und integrieren sich durch eine an Komponentendiagramme angelehnte graphische Syntax

gut mit der Spezifikation der Komponententypen. Des Weiteren sind Komponentenstorydiagramme stark typisiert, semantisch geeignet definiert und können durch einen kontextsensitiv syntaxgesteuerten Editor umgesetzt werden, so dass es nicht möglich ist, Komponentenstorydiagramme zu spezifizieren, die bei der Ausführung nicht typkonforme Konfigurationen erzeugen.

Im Rahmen dieses Kapitels wird in Abschnitt 3.1 eine Erweiterung der Mechatronic UML zur Spezifikation von Komponententypen mit einer variablen Anzahl an unterlagerten Komponenteninstanzen auf Basis der Kompositionsstrukturen der UML 2 [Obj07] beschrieben. Darauf aufbauend werden in Abschnitt 3.2 Komponentenstorydiagramme zur Spezifikation der Instantiierung und Rekonfiguration von Komponenteninstanzstrukturen vorgestellt. Abschnitt 3.3 betrachtet die Ausführung der Komponentenstorydiagramme zur Laufzeit in harten Echtzeitsystemen.

3.1 Strukturspezifikation

Die Struktur mechatronischer Systeme wird in der Mechatronic UML mittels Komponentendiagrammen spezifiziert. Wie in Abschnitt 2.3 dargestellt, unterscheidet die Mechatronic UML zwischen der Definition eines Komponententyps und dessen Instantiierung.

3.1.1 Komponententyp

Eine Komponente ist laut Szyperski [Szy98] ein eigenständiges Element einer Softwarearchitektur, welches über definierte Schnittstellen mit anderen Komponenten verbunden wird. Diese Definition unterscheidet nicht zwischen einer Typdefinition einer Komponente und deren Instantiierung. In Architekturbeschreibungssprachen wie Acme [GMW00], Darwin [MDK94], UML 2 [Obj07] und auch der Mechatronic UML werden Typen und Instanzen von Komponenten zum Zweck der Wiederverwendbarkeit unterschieden.

3.1.1.1 Ports

Ports sind die Interaktionspunkte einer Komponente zu anderen Komponenten. In der Mechatronic UML werden diskrete, kontinuierliche und hybride Ports unterschieden. Diese Unterscheidung ist im Rahmen dieser Arbeit nicht relevant. Es wird daher im Folgenden allgemein von Ports gesprochen. Ein Komponententyp besitzt eine Menge von Ports. Bisher hatten Ports in der Mechatronic UML implizit die Kardinalität 1 bzw. 0..1, wenn sie optional waren. Für die

Kommunikation mit mehreren Partnern, deren exakte Anzahl erst zur Laufzeit bekannt ist, wird in dieser Arbeit wie in [Hir08] die MECHATRONIC UML um Kardinalitäten für Ports erweitert. Mögliche Kardinalitäten sind hierbei 0..* und 0..1. Die genaue Anzahl der Portinstanzen wird durch die Ausführung von Komponentenstorydiagrammen (s. Abschnitt 3.2) zur Laufzeit festgelegt. Die Spezifikation von Kardinalitäten ermöglicht also die Spezifikation von Komponententypen, deren Instanzen eine unterschiedliche Anzahl an Kommunikationspartnern haben können, wie zum Beispiel das führende Fahrzeug in einem RailCab-Konvoi.

3.1.1.2 Schnittstellen

Die möglichen Interaktionen zwischen Komponenten über Ports werden mit Hilfe von Schnittstellen definiert. Eine Schnittstelle beschreibt hierbei in der UML eine öffentliche Merkmalseigenschaft (*feature*) oder die Verpflichtung (*obligation*), einen gewissen Dienst zur Verfügung zu stellen [Obj07, S. 86]. In der MECHATRONIC UML sind das die verschiedenen Nachrichten, die empfangen und verschickt werden (s. [Bur06, S. 12]). Es wird zwischen angebotenen sowie benötigten Schnittstellen unterschieden. Komponenten werden somit über ihre Ports miteinander verbunden, wenn eine benötigte und eine angebotene Schnittstelle verbunden werden.

3.1.1.3 Attribute

Komponententypen und deren Ports können analog zu Klassen im objektorientierten Entwurf Attribute besitzen. Attribute ermöglichen es, zusätzliche Informationen zu einem Komponententyp beziehungsweise Port abzulegen. Den Instanzen von Komponententypen und Ports werden bei oder nach der Instantiierung Werte für diese Attribute zugewiesen. Die Attribute können die folgenden Typen haben: ganze Zahl (Integer), Fließkommazahl (Double), Zeichenkette (String) sowie Wahrheitswert (Boolean).

3.1.1.4 Beispiel

Im Rahmen des RailCab-Beispiels werden in Abbildung 3.2 verschiedene Komponententypen des Softwaremodells eines RailCabs für die Konvoifahrt gezeigt.

Die Komponenten VCtrl und PosCtrl sind die Geschwindigkeits- und Positionsregler eines RailCabs. Die Komponenten berechnen die Soll-Kraft F* bzw. die Soll-Geschwindigkeit V*, die vom Linearmotor erzeugt werden soll. Der Positionsregler wird genutzt, wenn ein RailCab im Konvoi nicht vorne fährt und vom führenden RailCab kontinuierlich Sollpositionen für die Fahrt im Konvoi erhält.

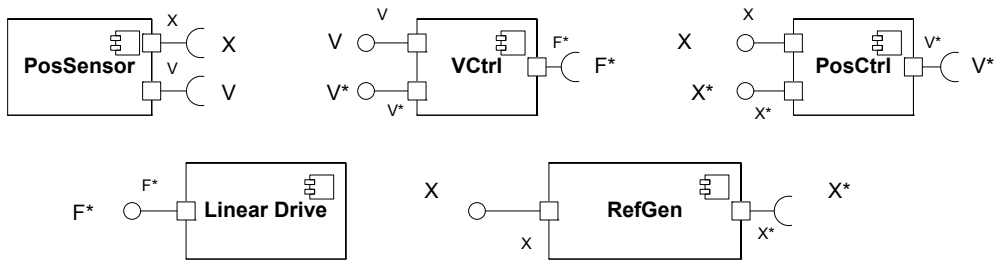


Abbildung 3.2: Komponententypen des RailCab-Beispiels

Die Komponente PosSensor liefert die aktuelle Geschwindigkeit V sowie die aktuelle Position X auf der Strecke. Die Komponente RefGen wird bei einer Konvoifahrt vom führenden Fahrzeug genutzt, um die Sollpositionen für die nachfolgenden RailCabs zu berechnen. Die Kardinalitäten der Ports sind aus Gründen der Übersichtlichkeit nicht im Diagramm dargestellt. Alle Ports haben die Kardinalität 0..1.

Abbildung 3.3: RailCab Komponententyp

Abbildung 3.3 zeigt den RailCab Komponententyp. Er besitzt mehrere Ports zur Verbindung mit anderen Komponententypen. Der Port X^*_{out} besitzt die Kardinalität 0..*, dargestellt durch den hinterlegten skizzierten Schatten. Eine Instanz des Komponententyps kann also beliebig viele Portinstanzen von diesem Port haben, um in diesem Fall als vorne fahrendes Fahrzeug die Sollpositionen an die übrigen im Konvoi fahrenden RailCabs zu senden. Diese Sollpositionen werden über den Port X^*_{in} mit der Kardinalität 0..1 empfangen.

Bisher wurden nur nicht-hierarchische Komponententypen beschrieben. Der RailCab Komponententyp besitzt allerdings mehrere eingebettete Komponenteninstanzen der Typen aus Abbildung 3.2, um seine Funktionalität zu erfüllen. Im Folgenden wird die Spezifikation hierarchischer Komponententypen vorgestellt.

3.1.2 Hierarchische Komponententypen

In der Mechatronic UML können hierarchische Komponententypen beschrieben werden. Diese hierarchischen Komponententypen definieren sich als Komposition unterlagerter Komponenteninstanzen (s. Abschnitt 2.3.3). Wie in der

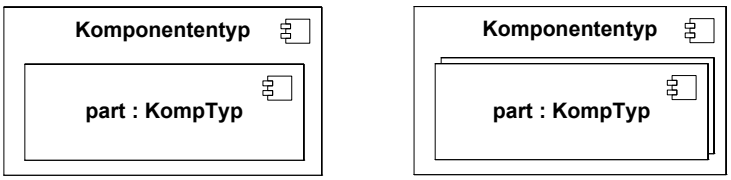
Einführung zu diesem Kapitel dargelegt, kann bisher nicht das mehrfache Vorkommen dieser Komponenteninstanzen spezifiziert werden.

In der Architekturbeschreibungssprache ROOM [SGW94] wurden für solche hierarchischen Komponenten *Replicated Actors* sowie *Replicated Ports* eingeführt. Dies ermöglicht die Spezifikation des mehrfachen Vorkommens von unterlagerten Komponenteninstanzen. Es kann neben dem z.B. 3-fachen Vorkommen einer unterlagerten Komponenteninstanz auch das *-fache Vorkommen der Instanz spezifiziert werden. In der UML 2 [Obj07] wurde dieses Konzept als *Kompositionsstrukturen* übernommen.

Zur Modellierung von Komponententypen mit einer variablen Anzahl an unterlagerten Komponenteninstanzen werden die Kompositionsstrukturen in die Typdefinition der MECHATRONIC UML integriert. Die resultierenden Komponententypmodelle definieren also Typgraphen für die zur Laufzeit existierenden unterlagerten Komponenteninstanzstrukturen. Die einzelnen in einen Komponententyp eingebetteten Elemente, die Komponenten, Ports und Konnektoren repräsentieren, werden wie in der UML 2 [Obj07] *Parts* genannt.

3.1.2.1 Komponentenparts

Die wichtigsten Elemente bei der Definition eines hierarchischen Komponententyps sind die unterlagerten Repräsentanten von Komponenteninstanzen – *Komponentenparts* genannt. Für diese Parts werden Kardinalitäten angegeben. Hier sind die Kardinalitäten `0..*` und `0..1` zulässig. Eine Kardinalität von `0..1` bedeutet, dass zur Laufzeit max. eine Komponenteninstanz existiert. Bei `0..*` kann eine beliebige Anzahl von Komponenteninstanzen zur Laufzeit existieren. Die konkrete Ausprägung eines Parts durch die Anzahl der Komponenteninstanzen zur Laufzeit wird durch die Ausführung von Komponentenstorydiagrammen (s. Abschnitt 3.2) festgelegt.



(a) Komponentenpart

(b) Komponentenpart mit Kardinalität `0..*`

Abbildung 3.4: Komponentenparts [Hol08]

Für die Darstellung der Parts sind verschiedene graphische Varianten möglich. Die gewählte Darstellung folgt der UML 2.0 Spezifikation, indem ein Part

als Rechteck innerhalb des umfassenden Komponententyps dargestellt wird. Der Name des Parts sowie der referenzierte Komponententyp werden innerhalb des Rechtecks dargestellt. Im Unterschied zu Komponenteninstanzen wird der Name nicht unterstrichen. Für die Kardinalität `0..*` wird das Rechteck in Anlehnung an die graphische Darstellung in ROOM doppelt gezeichnet. Abbildung 3.4 zeigt die Darstellung von Komponentenparts innerhalb eines Komponententyps.

3.1.2.2 Port- und Schnittstellenparts

Wenn ein Komponententyp als Part in einen anderen Komponententyp eingebettet wird, werden zusätzlich die Ports automatisch mit den zugehörigen Schnittstellen von den jeweiligen Komponententypen als Port- und Schnittstellenparts übernommen. Die Kardinalitäten der Ports werden automatisch übernommen. Abbildung 3.5 zeigt die Darstellung von Port- und Schnittstellenparts.

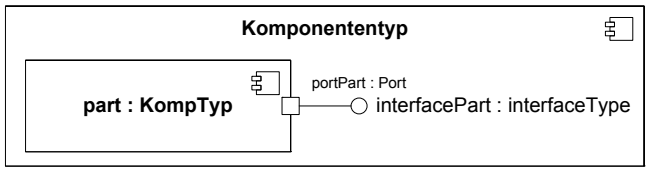
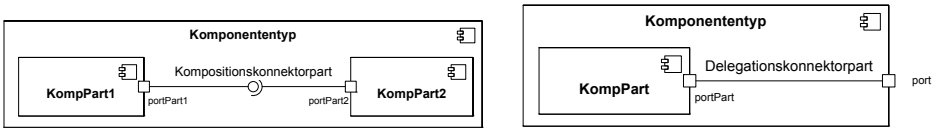


Abbildung 3.5: Port- und Schnittstellenparts [Hol08]

3.1.2.3 Konnektorparts

Konnektorparts werden in den Kompositionsstrukturen genutzt, um verschiedene Parts zu verbinden. Sie beschreiben mögliche Verbindungen zwischen den Ports der einzelnen Parts analog zu Assoziationen in Klassendiagrammen. Im Unterschied zu den bisher eingeführten Komponentenparts beschreiben sie keinen Repräsentanten von einem bereits definierten Typen. Sie werden als eingebettetes Element eines hierarchischen Komponententyps dennoch aus Gründen der Konsistenz Part genannt.

Konnektoren haben einen Namen. An den Enden der Konnektoren zur Verbindung mit den Ports werden Kardinalitäten annotiert. Diese Kardinalitäten beschreiben wie bei Assoziationsrollen, wie viele Portinstanzen über diesen Konnektor mit anderen Portinstanzen verbunden werden können. Falls die Quell- oder Zielkardinalität `0..*` beträgt, muss der zugehörige Portpart oder Komponentenpart auch die Kardinalität `0..*` haben, da sonst die Kardinalität des Konnektors erlaubt mehrere Elemente zu verbinden, diese Elemente allerdings nicht mehrfach vorkommen können.



(a) Kompositionskonnektorpart (b) Delegationskonnektorpart

Abbildung 3.6: Konnnektorparts [Hol08]

Kompositionskonnektorparts verbinden zwei Portparts innerhalb eines hierarchischen Komponententyps. *Delegationskonnektorparts* sind Konnektoren zwischen dem Port eines Komponententyps und einem Portpart eines untergeordneten Komponentenparts. Abbildung 3.6 stellt diese dar.

3.1.2.4 Systemdefinition

Der oberste Komponententyp einer Hierarchie wird als System bezeichnet. Er darf keine Ports haben, da er als oberstes Element nicht mit weiteren Komponenten verbunden werden kann, und darf nicht in andere Komponententypen als Part eingebettet werden. Dieser Komponententyp wird zusätzlich mit dem Stereotyp ≪system≫ gekennzeichnet.

3.1.2.5 Beispiel

Abbildung 3.7 zeigt die Spezifikation der unterlagerten Partstruktur des Komponententyps RailCab. Im Komponententyp sind Parts der in Abbildung 3.2 dargestellten Komponententypen eingebettet. Diese Parts haben bis auf den Part refGen:RefGen die Kardinalität 0..1; sie können also für eine Instanz des Komponententyps RailCab nur einmal instantiiert werden. Der Part refGen:RefGen hat die Kardinalität 0..*. Wie beschrieben, berechnet er die Sollpositionen der übrigen Fahrzeuge in einem Konvoi. In einem Konvoi der Länge n existieren also $n-1$ Instanzen dieses Parts im führenden Fahrzeug.

Die einzelnen Parts sind über Kompositions- und Delegationskonnektorparts miteinander und mit den außenliegenden Ports des Komponententyps RailCab verbunden. Im Beispiel sind alle Kardinalitäten der Konnektorparts 0..1. Der next Kompositionskonnektorpart verknüpft zwei refGen:RefGen Komponentenparts miteinander. Er ermöglicht die Instantiierung einer Kette von RefGen Komponenten analog zu einer Selbstassoziation in einem Klassendiagramm.

Abbildung 3.8 zeigt die oberste Komponentenebene des Beispiels. Ein ConvoySystem besteht aus zwei Parts. Der Part convoyLeader:RailCab repräsentiert das führende Fahrzeug in einem Konvoi. Die hinterherfahrenden Fahrzeuge werden

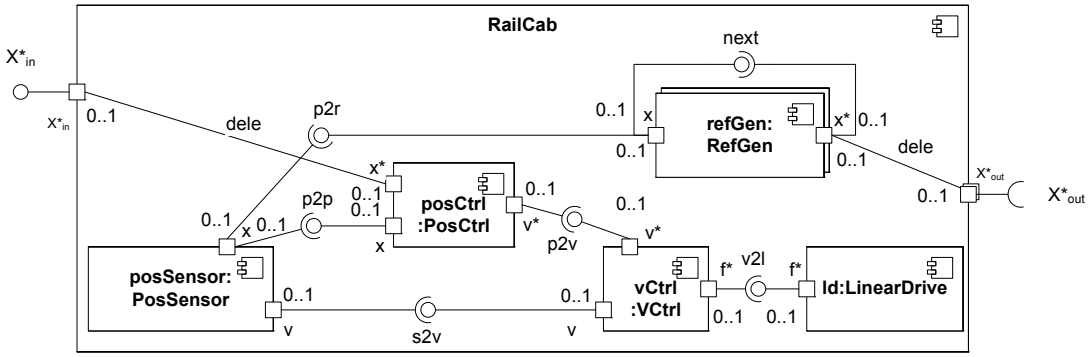


Abbildung 3.7: Hierarchischer RailCab Komponententyp

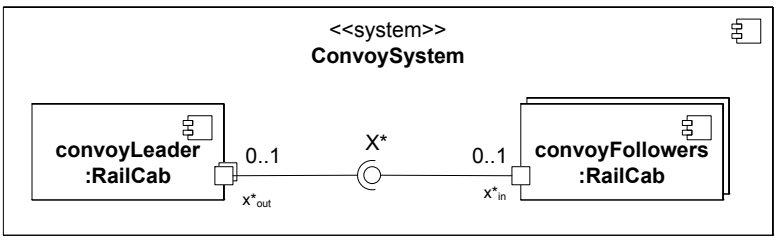


Abbildung 3.8: Spezifikation des Konvoisystems

durch den Part convoyFollowers:RailCab repräsentiert. Die beiden Parts werden durch den X^* Kompositionskonnektorpart verbunden. Er hat die Quell- und Zielkardinalität 0..1 und verbindet damit jeweils eine Portinstanz X^*_{out} des Parts convoyLeader mit der X^*_{in} Portinstanz eines der folgenden Fahrzeuge. Mit diesem Konnektorpart wird der Versand der Sollpositionen vom führenden Fahrzeug an die folgenden Fahrzeuge modelliert.

3.1.3 Komponenteninstanzen

Komponenteninstanzen werden von Komponententypen gebildet. Eine Komponenteninstanzstruktur entspricht einer Konfiguration. Bei der Instantiierung werden die variablen Anteile der Typdefinition der Komponenten festgelegt. Dies betrifft zum einen für die Komponente selber die Anzahl der aus Ports mit der Kardinalität 0..* erzeugten *Portinstanzen*. Zum anderen betrifft dies die der Komponente unterlagerte Partstruktur.

Für alle Komponentenparts der Kompositionsstruktur existieren *Komponenteninstanzen* der über den Part referenzierten Typen. Es können zur Laufzeit auch mehrere Instanzen eines Parts existieren, solange die für den Part festgelegte Kardinalität eingehalten wird. Diese Komponenteninstanzen besitzen Portinstanzen der beim Komponententyp angegebenen Ports; deren Anzahl wird ebenfalls durch die festgelegte Kardinalität eingeschränkt.

Die Komponenteninstanzen werden in Abhängigkeit zur Spezifikation der Konnektorparts in der Kompositionsstruktur mit *Konnektoren* verbunden. *Delegationskonnektoren* sind Konnektoren zwischen einer Portinstanz einer Komponenteninstanz und einer Portinstanz einer untergeordneten Komponenteninstanz. Sie delegieren alle von außen an die übergeordnete Komponenteninstanz eingehende Informationen an die untergeordnete Komponenteninstanz zur weiteren Verarbeitung. *Kompositionskonnektoren* verbinden Portinstanzen von Komponenteninstanzen innerhalb einer Hierarchieebene.

Es ergibt sich insgesamt also eine hierarchische Komponenteninstanzstruktur, die aus Komponenteninstanzen besteht, die über Portinstanzen und Konnektorinstanzen verbunden sind.

3.1.3.1 Implizite Kompositionsbeziehungen zwischen Elementen der Instanzstruktur

In Komponenteninstanzstrukturen können bestimmte Elemente nicht ohne andere Elemente existieren – sie sind Teil eines anderen Elements. Beziehungen zwischen diesen Elementen werden im Folgenden in Anlehnung an Kompositionen in UML Klassendiagrammen *Kompositionsbeziehungen* genannt. Sie sind impli-

zit durch die Semantik der Komponenteninstanzstrukturen gegeben und müssen nicht explizit modelliert werden.

Kompositionsbeziehungen existieren zwischen den verschiedenen Elementen der Komponenteninstanzstruktur. Diese Kompositionsbeziehungen existieren zum einen zwischen den Elementen unterschiedlicher Hierarchien und zum anderen auch zwischen den Elementen der gleichen Hierarchie.

Ersteres betrifft die einer Komponenteninstanz untergeordneten Instanzen (z.B. Komponenteninstanzen), die nicht ohne diese übergeordnete Komponenteninstanz existieren können (mit Ausnahme der Instanzen der Systemebene, die keine übergeordnete Hierarchie besitzen).

Die zweite Kompositionsbeziehung existiert zwischen Portinstanzen und Komponenteninstanzen sowie zwischen Konnektorinstanzen und Portinstanzen. Portinstanzen sind die Interaktionspunkte zwischen einer Komponenteninstanz und der Umwelt (sprich anderen Komponenteninstanzen) [GMW00]. Dementsprechend existiert eine Portinstanz nicht ohne die zugehörige Komponenteninstanz. Da Konnektorinstanzen die Verknüpfung zwischen Portinstanzen darstellen, existieren sie ebenfalls nicht ohne die zugehörigen Portinstanzen.

3.1.3.2 Beispiel

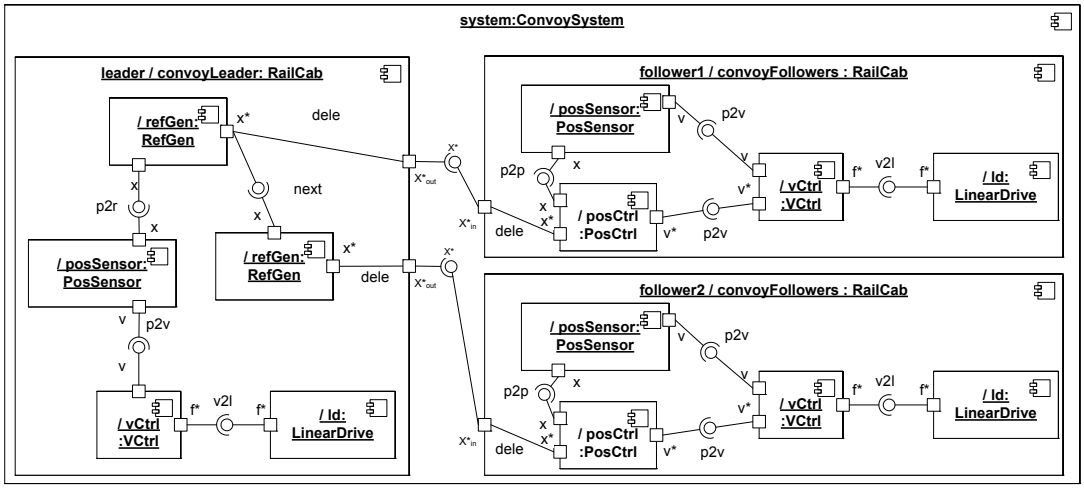


Abbildung 3.9: Instanzsituation eines Konvois aus drei Fahrzeugen

Abbildung 3.9 zeigt die Komponenteninstanzstruktur eines Konvois aus drei Fahrzeugen. Auf der linken Seite der Abbildung ist die Komponenteninstanz des führenden Fahrzeugs leader/convoyLeader:RailCab abgebildet. Diese Komponenteninstanz repräsentiert den Part convoyLeader vom Typ RailCab in einer Instanz des

Komponententyps ConvoySystem. Sie enthält Komponenteninstanzen für die Geschwindigkeitsregelung und den Linearmotor sowie die Komponenteninstanzen zur Berechnung der Sollpositionen für die folgenden RailCabs, die über die beiden Portinstanzen X^*_{out} gesendet werden.

Auf der rechten Seite sind die Komponenteninstanzen der zwei folgenden Fahrzeuge abgebildet. Beide Komponenteninstanzen enthalten sowohl den Positionssensor und Linearmotor als auch den Positionsregler und den Geschwindigkeitsregler, welche die empfangenen Sollpositionen zur Regelung nutzen.

3.1.4 Abstrakte Syntax

Abbildung 3.10 zeigt die Paketstruktur der Metamodelle für die Modellierung der beschriebenen erweiterten Komponentenstrukturen. Die Metamodelle formalisieren zusammen mit in OCL [OMG03] beschriebenen Invarianten die abstrakte Syntax.[1] Das Paket Structure::Type enthält die Metamodellklassen für die Typdefinition einer nicht-hierarchischen Komponente. Das Paket Structure::Part erweitert das Type Metamodell um die Definition hierarchischer Komponententypen in Bezug auf die Definition der eingebetteten Parts. Das Paket Structure::Instance enthält schließlich die Metamodellklassen für Komponenteninstanzstrukturen.

Um eine möglichst nahtlose Integration mit den bisherigen Metamodellen der Mechatronic UML zu erreichen, wurde darauf verzichtet das in [Obj07] beschriebene Metamodell der UML Kompositionsstrukturen zu nutzen. Statt dessen wurden nur die relevanten Anpassungen am derzeitigen Metamodell der Mechatronic UML durchgeführt.

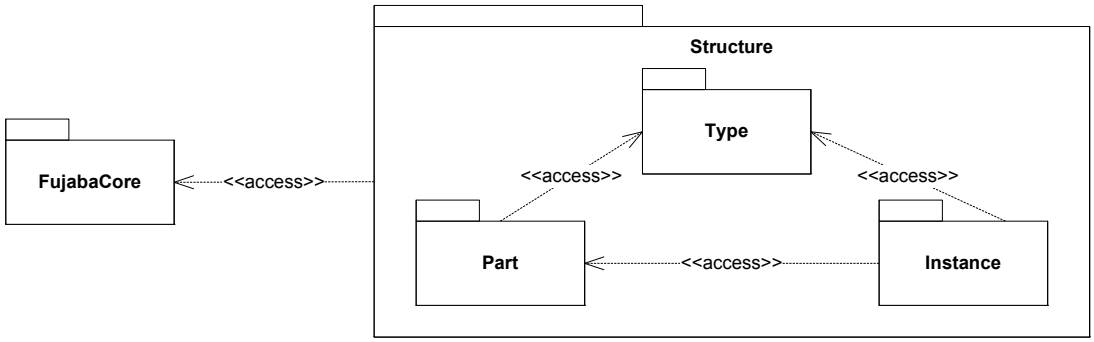


Abbildung 3.10: Pakete der Metamodelle der Typdefinition

[1] Aus Gründen der besseren Lesbarkeit werden keine Rollennamen an Assoziationen dargestellt. Dementsprechend werden in den OCL-Invarianten zur Navigation die Assoziationsnamen und nicht die Rollennamen genutzt.

3.1.4.1 Typdefinition

Die Metamodellklassen zur Typdefinition einer nicht-hierarchischen Komponente entsprechen weitestgehend den aktuellen Metamodellklassen aus der MECHATRONIC UML (s. Abbildung 3.11). Der Komponententyp wird durch die Metamodellklasse Component definiert. Ein Komponententyp kann mehrere Ports haben (Port), die wiederum über mehrere benötigte und angebotene Schnittstellen (RequiredInterface bzw. ProvidedInterface) verfügen. Für die Schnittstellen werden, wie in der bisherigen MECHATRONIC UML, die erlaubten Methoden (bzw. Nachrichten) mittels Interface-Klassen (FujabaCore::UMLClass) angegeben.

Erweiterungen gegenüber der bisherigen MECHATRONIC UML sind die Unterstützung von Attributen für Typen sowie Kardinalitäten für Ports. Ersteres ermöglicht weitergehende Möglichkeiten zur Modellierung der Eigenschaften von Komponenten, Ports und Schnittstellen analog zu attributierten Graphen (s. [Roz97]). Instanzen der Typen besitzen dann die konkreten Werte der Attribute (s. Abschnitt 3.1.4.3).

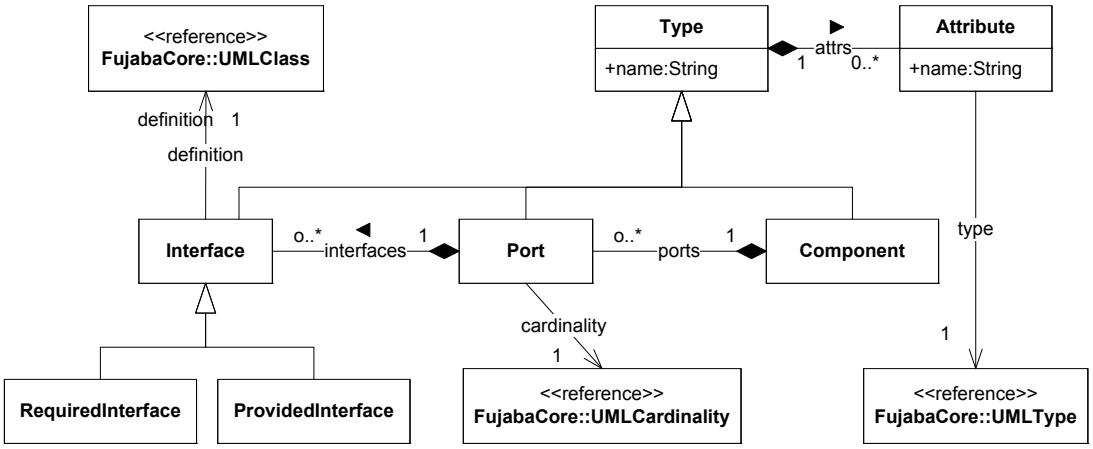


Abbildung 3.11: Metamodell der Typdefinition. Paket: Structure::Type

3.1.4.2 Hierarchische Komponententypen

Abbildung 3.12 zeigt die Metamodellklassen für die Definition der Struktur der in einen Komponententyp eingebetteten Parts in Anlehnung an die UML Kompositionsstrukturen. Ein Komponententyp (Component) enthält die verschiedenen Parts über die Komposition parts. Jeder Part ist typisiert über die Beziehung zur Metaklasse Type. Neben den Parts für Schnittstellen, Ports und Komponenten werden hier die möglichen Verbindungen zwischen Elementen durch die Meta-

klassen DelegationPart und AssemblyPart für Delegations- sowie Kompositionskonnektorparts definiert. Die abstrakte Metaklasse ConnectorPart definiert durch die unidirektionalen Assoziationen zur aus dem Kern von Fujaba stammenden Metaklasse UMLCardinality deren Kardinalitäten. Analog gilt dies für die Kardinalität der übrigen Parts.

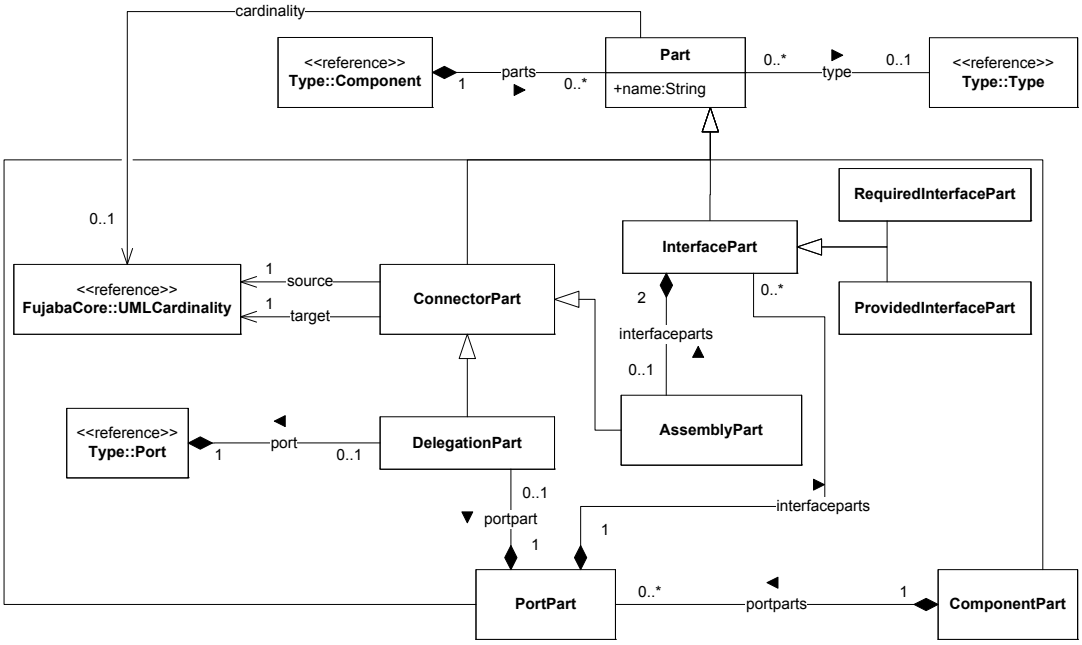


Abbildung 3.12: Metamodell der hierarchischen Typdefinition. Paket: Structure::Part

Für die Metamodellklassen des Pakets Structure::Part gelten die folgenden zusätzlichen Bedingungen:

Invariante 3.1 *Subklassen der Klasse Part dürfen nur über kompatible Metaklassen typisiert werden.*

context PortPart **inv**:
type.oclType(Port)

context ComponentPart **inv**:
type.oclType(Component)

context RequiredInterfacePart **inv**:

type.oclType(RequiredInterface)

context ProvidedInterfacePart **inv**:
type.oclType(ProvidedInterface)

Invariante 3.2 *ConnectorParts besitzen keinen Typ.*

context ConnectorPart **inv**:
type→isEmpty()

Invariante 3.3 *AssemblyParts müssen mit je einem ProvidedInterfacePart und einem RequiredInterfacePart verbunden werden.*

context AssemblyPart **inv**:
interfaceparts→**exists**(i1 |
interfaceparts→**exists**(i2 | i1 <> i2 **and**
i1.oclType(ProvidedInterfacePart) **and** i2.oclType(RequiredInterfacePart)))

Invariante 3.4 *AssemblyParts dürfen nur InterfaceParts verbinden, welche die gleiche Interface-Klasse haben.*

context AssemblyPart **inv**:
interfaceparts.type.definition→size() = 1

Invariante 3.5 *ConnectorParts nutzen nicht die Assoziation cardinality zur Definition der Kardinalität, sondern nur die Assoziationen source und target.*

context ConnectorPart **inv**:
cardinality→isEmpty()

3.1.4.3 Instanzstrukturen

Abbildung 3.13 zeigt die Metaklassen des Pakets Structure::Instance für die Definition von hierarchischen Komponenteninstanzstrukturen. Instanzen sind allgemein typisiert über die Assoziationen part und type. Für die auf Typebene definierten Attribute können Werte (Metaklasse AttributeValue) für die Instanzen angegeben werden. Eine Komponenteninstanz beinhaltet Komponenten-, Port- und Schnittstelleninstanzen. Die Metaklassen Delegation und Assembly stellen die Delegations- und Kompositionskonnektoren auf Instanzebene zwischen Portinstanzen dar.

Für die Metamodellklassen des Pakets Structure::Instance gelten die folgenden zusätzlichen Bedingungen:

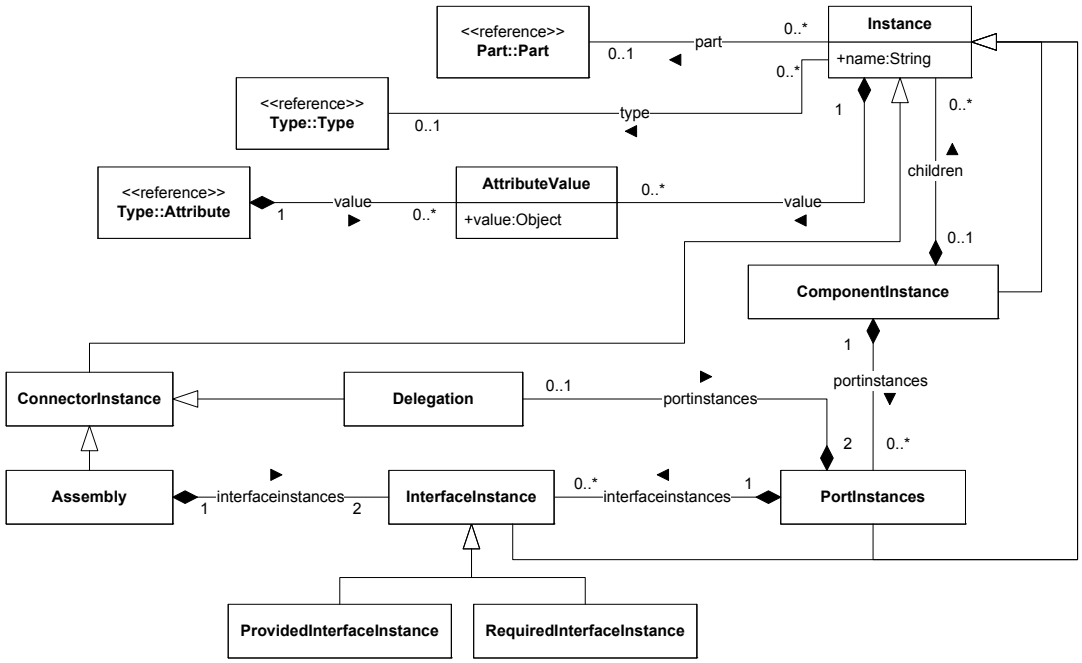


Abbildung 3.13: Metamodell der hierarchischen Instanzstrukturen. Paket: Structure::Instance

Invariante 3.6 *Subklassen der Klasse Instance dürfen nur mit kompatiblen Subklassen der Metaklassen Part und Type verbunden werden.*

context PortInstance **inv**:
type.oclType(Port) **and** part.oclType(PortPart)

context ComponentInstance **inv**:
type.oclType(Component) **and** part.oclType(ComponentPart)

context RequiredInterfaceInstance **inv**:
type.oclType(RequiredInterface) **and** part.oclType(RequiredInterfacePart)

context ProvidedInterfaceInstance **inv**:
type.oclType(ProvidedInterface) **and** part.oclType(ProvidedInterfacePart)

context Assembly **inv**:

part.oclType(AssemblyPart)

context Delegation **inv**:
part.oclType(DelegationPart)

Invariante 3.7 *Assemblys und Delegations besitzen keinen Typ.*

context Assembly **inv**:
type→isEmpty()

context Delegation **inv**:
type→isEmpty()

Invariante 3.8 *Die Typisierung der Instances muss konsistent sein.*

context Instance **inv**:
part.type = type

Invariante 3.9 *Assemblys müssen mit je einer ProvidedInterfaceInstance und einer RequiredInterfaceInstance verbunden werden.*

context Assembly **inv**:
interfaceinstances→**exists**(i1 |
interfaceinstances→**exists**(i2 | i1 <> i2 **and**
i1.oclType(ProvidedInterfaceInstance) **and**
i2.oclType(RequiredInterfaceInstance)))

Invariante 3.10 *Assemblys dürfen nur InterfaceInstances verbinden, welche die gleiche Interface-Klasse haben.*

context Assembly **inv**:
interfaceinstances.type.definition→size() = 1

3.2 Strukturtransformationen

Strukturanpassung (vgl. Abschnitt 2.1) stellt neben der Parameteranpassung die zweite Möglichkeit der Verhaltensanpassung für selbstoptimierende Systeme dar. Die in Abschnitt 3.1 dargestellte Sprache zur Modellierung von Komponententypen und Komponenteninstanzstrukturen wird in diesem Abschnitt um eine

stark typisierte Sprache zur Modellierung der Transformation von Komponenteninstanzstrukturen unter Beachtung der in einem hierarchischen Komponententypen definierten Parts ergänzt.

Die Sprache wird zu Beginn informal erläutert, danach wird die Syntax durch ein Metamodell formalisiert. Schlussendlich wird die Sprache durch eine Übersetzung auf Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen semantisch fundiert.

3.2.1 Anforderungen

Bradbury et al. haben in [Bra04, BCDW04] Kriterien für Techniken zur Entwicklung dynamischer Softwarearchitekturen eingeführt und verschiedene Ansätze diesbezüglich eingeordnet. Bradbury stellt 3 grundlegende Klassifikationsaspekte vor:

1. „What type of change is supported?“
2. „What kind of process implements this change?“
3. „What infrastructure is available to support the change process?“

Für die hier dargestellten Transformationen der Komponenteninstanzstrukturen ist vor allem der erste Klassifikationsaspekt relevant. Mit diesem Aspekt wird die Mächtigkeit der Transformationssprache beschrieben. Die beiden weiteren Aspekte zielen vor allem auf die Infrastruktur für die Ausführung der Transformationen zur Laufzeit bzw. die Einbettung in andere Verhaltensmodelle, die festlegen, wann eine Transformation ausgeführt wird. Diese Aspekte werden hier nicht weiter betrachtet. Im Folgenden werden die Anforderungen für die Sprache definiert:

3.2.1.1 Einfache Änderungsoperationen

Einfache Änderungsoperationen sind das Hinzufügen von Komponenteninstanzen und Konnektoren sowie das Löschen dieser Elemente.

3.2.1.2 Zusammengesetzte Änderungsoperationen

Unter zusammengesetzten Änderungsoperationen werden von Bradbury et al. sowohl einfache Änderungsoperationen auf Gruppen bzw. hierarchischen Substrukturen als auch eine Sequenz von Änderungsoperationen verstanden. Diese Sequenz von Änderungsoperationen kann auch Entscheidungen und Schleifen

beinhalten. Der (möglicherweise rekursive) Aufruf von zusammengesetzten Änderungsoperationen wird zwar nicht explizit genannt, kann aber auch als zusammengesetzte Änderungsoperation verstanden werden.

3.2.1.3 Variabilität der Architekturelemente

Die Variabilität der Architekturelemente beschreibt, ob neue Architekturelemente während der Änderungen hinzukommen können. Die Menge der Architekturelemente kann entweder (1) fest („fix“) sein, dass heißt es dürfen keine neuen Elemente hinzukommen, (2) teilweise variabel („partial variable“) sein, dass heißt es dürfen neue Instanzen der Architekturelemente hinzugefügt werden, die Menge der Architekturelementtypen ist allerdings fest, und (3) es können auch neue Typen zur Laufzeit hinzukommen. In dieser Arbeit wird der Fokus auf Variante (2) gesetzt.

3.2.1.4 Zusätzliche Anforderungen

Die in [Bra04, BCDW04] dargestellten Anforderungen befassen sich vor allem mit der Mächtigkeit der verschiedenen Ansätze zur Beschreibung und Umsetzung von Änderungen der Architektur und nur teilweise mit der Möglichkeit zur Definition, wo eine Änderung durchgeführt werden muss, oder weitergehende Konzepte bzgl. der Typisierung, der Einbettung in Architekturbeschreibungssprachen und graphische im Gegensatz zu textueller Syntax. Diese zusätzlichen Anforderungen werden im Folgenden kurz charakterisiert.

Typisierung Architekturelemente wie Komponenten, Ports und Konnektoren sind in Architekturbeschreibungssprachen üblicherweise typisiert (vgl. [SGW94, Obj07, MDK94]). Eine Sprache zur Modellierung von Architekturtransformationen sollte daher auch diese Typisierungsinformationen bei der Auswahl und Änderung der Architektur mitbetrachten. Dies folgt dem bei der Sprachentwicklung weit verbreiteten Konzept der *starken Typisierung*, die es ermöglicht, während der Übersetzung bereits Typüberprüfungen durchzuführen und nicht erst zur Laufzeit (vgl. [Mey97]).

Die starke Typisierung der Sprache erlaubt die Implementierung eines syntaxgesteuerten kontextsensitiven Editors. Dieser garantiert, dass die durch die Ausführung der spezifizierten Transformationen resultierende Komponenteninstanzstruktur zur Typdefinition *konform* ist, also auch die spezifizierten Kardinalitäten und Kompositionsbeziehungen einhält.

Definition der Anwendungsstelle Bevor eine Änderung durchgeführt wird, muss die Anwendungsstelle für die Änderung festgelegt werden. Eine Transformationssprache kann hier neben der Spezifikation der für die Anwendung der Transformation benötigten Architekturelemente zusätzlich auch die Möglichkeit von verbotenen oder optionalen Architekturelementen unterstützen.

Des Weiteren ist es hilfreich, wenn Architekturelemente Attribute besitzen, deren aktuelle Werte in der Definition der Anwendungsstelle berücksichtigt werden. Dies erleichtert es, ein Element mit einem bestimmten Attributwert direkt auszuwählen, ohne alle Elemente durchsuchen zu müssen.

Unterstützung von Ports Architekturbeschreibungssprachen beschreiben die Architektur üblicherweise [SGW94, MDK94, Obj07, GBSO04] mit Komponenten, Ports und Konnektoren. Während Bradbury die Unterstützung von Hinzufügen und Löschen von Konnektoren und Komponenten in die Anforderungen aufnimmt, werden Ports nicht explizit betrachtet. Da die Mechatronic UML auch explizit Ports unterstützt, sollte eine Transformationssprache für die oben dargestellten erweiterten Mechatronic UML Modelle Ports genauso wie Komponenten und Konnektoren unterstützen.

Geeignete visuelle Darstellung Für eine gute Verständlichkeit der modellierten Transformationen ist es hilfreich, wenn die Syntax der Sprache zur Strukturtransformation so weit wie möglich an die Syntax der Strukturdefinition angelehnt ist. Dies führt dazu, dass es keinen Bruch zwischen den Modellierungssprachen gibt. Dies steht im Kontrast zu früheren verwandten Arbeit im Kontext der Mechatronic UML [Sch06, Kle08]. In diesen Arbeiten werden Storydiagramme auf Objektstrukturen zur Modellierung von Änderungen an Komponentenstrukturen genutzt. Des Weiteren ist es oftmals einfacher graph-basierte Strukturen (wie hierarchische Komponenteninstanzstrukturen) graphisch darzustellen, da hierbei direkt die Verknüpfung der Komponenten über Ports und Konnektoren erkennbar ist. Dies gilt im selben Maße für eine Transformationssprache solcher Strukturen, da hier schnell die Strukturänderungen erkennbar sind.

3.2.2 Komponentenstorydiagramme

Im Folgenden wird eine Modellierungssprache für die Transformation hierarchischer Komponenteninstanzstrukturen beschrieben, welche die obigen Anforderungen erfüllt.

Storydiagramme [FNTZ98, Zün02] erfüllen bereits einige der oben dargestellten Anforderungen, wie zusammengesetzte Änderungsoperationen und visuel-

le Darstellung. Storypattern können auf Metamodellebene auch zur Spezifikation von Transformationen von hierarchischen Komponenteninstanzstrukturen genutzt werden. Allerdings ergibt sich hierbei ein Bruch, da bei der Modellierung der Transformation im Vergleich zur Strukturmodellierung von Modell auf Metamodellebene gewechselt werden muss. Stattdessen wird für die Spezifikation einer einzelnen Transformation in dieser Arbeit die konkrete Syntax der Komponenteninstanzstrukturen genutzt und mit den Konzepten der Storypattern kombiniert.

Die Sprache basiert auf UML Aktivitätsdiagrammen für die Beschreibung des Ablaufs einzelner Aktivitäten. Neben einfachen sequentiellen Abläufen werden Entscheidungen und Schleifen unterstützt. Im Wesentlichen werden hier die bereits aus Storydiagrammen bekannten Syntaxelemente zur Definition des Kontrollflusses unterstützt. Durch die Nähe zu Storydiagrammen wird diese Sprache *Komponentenstorydiagramme* genannt.

Die in Abschnitt 3.1 definierten Komponententypen sind hierarchisch. Dies bedeutet, dass ein Komponententyp Parts eines anderen Komponententyps beinhaltet. Ein Komponententyp ist dabei für die Struktur der ihm unterlagerten Komponentenparts verantwortlich. Komponentenstorydiagramme werden daher in Bezug auf die Komponententypen definiert. Die zu einem Komponententyp gehörigen Komponentenstorydiagramme können nur auf den direkt in diesem Typ enthaltenen Parts arbeiten. Ein Überspringen der Hierarchien und ein direkter Zugriff auf die Strukturen innerhalb eines Parts sind hierbei aus Gründen der Kapselung nicht erlaubt. Verschiedene Komponentenstorydiagramme werden durch ihren Namen unterschieden.

In den einzelnen Aktivitäten eines Komponentenstorydiagramms wird graphisch die strukturelle Transformation einer Komponenteninstanzstruktur spezifiziert analog zu Storypattern, die Transformationen auf Objektstrukturen beschreiben. Diese Aktivitäten werden als *Komponentenstorypatterns* bezeichnet. Weitere Aktivitätsarten sind u.a. Start- und Stopaktivitäten.

Analog zu Kontrukturen und Methoden in gängigen objektorientierten Programmiersprachen werden *Konstruktoren* und *Rekonfigurationen* unterschieden. Erstere werden nach der Instantiierung eines Komponententyps aufgerufen, um die unterlagerte Komponenteninstanzstruktur zu erzeugen; letztere werden genutzt, um eine bereits existierende unterlagerte Komponenteninstanzstruktur anzupassen.

Die Semantik der Komponentenstorydiagramme wird durch eine Übersetzung in Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen formal definiert. Diese Übersetzung wird auch bei der automatischen Quelltextsynthese genutzt. Ergebnisse aus [Hol08] zeigen, dass die Anzahl der zu modellierenden Elemente in Komponentenstorydiagrammen um den Faktor 9 geringer ist,

als wenn direkt Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen spezifiziert werden. Es ergeben sich somit schneller zu erstellende und wartbarere Modelle.

3.2.2.1 Signatur eines Komponentenstorydiagramms

Analog zu Methoden in objektorientierten Programmiersprachen besitzt ein Komponentenstorydiagramm benannte Parameter, die entweder über die Komponentenstruktur oder über Basistypen wie z.B. Integer und String typisiert sind. Mögliche Typen sind hierbei die Ports des zum Komponentenstorydiagramm gehörigen Komponententyps sowie die Typen der dem zugehörigen Komponententyp unterlagerten Parts – also Komponenten-, Port- sowie Konnektorparts.

Ein Komponentenstorydiagramm kann ein Ergebnis zurückliefern. Erlaubte Ergebnisse sind Komponenten-, Port- und Konnektorinstanzen. Hierbei gilt die gleiche Einschränkung bzgl. der Typisierung wie bei Parametern.

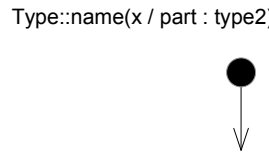


Abbildung 3.14: Signatur eines Komponentenstorydiagramms

Eine Signatur eines Komponentenstorydiagramms besteht also aus dem Namen des zugehörigen Komponententyps, dem Namen des Komponentenstorydiagramms, einer Menge von Parametern (Name und Typ) sowie dem Ergebnistyp. Abbildung 3.14 zeigt die Signatur des Komponentenstorydiagramms name des Komponententyps Type. Der Parameter x ist vom Typ part:type2, d.h. es dürfen nur Komponenteninstanzen des Komponententyps type2, die als Part part in den Komponententyp eingebettet sind, als Argument übergeben werden; der Rückgabeparameter result ist vom Typ part3:type3. Die Signatur wird graphisch wie bei Storydiagrammen an der Startaktivität dargestellt.

3.2.3 Komponentenstorypatterns

Ein Komponentenstorypattern beschreibt eine einzelne Strukturänderung auf der einer Komponenteninstanz unterlagerten Komponenteninstanzstruktur. Eine Strukturänderung ist hierbei das Entfernen oder Hinzufügen einer Komponenten-, Port-, oder Konnektorinstanz. Die Sprache für die Strukturänderung ist an Storypatterns angelehnt und für hierarchische Komponentenstrukturen angepasst.

Die Strukturänderung wird analog zu Storypatterns visuell mit Hilfe von erweiterten Komponenteninstanzstrukturen dargestellt. Die Elemente der Struktur, so genannte *Variablen*, können als zu löschen oder zu erzeugen markiert werden. Durch diese Markierung wird implizit die Situation vor und nach der Transformation beschrieben analog zur linken bzw. rechten Regelseite in Graphtransformationen.

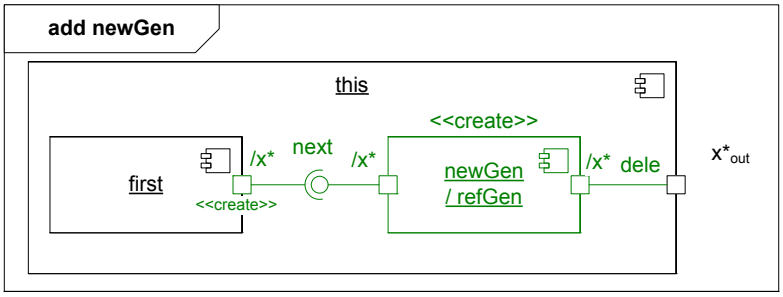


Abbildung 3.15: Beispiel eines Komponenstorypattern

Abbildung 3.15 zeigt das Komponentenstorypattern add newGen des RailCab-Komponententyps. Es ist Teil der Rekonfiguration, wenn ein neues RailCab dem Konvoi beitritt und Sollpositionsdaten benötigt. Das Komponentenstorypattern wird auf einer Komponenteninstanz des Typs RailCab aufgerufen. Diese Instanz wird durch die this-Komponentenvariable repräsentiert. Innerhalb dieser Instanz wird ausgehend von einer bereits gebundenen first-Komponentenvariable (vom Typ refGen:RefGen) eine neue RefGen-Komponenteninstanz mit dem Variablennamen newGen erzeugt und über den next-Kompositionskonnektor an first angefügt. Weiterhin wird eine Delegation zur Portvariable x^*_{out} erzeugt.

3.2.3.1 Variablen

Variablen sind Repräsentanten von Elementen der Komponenteninstanzstruktur. Sie werden bei der Ausführung eines Komponentenstorydiagramms an Elemente der Komponenteninstanzstruktur gebunden. Die Variablen eines Komponentenstorypatterns lassen sich wie bei Graphtransformationen üblich in zwei Mengen unterteilen:

- *Linke Regelseite*: Variablen, die vor Ausführung des Komponentenstorypatterns auf ein Element der Komponenteninstanzstruktur abgebildet werden.
- *Rechte Regelseite*: Variablen, die nach Ausführung des Komponentenstorypatterns auf ein Element der Komponenteninstanzstruktur abgebildet werden.

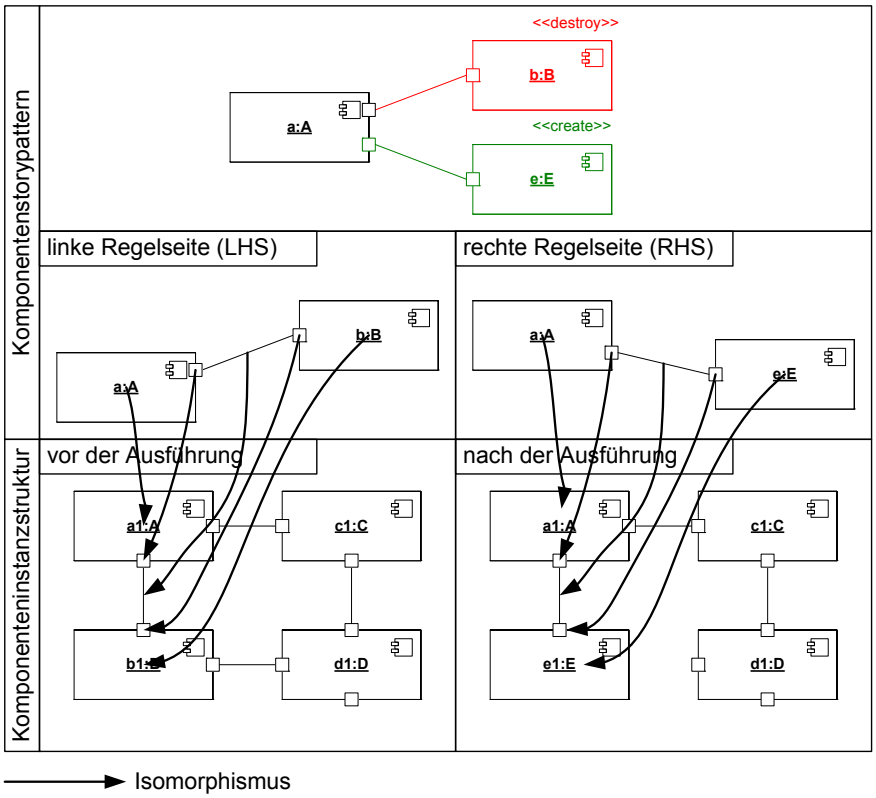


Abbildung 3.16: Linke und rechte Seite eines Komponentenstorypatterns und die Abbildung der Elemente des Patterns auf eine Komponenteninstanzstruktur, den Wirtsgraph

Wichtig ist, dass Variablen, die Teil beider Regelseiten sind, vor und nach der Ausführung des Komponentenstorypatterns auf das gleiche Element abgebildet werden. Analog zu Storypatterns werden die Variablen beider Mengen syntaktisch zusammen dargestellt. Variablen, die in der linken aber nicht in der rechten Regelseite vorkommen, also nach der Ausführung des Komponentenstorypatterns nicht mehr existieren, werden mit dem Stereotypen ≪destroy≫ versehen. Variablen, die in der rechten aber nicht in der linken Regelseite vorkommen, also durch die Ausführung des Komponentenstorypatterns entstehen, werden dementsprechend mit dem Stereotypen ≪create≫ annotiert. Variablen, die in beiden Mengen vorkommen, haben keinen Stereotypen. Zur einfacheren Lesbarkeit der Diagramme werden alle Variablen für zu erzeugende Elemente grün und für zu löschende Elemente rot dargestellt.

Ein Komponentenstorypattern wird ausgeführt, wenn ein Morphismus existiert, der alle Variablen der linken Regelseite auf Elemente der Komponenten-

instanzstruktur (auch Wirtsgraph genannt) abbildet und dabei die durch die Variablenstruktur vorgegebenen Zusammenhänge zwischen den Variablen auch für die Elemente gelten. So müssen eine Komponentenvariable und eine zugehörige Portvariable auf eine Komponenteninstanz und eine zugehörige Portinstanz abgebildet werden. Wie bei Storypatterns wird im speziellen ein Isomorphismus genutzt (im Gegensatz zum Beispiel zum Homomorphismus, der in GROOVE [Ren04] eingesetzt wird), der bestimmt, dass kein Element der Komponenteninstanzstruktur auf zwei unterschiedliche Variablen abgebildet werden darf.

Ist die Ausführung des Komponentenstorypatterns möglich, wird die Komponenteninstanzstruktur hinsichtlich der Löschungen und Hinzufügungen so angepasst, dass nach dieser Anpassung ein Isomorphismus zwischen den Variablen der rechten Regelseite des Komponentenstorypatterns auf die angepasste Komponenteninstanzstruktur existiert (s. Abbildung 3.16). Hierbei müssen die sich nicht ändernden Variablen des Komponentenstorypatterns vor und nach der Anpassung der Komponenteninstanzstruktur auf die gleichen Elemente der Komponenteninstanzstruktur abgebildet werden.

Bei der Ausführung der Komponentenstorydiagramme wird wie bei Storydiagrammen das Single-Pushout (SPO) [LE91] Prinzip bei Graphtransformationen verfolgt. Dies hat vor allem Auswirkung auf die Löschung von Instanzen. Wenn eine Komponenten- oder Portinstanz gelöscht wird, werden implizit alle über Kompositionsbeziehungen verbundene Port- oder Konnektorinstanzen ebenfalls gelöscht, so dass es nach der Ausführung eines Komponentenstorypatterns keine inkonsistente Struktur, wie zum Beispiel *dangling edges* in Form von unverbundenen Konnektoren, gibt. Der Konnektor zwischen b1 und d1 in Abbildung 3.16 ist ein Beispiel für eine solche implizit gelöschte Kante.

3.2.3.2 this-Variable

Elemente eines Komponentenstorypatterns sind die Variablen, die bei der Ausführung gebunden (linke Regelseite) und dann je nach Spezifikation verarbeitet werden.

Komponentenstorypatterns beschreiben die Strukturänderung der unterlagerten Elemente eines Komponententyps. Sie werden dementsprechend auf einer Instanz dieses Komponententyps aufgerufen. Die this-Variable repräsentiert diese Instanz. Jedes Komponentenstorypattern muss diese Variable enthalten. Alle weiteren Variablen des Komponentenstorypatterns sind entweder unterlagerte Elemente oder als Sonderfall Portvariablen der this-Variable. In Abbildung 3.15 ist die this-Variable vom Typ RailCab und enthält weitere Variablen für die refGen-Komponenteninstanzen. Des Weiteren enthält sie die x^*_{out}-Portvariable sowie die Variablen für die Konnektoren.

3.2.3.3 Weitere Variablen

Neben der this-Variable gibt es weitere Variablen für die unterschiedlichen Elemente einer Komponenteninstanzstruktur:

- *Komponentenvariablen* sind Repräsentanten für die der this-Variable unterlagerten Komponenteninstanzen. In Abbildung 3.15 sind first und newGen/refGen Komponentenvariablen.
- *Portvariablen* sind Repräsentanten entweder für Portinstanzen der unterlagerten Komponenteninstanzen oder für Portinstanzen der this-Variable. Die unbenannten Ports bzw. x^*_{out} sind Beispiele hiefür.
- Delegationskonnektoren werden mit *Delegationskonnektorvariablen* im Komponentenstorypattern repräsentiert. Die Verbindung zwischen der x^*_{out}-Portvariable und der (unbenannten) Portvariable der newGen/refGen-Variable in Abbildung 3.15 ist eine Delegationskonnektorvariable.
- *Kompositionskonnektorvariablen* ermöglichen die Betrachtung von Kompositionskonnektoren in einem Komponentenstorypattern, wie beispielsweise die next-Kompositionskonnektorvariable zwischen den Komponentenvariablen first und newGen/refGen in Abbildung 3.15.

3.2.3.4 Typisierung

In Abschnitt 3.2.1 wurde eine starke Typisierung der Beschreibungselemente eines Komponentenstorypatterns gefordert. Alle Variablen eines Komponentenstorypatterns werden zum Modellierungszeitpunkt über Parts und den Komponententyp des Komponentenstorydiagramms typisiert. Diese starke Typisierung ermöglicht zur Entwurfszeit im Zusammenhang mit einem syntaxgesteuerten kontextsensitiven Editor, Typisierungsfehler zu vermeiden bzw. automatisch zu erkennen. Zum Beispiel muss eine unterlagerte Komponentenvariable über einen Komponentenpart des Komponententyps der this-Variable typisiert sein, damit diese Komponentenvariable zur Typdefinition passt. Bei der Bindung der Variablen auf Instanzen während der Ausführung werden diese Typinformationen beachtet, so dass nur Instanzen an Variablen gebunden werden, die dem spezifizierten Variablentyp entsprechen.

Abbildung 3.17 zeigt die Typisierung einiger, aus Gründen der Übersichtlichkeit nicht aller, Variablen eines Komponentenstorypatterns in Bezug auf den zugehörigen Komponententyp. Die this-Variable ist vom Typ RailCab, die Variable newGen ist typisiert über den Part refGen innerhalb des Komponententyps

RailCab und die p2r-Kompositionskonnektorvariable ist typisiert über den Kompositionskonnektor p2r. Die Komponentenvariable ps ist schließlich typisiert über den Part posSensor.

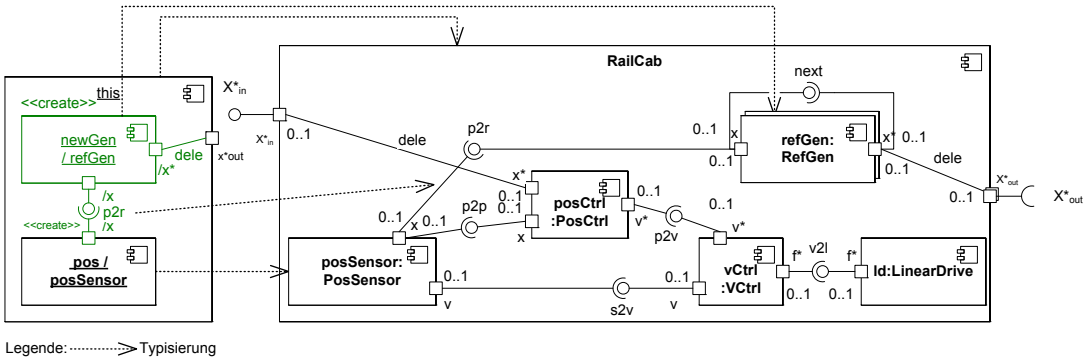


Abbildung 3.17: Typisierung der Variablen eines Komponentenstorypatterns in Bezug auf den zugehörigen Komponententyp

3.2.3.5 Bindung

Bei der Ausführung eines Komponentenstorypatterns werden die Variablen der linken Regelseite auf Elemente der Komponenteninstanzstruktur abgebildet. Dies wird *Bindung* genannt. Es werden alle Variablen der linken Regelseite mit Ausnahme der this-Variable gebunden, deren Bindung durch den Aufruf des Komponentenstorydiagramms bereits vorgegeben ist. Da die einzelnen Komponentenstorypatterns durch Kontrollfluss in einem Zusammenhang stehen, können die Bindungen von Variablen von einem Komponentenstorypattern in die folgenden Patterns durch die Markierung einer Variable als *gebunden* übernommen werden. Dies bedeutet, dass in einem vorherigen Komponentenstorypattern innerhalb eines Komponentenstorydiagramms bereits ein Element des Wirtsgraphen an diese Variable gebunden wurde und diese Bindung des Elements (nach einer Typprüfung) übernommen wird. Somit ist ein Teil der Abbildung der linken Regelseite auf den Wirtsgraphen bereits vorgegeben. Eine nicht gebundene Variable wird in der Form Name/Part:Typ dargestellt, wobei der Typ optional ist, da er durch die Angabe des Parts bereits automatisch feststeht. Bei einer gebundenen Variable wird nur der unterstrichene Name dargestellt.

3.2.3.6 Negative Variablen

In gewissen Situationen muss das Ausführen eines Komponentenstorypatterns vom nicht erfolgreichen Binden einer oder mehrerer Variablen abhängig gemacht

werden. Dies wird üblicherweise als *Negative Application Condition* (NAC) bezeichnet. Diese negative Anwendungsbedingung wird allerdings nicht als extra Diagramm modelliert, sondern es werden einzelne Variablen als *negativ* markiert. Graphisch werden solche Variablen durchgestrichen dargestellt (vgl. Abbildung 3.18). Das Komponentenstorypattern aus Abbildung 3.18 kann nur ausgeführt werden, wenn die an this gebundene Komponenteninstanz keine unterlagerte Komponenteninstanz vom Typ part:type hat.

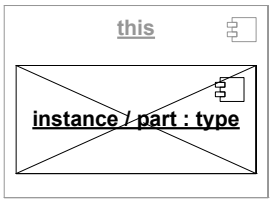


Abbildung 3.18: Beispiel für eine negative Komponentenvariable

3.2.3.7 Optionale Variablen

Variablen können als optional markiert werden. Dies bedeutet, dass diese Variablen auf ein Element der Komponenteninstanzstruktur abgebildet werden können, aber nicht müssen. Optionale Variablen werden mit einer gestrichelten Umrandung dargestellt. Die Variable instance/part:type des Komponentenstorypatterns aus Abbildung 3.19 ist optional. Das Komponentenstorypattern kann also ausgeführt werden, egal ob die an this gebundene Komponenteninstanz eine unterlagerte Komponenteninstanz vom Typ part:type hat oder nicht.

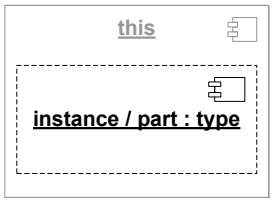


Abbildung 3.19: Beispiel für eine optionale Komponentenvariable

3.2.3.8 Auswirkung der Kompositionsbeziehung

In Abschnitt 3.1.3.1 wurden verschiedene Kompositionsbeziehungen zwischen Elementen beschrieben. Diese Kompositionsbeziehungen haben spezielle Auswir-

kungen bei der Ausführung eines Komponentenstorypatterns, wenn die Variablen zu erzeugen oder zu löschen bzw. negativ oder optional sind:

create Werden Variablen mit ≪create≫ markiert, dann bezieht sich dies nicht nur auf die Variable selber, sondern auf alle über Kompositionsbeziehungen verbundenen Variablen. Für eine zu erzeugende Komponenteninstanz werden dementsprechend auch alle Portvariablen an der Komponentenvariable als zu erzeugend markiert. Dies gilt analog und transitiv auch für Portvariablen und deren anliegende Kompositionskonnektor- und Delegationskonnektorvariablen.

destroy Dies gilt ebenfalls für Variablen, die mittels des Stereotypen ≪destroy≫ als zu löschend angegeben sind. Hier sind alle zu einer Variable in Kompositionsbeziehung stehenden Variablen ebenfalls implizit als zu löschend markiert.

Die Kompositionsbeziehung hat neben dieser impliziten Markierung der Variablen auch weitere Auswirkungen. Wenn die an eine zu löschende Variable gebundene Instanz bei der Ausführung gelöscht wird, werden auch alle über eine Kompositionsbeziehung verbundenen Instanzen gelöscht, da diese nicht alleine existieren können (analog zum SPO-Ansatz [LE91]). Dies gilt auch, wenn diese Instanzen nicht an Variablen im Komponentenstorypattern gebunden worden sind. Beim Löschen einer Komponenteninstanz werden daher auch alle zu dieser Komponenteninstanz gehörenden Portinstanzen und die zu den Portinstanzen gehörenden Kompositionskonnektoren und Delegationen gelöscht. Es werden ebenfalls alle unterlagerten Komponenteninstanzstrukturen der gelöschten Komponenteninstanz gelöscht.

negativ Ist eine Variable als negativ markiert, bildet sie mit den über eine Kompositionsbeziehung verbundenen Variablen eine negative Gruppe. Dies bedeutet, dass die negative Gruppe als Ganzes nicht gebunden werden darf.

optional Optionale Variablen werden mit den per Komposition verbundenen Variablen ebenfalls als Gruppe betrachtet. Eine optionale Gruppe wird nur dann erfolgreich gebunden, wenn alle Variablen der Gruppe als Ganzes gebunden werden können.

3.2.3.9 Attribute der Variablen

Komponententypen und Ports können Attribute besitzen (s. Abschnitt 3.1.1.3), die sie näher beschreiben. Die Werte dieser Attribute können nun in den Komponentenstorypatterns geprüft und gesetzt werden. Das Überprüfen geschieht beim

Binden der Variablen der linken Regelseite. Das Setzen wird bei der Ausführung des Komponentenstorypatterns also bei der Herstellung der rechten Regelseite durchgeführt. Syntaktisch wird dies durch die Angabe des Attributes, des Wertes und eines Operators (dargestellt in Abbildung 3.20) umgesetzt. Operatoren sind $=$, $\neq$, $<$, $\leq$, $>$, $\geq$ und $:=$. Die ersteren werden für die Überprüfung des Attributwertes, der Zuweisungsoperator $:=$ für das Setzen eines Wertes genutzt. Während $=$, $\neq$ und $:=$ für alle Typen der Attribute unterstützt werden, können $<$, $\leq$, $>$ und $\geq$ nur für Zahlen genutzt werden.

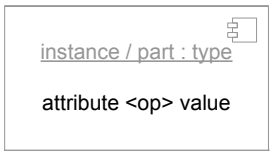


Abbildung 3.20: Beispiel für eine Attributbedingung einer Komponentenvariable

3.2.3.10 Aufrufe von Komponentenstorydiagrammen

Komponentenstorydiagramme können aus anderen Komponentenstorydiagrammen aufgerufen werden. Das dem zugrunde liegende Paradigma der Kapselung durch Unterprogramme [PZ97, S. 266] ermöglicht eine Modularisierung der Komponentenstorydiagramme und damit eine einfachere Wiederverwendung. Des Weiteren ist hierdurch die Modellierung von Transformationen über mehrere Hierarchieebenen möglich.

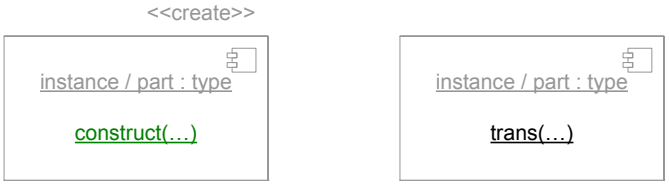


Abbildung 3.21: Beispiele für den Aufruf von Komponentenstorydiagrammen

Abbildung 3.21 zeigt zwei Aufrufe von Komponentenstorydiagrammen. Auf der linken Seite wird für eine zu erzeugende Komponenteninstanz ein Konstruktor aufgerufen. Nach der Erzeugung der Komponenteninstanz wird die initiale unterlagerte Komponenteninstanzstruktur der erzeugten Komponenteninstanz durch ein Komponentenstorydiagramm angelegt. Dies ermöglicht eine einfache Erzeugung einer hierarchischen Komponenteninstanzstruktur. Auf der rechten Seite

der Abbildung wird ein Komponentenstorydiagramm auf einer Komponenteninstanz aufgerufen, um eine Rekonfiguration dieser Komponenteninstanz durchzuführen.

Aufrufe von weiteren Komponentenstorydiagrammen werden am Ende eines Komponentenstorypatterns ausgeführt, also nachdem die rechte Regelseite hergestellt worden ist.

Komponentenstorydiagramme haben Parameter und einen Rückgabetyp (s. Abschnitt 3.2.2.1). Beim Aufruf eines Komponentenstorydiagramms werden die Argumente übergeben. Die im Komponentenstorypattern verwendeten Variablen sowie die in den vorherigen Aktivitäten gebundenen Variablen können als Argumente verwendet werden. Da das angegebene Komponentenstorydiagramm nach der Ausführung der Strukturänderung aufgerufen wird, sind Variablen, die zerstört werden, also nur in der rechten Regelseite vorhanden sind, von der Übergabe als Argument ausgeschlossen.

Rückgabewerte von aufgerufenen Komponentenstorydiagrammen dürfen nicht im gleichen Komponentenstorypattern zur Bindung genutzt werden, da der Aufruf am Ende des Komponentenstorypatterns ausgeführt wird. Sie stehen allerdings als gebundene Variablen ab der nächsten Aktivität zur Verfügung.

3.2.4 Iterierte Komponentenstorypatterns

Die Komponentenstorypatterns aus Abschnitt 3.2.3 ändern die Struktur einer passenden Anwendungsstelle im Wirtsgraphen. Durch die Integration von Kardinalitäten an den unterlagerten Parts eines Komponententypen besteht nun die Möglichkeit, dass es mehrere passende Anwendungsstellen für die Variablen eines Komponentenstorypatterns gibt. Die bisher erläuterten Komponentenstorypatterns werden auf eine beliebige passende Anwendungsstelle angewendet. Für den Fall, dass jedoch ein Komponentenstorypattern auf alle Anwendungsstellen angewendet werden soll, werden analog zu Storydiagrammen [Zün02, S. 28] *iterierte* Komponentenstorypatterns unterstützt. Diese iterierten Komponentenstorypatterns werden in einer Schleife auf alle Anwendungsstellen angewendet.

Abbildung 3.22 zeigt zwei Varianten eines iterierten Komponentenstorypatterns. Die obere Variante beschreibt den oben dargestellten Fall, dass ein Komponentenstorypattern für alle möglichen Anwendungsstellen ausgeführt wird. Es werden alle unterlagerten Komponenteninstanzen typisiert über part : type nacheinander gebunden und gelöscht. Danach wird die Aktivität über die Transition mit der Bedingung end verlassen.

Die untere Variante beschreibt eine Erweiterung der iterierten Komponentenstorypatterns, die es ermöglicht, nicht nur ein Komponentenstorypattern pro Anwendungsstelle auszuführen, sondern zusätzlich eine Sequenz von weiteren Ak-

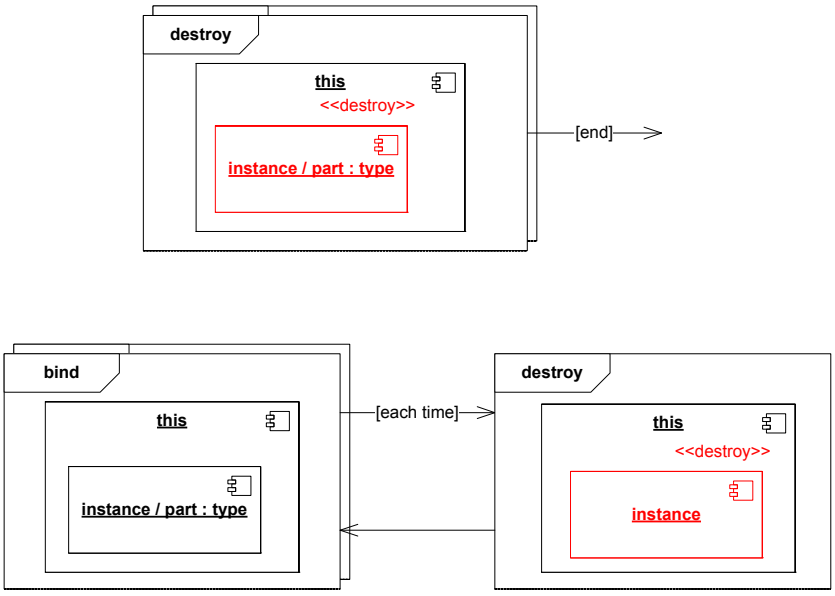


Abbildung 3.22: Beispiele für iterierte Komponentenstorypatterns

tivitäten. In diesem Beispiel wird im iterierten Komponentenstorypattern bind die Variable instance gebunden und danach die Aktivität destroy ausgeführt, die das an die Variable gebundene Element löscht. Für jede Anwendungsstelle des Komponentenstorypatterns bind werden die Aktivitäten, die durch die Transition mit der Bedingung each time verbunden sind, ausgeführt. Danach wird die Aktivität bind über die Transition mit der Bedingung end verlassen. Die beiden Varianten des Beispiels beschreiben also die selbe Rekonfiguration. Die zweite Variante erlaubt allerdings komplexere Operationen.

3.2.5 Start- und Stopaktivitäten

Start- und Stopaktivitäten sind die Aktivitäten mit denen ein Komponentenstorydiagramm beginnt bzw. endet. Wie bei Storydiagrammen hat ein Komponentenstorydiagramm genau eine Startaktivität, die keine Vorgänger hat. Ebenso können beliebig viele Stopaktivitäten existieren, die alle keinen Nachfolger haben. An der Startaktivität wird die Signatur des Komponentenstorydiagramms dargestellt (s. Abschnitt 3.2.2.1).

An Stopaktivitäten werden die Rückgabewerte des Komponentenstorydiagramms festgelegt. Dies geschieht durch die Angabe von Rückgabeparametern und der ausgewählten Variablen aus einem der Komponentenstorypatterns, deren gebundenes Element zurückgegeben werden soll.

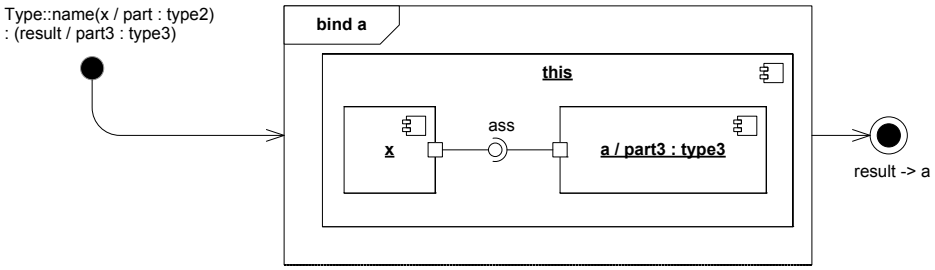


Abbildung 3.23: Beispiel für Start- und Stopaktivitäten mit Parametern und Rückgabewerten.

Abbildung 3.23 enthält ein einfaches Komponentenstorydiagramm mit einer Start- und einer Stopaktivität sowie einem Komponentenstorypattern. Die Startaktivität zeigt die Signatur des Komponentenstorydiagramms. Das Diagramm ist spezifiziert für den Komponententyp Type, hat einen Parameter x typisiert über part : type2 und einen Rückgabeparameter result typisiert über part3 : type3.

In der Aktivität bind a wird ausgehend von der this-Variable und der durch den Parameter bereits gebundenen Variable x die Variable a gebunden. Am Ende des Diagramms wird der Wert der Variablen a, also das Element im Wirtsgraphen, als Wert für den Rückgabeparameter result zurückgegeben.

3.2.6 Weitere Aktivitäten

Die Erfahrung bei der Anwendung von Graphtransformationen auf Objektstrukturen im Rahmen der Arbeiten an Story Driven Modeling [FNTZ98, Zün02] hat gezeigt, dass zusätzlich zu den oben dargestellten Komponentenstorypatterns auf der Komponenteninstanzstruktur die Einbettung von programmiersprachlichen Konstrukten, zum Beispiel zur Berechnung mathematischer Formeln, die Anwendbarkeit der Sprache erhöht [Zün02, S. 32]. Zu diesem Zweck können zusätzlich zu den Komponentenstorypatterns textuelle Aktivitäten modelliert werden. In diesen Aktivitäten werden programmiersprachliche Konstrukte der Zielplattform für die Ausführung der Komponentenstorydiagramme eingebettet, die bei der Übersetzung des Komponentenstorydiagramms in Quelltext direkt übernommen werden.

In [Opp06] wurde für Storydiagramme eine plattform- und programmiersprachenunabhängige Sprache für textuelle Aktivitäten in Fujaba beschrieben. Diese Arbeit könnte für Komponentenstorydiagramme angepasst und übernommen werden.

Für die Modellierung von Schleifen und Entscheidungen werden weitere Akti-

vitäten, so genannte Entscheidungs-/Vereinigungsaktivitäten, unterstützt.

3.2.7 Kontrollfluss

Der Kontrollfluss definiert die Reihenfolge der einzelnen Aktivitäten eines Komponentenstorydiagramms. Neben der einfachen Sequenz von Aktivitäten unterstützt er auch Entscheidungen und Schleifen. Die Spezifikation des Kontrollflusses von Komponentenstorydiagrammen ist identisch zu der bei Storydiagrammen, die wiederum auf UML Aktivitätsdiagrammen basiert.

Der Kontrollfluss verbindet die Aktivitäten eines Komponentenstorydiagramms durch gerichtete Kontrollflusskanten (Transitionen). Es ergibt sich somit ein gerichteter Graph. Der Graph hat mit der Startaktivität genau einen Knoten, der keine Vorgängerknoten hat. Er ist von diesem Knoten aus schwach zusammenhängend. Knoten ohne Nachfolger sind Stopaktivitäten.

Zwischen Start- und Stopaktivitäten dürfen die oben vorgestellten Aktivitäten genutzt werden. Hierbei dürfen allerdings, wie in Storydiagrammen, unterschiedliche Schleifen bzw. Entscheidungen sich nicht überschneiden. Dies wird in [Zün02] als *well-formedness* Eigenschaft beschrieben. Diese Einschränkung ermöglicht eine einfache und direkte Übersetzung des Kontrollflusses der Komponentenstorydiagramme auf Storydiagramme (siehe Abschnitt 3.2.10) auf dem Metamodell der Komponenteninstanzstrukturen. Die für den Kontrollfluss von Storydiagrammen definierte Graphgrammatik [Kle99, S. 36] gilt auch für Komponentenstorydiagramme.

Für die Modellierung von Entscheidungen und Schleifen in Komponentenstorydiagrammen müssen deren Bedingungen beschrieben werden. Komponentenstorydiagramme übernehmen hier die aus Storydiagrammen bereits bekannten Konstrukte. Abbildung 3.24 zeigt eine Übersicht der Transitionsbedingungen, die im Folgenden erläutert werden.

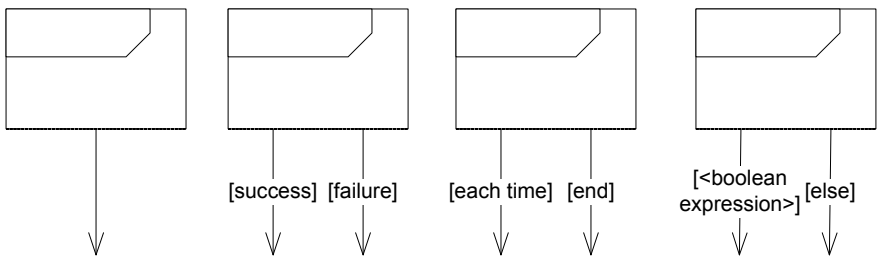


Abbildung 3.24: Übersicht der Transitionsbedingungen.

success / failure Bei der Ausführung eines Komponentenstorypatterns wird, wie in Abschnitt 3.2.3.1 erläutert, zuerst versucht, die linke Regelseite zu binden. Wenn dies erfolgreich war und nur dann, wird die rechte Regelseite hergestellt, indem die Änderungen durchgeführt werden. Um je nach dem, ob das Pattern erfolgreich ausgeführt wurde oder nicht, unterschiedlich zu reagieren, besteht die Möglichkeit, eine Aktivität über Transitionen mit den Bedingungen success bzw. failure zu verlassen. Wenn das Komponentenstorypattern erfolgreich ausgeführt wurde, wird die Transition mit der Bedingung success gewählt. Im anderen Fall wird die failure-Transition ausgewählt. Beide Transitionsbedingungen können nur zusammen genutzt werden.

each time / end Für die in Abschnitt 3.2.4 dargestellten iterierten Komponentenstorypatterns existieren die Bedingungen each time und end. Die Transition each time wird für jede Anwendungsstelle des iterierten Komponentenstorypatterns gewählt. Wenn alle Anwendungsstellen abgearbeitet wurden, wird das iterierte Komponentenstorypattern über die Transition end verlassen. An einem iterierten Komponentenstorypattern muss eine ausgehende Transition mit der Bedingung end existieren. Die each time-Transition ist optional.

boolesche Bedingungen / else Die textuellen Aktivitäten aus Abschnitt 3.2.6 ermöglichen die Nutzung programmiersprachlicher Konstrukte der Zielplattform z.B. für die Berechnung mathematischer Formeln. Dies ist oft auch bei der Definition der Bedingungen von Entscheidungen und Schleifen hilfreich. Daher werden für Transitionen auch Bedingungen in der Programmiersprache der Zielplattform unterstützt, die bei der Übersetzung auf Quelltext direkt übernommen werden. Ergänzend zu diesen booleschen Bedingungen existiert die Möglichkeit, eine Transition mit der Bedingung else zu versehen. Diese Transition wird dann gewählt, wenn die booleschen Bedingungen der übrigen Transitionen nicht erfüllt sind.

3.2.8 Beispiel

Abbildung 3.25 zeigt ein Komponentenstorydiagramm, welches im Rahmen des Beispiels die Struktur des führenden RailCabs anpasst, wenn ein neues Fahrzeug dem Konvoi beitritt. Hierbei wird an der richtigen Stelle in der Kette von RefGen-Komponenteninstanzen eine neue Komponenteninstanz für das zusätzliche Fahrzeug eingefügt. Als Parameter erhält das Komponentenstorydiagramm den Namen der zu erzeugenden Komponenteninstanz, die Position des neuen

Fahrzeugs im Konvoi sowie die Portinstanz, über welche die berechneten Positionssolldaten an das neue Fahrzeug geschickt werden sollen.

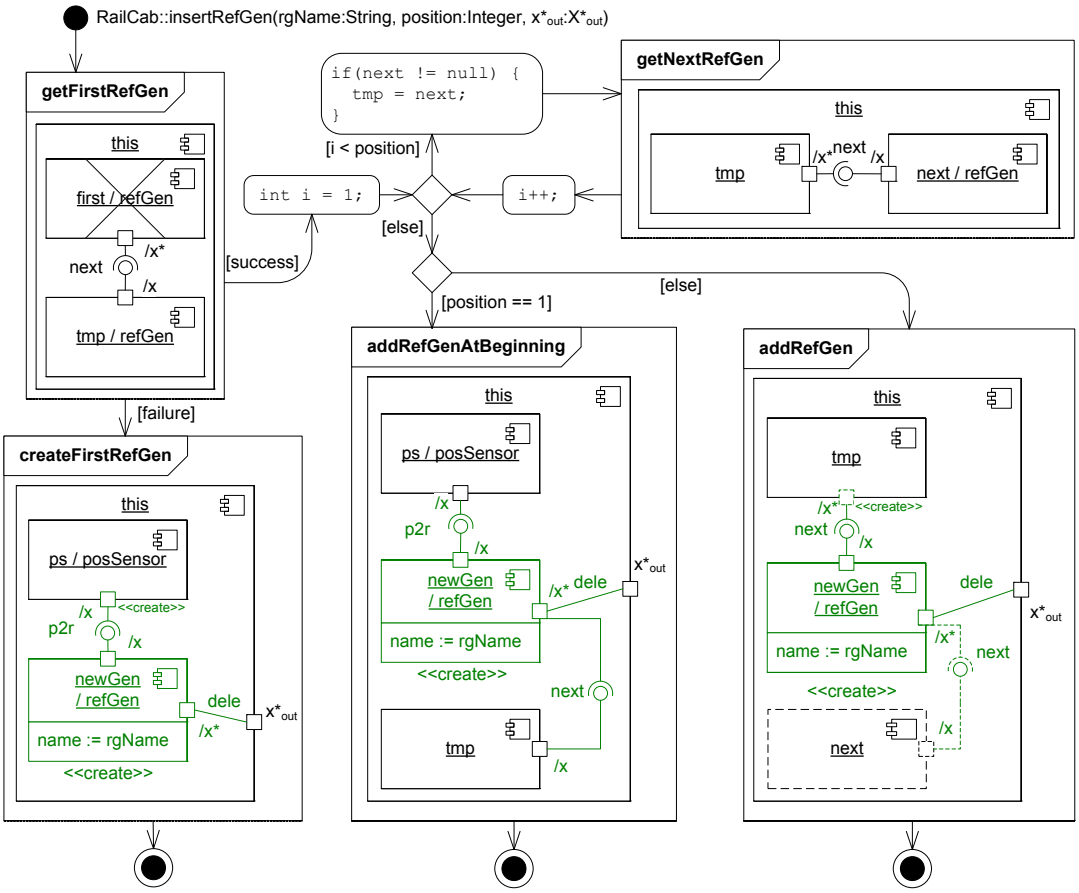


Abbildung 3.25: Komponentenstorydiagramm RailCab::insertRefGen

In den oberen Aktivitäten des Komponentenstorydiagramms wird die korrekte Position der zu erzeugenden RefGen-Komponenteninstanz gesucht. In den unteren Aktivitäten wird die Komponenteninstanz erzeugt und mit der übergebenen Portinstanz sowie den restlichen Komponenteninstanzen verbunden.

Die Suche nach der korrekten Position beginnt in der Aktivität getFirstRefGen. Hier wird die RefGen-Komponenteninstanz gesucht, die keinen Vorgänger hat. Falls dies nicht erfolgreich ist, gibt es noch keine RefGen-Komponenteninstanz und eine neue wird in der Aktivität createFirstRefGen erzeugt. Der Fall dass die Aktivität getFirstRefGen fehlschlägt, da zwar bereits RefGen-Komponenteninstanzen existieren, diese aber in einem Kreis miteinander verbunden sind, wird nicht

betrachtet, da diese Instanzstruktur nicht durch die modellierten Komponentenstorydiagramme erzeugt wird.

Falls die erste RefGen-Komponenteninstanz gefunden wurde, wird nun in der folgenden Schleife über die Kette der RefGen-Komponenteninstanzen iteriert, bis die Stelle erreicht wurde, an der die neue Komponenteninstanz festgelegt durch den Parameter position erzeugt und in die Struktur eingefügt werden soll. Dies wird in den Aktivitäten addRefGenAtBeginning sowie addRefGen durchgeführt.

3.2.9 Abstrakte Syntax

Die Syntax der Komponentenstorydiagramme ist definiert durch ein Metamodell und zugehörige Invarianten, die in OCL beschrieben wurden. Abbildung 3.26 zeigt die verschiedenen Pakete, in die das Metamodell der Komponentenstorydiagramme aus Gründen der Übersichtlichkeit aufgesplittet wurde. Das Paket ControlFlow enthält die Metaklassen für die Signatur der Komponentenstorydiagramme sowie für die Kontrollflusselemente. Das Paket Component Story Patterns enthält die Metaklassen für die Modellierung der Komponentenstorypatterns. Beide Pakete greifen auf das Paket Structure aus Abbildung 3.10 zu, in dem die Metaklassen zur Definition der Komponententypen enthalten sind. Des Weiteren greift das Metamodell auf diverse Klassen aus dem Fujaba-Metamodell zu, insbesondere die dort bereits enthaltenen Metaklassen der Storydiagramme. Bidirektionale Verbindungen zu diesen Klassen sind technisch über das Metamodell-Erweiterungs-Muster realisiert [BGN+03, BGN+04].

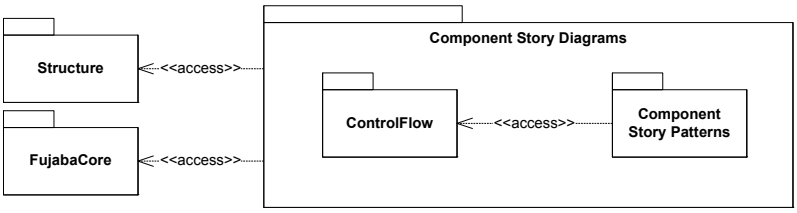


Abbildung 3.26: Pakete der Metamodelle der Komponentenstorydiagramme

Der größte Teil des Kontrollfluss-Metamodells für die Komponentenstorydiagramme in Abbildung 3.27 enthält die Definition der Signatur von Komponentenstorydiagrammen. Ein Komponententyp kann eine beliebige Anzahl an Komponentenstorydiagrammen, definiert durch die Metaklasse ComponentActivityDiagram, haben. Die Metaklassen Constructor und Reconfiguration werden für die Modellierung der entsprechenden Arten von Komponentenstorydiagrammen genutzt. Die Parameterliste eines Komponentenstorydiagramms ist realisiert über die geordnete Assoziation zur Klasse Parameter. Die Eigenschaft der Ordnung

(dargestellt durch den Stereotypen {ordered}) der Assoziation garantiert die gleichbleibende Reihenfolge der Parameter. Parameter, die über die Komponentenstruktur typisiert sind, werden durch die Metaklasse ComplexParam realisiert, während Parameter von Basistypen durch die Metaklasse PrimitiveParam umgesetzt werden. Die Kontrollflusselemente werden direkt von den Fujaba Storydiagrammen übernommen.

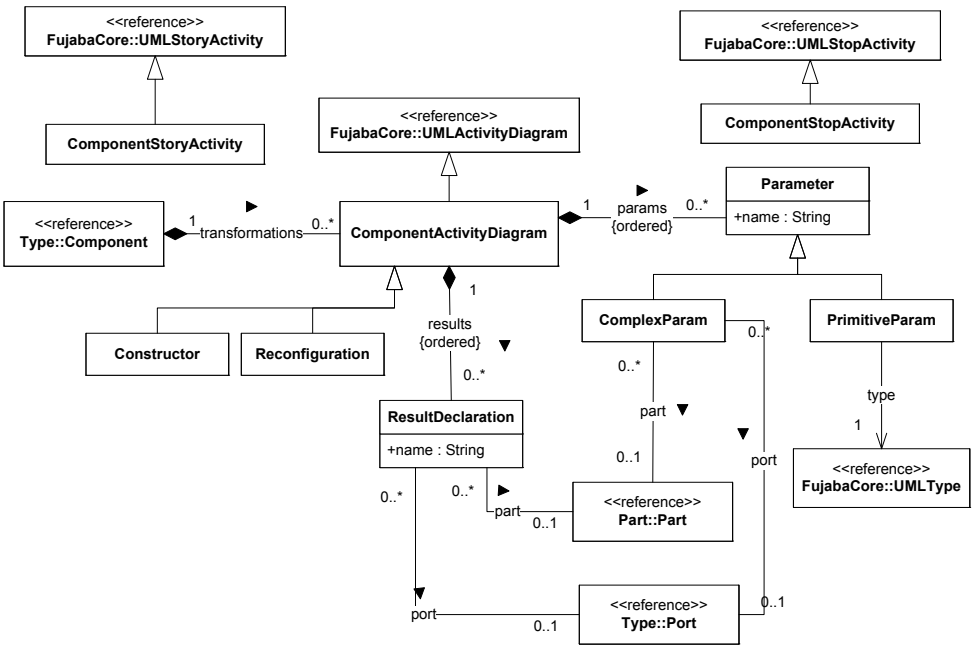


Abbildung 3.27: Metamodell der Komponentenstorydiagramme – Kontrollfluss

Es gelten zusätzlichen die folgenden Invarianten:

Invariante 3.11 *Parameter und Ergebnistypen eines Komponentenstorydiagramms dürfen nur entweder über die unterlagerten Parts eines Komponententyps oder einen Port des Komponententyps typisiert sein.*

context ComplexParam **inv**:
params.transformations.parts→**exists**(p | p = part) **xor**
params.transformations.ports→**exists**(p | p = port)

context ResultDeclaration **inv**:
results.transformations.parts→**exists**(p | p = part) **xor**
results.transformations.ports→**exists**(p | p = port)

Abbildung 3.28 zeigt den Teil des Metamodells für Komponentenstorypatterns. Die Variablen eines Patterns werden durch die abstrakten Metaklassen Variable und ConnectorVariable sowie die Subklassen ComponentVariable, PortVariable, DelegationVariable sowie AssemblyVariable realisiert. Variablen sind enthalten in einem ComponentStoryPattern und typisiert über die Metaklasse Part. Die Metaklasse Modifier enthält die Aufzählung der unterschiedlichen Modifizierer von Variablen. Die this-Variable wird dargestellt durch die Metaklasse ComponentVariable und die Assoziation thisvariable zur Metaklasse ComponentStoryActivity. Der Typ der this-Variable wird definiert durch die Assoziation zur Metaklasse Type.

Die Metaklasse AttrExprPair wird für die Attributbedingungen genutzt und verbindet daher einen Ausdruck (Klassenattribut expr) mit der Metaklasse Attribute sowie der Metaklasse Operator.

Im unteren Teil der Abbildung sind die Aufrufe anderer Komponentenstorydiagramme durch die Metaklasse Call modelliert. Das aufzurufende Komponentenstorydiagramm ist durch die Assoziation zur Metaklasse ComponentActivityDiagram abgebildet. Als Argumente, Metaklasse Argument, und Rückgabewerte, Metaklasse Result, werden die Variablen des Patterns über die Assoziationen variable und value genutzt.

Es gelten zusätzlich die folgenden Invarianten:

Invariante 3.12 *Zuweisungsoperatoren dürfen nicht in Attributbedingungen verwendet werden, die zu einer negativen oder zu löschenden Variablen gehören.*

context AttrExprPair **inv**:
  operator = Operator::assignment **implies**
    ((variable.modifier <> Modifier::negative)
    and (variable.modifier <> Modifier::destroy))

Invariante 3.13 *Vergleichsoperatoren dürfen nicht in Attributbedingungen verwendet werden, die zu erzeugenden Variablen gehören.*

context AttrExprPair **inv**:
  operator <> Operator::assignment **implies**
    variable.modifier <> Modifier::create

3.2.10 Formale Semantik

Die zuvor informal beschriebene Sprache zur Transformation von Komponenteninstanzstrukturen hat mit dem dargestellten Metamodell eine formal definierte

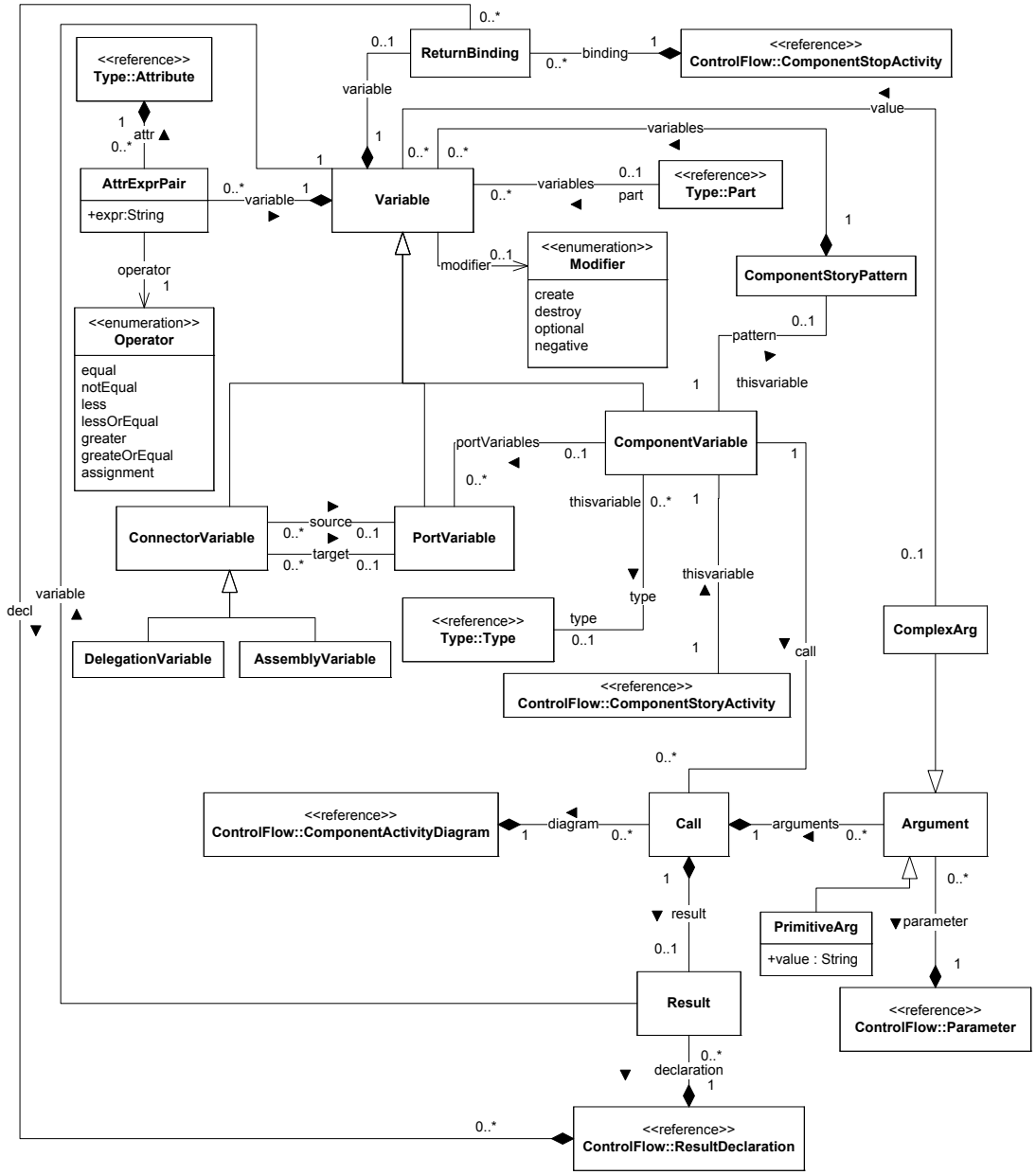


Abbildung 3.28: Metamodell der Komponentenstorydiagramm – Komponentenstorypattern

Syntax. Die informal beschriebene Semantik der Sprache muss für eine Ausführung der Komponentenstorydiagramme formal definiert werden. Im Rahmen dieser Arbeit wird die Ausführung der Komponentenstorydiagramme auf den in Abschnitt 3.1.3 definierten Komponenteninstanzstrukturen betrachtet. Die Ausführung der Komponentenstorydiagramme wird hierbei durch eine Übersetzung der Komponentenstorydiagramme auf Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen und die anschließende Ausführung dieser Storydiagramme realisiert. Die Semantik der Komponentenstorydiagramme wird somit durch diese Übersetzung (translational semantics [SK94]) formal definiert.

Abbildung 3.29 zeigt die Übersetzung an einem vereinfachten Beispiel. Auf der linken Seite sind die Komponenteninstanzstrukturen sowie ein Komponentenstorypattern auf Modellebene in konkreter Syntax dargestellt. Die rechte Seite zeigt dieselben Komponenteninstanzstrukturen (aus Platzgründen ohne die auf der linken Seite ausgegrauten Elemente) auf Metamodellebene also in abstrakter Syntax. Das Komponentenstorypattern in der Mitte wird als Storypattern auf dem Metamodell der Komponenteninstanzen modelliert. Dieses Storypattern transformiert die oben abgebildete Objektstruktur der Komponenteninstanzstruktur in die unten abgebildete Objektstruktur. Das Storypattern führt also das links in der Mitte abgebildete Komponentenstorypattern aus.

Ziel der im Folgenden beschriebenen Regeln ist die Übersetzung des links in der Mitte dargestellten Komponentenstorypattern auf das rechts dargestellte Storypattern sowie die Übersetzung der Syntaxelemente von Komponentenstorydiagrammen wie Transitionen zwischen den Aktivitäten auf entsprechende Elemente der Storydiagramme. Die Übersetzungsregeln werden aus Gründen der besseren Verständlichkeit in konkreter Syntax dargestellt. Technisch sind diese Regeln überwiegend durch Triple-Graph-Grammatiken (TGG) [Sch94, WGN03, KW07] sowie zu einem kleinen Teil durch Storydiagramme spezifiziert. Beide Übersetzungen werden auf dem Metamodell der Komponentenstorydiagramme sowie der Storydiagramme, also auf der abstrakten Syntax, spezifiziert [Hol08].[2]

3.2.10.1 Kontrollfluss

Auf der linken Seite der folgenden Übersetzungsregeln ist die konkrete Syntax der Komponentenstorydiagramme dargestellt; auf der rechten Seite sind die entsprechenden Storydiagramme ebenfalls in konkreter Syntax dargestellt. Der Pfeil von links nach rechts symbolisiert die Übersetzung. Auf beiden Seiten sind die Elemente entweder farbig, schwarz oder grau markiert. Farbig bzw. schwarz werden die für die Übersetzung relevanten Elemente dargestellt, während die grauen

[2] Um die Übersetzungsregeln mittels TGGs umsetzen zu können, wurden kleinere Erweiterungen an den Storydiagrammen vorgenommen, die in Anhang A näher erläutert werden.

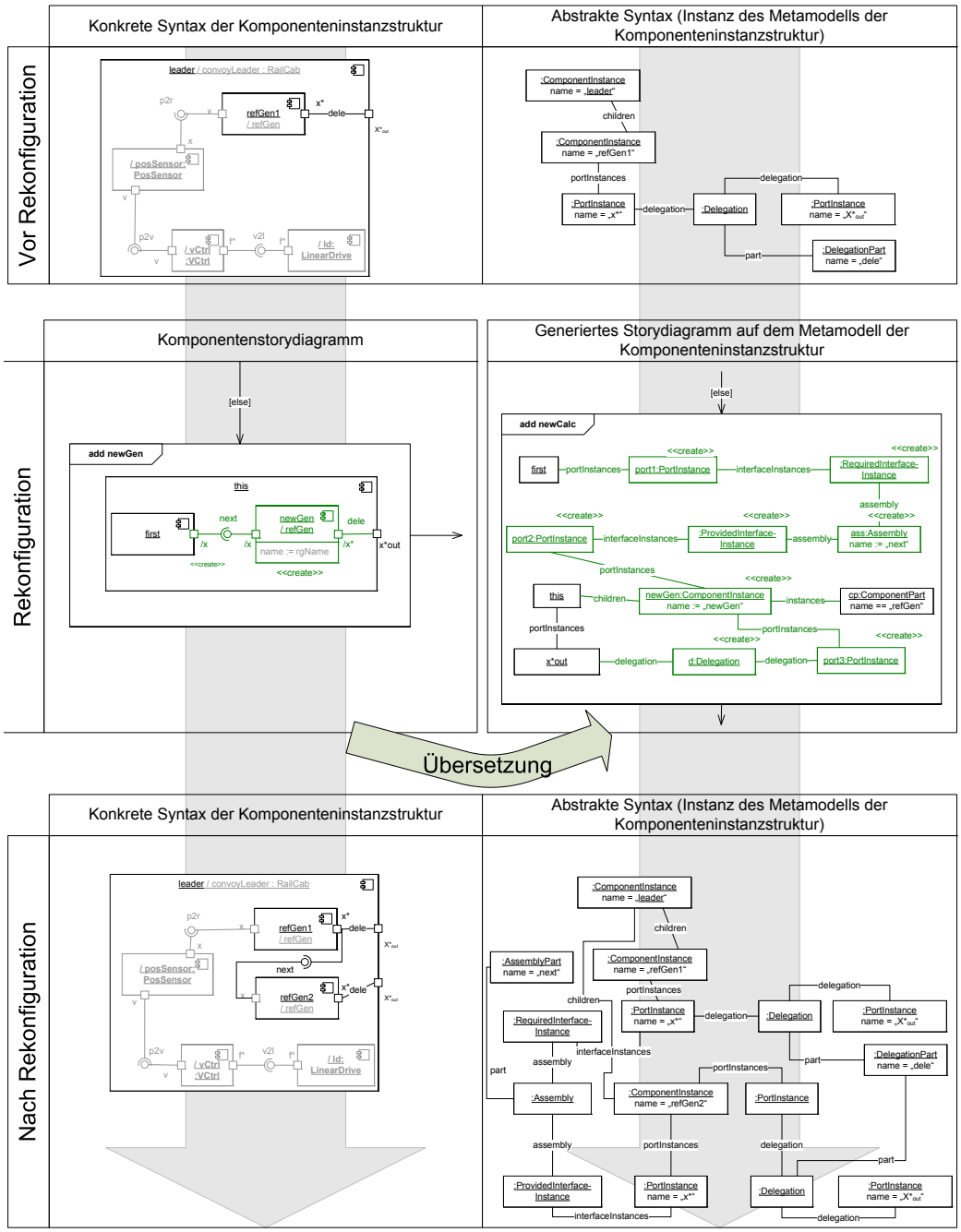


Abbildung 3.29: Übersetzung der Komponentenstorydiagramme auf Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen.

Elemente den Kontext beschreiben. Die Übersetzung der Kontextelemente wird in anderen Übersetzungsregeln dargestellt.

Der Kontrollfluss der Komponentenstorydiagramme, bestehend aus den Aktivitäten und den Kanten zwischen den Aktivitäten, wird in den meisten Fällen identisch auf den Kontrollfluss der Storydiagramme übersetzt. Abbildung 3.30 zeigt die direkte Übersetzung einer (beliebigen) Aktivität. Die nächste Abbildung zeigt die Übersetzung der Transitionskanten. Die an einer Transition annotierten Transitionsbedingungen werden ebenfalls direkt übernommen.

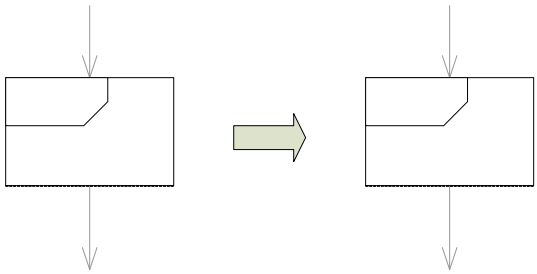


Abbildung 3.30: Übersetzung einer Aktivität

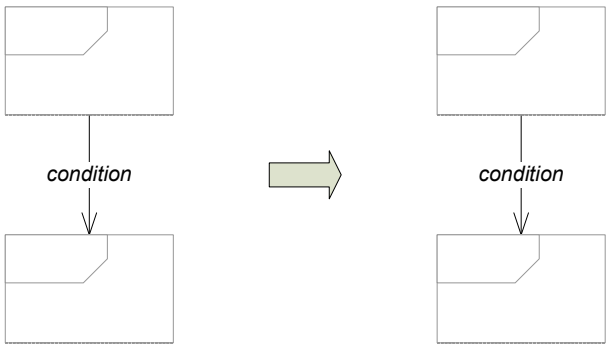


Abbildung 3.31: Übersetzung einer Transition

Die Startaktivität eines Komponentenstorydiagramms mit dem Namen name einer Komponente mit dem Namen Type wird übersetzt auf eine Startaktivität eines erweiterten Storydiagramms mit dem Namen Type_name (s. Abbildung 3.32). Dieses erweiterte Storydiagramm hat einen Parameter vom Typ ComponentInstance mit dem Namen self. Dieser Parameter wird genutzt, um der this-Variable eines Komponentenstorydiagramms die Komponenteninstanz als Wert zuzuweisen, auf der das Komponentenstorydiagramm aufgerufen wird.

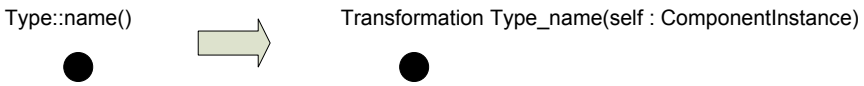


Abbildung 3.32: Übersetzung einer Startaktivität

Abbildung 3.33 zeigt die Übersetzung eines Parameters x typisiert über part:type am Beispiel einer Komponenteninstanz. Der Parameter wird mit dem gleichen Namen x und dem Typ ComponentInstance als Parameter der Startaktivität des erweiterten Storydiagramms übernommen. Portinstanzen, Kompositions- und Delegationskonnektoren werden als Parameter vom Typ PortInstance, Assembly bzw. Delegation übernommen. Die Übersetzung der Rückgabeparameter geschieht analog zur Übersetzung der Parameter.

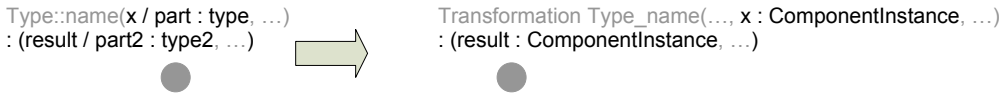


Abbildung 3.33: Übersetzung eines Parameters

Stopaktivitäten der Komponentenstorydiagramme werden in Stopaktivitäten der erweiterten Storydiagramme übersetzt (s. Abbildung 3.34). An Stopaktivitäten werden die Rückgabewerte der Komponentenstorydiagramme angegeben. Rückgabewerte sind Variablen aus den Aktivitäten der Komponentenstorydiagramme. Bei der Übersetzung dieser Rückgabewerte wird der Name des Rückgabeparameters, der sich aus der Signatur des Komponentenstorydiagramms bereits ergibt, übernommen. Eine Variable eines Komponentenstorypatterns wird nach Abbildung 3.39 auf ein Objekt eines Storydiagramms mit dem gleichen Namen wie die Variable übersetzt. Da die Namen der Variable und des Objektes identisch sind, wird als Rückgabewert der Stopaktivität des Storydiagramms der Name des Objekts genutzt.

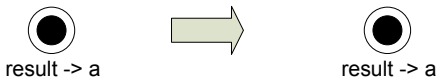


Abbildung 3.34: Übersetzung einer Stopaktivität

Entscheidungs-/Vereinigungsaktivitäten zur Modellierung von Entscheidungen und Schleifen von Komponentenstorydiagrammen werden direkt auf ihre Entsprechung in den erweiterten Storydiagrammen übersetzt (s. Abbildung 3.35).

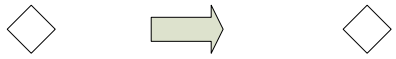


Abbildung 3.35: Übersetzung einer Entscheidungs-/Vereinigungsaktivität

Iterierte Komponentenstorypatterns werden identisch in die erweiterten Storydiagrammen übersetzt (s. Abbildung 3.36).

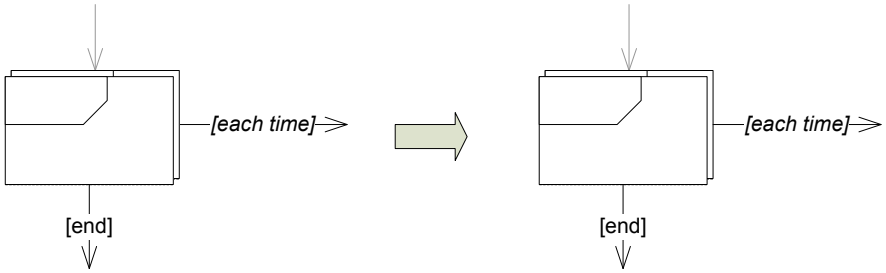


Abbildung 3.36: Übersetzung eines iterierten Komponentenstorypatterns

3.2.10.2 Komponentenstorypatterns

Der this-Variable wird bei der Ausführung eines Komponentenstorypatterns die Komponenteninstanz zugewiesen, auf der das Komponentenstorydiagramm ausgeführt wird. Nach Abbildung 3.32 wird bei der Übersetzung der Startaktivität der Parameter self:ComponentInstance hinzugefügt, der diese Komponenteninstanz repräsentiert. In einer Aktivität wird die this-Variable nun auf ein Objekt mit dem Namen self vom Typ ComponentInstance übersetzt. Dieses Objekt wird als gebunden markiert, da der Wert des Objekts bereits durch den Parameter self festgelegt ist und daher nicht neu gebunden werden muss. Zusätzlich wird ein Objekt für den zugehörigen Typ der Komponenteninstanz gebunden, welches mit dem self-Objekt über die type-Assoziation verbunden ist.

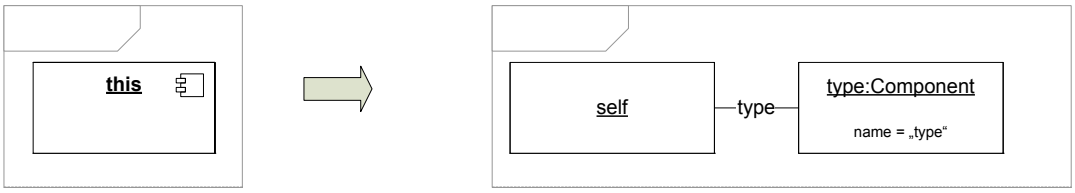


Abbildung 3.37: Übersetzung der this-Komponentenvariable

Eine nicht gebundene Variable wird, wie in Abbildung 3.38 beispielhaft eine Komponentenvariable, auf mehrere Objekte eines Storypatterns übersetzt. Dies sind ein Objekt für das Element der Komponenteninstanzstruktur selber (Objekt instance) sowie zusätzliche Objekte für den zugehörigen Part (Objekt part) und den Typ (Objekt type). Der Part und der Typ sind mit Attributbedingungen versehen, die den Namen des Parts und des Typs bzgl. der bei der Variable angegebenen Namen (im Beispiel part und type) überprüfen. Diese zusätzlichen Objekte sind notwendig, da nicht irgendeine Komponenteninstanz gebunden werden soll, sondern eine vom gewünschten Part und Typ.

Die zu bindende Komponenteninstanz muss über die children-Assoziation mit dem self-Objekt verbunden sein, da die Variable in die this-Variable eingebettet ist. Des Weiteren muss der zu bindende Part im Komponententyp der this-Variable enthalten sein. Falls diese Bedingung nicht überprüft würde, wäre es möglich, dass ein Part gebunden wird, der Teil eines anderen Komponententyps ist.

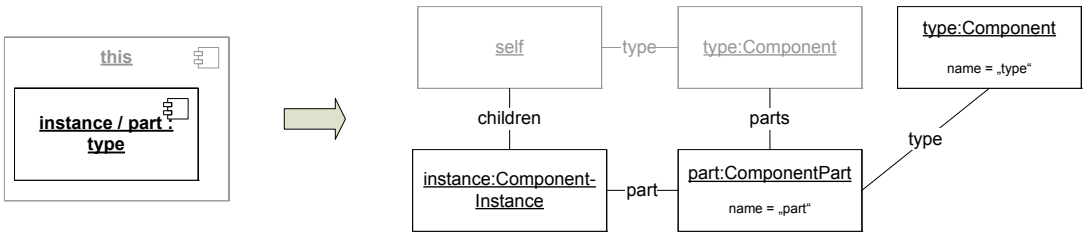


Abbildung 3.38: Übersetzung einer nicht gebundenen Komponentenvariable

Die Übersetzung (s. Abbildung 3.39) einer gebundenen Variable ist überwiegend analog zur oben beschrieben Übersetzung einer ungebundenen Variable. Im Unterschied zu oben wird das Objekt instance im Storypattern wie die Variable ebenfalls als gebunden markiert.

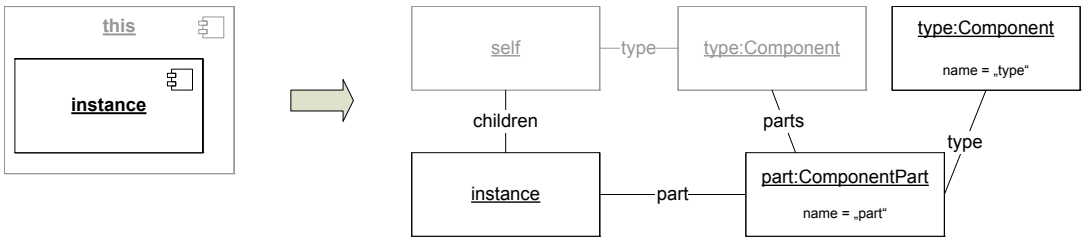


Abbildung 3.39: Übersetzung einer gebundenen Komponentenvariable

Abbildung 3.40 zeigt die Übersetzung einer Portvariablen in das zugehörige

Storypattern. Die Portvariable port vom Part part und Typ porttype gehört zur (gebundenen) Komponentenvariable instance. Die Übersetzung von instance nach der oben beschriebenen Übersetzungsregel ist in grau dargestellt. Für die Portvariable ist im Storypattern ein Objekt vom Typ PortInstance sowie Objekte für den Part (part) und den Typ (typeP) der Portvariablen mit entsprechenden Attributbedingungen enthalten. Die neuen Objekte sind mit den Objekten für die instance-Variable verbunden. So muss die gesuchte Portinstanz zur Komponenteninstanz instance gehören, der Portpart Teil der hierarchischen Definition des Komponententyps der self-Komponenteninstanz sein und der Port zum Komponententyp der instance-Komponenteninstanz gehören.

Eine gebundene Portvariable wird fast identisch auf eine Objektstruktur im Storypattern übersetzt. Einzig die Markierung als gebundenes Objekt und die nicht notwendigen Attributbedingungen machen den Unterschied aus.

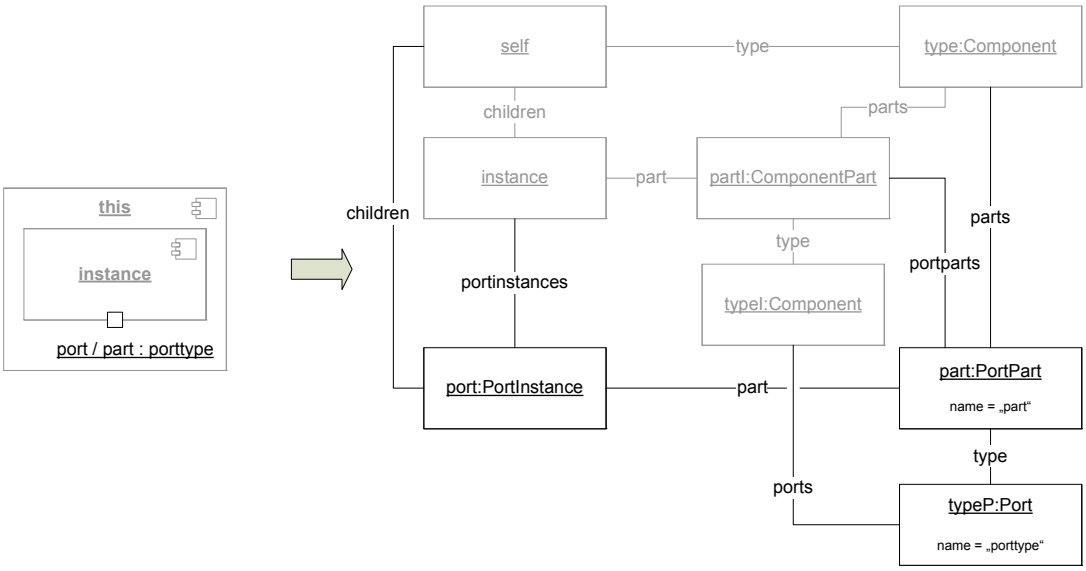


Abbildung 3.40: Übersetzung einer nicht gebundenen Portvariablen

Eine Portvariable kann neben dem eben dargestellten Fall, dass sie zu einer eingebetteten Komponentenvariable gehört, auch direkt an der this-Variable angegeben werden. In diesem Fall muss bei der Übersetzung in das Storypattern auf das Part-Objekt verzichtet werden. Dies liegt daran, dass die Portinstanz nicht in die Komponenteninstanz eingebettet ist, welches die Vorrausetzung für eine Bindung über einen Part ist.

Stattdessen wird die Portvariable direkt in ein Objekt vom Typ PortInstance sowie ein Objekt vom Typ Port übersetzt unter der Bedingung, dass das Port-Objekt

zum Komponententyp des self-Objekts gehört. Damit die gewünschte Portinstanz gebunden wird, erhält das Port-Objekt zusätzlich eine Attributbedingung über den gewünschten Namen „porttype“.

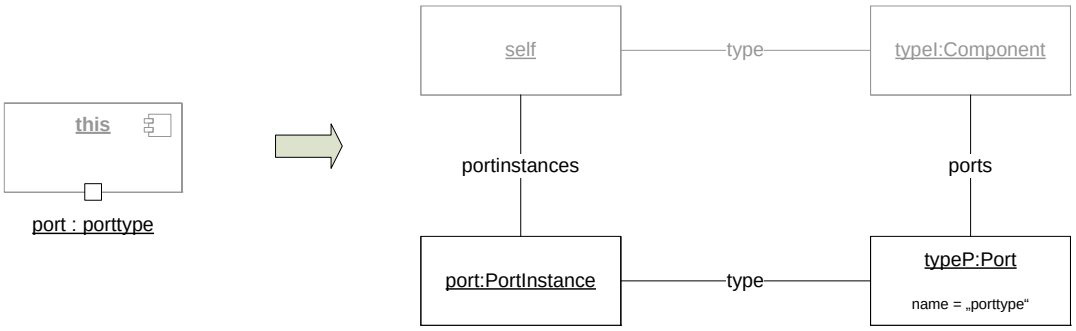


Abbildung 3.41: Übersetzung einer Portvariablen an der this-Variablen

Abbildung 3.42 zeigt die Übersetzung einer Kompositionskonnektorvariable. Ein Kompositionskonnektor verbindet in einer Komponenteninstanzstruktur eine benötigte und eine angebotene Schnittstelleninstanz. Dementsprechend wird eine Kompositionskonnektorvariable auf ein Objekt ass vom Typ Assembly, ein Objekt pii vom Typ ProvidedInterfaceInstance sowie ein Objekt rii vom Typ RequiredInterfaceInstance übersetzt. Im Beispiel gehört die Kompositionskonnektorvariable zum Kompositionskonnektorpart X. Dementsprechend werden bei der Übersetzung auch die Objekte zur Bindung der dementsprechenden Parts für Kompositionskonnektor und Schnittstellen angelegt. Weiterhin werden Objekte zur Bindung der Schnittstellen auf Typebene angelegt. Die Namen der Schnittstellen wurden aus Gründen der Übersichtlichkeit weggelassen.

Auf Komponentenvariablen können Komponentenstorydiagramme aufgerufen werden. Abschnitt 3.2.10.1 enthält eine Beschreibung der Übersetzung von Komponentenstorydiagrammen auf erweiterte Storydiagramme (s. Anhang A). Erweiterte Storydiagramme erlauben den Aufruf von anderen Storydiagrammen. Dies wird graphisch mit so genannten Transformation Call Knoten modelliert.

Dementsprechend wird also der Aufruf von Komponentenstorydiagrammen auf solche Transformation Calls übersetzt (s. Abbildung 3.43). Der Name des aufzurufenden erweiterten Storydiagramms ergibt sich nach Abbildung 3.32 durch den Namen des Komponententypen sowie dem Namen des Komponentenstorydiagramms. Als Argument für den immer vorhandenen Parameter self wird das Objekt für die übersetzte Komponentenvariable, auf der das Komponentenstorydiagramm aufgerufen werden soll, angegeben.

Weitere Argumente für einen Aufruf eines Komponentenstorydiagramms wer-

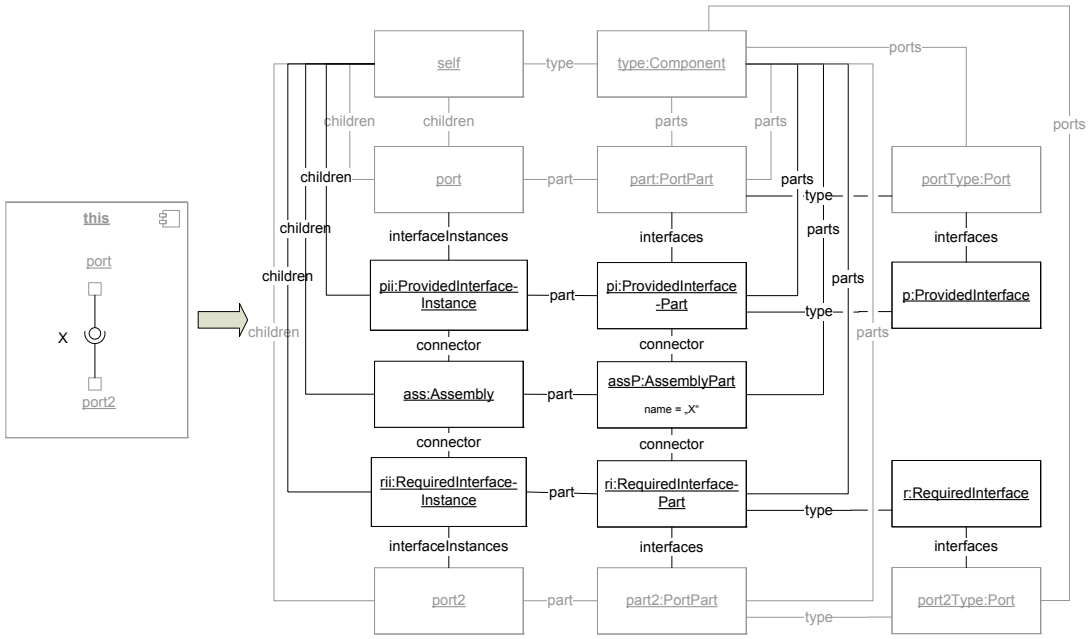


Abbildung 3.42: Übersetzung einer Kompositionskonnektorvariable

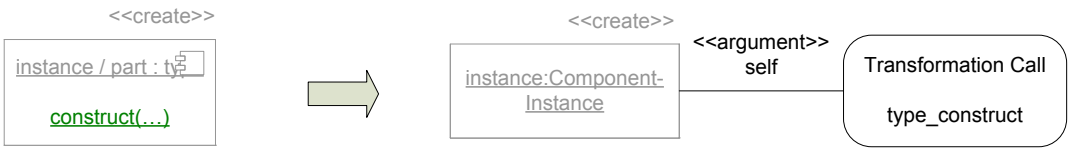


Abbildung 3.43: Übersetzung des Aufrufs eines Komponentenstorydiagramms

den analog zum self-Argument entsprechend der Reihenfolge der Signatur übersetzt (s. Abbildung 3.44). Komponentenstorydiagramme können auch Rückgabewerte zurückliefern. Diese Rückgabewerte werden an Variablen gebunden, die dann in nachfolgenden Komponentenstorypatterns als gebundene Variablen genutzt werden können. Für jeden Rückgabewert werden im erweiterten Storydiagramm Objekte für die jeweiligen Rückgabewerte (Komponenteninstanzen, Portinstanzen, Kompositions- oder Delegationskonnektoren) angelegt und entsprechend der Reihenfolge in der Signatur mit result-Links mit dem Transformation Call verbunden.

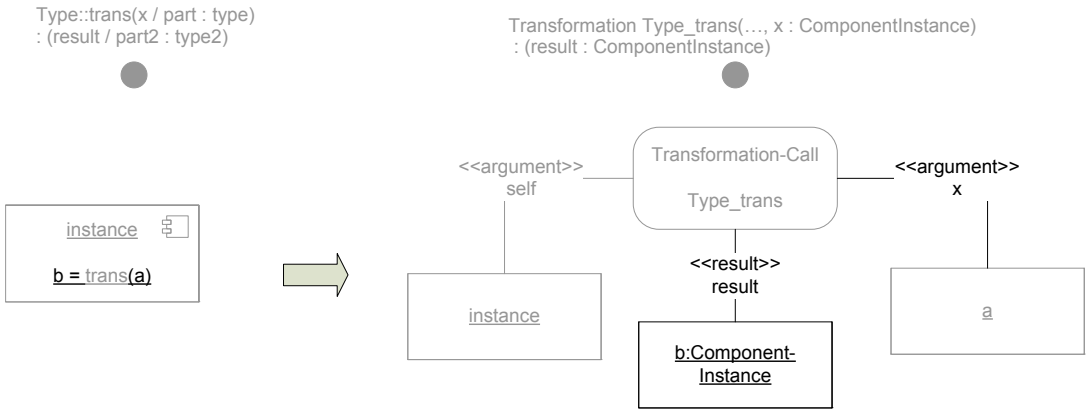


Abbildung 3.44: Übersetzung der Aufrufargumente und des Rückgabewertes

Attributbedingungen und -zuweisungen werden in der Form „Attribut Operator Wert“ bei Variablen angegeben und schränken die Bindungsmöglichkeit der Variablen ein. Sie werden übersetzt auf Objekte der Metaklassen AttributeValue und Attribute. Das zweite Objekt wird mit einer Attributbedingung versehen, die den Namen des Attributs einschränkt (im Beispiel „x“). Das Objekt value speichert den Wert des Attributs für die Instanz. Dementsprechend wird dem Objekt dann eine Attributbedingung bzw. -zuweisung für das Attribut value mit dem in der Variable angegebenen „Operator“ und „Wert“ hinzugefügt (s. Abbildung 3.45).

3.2.10.3 Übersetzung von ≪create≫/≪destroy≫

Wenn Variablen mit den Stereotypen ≪create≫ oder ≪destroy≫ versehen wurden, müssen bei der oben dargestellten Übersetzung der Variablen in Objekte eines Storypatterns zusätzlich diese Stereotypen korrekt übersetzt werden. Nach Abschnitt 3.2.3.8 gelten diese beiden Stereotypen auch für mit einer Variablen durch Kompositionsbeziehungen verbundene weitere Variablen, wie zum Beispiel

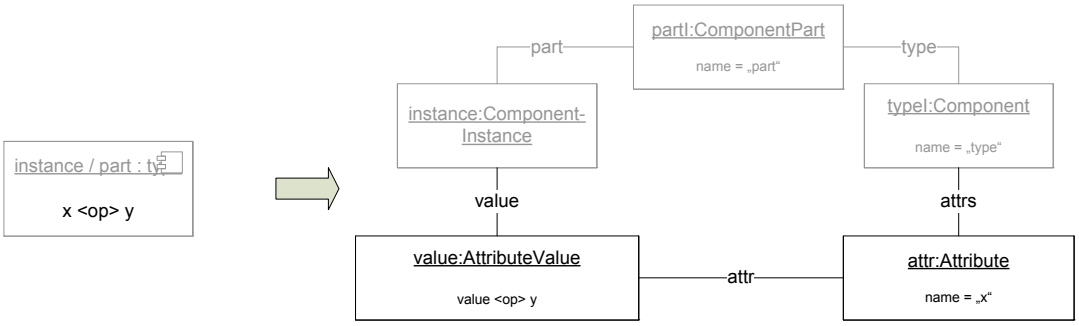


Abbildung 3.45: Übersetzung von Attributbedingungen und -zuweisungen

die Portvariablen an einer Komponentenvariable. Abbildung 3.46 zeigt wie eine zu erzeugende Komponentenvariable sowie eine zugehörige Portvariable auf Objekte eines Storypatterns übersetzt wird. Obwohl die Portvariable nicht mit ≪create≫ annotiert wurde, wird sie wegen der Kompositionsbeziehung wie eine zu erzeugende Variable auf Objekte eines Storydiagramms übersetzt.

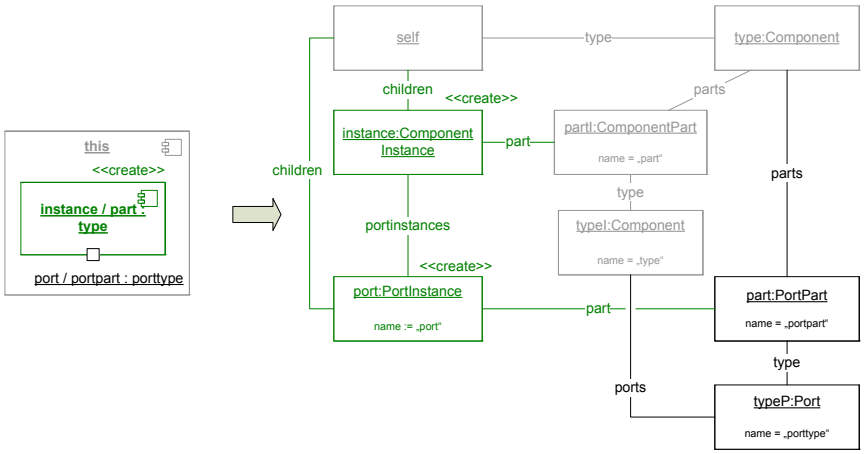


Abbildung 3.46: ≪create≫ Stereotype wird auf zusätzliche Objekte angewandt

Wenn eine Variable als zu löschend markiert wurde, wird die Übersetzung der zugehörigen Variablen analog durchgeführt. Bei der Ausführung der Löschung der an die Variable gebundenen Instanz muss allerdings zusätzlich die Situation betrachtet werden, dass nicht für alle der Instanz per Kompositionsbeziehung zugehörigen Instanzen (beispielsweise Portinstanz an einer Komponenteninstanz) Variablen vom Modellierer angelegt wurden. Dies würde dazu führen, dass nicht alle Instanzen gelöscht werden, die über eine Kompositionsbeziehung mit der zu

löschenden Instanz verbunden sind. Dies stellt einen Verstoß gegen die Kompositionsbeziehung dar. Um diesen Fall korrekt (im Sinne des gewählten SPO-Ansatzes [LE91]) zu behandeln, sind diese Beziehungen, wie z.B. zwischen Komponenteninstanz und Portinstanz sowie zwischen Komponenteninstanz und den unterlagerten Instanzen, im Metamodell durch Kompositionen [Obj07, S. 39] modelliert. Der aus diesen Metamodellen generierte Quelltext garantiert, dass beim Löschen einer Instanz auch alle über Kompositionsbeziehungen verbundene Instanzen mitgelöscht werden, also kein inkonsistentes Modell entsteht.

3.2.10.4 Kardinalitäten

Für Komponentenports und einer Komponente unterlagerte Parts werden bei der Typdefinition Kardinalitäten angegeben. Kardinalitäten beschreiben wie oft eine Instanz des jeweiligen Typs im spezifizierten Kontext existieren darf. Wenn z.B. ein Port die Kardinalität 0..1 besitzt, darf pro Instanz des zugehörigen Komponententyps nur maximal eine Instanz des Ports existieren. Betrachtet man alle Instanzen des Komponententyps, können insgesamt natürlich mehr als eine Portinstanz existieren. Analoges gilt auch für die für einen Komponententyp angegebenen Parts.

Bei der Ausführung der Komponentenstorypatterns müssen die spezifizierten Kardinalitäten eingehalten werden. Das gleiche Problem muss auch bei der Ausführung von Storypatterns berücksichtigt werden, da für Assoziationen in Klassendiagrammen ebenfalls beliebige Kardinalitäten angegeben werden können. Bei der Ausführung von Storypatterns werden nach [Zün02, S. 49] nur die Kardinalitäten 0..1 und 0..* unterschieden und bei der Ausführung beachtet. Auf Grund der Erfahrung mit Storypatterns und den in [Zün02, S. 48] beschriebenen Gründen werden für Kardinalitäten bei der Ausführung von Komponentenstorypatterns ebenfalls nur diese Kardinalitäten beachtet. Die Kardinalität 0..* ist bei der Ausführung unproblematisch, da sie keinerlei Einschränkung für die Anzahl der Instanzen angibt. Im Folgenden wird daher nur der Sonderfall der Kardinalität 0..1 beschrieben.

Abbildung 3.47 zeigt die Übersetzung einer zu erzeugenden Komponentenvariable. In diesem Beispiel wird angenommen, dass bei der Definition des Komponententyps der this-Variable vom Modellierer festgelegt wurde, dass maximal eine Instanz vom Part part:type existieren darf. Bei der Übersetzung der Komponentenvariable instance/part:type wird neben der normalen Übersetzung zusätzlich das optionale, zu löschende Objekt instanceOld angelegt. Bei der Ausführung des resultierenden Storydiagramms wird dann versucht, dieses optionale Objekt zu binden. Wenn dies erfolgreich ist, wird es gelöscht, damit nach Erzeugung des Objekts instance nur eine Komponenteninstanz innerhalb der an das Objekt self

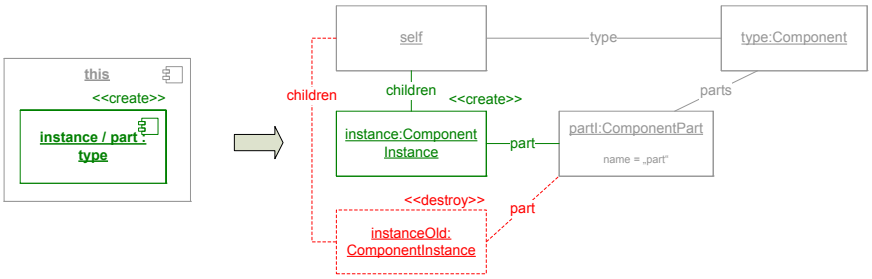


Abbildung 3.47: Übersetzung einer zu erzeugenden Komponentenvariable im Fall einer Kardinalität von 0..1

gebundenen Komponenteninstanz existiert.

Dieses Vorgehen wird neben dem dargestellten Fall einer Komponenteninstanz ebenfalls für Portinstanzen sowie Kompositions- und Delegationskonnektoren angewandt. Da Konnektoren zwei Kardinalitäten besitzen (an der Quell- und an der Zielseite), werden beide Seiten betrachtet.

3.2.10.5 Negative Variablen

Negative Variablen eines Komponentenstorypatterns können im Prinzip direkt in negative Objekte eines Storypatterns übersetzt werden. Allerdings sind Komponenten- und Portinstanzen sowie Kompositionskonnektoren und Delegationskonnektoren nicht unabhängig voneinander sondern durch Kompositionsbeziehungen verbunden (siehe Abschnitt 3.1.3.1). In Abschnitt 3.2.3.8 wurde erläutert, dass die Negation einer Komponentenvariable dementsprechend auch auf zugehörige Portvariablen und etwaige Kompositions- und Delegationskonnektorvariablen ausgedehnt wird.

Würde dies nicht gemacht, würde im Beispiel aus Abbildung 3.48 versucht, die Kompositionskonnektorvariable ass zu binden, die zu einer Portinstanz gehören muss, die wegen der Negation der Portinstanzvariable port2 allerdings nicht existieren darf. Solch eine Situation ist widersprüchlich und kann dementsprechend nie im Wirtsgraph gefunden werden. In Abbildung 3.48 wird daher die Negation auf die Kompositionskonnektorvariable ass erweitert. Durch die Kompositionsbeziehung zwischen Portinstanz und Kompositionskonnektor stellt dies eine negative Gruppe dar. Es wird also versucht port und instance2 so zu binden, dass instance2 keine Portinstanz besitzt, die per Kompositionskonnektor mit port verbunden ist.

Abbildung 3.48 zeigt auf der rechten Seite zwei Komponenteninstanzstrukturen als Beispiele für die Überprüfung der negativen Variablen. Die linke Beispielstruktur wird erfolgreich gebunden, unabhängig davon ob die Portvariable und

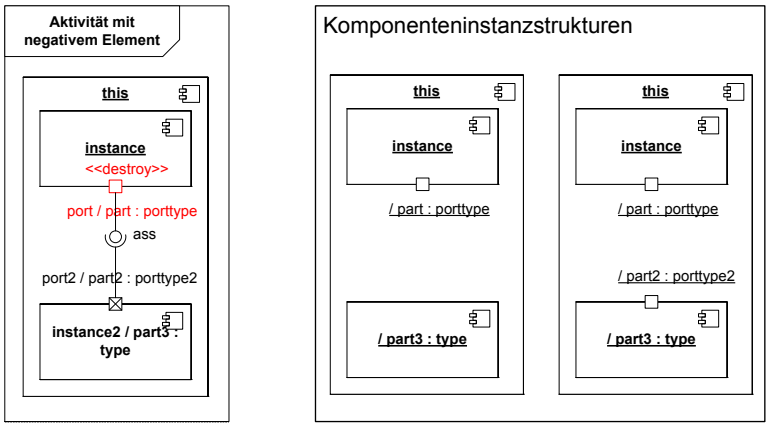


Abbildung 3.48: Beispiel für negative Variablen

die Kompositionskonnektorvariable als negative Gruppe betrachtet wird oder jede negative Variable einzeln betrachtet wird. Die rechte Beispielstruktur wird allerdings nur korrekt gebunden, wenn die beiden Variablen als negative Gruppe betrachtet werden. Wenn die negativen Variablen einzeln betrachtet werden würden, würde die Portinstanz /part2:porttype2 dazu führen, dass die negative Portvariable eine Bindung fälschlicherweise verhindert.

Falls dieses Verhalten gewünscht ist, muss der Entwickler die Kompositionskonnektorvariable im Komponentenstorypattern weglassen. Dies führt dann dazu, dass die negative Portvariable nur die Existenz einer Portinstanz überprüft und nicht zusätzlich einen zugehörigen Kompositionskonnektor.

Bei der Übersetzung dieser beiden negativen Variablen in Storypatterns muss die Semantik negativer Objekte in Storypatterns beachtet werden. Nach [Zün02, S. 137] wird jedes negative Element einzeln auf Existenz überprüft. Eine 1-zu-1 Übersetzung der negativen Variablen würde dazu führen, dass die oben dargestellte negative Gruppe falsch übersetzt wird.

Grundlage für eine korrekte Übersetzung ist die Ersetzung der negativen Variablen in einem Komponentenstorypattern durch zusätzliche Überprüfungsaktivitäten für negative Variablen analog zu [Zün02, S. 138]. Diese Übersetzung ist dargestellt in Abbildung 3.49. In einem ersten iterierten Komponentenstorypattern wird versucht, die linke Regelseite ohne die negativen Variablen zu binden (Aktivität Ohne negatives Element). Für jede erfolgreiche Bindung wird dann in der Aktivität NAC versucht, die negativen Variablen zu binden. Dies realisiert die korrekte Übersetzung der negativen Gruppe, da bei der Bindung versucht wird, alle Variablen im Zusammenhang zu binden und nicht jedes Element einzeln.

Ist diese Bindung erfolgreich (Bedingung success) bedeutet dies, dass die ne-

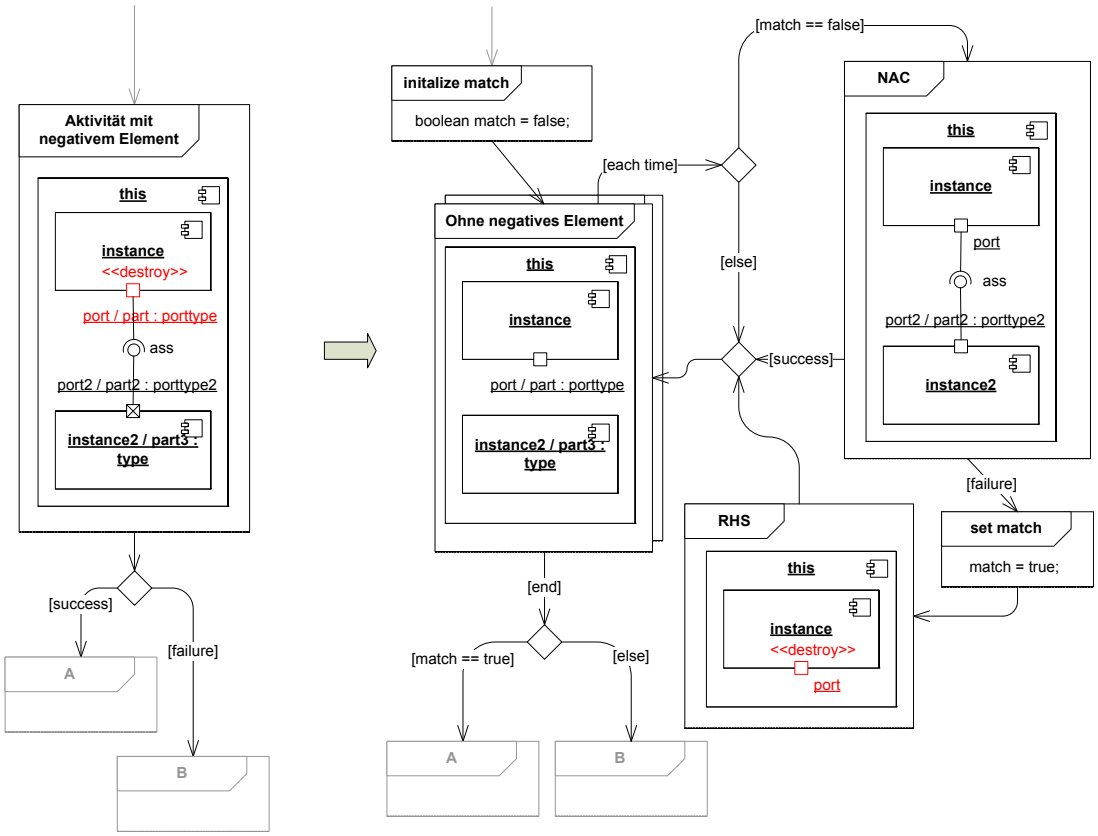


Abbildung 3.49: Übersetzung negativer Elemente

gative Gruppe gefunden werden konnte, also die Bindung unter Beachtung der negativen Variablen fehlgeschlagen ist. Dementsprechend wird im iterierten Komponentenstorypattern versucht, die nächste linke Regelseite ohne negative Elemente zu binden und nachfolgend wieder die negativen Elemente zu überprüfen. Dies geschieht so lange, bis entweder keine Bindungen der linken Regelseite ohne negative Variablen gefunden werden konnten oder die Prüfung der negativen Elemente fehlschlägt und damit eine insgesamt korrekte Bindung gefunden wurde. In diesem Fall wird die rechte Regelseite ausgeführt, dass heißt die an die Variable port gebundene Portinstanz gelöscht, und eine boolesche Variable match auf wahr gesetzt, da das ursprüngliche Komponentenstorypattern nur für die erste erfolgreiche Bindung ausgeführt werden soll. Falls das ursprüngliche Komponentenstorypattern mit negativen Elementen ein iteriertes Komponentenstorypattern ist, dann wird auf die Variable match verzichtet, so dass für alle erfolgreichen Bindungen die rechte Regelseite hergestellt wird.

Durch die vorgestellte Übersetzung werden alle negativen Variablen eines Kom-

ponentenstorypatterns als Gruppe betrachtet. Falls dies vom Entwickler nicht gewünscht ist, kann er die einzelnen negativen Variablen entsprechend zu der beschriebenen Übersetzung durch mehrere sequentielle Aktivitäten analog zur Aktivität NAC abbilden. Diese Aktivitäten überpüfen dann jeweils einzeln eine negative Variable.

Die Übersetzung negativer Elemente mit Attributbedingungen muss analog durchgeführt werden, da Attributbedingungen (nach Abbildung 3.45) in mehrere Objekte in einem Storypattern übersetzt werden. Dies führt dazu, dass alle Objekte als negative Gruppe im Storypattern betrachtet werden müssten. Da Storypatterns dies nicht erlauben, wird obige Übersetzung analog angewendet.

3.2.10.6 Optionale Variablen

Optionale Variablen müssen analog behandelt werden. Für sie gelten ebenfalls die Auswirkungen der Kompositionsbeziehung zwischen verschiedenen Elementen (siehe Abschnitt 3.2.3.8). Da Storydiagramme nach [Zün02, S. 146] ebenfalls versuchen optionale Elemente einzeln zu binden, müssen für eine korrekte Übersetzung der optionalen Variablen optionale Gruppen, analog zu den negativen Gruppen, realisiert werden.

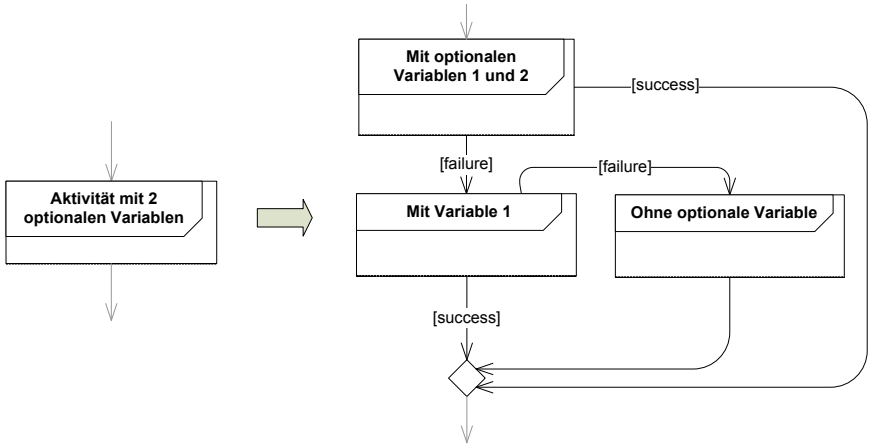


Abbildung 3.50: Übersetzung einer Aktivität mit zwei optionalen Variablen

Optionale Objekte werden in Storydiagrammen bei der Bindung iterativ verarbeitet. Zuerst wird versucht, zusätzlich zu den nicht-optionalen Objekten alle optionalen Objekte zu binden. Gelingt dies nicht, werden sukzessiv die optionalen Objekte entfernt, so dass bei n optionalen Objekten nach $n + 1$ Bindungsversuchen schließlich eine Bindung nur der nicht-optionalen Objekte versucht wird. Die Reihenfolge der Entfernung optionaler Objekte ist nicht-deterministisch.

Abbildung 3.50 zeigt die Übersetzung eines Komponentenstorypatterns mit zwei optionalen Variablen, die aus Gründen der Übersichtlichkeit nicht im Diagramm dargestellt werden, auf mehrere Aktivitäten ohne optionale Variablen. In einer ersten Aktivität wird versucht alle Variablen zu binden. Falls dies nicht gelingt, wird eine der optionalen Variablen aus der Aktivität entfernt und nochmalig versucht die Variablen zu binden. Wenn dies ebenfalls mißlingt, wird als letztes versucht, nur die nicht-optionalen Variablen zu binden. Bei dieser Übersetzung der optionalen Variablen wird die Kompositionsbeziehung zwischen Variablen beachtet. Aktivitäten mit mehr als 2 optionalen Variablen werden analog übersetzt. Optionale Variablen mit Attributbedingungen werden identisch übersetzt.

Sind negative und optionale Elemente in einem Komponentenstorypattern enthalten, werden zuerst die negativen Elemente und nachfolgend die optionalen Elemente betrachtet.

3.3 Ausführung in Echtzeitsystemen

Komponentenstorydiagramme modellieren eine Rekonfiguration der Komponenteninstanzstruktur. Die vorgestellten Übersetzungsregeln zur formalen Definition der Semantik werden auch eingesetzt, um aus Komponentenstorydiagrammen Quelltext zur Ausführung zu generieren. In einem ersten Schritt werden die Komponentenstorydiagramme mittels den vorgestellten Übersetzungsregeln in Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen übersetzt. Aus den Storydiagrammen wird dann nachfolgend Quelltext mit der bereits existierenden Quelltextgenerierung für Storydiagramme erzeugt, der dann ausgeführt werden kann.

Selbstoptimierende Systeme müssen typischerweise harte Echtzeitanforderungen einhalten. Um Systeme, deren Rekonfiguration mit Komponentenstorydiagrammen spezifiziert worden sind, einsetzen zu können, wird im Folgenden eine Analyse der maximalen Ausführungszeiten (Worst-Case-Execution-Time = WCET) des aus Komponentenstorydiagrammen generierten Quelltextes vorgestellt. Darauf aufbauend wird dann gezeigt, wie das Scheduling für solche Systeme analysiert wird.

3.3.1 Worst-Case-Execution-Time Analyse

Die Berechnung der WCET des generierten Quelltextes wird auf Basis der generierten Storydiagramme durchgeführt. Die Fujaba Quelltextgenerierung für Storydiagramme für Nicht-Echtzeitsysteme nutzt eine Heuristik [FNT98, S. 177],

die auf Basis der Assoziationsrollenkardinalitäten versucht, eine möglichst gute Reihenfolge hinsichtlich der Ausführungszeit zu wählen. Diese Heuristik basiert im Wesentlichen darauf, dass versucht wird, möglichst frühzeitig Objekte auszuschließen, indem zum Beispiel zuerst Links überprüft werden, deren Assoziationsrollen eine Kardinalität `0..1` haben, und erst danach Links mit einer Kardinalität `0..*`. Diese Heuristik kennt allerdings keine Zahlen als feste Obergrenzen für die Kardinalitäten und kann daher keine optimale Bindungsreihenfolge berechnen. Des Weiteren kann ohne diese Information keine maximale Ausführungszeit berechnet werden.

Es wurde daher in [Sei05, BGST05] eine Technik zur Bestimmung der maximalen Ausführungszeit (WCET) von Storypatterns vorgestellt, die feste Zahlen für die Kardinalitäten und die Instanzen der einzelnen Klassen benötigt. Mit Hilfe dieser Informationen kann dann die WCET berechnet werden. Neben der Bestimmung der WCET wurde auch deren Minimierung, durch eine Berechnung der optimalen Bindungsreihenfolge im schlechtesten Fall, vorgestellt. In [Sei05] wurde eine Erweiterung dieser Technik für die Berechnung der WCET nicht nur von einzelnen Storypatterns sondern eines kompletten Storydiagramms im Ausblick skizziert, aber noch nicht voll umgesetzt. In [Bur06] wurde dies ebenfalls skizziert und für Storydiagramme mit maximal einer Aktivität umgesetzt.

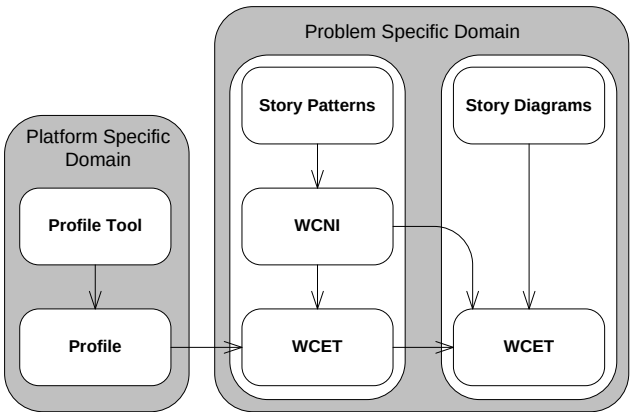


Abbildung 3.51: Analyseprozess zur Berechnung der WCET von Storydiagrammen [TGS06]

Aufbauend auf diesen Arbeiten wurde in [TGS06] die Berechnung der WCET für Storydiagramme dargestellt. Abbildung 3.51 zeigt den Analyseprozess im Überblick. Die Ausführung von Storypatterns basiert auf einer Menge von Basisoperationen, wie die Bindung eines Objektes oder die Überprüfung einer Menge von Objekten über eine Assoziationsrolle mit der Kardinalität `0..*`. Bei der Ausführung von Storypatterns und Storydiagrammen werden diese Basisoperationen

in einer bestimmten Reihenfolge benutzt. Im Rahmen einer plattformspezifischen Analyse mittels herkömmlicher WCET-Analysewerkzeuge wird für jede Basisoperation eine plattformspezifische WCET berechnet. Alle diese WCETs ergeben das Profil einer Plattform.

Durch den in selbstoptimierenden Systemen eingeschränkten Speicherplatz und die Notwendigkeit harte Echtzeitbedingungen zu erfüllen, müssen feste Obergrenzen für die Kardinalität 0..* festgelegt werden. Die Anzahl der Instanzen und maximalen Kardinalitäten der Assoziationsrollen für das Metamodell der Komponenteninstanzstrukturen wird dann auf Basis der festen Kardinalitäten der Elemente der hierarchischen Komponententypdefinition und den Übersetzungsregeln berechnet.

Für ein einzelnes Storypattern wird nach [Sei05, BGST05] dann die maximale Anzahl an Schleifendurchläufen (WCNI: Worst-Case-Number-of-Iterations) der verschiedenen Schleifen, die für die Bindung der Objekte notwendig sind, auf Basis der bekannten maximalen Anzahl an Instanzen und festen Kardinalitäten der Assoziationen berechnet. In Kombination mit dem Profil kann damit die WCET eines Storypatterns berechnet werden.

Eine Erweiterung des Ansatzes für die genauere Berechnung der WCNI wurde in [TGS06] beschrieben. Durch den in Storypatterns eingesetzten Isomorphismus [Zün02, S. 110] verringern sich im Falle von als gebunden markierten Objekten die maximal möglichen Bindungen für Objekte des gleichen Typs in einem Storypattern um die Anzahl der als gebunden markierten Objekte. Da bei der Übersetzung eines Komponentenstorypatterns auf ein Storypattern, das Storypattern viele Objekte der gleichen Klasse enthält (z.B. ComponentInstance für alle Komponenteninstanzen) und bei der Übersetzung von z.B. negativen Variablen viele gebundene Objekte genutzt werden, hat diese Verbesserung einen starken Effekt und resultiert in einer deutlich genaueren Berechnung der maximalen Anzahl an Schleifendurchläufen.

Für die Berechnung der WCET von Storydiagrammen ist der zeitlich längste Pfad von der Start- zur Stopaktivität zu betrachten. Storydiagramme ohne Entscheidungen oder Schleifen haben nur einen einzelnen Pfad. Daher ist die WCET dann einfach die Summe der WCETs aller Aktivitäten. Bei Storydiagrammen mit Entscheidungen ist die WCET des Storydiagramms das Maximum der WCETs aller Pfade des Storydiagramms.

Wenn ein Storydiagramm Schleifen enthält, muss die maximale Anzahl an Schleifendurchläufen bekannt sein, um den längsten Pfad zu bestimmen und die WCET zu berechnen. Für iterierte Storypatterns entspricht die Anzahl der Durchläufe der Anzahl der möglichen Belegungen für die linke Regelseite des iterierten Storypatterns (unter Nutzung der *Pre-Select* Semantik [Zün02, TMG06]), die auf Basis der festen Kardinalitäten automatisch berechnet werden. Wenn die

modellierten Komponentenstorydiagramme als Schleifenkonstrukte nur iterierte Komponentenstorypatterns beinhalten, enthalten auch die auf Basis der Übersetzungsregeln daraus erzeugten Storydiagramme nur iterierte Storypatterns. Falls der Entwickler Schleifen nicht mit iterierten Komponentenstorypatterns, sondern durch den Kontrollfluss modelliert, muss er manuell für die WCET-Analyse die maximale Anzahl an Schleifendurchläufen angeben.

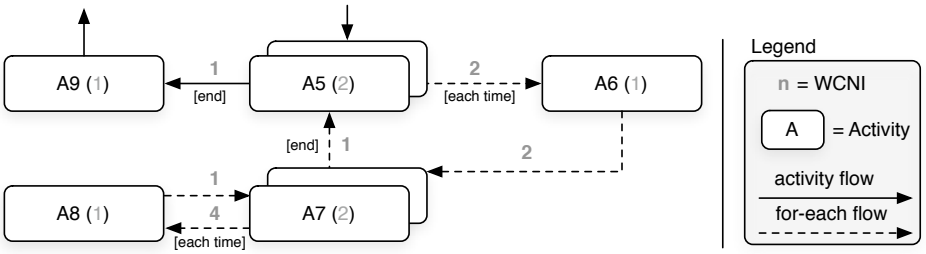


Abbildung 3.52: Berechnung der WCNI [TGS06]

Abbildung 3.52 zeigt an einem Beispiel die Berechnung der WCNI für ein Storydiagramm mit zwei iterierten geschachtelten Storypatterns A5 und A7. Es existieren im Beispiel zwei verschiedene Belegungen für A5. Dementsprechend werden A6 und A7 jeweils 2 mal aufgerufen; die WCNI der Schleife von A5 beträgt also 2. Die geschachtelte Schleife durch das iterierte Storypattern A7 hat ebenfalls eine WCNI von 2 durch 2 mögliche Belegungen des Storypatterns A7. Durch die Schachtelung von A7 in die Schleife von A5 kann es nun im schlechtesten Fall 2 Belegungen von A7 für jede der 2 Belegungen von A5 geben. Dies führt dazu, dass das Storypattern A8 insgesamt 4 mal ausgeführt wird.

Mit Hilfe dieser Informationen wird wie in [TGS06] beschrieben die WCET eines Storydiagramms mit (auch verschachtelten) Schleifen automatisch berechnet. Die experimentellen Ergebnisse auf Basis eingebetteter Hardware (phyCORE-MPC555 Prozessorboard mit einem 40MHz 32-Bit PowerPC Mikroprozessor von Motorola) und unter Benutzung des DREAMS Echtzeitbetriebssystems [Dit00] in [TGS06] zeigen, dass die Berechnung auch im realen System zu guten Ergebnissen führt. Die berechneten Zeiten sind zwar etwas höher (bis zu 15%) als die gemessenen Zeiten, können als WCET allerdings trotzdem genutzt werden, da sie eine pessimistische Abschätzung darstellen.

Alternativ kann die WCET auch mit dem Werkzeug bound-t [HLS00] berechnet werden. Die berechneten WCNI werden dabei dem Werkzeug übergeben, dass auf dieser Grundlage dann die WCET berechnet [Ric08].

3.3.2 Scheduling im Echtzeitbetriebssystem

Die Ausführung der Komponentenstorydiagramme zur Laufzeit resultiert in unterschiedlichen Ressourcenanforderungen der Anwendung an das Betriebssystem. Um diese Änderungen zur Laufzeit in einem Echtzeitsystem umzusetzen, ist zuvor eine Scheduling Analyse notwendig. Diese Analyse überprüft, ob das System auch nach der Ausführung eines Komponentenstorydiagramms auf einen Prozess noch alle Echtzeitanforderungen (deadlines) einhält. In [THHO08] wurde beschrieben, wie die Ausführung der Komponentenstorydiagramme und die daraus resultierenden Konfigurationen in eine Scheduling Analyse integriert werden kann. Im Folgenden wird dies skizziert.

Für den Fall einer Menge von Prozessen $\tau_1, ..., \tau_n$ mit einer festen Periode T_i, einer Deadline, die der Periode entspricht, sowie einer WCET Zeit C_i für die Ausführung des Prozesses τ_i hat die Scheduling Analyse bei der Nutzung eines *Earliest Deadline First* (EDF) Schedulers nur eine lineare Komplexität bezüglich der Anzahl der Prozesse n [LL73]. Wenn Bedingung 3.1 erfüllt ist, können alle Prozesse mit EDF geplant werden. Dies ermöglicht die Überprüfung des Schedulings vor einer Ausführung eines Komponentenstorydiagramms.

$$\sum_{i=1}^{n} C_i/T_i \leq 1 \tag{3.1}$$

Die Ausführung eines Komponentenstorydiagramms ist typischerweise aperiodisch. Unter Nutzung des Total Bandwidth Servers (TBS) [SB96] können auch aperiodische Prozesse, die ein Komponentenstorydiagramm ausführen, bei der Scheduling Analyse betrachtet werden, ohne dabei die Komplexitätsklasse des Problems zu ändern. Grundlegende Idee des TBS ist, für aperiodische Prozesse ebenfalls eine WCET (unter Benutzung der Ergebnisse aus Abschnitt 3.3.1) und eine Deadline zu definieren und sie dann wie periodische Prozesse bei der Scheduling Analyse zu behandeln. Da Komponentenstorydiagramme ja nur aperiodisch ausgeführt werden, sie aber als periodische Prozesse betrachtet werden, wird Prozessorzeit vergeudet, wenn kein Komponentenstorydiagramm ausgeführt wird. Um diese Prozessorzeit sinnvoll zu verwenden, kann der Flexible Ressource Manager (FRM) [OB04] genutzt werden. Er ermöglicht im Rahmen von selbstoptimierenden Systemen, derzeit nicht benutzte Prozessorzeit anderen Prozessen zur Verfügung zu stellen.

3.4 Zusammenfassung

In diesem Kapitel wurde die Erweiterung der MECHATRONIC UML in Bezug auf die Modellierung der Struktur und deren Rekonfiguration beschrieben. Die Strukturmodellierung der MECHATRONIC UML wurde dabei um das Konzept der Kompositionsstrukturen der UML erweitert, die es ermöglicht, hierarchische Komponentenstrukturen zu definieren, in denen für jede Hierarchieebene Typstrukturen definiert werden.

Auf Basis dieser Strukturmodellierung wurden Komponentenstorydiagramme als Spezifikationssprache für Rekonfigurationen vorgestellt. Neben der Erfüllung der Anforderungen aus [BCDW04] sowie der intuitiven graphischen Syntax ist die starke Typisierung der Sprache herauszustellen. Dies ermöglicht die Implementierung eines syntaxgesteuerten kontextsensitiven Editors, so dass nur Komponentenstorydiagramme modelliert werden können, die typkonforme Komponentenstrukturen erzeugen.

Quelltext für die Ausführung der Komponentenstorydiagramme wird durch die Anwendung der Übersetzungsregeln und der nachfolgenden Generierung von Quelltext aus den resultierenden Storydiagrammen erzeugt. Der Quelltext kann in harten Echtzeitsystemen ausgeführt werden, da eine maximale Ausführungszeit für ihn berechnet wird.

4 Komponentenbasierte Gefahrenanalyse

Mechatronische Systeme werden oftmals in Bereichen eingesetzt, in denen die Verlässlichkeit des Systems wichtig ist. Dies ist zum Beispiel der Einsatz in Flugzeugen, Industrieanlagen und Automobilen. In diesen Bereichen muss die Verlässlichkeit des entwickelten Systems gebenüber externen Gutachtern wie dem TÜV oder Regierungsstellen wie dem Bundesamt für Strahlenschutz und dem Eisenbahnbundesamt nachgewiesen werden.

Die Verlässlichkeit lässt sich, wie in Kapitel 2 dargestellt, in die verschiedenen Kenngrößen Verfügbarkeit, Zuverlässigkeit, Sicherheit und Vertraulichkeit unterteilen. Verschiedene Techniken werden für den Nachweis der Verlässlichkeit in Bezug auf die Anforderungen eingesetzt. Dies sind zum Beispiel Fehlerbaumanalyse (FTA), Hazard and Operability Study (HAZOP), Failure Mode Effect Analysis (FMEA) und Zuverlässigkeitsblockdiagramme (vgl. [Sto96]).

In diesem Kapitel wird eine komponentenbasierte Gefahrenanalyse zur qualitativen und quantitativen Analyse der Sicherheit [GTS04, GT06] als einer Kenngröße der Verlässlichkeit vorgestellt. Die qualitative Analyse ergibt, ob eine Gefahrensituation eintreten kann, während eine quantitative Analyse auf Basis von Wahrscheinlichkeiten von einzelnen Fehlerzuständen die Wahrscheinlichkeit für die Gefahrensituation berechnet. Sie basiert auf dem Konzept der Fehlerpathologie von Laprie in komponentenbasierten Systemen (vgl. Abschnitt 2.2.2 und [Lap92]).

Grundlage für die Gefahrenanalyse sind die Modelle des mechatronischen Systems, deren Struktur mit der in Kapitel 3 vorgestellten Modellierungssprache spezifiziert wurden. Für die einzelnen Komponententypen wird zusätzlich das Verhalten im Fehlerfall modelliert. Die Analysetechnik nutzt dann die modellierte Komponenteninstanzstruktur, um die Fehlerverhaltensmodelle der einzelnen Teilkomponenten automatisiert zu verknüpfen und so das Fehlerverhaltensmodell des Gesamtsystems zu bestimmen.

Die hier vorgestellte Technik basiert auf der Fehlerbaumanalyse (FTA) (s. Abschnitt 2.2.3). Ein Fehlerbaum beschreibt das Zustandekommen einer Gefahr auf Basis einer booleschen Kombination von Basisereignissen. Fehlerbäume werden typischerweise in der Praxis per Hand erstellt und basieren nicht auf der Ar-

chitektur des Systems. Dies bedeutet, dass neben dem höheren Aufwand gegenüber einer automatischen Analyse auch Fehler bei der Analyse gemacht werden können. Hier sind die fehlende Betrachtung von Fehlerzuständen in einzelnen Komponenten des Systems oder die Auslassung von Zusammenhängen zwischen Komponenten zu nennen. Dies führt dazu, dass die Analyse nicht das entworfene System widerspiegelt und die aus der Analyse gezogenen Folgerungen für die Verlässlichkeit des Systems falsch bzw. unzureichend sind. Eine automatische Analysetechnik auf Struktur- und Fehlerverhaltensmodellen des Systems verringert diese Probleme.

Verwandte Arbeiten [FM93, KLM03, PM99, Wal05, Gru03b] unterstützen nicht die speziellen Eigenschaften selbstoptimierender Systeme, wie zyklische Strukturen und Rekonfiguration. In diesem Abschnitt wird die Rekonfiguration ebenfalls nicht betrachtet, allerdings ermöglicht die vorgestellte Technik die Erweiterung um die Betrachtung rekonfigurierbarer Systeme in Kapitel 5.

Die im Folgenden vorgestellte Technik ermöglicht die Gefahrenanalyse ähnlich zu Fehlerbäumen auf Basis der Architektur des Systems. Die Idee basiert auf der Spezifikation des wiederverwendbarem Fehlerverhalten einzelner Komponententypen. Das Fehlerverhalten wird auf Basis von Fehlerzuständen und Ausfällen definiert. Fehlerzustände entstehen innerhalb von Komponenten und manifestieren sich als Ausfälle an den Ports der Komponente (vgl. Fehlerpathologie [Lap92] und Abschnitt 2.2.2). Die Ausfälle an den Ports der Komponenten, also den Schnittstellen zu anderen Komponenten, propagieren sich über die Verbindungen zwischen den Komponenten zu anderen Komponenten und können dort auch zu Ausfällen führen. Das Fehlerverhalten einer Komponente beschreibt also, wie sich eine Komponente bezüglich (eingehenden) Ausfällen anderer Komponenten sowie internen Fehlerzuständen verhält.

Aus der Fehlerpropagierung der Komponententypen sowie der Komponenteninstanzstruktur wird die Fehlerpropagierung des Gesamtsystems erzeugt. Die Fehlerpropagierung des Gesamtsystems wird dann hinsichtlich des Auftretens einer Gefahr analysiert. Die Wahrscheinlichkeit dafür wird auf Grundlage von angenommenen Wahrscheinlichkeiten für die Fehlerzustände berechnet.

Abbildung 4.1 zeigt eine Übersicht über die komponentenbasierte Gefahrenanalyse. Die Gefahr wird mit einem Fehlerbaum spezifiziert. Das Topereignis wird dann manuell bis zu Ausfällen an einzelnen Ports der Komponenteninstanzstruktur verfeinert. Die restliche Verfeinerung von diesen Ausfällen an den Ports bis zu den Fehlerzuständen in den Komponenten wird dann automatisch aus der Komponentenstruktur sowie den spezifizierten Fehlerpropagierungen hergeleitet. Resultat sind die Kombinationen der Fehlerzuständen der Komponenten, die schließlich zur Gefahr führen.

Die Konvoifahrt von zwei RailCabs basierend auf der Modellierung in Kapitel

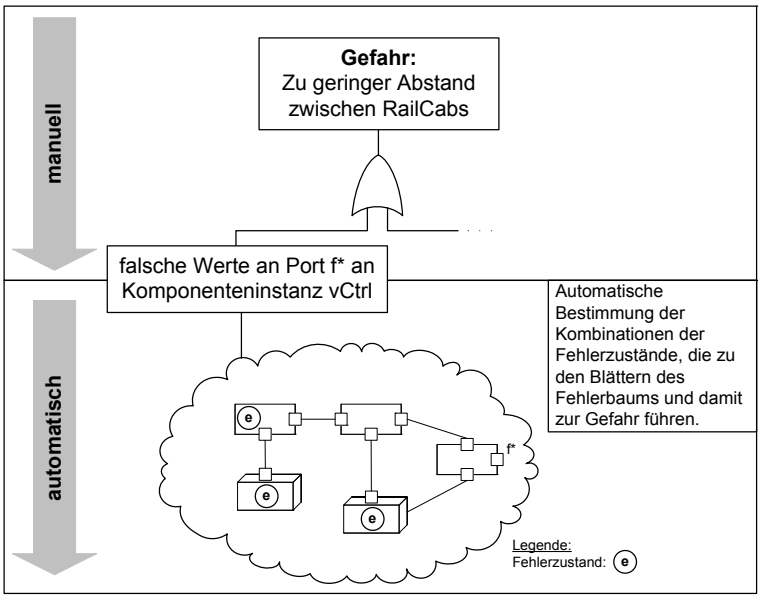


Abbildung 4.1: Komponentenbasierte Gefahrenanalyse

3 wird als Beispiel für die Gefahrenanalyse genutzt. Bei der Konvoifahrt zweier RailCabs ist eine Kollision zwischen den Fahrzeugen ein Unfall. In einer Gefahrenanalyse sind nun die Zustände des Systems (und der Umgebung) sowie deren Kombinationen zu identifizieren, die zu diesem Unfall führen.

Abbildung 4.2 zeigt die Komponenteninstanzstruktur des skizzierten Beispiels. Diese Komponenteninstanzstruktur wurde mit der in Abschnitt 3.1 beschriebenen Sprache modelliert.

Die Komponenteninstanzstruktur entspricht weitestgehend der Version aus Abbildung 3.9. Wie zu Beginn dieses Kapitels beschrieben, betrachtet die Gefahrenanalyse die Fehlerpropagierung durch die Komponenteninstanzen. Um mehr Möglichkeiten zur Spezifikation der Fehlerpropagierung zu zeigen, wurde die Komponenteninstanzstruktur um einen GPS Sensor /gpsSensor:GpsSensor erweitert, der ebenfalls die aktuelle Position auf der Strecke sowie die aktuelle Geschwindigkeit berechnet. Die Komponenteninstanz /xvCalc:xvCalc nutzt beide Sensoren, um die aktuelle Geschwindigkeit und Position zu berechnen. Hierdurch kann diese Komponente fehlende Daten eines der beiden Sensoren tolerieren.

Im folgenden Abschnitt werden, zusätzlich zu den in Kapitel 3 beschriebenen Strukturmodellen, die für die Gefahrenanalyse nötigen Fehlerverhaltensmodelle beschrieben. Danach werden in den Abschnitten 4.2 und 4.3 eine qualitative und eine quantitative Gefahrenanalyse auf Basis der Modelle dargestellt. Die quantitative Analyse wird in Abschnitt 4.4 erweitert, um Ansatzpunkte für die

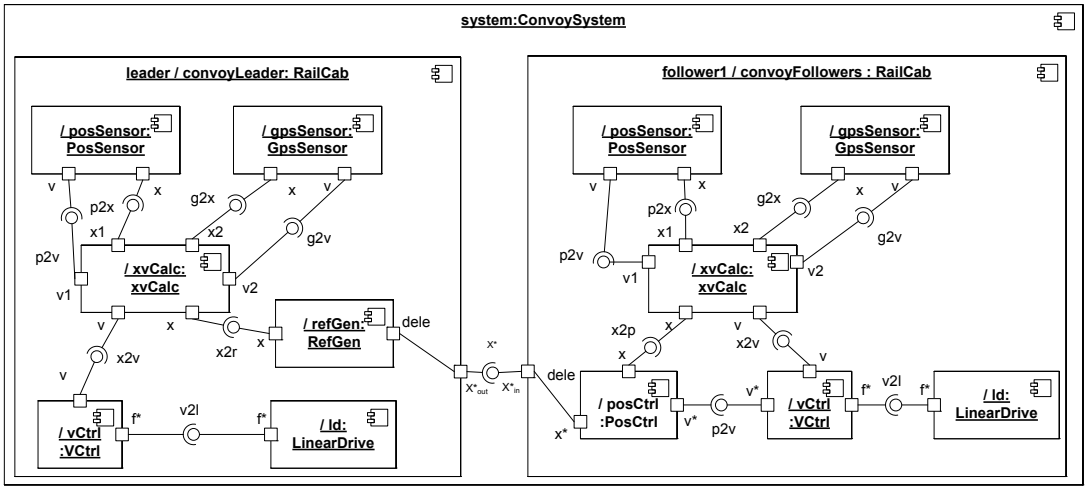


Abbildung 4.2: Komponenteninstanzstruktur für das Beispiel

Anwendung von Fehlertoleranztechniken zu bestimmen.

4.1 Modelle

Zusätzlich zu den Komponenteninstanzstrukturen aus Abschnitt 3.1 muss das abstrakte Fehlerverhalten der Komponenten und Konnektoren zwischen diesen beschrieben werden, um eine Gefahrenanalyse durchführen zu können.

4.1.1 Fehlertyphierachie

Grundlage der Gefahrenanalyse ist die Definition der verschiedenen Fehlertypen. In Abschnitt 2.2.2 wurden die verschiedenen in der Literatur typischerweise genutzten Fehlertypen eingeführt. Auf Basis dieser Fehlertypen kann eine anwendungsspezifische Fehlertyphierarchie modelliert werden.

Neben der Definition der Fehlertypen wird die Spezifikation von Generalisierungsbeziehungen unterstützt. Diese erlaubt es, die Ersetzbarkeit von spezifischeren Fehlertypen durch allgemeinere Fehlertypen, in Anlehnung an die Konzepte objektorientierter Programmiersprachen, zu modellieren. Dies ist hilfreich, wenn Komponenten bzw. ihre Fehlerpropagierungen verbunden werden. Sie können verbunden werden, wenn die Fehlertypen *kompatibel* sind. Kompatibel bedeutet, dass entweder der selbe Fehlertyp genutzt wird oder ein Fehlertyp mit einem allgemeineren Fehlertyp verbunden werden soll.

Definition 4.1 *Alle in einer Fehlertyphierarchie enthaltenen Fehlertypen t sind in der Menge T enthalten.*

Abbildung 4.3 zeigt eine Beispielfehlertyphierarchie [GTS04] mit der aus der Literatur bekannten Aufteilung in Dienstfehler (service), Zeitfehler (timing) und Wertefehlern (value) [FMNP94, MP94]. Alle diese Fehlertypen sind weiter verfeinert wie zum Beispiel Zeitfehler in zu früh (early) und zu spät (late).

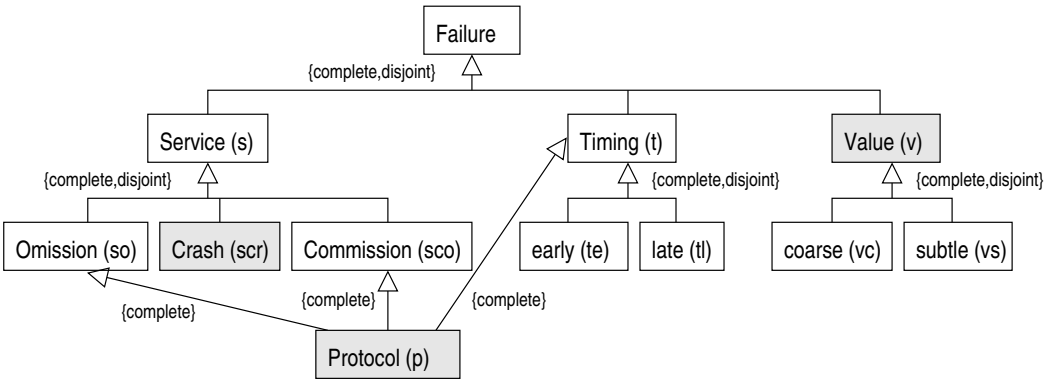


Abbildung 4.3: Beispiel für eine Fehlertyphierarchie

Wichtig für die Fehlertyphierarchie ist, dass die Verfeinerung komplett ist und die einzelnen Untertypen sich nicht überschneiden, da ansonsten die Analyse auf Basis fehlerhafter Informationen durchgeführt wird. Wenn die Untertypen sich überschneiden würden, bedeutet dies, dass verschiedene Fehlerzustände unterschiedlichen Typs unter Umständen nicht unabhängig voneinander sind. Da die später vorgestellten Analysen dies voraussetzen, würde eine Überschneidung der Fehlertypen dazu führen, dass Analyseergebnisse besser als die Wirklichkeit sind.

Wenn die Verfeinerung komplett ist, bedeutet dies, dass es keine weiteren Typen gibt, die nicht in der Hierarchie spezifiziert worden sind. Bei der Modellierung der Fehlerpropagierung einer Komponente kann dann überprüft werden, ob alle in der Fehlertyphierarchie modellierten Fehlertypen in der Fehlerpropagierung betrachtet wurden, das Fehlerpropagierungsmodell also vollständig ist.

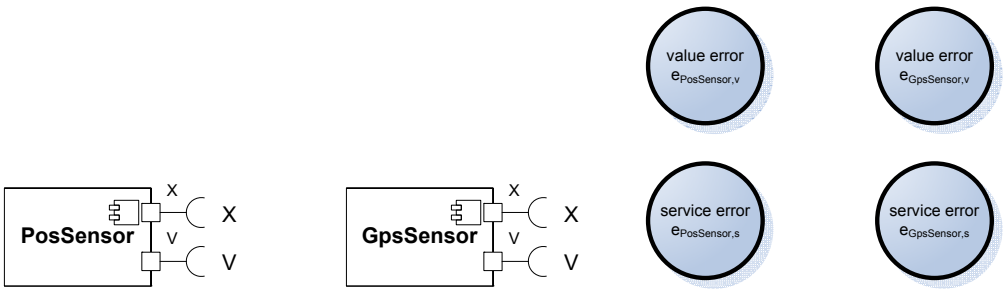
4.1.2 Basisereignisse

Basisereignisse bilden in der klassischen Fehlerbaumanalyse die Blätter der Fehlerbäume. Sie entsprechen den Fehlerzuständen in der Fehlerpathologie von Laprie (vgl. Abschnitt 2.2.2). Ein Fehlerzustand ist die beobachtbare Manifestation einer Fehlerursache innerhalb einer Komponente. Einem Basisereignis wird ein

Fehlertyp zugeordnet. Für einen Fehlerzustand können Wahrscheinlichkeiten angegeben werden, die es später ermöglichen die Wahrscheinlichkeit einer Gefahr zu berechnen.

Definition 4.2 *V_E ist die Menge der Basisereignisvariablen. Die boolesche Variable $e_{c,t} \in V_E$ beschreibt das Auftreten des Basisereignisses vom Typ $t \in T$ in der Komponente c.*

Verschiedene Basisereignisse müssen unabhängig voneinander sein. Andernfalls kann es passieren, dass die Analyse berechnet, dass eine Gefahr nur eintritt, wenn zwei als unabhängig angenommene Ereignisse zusammen eintreten. Tatsächlich würde allerdings die Gefahr schon auftreten, wenn nur eines der Basisereignisse auftritt.



(a) Komponententypen der Sensoren (b) Fehlerzustände der Sensoren

Abbildung 4.4: Fehlerzustände/Basisereignisse der beiden Sensoren

Abbildung 4.4 zeigt die Basisereignisse für die Sensoren des Beispiels. Bei beiden Sensoren können sowohl ein Fehlerzustand vom Typ Dienstfehler (service) als auch ein Fehlerzustand vom Typ Wertefehler (value) auftreten.

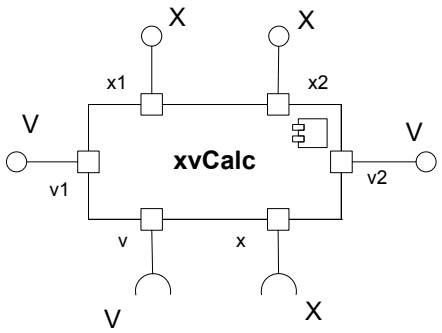
Die Basisereignisse sind für die Komponententypen definiert. Alle Instanzen eines Typs besitzen daher die gleichen Basisereignisse. Die Basisereignisse der verschiedenen Komponenteninstanzen sind unterschiedlich und damit unabhängig.

4.1.3 Ausfälle

Ein Ausfall ergibt sich als Manifestation von Basisereignissen an den Schnittstellen einer Komponente zu anderen Komponenten. In der MECHATRONIC UML sind die Ports diese Schnittstellen zur Außenwelt (vgl. Abschnitt 2.3.3). Ein Ausfall wird wie ein Basisereignis über einen Fehlertyp klassifiziert. Ausfälle einer

Komponente werden über die Kommunikationsverbindungen zwischen Komponenten an andere Komponenten propagiert. Daher werden eingehende Ausfälle und ausgehende Ausfälle unterschieden.

Definition 4.3 *Die Menge V_F ist die Menge der Ausfallvariablen. Die boolesche Variable $f_{c,p,t,d} \in V_F$ beschreibt das Auftreten des (eingehenden $d = i$ bzw. ausgehenden $d = o$) Ausfalls vom Typ $t \in T$ am Port p der Komponente c. Die Mengen V_E und V_F sind disjunkt, also gilt $V_E \cap V_F = \emptyset$.*



(a) Komponententyp xvCalc

$f_{xvCalc,x1,v,i}$ $f_{xvCalc,x1,s,i}$
$f_{xvCalc,x2,v,i}$ $f_{xvCalc,x2,s,i}$
$f_{xvCalc,v1,v,i}$ $f_{xvCalc,v1,s,i}$
$f_{xvCalc,v2,v,i}$ $f_{xvCalc,v2,s,i}$
$f_{xvCalc,x,v,o}$ $f_{xvCalc,x,s,o}$
$f_{xvCalc,v,v,o}$ $f_{xvCalc,v,s,o}$

(b) Ausfälle an Ports

Abbildung 4.5: Ausfälle an den Ports des Komponententyps xvCalc

Abbildung 4.5 zeigt die an den Ports der xvCalc-Komponente angegebenen ausgehenden und eingehenden Ausfallvariablen. Es können in diesem Beispiel Ausfälle der Fehlerarten Dienst (service (s)) und Werte (value (v)) auftreten.

4.1.4 Fehlerpropagierung innerhalb einer Komponente

Die Fehlerpropagierung innerhalb einer Komponente wird mit Fehlerbäumen modelliert. In diesen Fehlerbäumen wird beschrieben, wie die ausgehenden Ausfälle auf Basis der eingehenden Ausfälle und der Basisereignisse, verknüpft mit booleschen Operatoren, entstehen (s. Abbildung 4.6). Die eingehenden Ausfälle und Basisereignisse bilden die Blätter des Fehlerbaums, während ein ausgehender Ausfall die Wurzel des Baums bildet. Die inneren Knoten des Baumes sind boolesche Operatoren wie Und ($\wedge$), Oder ($\wedge$) und Nicht ($\neg$). Für jeden der ausgehenden Ausfälle der Komponente wird ein Fehlerbaum modelliert.

Die Blätter und inneren Knoten dieses Baums stellen den abstrakten Syntaxgraphen einer booleschen Formel dar. Der ausgehende Ausfall tritt also genau dann auf, wenn die boolesche Formel über den Basisereignissen und eingehenden Ausfällen wahr ist.

Definition 4.4 *Die Menge $O_F^c \subseteq V_F$ beinhaltet alle ausgehenden Fehler der Komponente c, die Menge $I_F^c \subseteq V_F$ alle eingehenden Ausfälle der Komponente c, die Menge $V_E^c \subseteq V_E$ die Menge aller Basisereignisvariablen der Komponente c. Es gilt $I_F^c \cap O_F^c = \emptyset$.*

Definition 4.5 *Wenn ein eingehender Ausfall, dargestellt durch die Variable $f_k \in I_F^c$, den ausgehenden Ausfall, dargestellt durch die Variable $f_l \in O_F^c$, auslösen kann, muss die Fehlerpropagierung ψ_c die boolesche Formel $f_l \Leftrightarrow f_k$ beinhalten. Mehrere Fehlerpropagierungen einer Komponente, z.B. durch unterschiedliche Fehlertypen, werden durch ein logisches Und verknüpft.*

Für die Modellierung der Fehlerpropagierung zwischen dem ausgehenden Ausfall und der booleschen Formel aus eingehenden Ausfällen und Basisereignissen wird die Bi-Implikation (bzw. Äquivalenz), also $f_l \Leftrightarrow f_k$, genutzt, obwohl eine reine Implikation naheliegender wäre. Schließlich „impliziert“ die boolesche Formel (in diesem Fall f_k) den ausgehenden Ausfall (f_l). Problematisch ist hierbei allerdings, dass bei einer reinen Implikation, also $f_l \Leftarrow f_k$, der ausgehende Fehler f_l auch bei einem nicht eingetretenen f_k auftritt. Dies liegt an der Bedeutung der Implikation, die auch insgesamt wahr wird, wenn gilt $f_k = \text{false} \wedge f_l = \text{true}$. Dies entspricht nicht einer sinnvollen Semantik der Fehlerpropagierung. Wenn man diesen Fall ausschließt, erhält man die Bi-Implikation:

$$
\begin{aligned}
& (f_l \Leftarrow f_k) \wedge \neg(\neg f_k \wedge f_l) \\
\Leftrightarrow \quad & (f_l \Leftarrow f_k) \wedge (f_k \vee \neg f_l) \\
\Leftrightarrow \quad & (f_l \Leftarrow f_k) \wedge (f_k \Leftarrow f_l) \\
\Leftrightarrow \quad & (f_l \Leftrightarrow f_k)
\end{aligned}
$$

Abbildung 4.6 zeigt die Fehlerpropagierung des xvCalc-Komponententyps für Wertefehler und Dienstfehler in Bezug auf den Port x. Dargestellt wird die Fehlerpropagierung in Form mehrerer Fehlerbäume, die ausgehende Ausfälle (als Wurzel des Baums) mit eingehenden Ausfällen, Basisereignissen und booleschen Operatoren verbinden.

Der xvCalc-Komponententyp berechnet die aktuelle Geschwindigkeit sowie Position auf der Strecke auf Basis der Daten des Positions- und des GPS-Sensors. Ein Ausfall vom Typ Wertefehler tritt am x-Port genau dann auf, wenn einer der beiden Sensoren falsche Werte liefert, da der xvCalc-Komponententyp dann nicht entscheiden kann, welche der beiden Werte richtig sind. Der linke Fehlerbaum in Abbildung 4.6 modelliert diese Fehlerpropagierung. Ein Dienstausfall tritt im Gegensatz dazu nur dann auf, wenn beide Sensoren keinen Dienst mehr anbieten

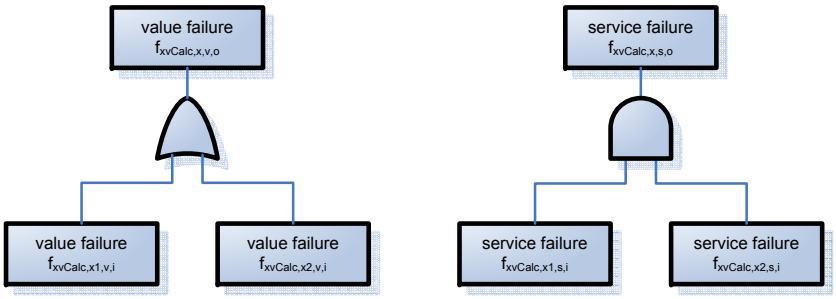


Abbildung 4.6: Fehlerpropagierung des xvCalc-Komponententyps für den Portx

(s. rechter Fehlerbaum im Bild), da die Daten von einem Sensor ausreichen. Die beschriebene Fehlerpropagierung wird, wie oben dargestellt, folgendermaßen auf boolesche Logik abgebildet:

$$
\begin{aligned}
\psi_{xvCalc}: \quad & f_{xcCalc,x,v,o} \Leftrightarrow f_{xvCalc,x1,v,i} \vee f_{xvCalc,x2,v,i} \\
\wedge \quad & f_{xcCalc,x,s,o} \Leftrightarrow f_{xvCalc,x1,s,i} \wedge f_{xvCalc,x2,s,i} \\
\wedge \quad & f_{xcCalc,v,v,o} \Leftrightarrow f_{xvCalc,v1,v,i} \vee f_{xvCalc,v2,v,i} \\
\wedge \quad & f_{xcCalc,v,s,o} \Leftrightarrow f_{xvCalc,v1,s,i} \wedge f_{xvCalc,v2,s,i}
\end{aligned}
$$

Die Fehlerpropagierung wird für einen Komponententyp auf Basis der an den Ports auftretenden Ausfälle definiert. Die Komponentenstrukturen aus Kapitel 3 erlauben eine unbestimmte Anzahl an Portinstanzen (und damit Ausfallvariablen) eines Ports eines Komponententypen. Dies bedeutet, dass die Fehlerpropagierung bei der Instantiierung des Komponententyps hinsichtlich der Anzahl der Portinstanzen angepasst werden muss.

Für den Fall eines ausgehenden Ausfalls $f_{p,o}$ am Port p wird die spezifizierte Fehlerpropagierung für jede Portinstanz p_i ebenfalls instantiiert und mit Und verknüpft, so dass sich bei n Portinstanzen auch n Fehlerpropagierungen ergeben.

Für den Fall eines eingehenden Ausfalls $f_{p,i}$ am Port p werden die Ausfallvariablen aller n Portinstanzen p_j mit Oder verknüpft in der Fehlerpropagierung ersetzt, d.h. $f_{p,i} : \bigvee_{p_j=p_1 \ldots p_n} f_{p_j,i}$.

4.1.5 Fehlerpropagierung zwischen Komponenteninstanzen

Die Fehlerpathologie beschreibt nach Laprie [Lap92] die Auswirkungen eines Ausfalls einer Komponente auf eine abhängige Komponente. Eine abhängige Komponente ist hierbei eine Komponente, die über einen Konnektor mit der ausgefallenen Komponente verbunden ist.

Die Fehlerpropagierung zwischen Komponenteninstanzen wird automatisch auf Basis der Konnektoren zwischen den zugehörigen Portinstanzen erstellt. Für die Portinstanz p einer Komponente c, die über einen Konnektor mit einer Portinstanz q einer Komponente d verbunden ist, werden für alle Typen $t \in T$ am Zielport gemäß der Richtung des Konnektors die Fehlerpropagierungsformeln $f_{c,p,t,o} \Leftrightarrow f_{d,q,t,i}$ bzw. $f_{d,q,t,o} \Leftrightarrow f_{c,p,t,i}$ erstellt.

Wenn ein Port über mehrere Konnektoren mit anderen Ports verbunden ist, muss die Fehlerpropagierung über die Konnektoren speziell betrachtet werden; wichtig ist hierbei die Richtung der Konnektoren. Laut UML [Obj07, S. 155] ist der Quellport einer Kommunikation (und damit auch der Fehlerpropagierung) der Port mit der benötigten Schnittstelle. Der Zielport ist der Port mit der angebotenen Schnittstelle.

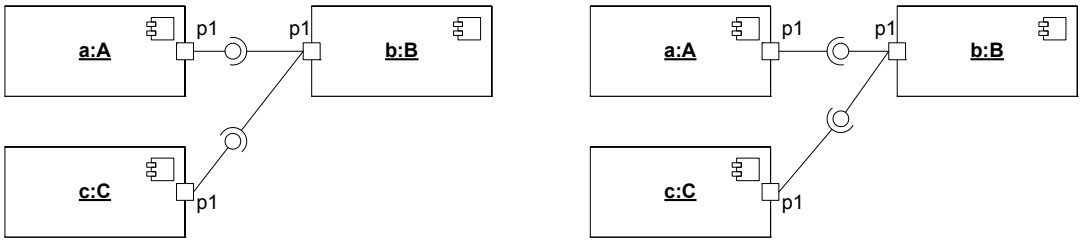


Abbildung 4.7: Fehlerpropagierung mit mehreren Konnektoren an einem Port

Abbildung 4.7 zeigt zwei Komponentenstrukturen. Auf der linken Seite ist der Port p1 der Komponente b:B ausgehend mit c:C,p1 und a:A,p1 verbunden. Ein Ausfall an b:B,p1 wird nun an beide Ports propagiert. Hierbei werden die oben dargestellten Fehlerpropagierungsformeln für die Konnektoren genutzt. Auf der rechten Seite ist die umgekehrte Situation dargestellt. Der Port p1 der Komponente b:B ist über zwei eingehende Konnektoren mit den beiden anderen Komponenten verbunden. Hierbei tritt der Ausfall $f_{b,p1,t,i}$ auf, wenn mindestens einer der beiden Ausfälle $f_{a,p1,t,o}$ und $f_{c,p1,t,o}$ auftritt. Dies wird abgebildet durch die Fehlerpropagierungsformel $f_{b,p1,t,i} \Leftrightarrow f_{a,p1,t,o} \vee f_{c,p1,t,o}$.

Ist eine andere Semantik gewünscht, wenn zum Beispiel bei Anwendung von Fehlertoleranz ein Ausfall nur auftritt, wenn beide Ausfälle auftreten, kann dies durch eine zusätzliche Komponente mit der gewünschten Fehlerpropagierung, die diese mehrfachen Konnektoren ersetzt, realisiert werden.

Bei der automatischen Erzeugung der Fehlerpropagierungsformeln werden ebenfalls die Verfeinerungsbeziehungen zwischen den verschiedenen Fehlertypen auf Basis der spezifizierten Fehlertyphierarchie beachtet. So werden nur Fehlerpropagierungsformeln zwischen kompatiblen (s. Abschnitt 4.1.1) Ausfällen an verbundenen Ports erzeugt [Sch07].

4.1.6 Gefahren

Eine Gefahr wird ebenfalls als Fehlerbaum modelliert. Abbildung 4.8 zeigt die Definition der Gefahr, die dazu führen kann, dass zwei RailCabs kollidieren. Das Topereignis, welches die Gefahr repräsentiert beschreibt, dass eines der Fahrzeuge eine falsche Geschwindigkeit hat.

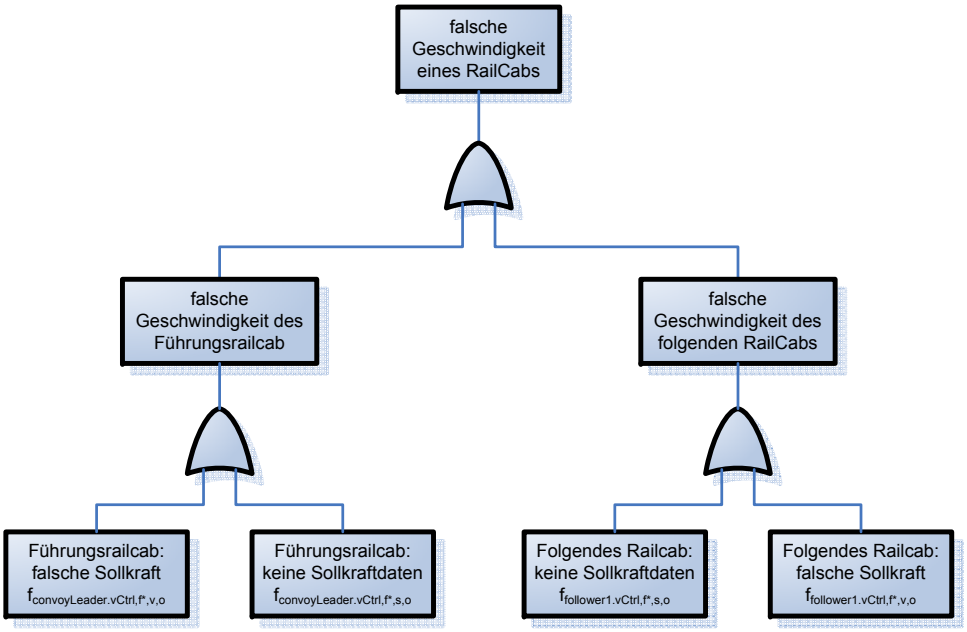


Abbildung 4.8: Gefahrendefinition für zwei RailCabs im Konvoi

Dieses Topereignis wird verfeinert, bis die Blätter des Baumes als Ausfallvariablen der Komponenteninstanzstruktur ausgedrückt werden können. Im Beispiel ist dies ein falscher Wert oder keine Werte am Port f* des Geschwindigkeitsreglers vCtrl des Führungsfahrzeuges bzw. des folgenden Fahrzeugs.

Aus den Ausfallvariablen und den booleschen Operatoren im Fehlerbaum ergibt sich dann die boolesche Formel γ als Gefahrendefinition. Der abgebildete Fehlerbaum des laufenden Beispiels ergibt Formel 4.1 auf Basis der Ausfälle an Ports der Komponenteninstanzen.

$$\gamma : \quad f_{convoyLeader.vCtrl,f^*,v,o} \vee f_{convoyLeader.vCtrl,f^*,s,o} \vee f_{follower1.vCtrl,f^*,s,o} \vee f_{follower1.vCtrl,f^*,v,o} \tag{4.1}$$

Das bedeutet, dass Gefahren auf der Komponenteninstanzstruktur definiert werden. Die Komponentenstrukturen aus Kapitel 3 erlauben eine unbestimmte

Anzahl an Portinstanzen (und damit Ausfallvariablen) eines Ports eines Komponententyps. Um die Gefahrendefinition unabhängig von einer speziellen Instanzstruktur und damit allgemeiner zu halten, können die Blätter des Fehlerbaums auch durch Ausfallvariablen an Porttypen beschrieben werden. Für eine gegebene Instanzsituation mit n Portinstanzen wird dann in der durch den Fehlerbaum definierten Gefahrenformel die Ausfallvariable für den Porttyp f_{pt} durch die Disjunktion der Ausfallvariablen der Portinstanzen f_{pi} ersetzt (s. Formel 4.2).

$$f_{pt} : \bigvee_{pi=p1...pn} f_{pi} \tag{4.2}$$

4.1.7 Abstrakte Syntax

Die abstrakte Syntax der Modelle für die Gefahrenanalyse ist mit einem Metamodell formalisiert. Das Metamodell ist mit dem Metamodell für die Komponententypen und Komponenteninstanzstrukturen (s. Abschnitt 3.1.4) durch einige Assoziationen zwischen den Metaklassen verbunden.

In der oberen Hälfte von Abbildung 4.9 sind die Metaklassen für die Definition der Ausfälle und Basisereignisse dargestellt. Die Metaklasse FailureType und die zugehörige Selbstassoziation modelliert die verschiedenen Ausfallsarten; analog wird die Metaklasse ErrorType zur Modellierung der verschiedenen Arten der Basisereignisse genutzt. Die Metaklassen Failure und Error beschreiben das mögliche Auftreten eines Ausfalls an einem Port des Komponententyps bzw. das mögliche Auftreten eines Basisereignisses innerhalb des Komponententyps. Die Metaklassen FailureOccurrence und ErrorOccurrence modellieren denselben Zusammenhang für Portinstanzen und Komponenteninstanzen. Für Basisereignisse werden Fließkommazahlen als Wahrscheinlichkeiten angegeben.

Der untere Teil der Abbildung zeigt die Metaklassen für die Definition der booleschen Formeln für die Fehlerpropagierung der Komponententypen (Metaklasse FailurePropagation). Grundlage ist die doppelte Anwendung des Composite-Entwurfsmusters, um durch die abstrakte Metaklasse BooleanFormula und der Unterklassen beliebige boolesche Formeln mit den binären Operatoren (Metaklasse BinaryExpression) AND, OR und Bi-Implikation sowie dem unären Operator NOT (Metaklasse UnaryExpression) abzubilden.

4.2 Qualitative Analyse

Die dargestellten Fehlerpropagierungsmodelle für die Komponenten sowie die Verbindung zwischen den Komponenten werden genutzt, um die zu Beginn skizzierten verschiedenen Gefahrenanalysen durchzuführen.

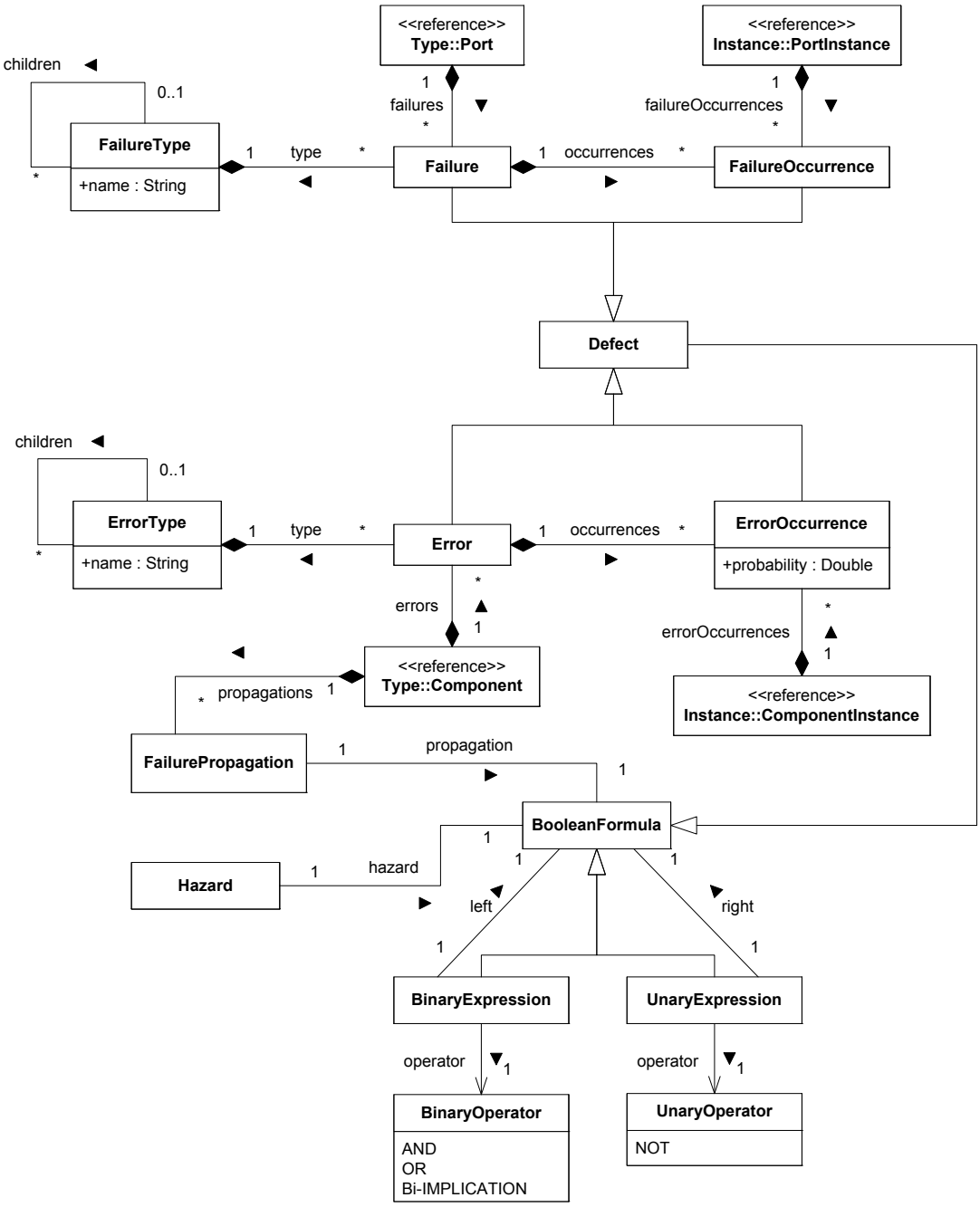


Abbildung 4.9: Metamodell für die Gefahrenanalyse

Im Rahmen der qualitativen Analyse werden zwei Fragestellungen untersucht. Zum einen wird auf Basis einer festen Menge vom Entwickler ausgewählter Basisereignisse analysiert, welche Gefahren auftreten können (im Folgenden als *Bottom-Up* bezeichnet). Zum anderen wird analysiert, welche Basisereignisse auftreten müssen, damit eine ausgewählte Gefahr auftritt (im Folgenden als *Top-Down* bezeichnet).

4.2.1 Bottom-Up

Die Bottom-Up Analyse wird durch eine Simulation der Fehlerpropagierung durchgeführt. Diese Simulation startet mit der vom Entwickler ausgewählten festen Menge an Basisereignissen und führt direkt rekursiv die Fehlerpropagierungsformeln aus. Ergebnis dieser Simulation sind alle Gefahren, die auftreten können, sowie der Verlauf der Fehlerpropagierung im System. Besonderer Vorteil dieser Analyse ist, dass die Propagierung der Fehler durch die Architektur visuell dargestellt werden kann. Dies erlaubt es dem Entwickler, die Auswirkungen von Basisereignissen in komplexen Architekturen besser zu verstehen.

Abbildung 4.10 zeigt die Fehlerpropagierung über einen Ausschnitt der Komponenteninstanzstruktur des Beispiels aus Abbildung 4.2. Ein Fehlerzustand (im Diagramm mit e dargestellt) vom Typ service im Positionssensor /posSensor:PosSensor führt zu einem entsprechenden Ausfall an den Ports x und v. Dieser Ausfall wird dann zur Komponente /xvCalc:xcCalc propagiert. Auf Grund der für den xvCalc-Komponententyp spezifizierten Fehlerpropagierung (s. Abbildung 4.6) wird der Ausfall nicht weiterpropagiert, da ein Dienstausfall am Port x nur genau dann auftritt, wenn an beiden Ports x1 und x2 Dienstausfälle auftreten. Analog gilt dies für die Ports v1, v2 und v.

Wenn zu dem oben betrachteten Dienst-Fehlerzustand im Geschwindigkeitssensor ein Fehlerzustand vom Typ service im GPS Sensors hinzukommt (wie in Abbildung 4.11 dargestellt), wird dieser ebenfalls bis zur Komponente /xvCalc:xvCalc propagiert. Da nun sowohl in Port x1 und in Port x2 ein Dienstausfall eingetreten ist, tritt auch am Port x ein Dienstausfall ein. Analog gilt dies für die Ports v1, v2 und v. Die Ausfälle werden an die Ports der übrigen Komponenteninstanzen inklusive dem Port f* der Komponenteninstanz /vCvtrl:VCtrl weiterpropagiert. Damit ist die spezifizierte Gefahr aus Abbildung 4.8 erfüllt.

4.2.2 Top-Down

Die Top-Down Analyse berechnet ausgehend von einer vom Entwickler festgelegten Gefahr (die oben in einem Fehlerbaum dargestellt wird) welche Kombinationen von Basisereignissen dafür auftreten müssen.

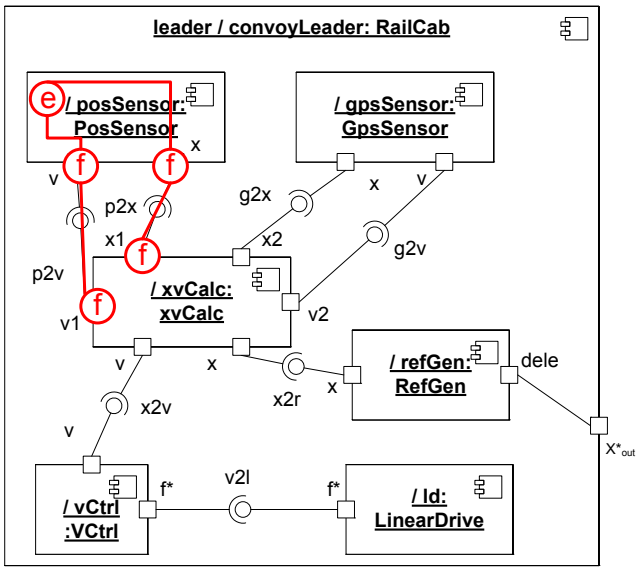


Abbildung 4.10: Fehlerpropagierung eines Dienstfehlers im Positionssensor

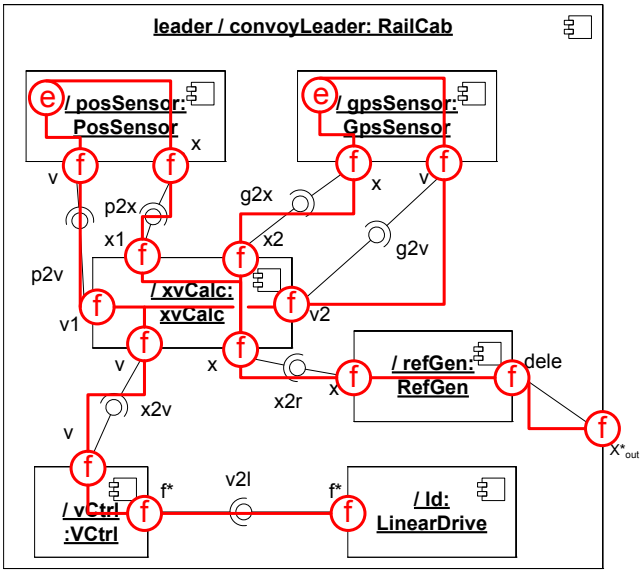


Abbildung 4.11: Fehlerpropagierung von Dienstfehlern im Geschwindigkeitssensor und im GPS

Bei der Top-Down Analyse wird die Fehlerpropagierung des Gesamtsystems, also die Fehlerpropagierungen aller Komponenteninstanzen und den Konnektoren

zwischen diesen, in einer booleschen Formel dargestellt. Diese boolesche Formel ist eine symbolische Kodierung der Fehlerpropagierung, da alle Abläufe der Fehlerpropagierung in einer booleschen Formel kodiert sind. Dies und die gewählte Darstellung als Binary Decision Diagram (BDD) [Ake78, Bry86, Bry92] erlaubt es, große Fehlerpropagierungsmodelle zu analysieren.

4.2.2.1 Fehlerpropagierung des Gesamtsystems

Die Fehlerpropagierung innerhalb einer Komponente und zwischen Komponenten wird mittels boolescher Logik (s. Abschnitt 4.1) modelliert. Die Fehlerpropagierung eines Systems ψ_s ergibt sich dabei nach [Rau03] durch eine Und-Verknüpfung der einzelnen Formeln ψ_c aller Komponenten und Konnektoren $c \in C$ des Systems.

$$\psi_s : \bigwedge_{c \in C} \psi_c \tag{4.3}$$

Die Und-Verknüpfung bewirkt, dass alle Fehlerpropagierungsformeln der einzelnen Komponenten und der Konnektoren wahr werden müssen, damit die Gesamtfehlerpropagierung wahr wird. Dies ist für die Gefahrenanalyse die gewünschte Semantik.

Als vereinfachtes Beispiel dienen die zwei folgenden Fehlerpropagierungen und deren Und-Verknüpfung:

$$e_1 \Leftrightarrow f_1 \tag{4.4}$$

$$f_2 \Leftrightarrow f_1 \tag{4.5}$$

$$(e_1 \Leftrightarrow f_1) \wedge (f_2 \Leftrightarrow f_1) \tag{4.6}$$

Die erste Fehlerpropagierung wird für die folgenden Belegungen der Variablen wahr: $e_1 = \text{true}, f_1 = \text{true}$ und $e_1 = \text{false}, f_1 = \text{false}$. Das heißt, wenn das Basisereignis e_1 eintritt, genau dann tritt auch der Ausfall f_1 auf und umgekehrt. Analog gilt dies auch für die zweite Fehlerpropagierung.

Für die Gesamtfehlerpropagierung der dritten Formel sollte natürlich gelten, dass wenn das Basisereignis e_1 auftritt, genau dann auch f_1 und daraus folgend f_2 auftritt. Wenn e_1 nicht auftritt, dann treten auch beide Ausfälle nicht auf. Durch die Und-Verknüpfung wird genau diese gewünschte Semantik umgesetzt, wie die Wahrheitstabelle in Tabelle 4.1 zeigt:

Für die Gefahrenanalyse wird die zu analysierende Gefahr mit der Fehlerpropagierung kombiniert. Hierzu wird die boolesche Formel der Gefahr γ ebenfalls, analog zu obiger Argumentation, mit einem booleschen Und mit der Fehlerpropagierung des Systems verknüpft.

e_1	f_1	f_2
true	true	true
false	false	false

Tabelle 4.1: Wahrheitstabelle der Formel 4.6

$$\psi : \psi_s \wedge \gamma \tag{4.7}$$

Nach dem Erstellen der Fehlerpropagierung des Gesamtsystems werden die Ausfallvariablen f_i daraus entfernt. Die Ausfallvariablen sind nur dafür da, um die Basisereignisse miteinander zu verknüpfen und sind eine Auswirkung aus dem komponentenbasierten Ansatz zur Modellierung der Fehlerpropagierung. Für die Gefahrenanalyse sind diese Ausfallvariablen nicht mehr wichtig, da nur die Kombinationen der Basisereignisse, die eine Gefahr eintreten lassen, interessieren. Die Ausfallvariablen f_i werden, wie in [Rau03] beschrieben, dann per Exist-Quantifikation aus der Formel entfernt.

$$\psi^{\exists} : \exists f_i \, (\psi) \tag{4.8}$$

Die Exist-Quantifikation auf booleschen Formeln wird mittels Ersetzung ($[y/x]$; ersetzt x durch y) auf eine Disjunktion abgebildet:

$$\exists v : \phi \qquad \text{equals} \qquad \phi[\text{true}/v] \vee \phi[\text{false}/v] \tag{4.9}$$

Das Result ist eine Formel, die nur noch die Basisereignisse beinhaltet. Alle Belegungen der Variablen der Basisereignisse, die die Formel wahr werden lassen, sind die Voraussetzungen für das Auftreten der Gefahr. Für die Betrachtung der Sicherheit ist nicht nur eine einzelne Kombination der Basisereignisse relevant, sondern alle Kombinationen, die die Gefahr auftreten lassen. Diese in der Literatur *Implikanten* genannten Kombinationen können mit Hilfe von Binary Decision Diagrams (BDDs) [Ake78, Bry86, Bry92] berechnet werden.

Binary Decision Diagrams sind eine Datenstruktur zur Darstellung und Manipulation von booleschen Formeln. Ein BDD ist ein gerichteter azyklischer Graph dessen Knoten aus den Variablen einer booleschen Formel besteht. Zusätzlich gibt es die Terminalknoten *true* (1) und *false* (0). Genau ein Knoten eines BDDs hat keinen Vorgänger und wird daher Wurzelknoten genannt. Dieser Knoten repräsentiert die boolesche Formel. Alle Knoten bis auf die Terminalknoten haben zwei Nachfolger, die entweder über die *low*-Kante oder die *high*-Kante erreicht werden. Die *low*- bzw. *high*-Kante beschreibt hierbei, dass die Knotenvariable den Wert *false* bzw. *true* hat. Dies wird als Shannon-Zerlegung bezeichnet.

Reduced Ordered Binary Decision Diagrams (ROBDDs) sind eine Variante von BDDs die bestimmte Eigenschaften bzgl. der Struktur einhalten. Diese Eigenschaften ermöglichen, eine boolesche Formel bzgl. des Speicherplatzes oftmals effizient darzustellen und boolesche Operationen auf dieser Datenstruktur durchzuführen. Hierbei ist zum einen die Variablenordnung auf allen Pfaden des BDDs von Wurzelknoten zu Terminalknoten gleich (Ordered). Zum anderen werden redundante Informationen aus dem Graphen entfernt. Für weitere Informationen zu BDDs und ROBDDs sei auf [DB98] verwiesen.

Formel 4.10 zeigt eine Beispielformel für eine Fehlerpropagierung, die aus den drei Fehlerzuständen e_1, e_2, e_3 besteht.

$$\psi^{\exists} : e_1 \wedge (e_3 \vee (\neg e_2 \wedge \neg e_3)) \tag{4.10}$$

Abbildung 4.12 zeigt die BDD-Repräsentation von $\psi^{\exists}$ aus Formel 4.10.

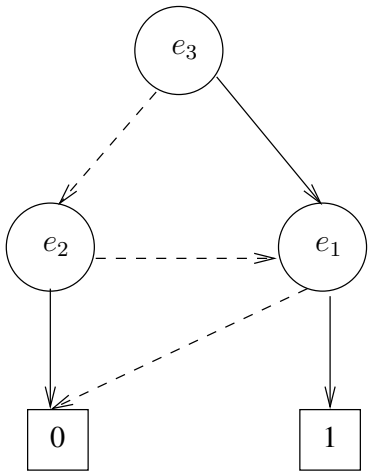


Abbildung 4.12: BDD Darstellung der Formel 4.10

4.2.2.2 Berechnung der Implikanten

Wenn der BDD für das Auftreten einer Gefahr nur noch aus dem false (0) Terminalknoten besteht, gibt es keine Kombination aus auftretenden Basisereignissen, die zu dieser Gefahr führen. Wenn dies nicht der Fall ist, müssen für die durch den BDD repräsentierte boolesche Formel die Implikanten bestimmt werden. Implikanten beschreiben im Kontext der qualitativen Gefahrenanalyse Fehlerszenarien, die zu einer Gefahr führen.

Abbildung 4.13 zeigt das Metamodell von BDDs mit weiteren Klassen für die Speicherung von Implikanten. Die Klassen und Assoziationen sind hierbei analog

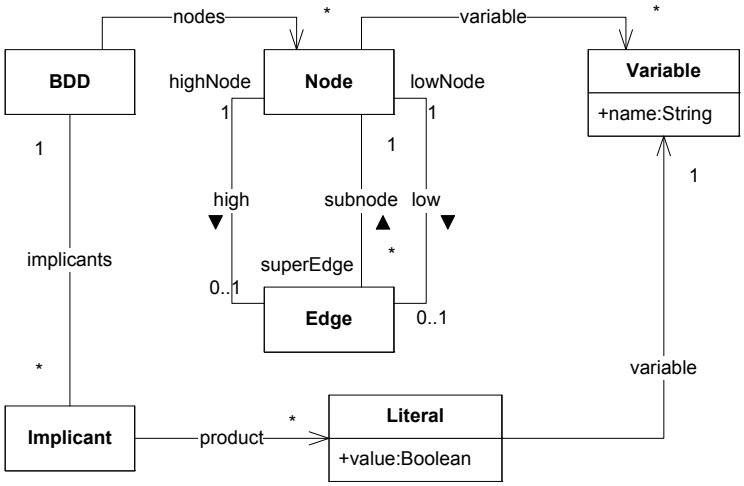


Abbildung 4.13: Metamodell für Binary Decision Diagrams

zur obigen textuellen Darstellung. Wenn die Fehlerpropagierungsformel in einem BDD abgebildet worden ist, werden mit Hilfe eines rekursiven Algorithmus auf einer Objektstruktur, die den BDD repräsentiert, die Implikanten berechnet.

Algorithmus 1 berechnet rekursiv die Implikanten einer booleschen Formel auf Basis deren BDD-Darstellung. Der Algorithmus startet auf dem *true*-Knoten. Für jede von diesem Knoten ausgehende Kante k zu Knoten mit einem geringeren Index wird an Hand des Typs der Kante (*high* bzw. *low*) die Belegung der Variable (Literal genannt [Rau03]) für den Implikanten gewählt (*true* bzw. *false*), einer Liste der Belegungen hinzugefügt und dann der Algorithmus rekursiv auf dem über die Kante k erreichbaren Knoten aufgerufen. Damit für jeden Pfad unterschiedliche Listen der Belegungen genutzt werden, wird die Liste beim rekursiven Aufruf geklont.

Rekursionsabbruch ist das Erreichen des Wurzelknotens. Hier wird für die auf dem Pfad erstellten Belegungen der Variablen ein Implikant erzeugt und an das Diagramm angehängt.

Abbildung 4.14 zeigt die Berechnung der Implikanten des Beispiels aus Abbildung 4.12. Ausgehend vom 1-Terminalknoten wird die high-Kante zum Knoten mit der Variable e_1 ausgewählt (s. Zeile 7) und in Zeile 9 der Liste das positive Literal e_1 hinzugefügt. Auf den Knoten mit der Variable e_1 zeigen zwei Kanten. Es folgt der rekursive Aufruf der Funktion auf dem Knoten der Variable e_1. Dieser Knoten besitzt zwei Kanten, die auf ihn zeigen. Die Schleife in Zeile 7 iteriert nun über diese beiden Kanten: (1) Die erste führt dazu, dass das positive Literal e_3 dem Produkt hinzugefügt wird; der darauf folgende Rekursionsaufruf auf dem obersten Knoten des BDD führt zum Rekursionsabbruch in Zeile 2 und

Algorithmus 1 : void createImplicants (BDDDiagram diagram, Node node, Set<Literal> list)

```
   begin
2      if node.superEdge = ∅ then
           Implicant implicant ← new Implicant();
           implicant.product ← list;
           diagram.implicants ← diagram.implicants ∪ implicant;
       else
7          foreach edge ∈ node.superEdge do
               Literal literal ← new Literal();
9              literal.value ← edge.highNode ≠ null;
               literal.variable ← node.variable;
               list ← list ∪ literal;
               if edge.highNode then
                   createImplicants(diagram, edge.highNode, list.clone());
               else
                   createImplicants(diagram, edge.lowNode, list.clone());
   end
```

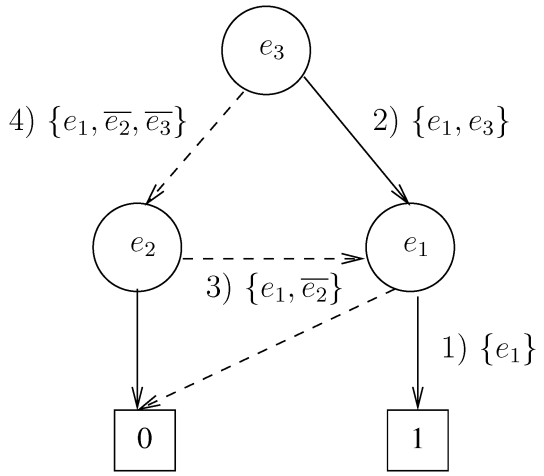


Abbildung 4.14: Beispiel für die Implikantenberechnung

resultiert in dem Implikanten $\{e_1, e_3\}$. (2) die zweite Kante vom Knoten e_1 führt dazu, dass das negative Literal $\overline{e_2}$ zum Produkt hinzugeführt wird. Der Rekursionsaufruf resultiert im Hinzufügen des negativen Literals $\overline{e_3}$, so dass sich als

zweiter Implikant $\{e_1, \overline{e_2}, \overline{e_3}\}$ ergibt.

Die dargestellte Berechnung der Implikanten ist aus Sicht der Gefahrenanalyse prinzipiell ausreichend, da sie alle Fehlerszenarien enthält. Von besonderem Interesse sind Primärimplikanten (prime implicants). Diese entsprechen minimalen Fehlerszenarien, die zu einer Gefahr führen (vgl. [RD97]). Es ist daher sinnvoll Primärimplikanten zu berechnen, da sie möglicherweise weniger Szenarien ergeben, da mehrere Implikanten durch einen Primärimplikanten ersetzt werden können. Hierfür kann das Quine-McCluskey-Verfahren eingesetzt werden [OQ55, McC56].

Die Berechnung der minimalen Menge an Primärimplikanten wird als NP-hart angesehen [Bol90, S. 108]. Der optimalen Berechnung nach Quine-McCluskey sind daher Heuristiken [BSVMH84, Rud84, Sed08] vorzuziehen, die versuchen sich der minimalen Menge an Primärimplikanten anzunähern.

4.2.2.3 Modelle mit Zyklen

Die bisher dargestellten Modelle der Fehlerpropagierung auf Basis boolescher Formeln können für nicht-zyklische Komponentenmodelle mit obiger Analysetechnik basierend auf BDDs analysiert werden.

Ein Fehlerpropagierungsmodell ψ_s ist hierbei zyklisch, wenn eine Ausfallvariable indirekt über mehrere Fehlerpropagierungsformeln von sich selbst abhängt. Dies resultiert dann darin, dass eine Belegung möglich ist, die die boolesche Formel der Fehlerpropagierung wahr macht, also die Gefahr eintritt, in der nur die Ausfallvariablen wahr sind, ohne dass ein Basisereignis eintritt.

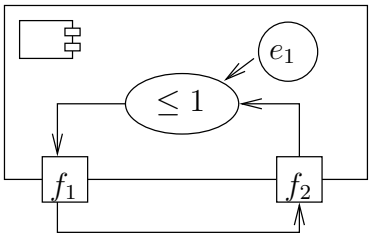


Abbildung 4.15: Zyklisches Fehlerpropagierungsmodell

Das Beispiel aus Abbildung 4.15 beinhaltet die folgenden beiden Fehlerpropagierungen:

$$\psi_1 : f_1 \Leftrightarrow f_2 \vee e_1 \tag{4.11}$$

$$\psi_2 : f_2 \Leftrightarrow f_1 \tag{4.12}$$

In der folgenden Tabelle sind die Belegungen der Variablen f_1, f_2, e_1 dargestellt, die die Fehlerpropagierung dieses Beispiels $(\psi_1 \wedge \psi_2)$ wahr werden lassen.

f_1	f_2	e_1
false	false	false
true	true	true
true	true	false

Tabelle 4.2: Wahre Belegungen der Fehlerpropagierung aus Abbildung 4.15

Diese Belegungen zeigen, dass $f_1 f_2$ ein Implikator der Fehlerpropagierung ist. Das heißt, dass wenn f_1 und f_2 wahr werden, die gesamte Fehlerpropagierung wahr wird, unabhängig von den Belegungen der restlichen Variablen (in diesem Beispiel e_1). Ein weiterer Implikator ist $\overline{f_1 f_2 e_1}$. Dies bedeutet, dass die Fehlerpropagierung erfüllt ist, wenn sowohl die Ereignisvariable e_1 als auch die Ausfallvariablen f_1 und f_2 nicht wahr sind.

Dies entspricht dem erwarteten Verhalten der Fehlerpropagierung, dass wenn das Ereignis e_1 nicht auftritt auch die Ausfälle f_1 und f_2 nicht auftreten. Die Belegung $f_1 f_2 \overline{e_1}$ erfüllt zwar die Formel, entspricht aber nicht dem erwarteten Fehlerverhalten. Hierbei werden sowohl der Ausfall f_1 als auch der Ausfall f_2 wahr, ohne dass das Ereignis e_1 eingetreten ist.

Dies liegt an dem Zyklus im oben dargestellten Fehlerpropagierungsmodell. Dieser Zyklus führt dazu, dass Belegungen der Variablen die Fehlerpropagierung erfüllen, bei denen die Ausfallvariablen wahr werden, ohne dass Ereignisse eintreten.

Zyklenüberprüfung Das Fehlerpropagierungsmodell des Gesamtsystems ψ_s besteht aus einer Menge von Fehlerpropagierungsformeln ψ_c. Die Funktion $ancestors_{\psi_s}(f)$ (Algorithmus 3) liefert die Menge aller Defekte (Basisereignisse und Ausfälle) zurück, von denen die Ausfallvariable f direkt oder indirekt abhängt. Gibt es ein f, so dass gilt $f \in ancestors_{\psi_s}(f)$, dann ist das Fehlerpropagierungsmodell zyklisch (vgl. [Rau03]).

Zyklenbehandlung Modelle, in denen die Fehlerpropagierung Zyklen aufweist, können nach [Rau03] ebenfalls mit obiger Analysetechnik analysiert werden. Grundlage dabei ist, dass die Implikanten der Fehlerpropagierung (s. Tabelle 4.2), die nicht dem erwarteten Fehlerverhalten entsprechen, bei der Analyse erkannt und verworfen werden. Dies funktioniert für Fehlerpropagierungen die monoton (steigend oder fallend) sind. Die Zyklenbehandlung kann effizient direkt auf der BDD-Repräsentation durchgeführt werden [Rau03]. Der dort vorgestellte

Algorithmus 2 : $ancestorsRek_{\psi_s}$ (Defect f, Set<Defect> set)

```
begin
    set ← set ∪ f;
    foreach d ∈ allDefectsInPropagationFormula (f) do
        if d ∉ set ∧ d instanceof Failure then
            ancestorsRek_ψs(d, set);
end
```

Algorithmus 3 : Set<Variable> $ancestors_{\psi_s}$ (Defect f)

```
begin
    Set<Defect> set ← ∅;
    ancestorsRek_ψs (f, set);
    return set;
end
```

BDD-Operator wurde im Rahmen von [Sch07] in den hier vorgestellten Ansatz integriert. Für weitergehende Details wird auf [Rau03, Sch07] verwiesen.

4.2.2.4 Abstraktionen

Die vorgestellten Modelle und Analysen erlauben zusätzliche Abstraktionen [GTS04], die genutzt werden können, um bei größeren Modellen von Details zu abstrahieren. Die Abstraktion muss hierbei pessimistisch sein. Dies bedeutet für die Gefahrenanalyse, dass wenn im nicht abstrahierten Modell eine Gefahr auftreten kann, dann muss dies auch für das abstrahierte Modell gelten.

Zwei Aspekte der vorgestellten Modelle können abstrahiert werden: (1) Die in Abschnitt 4.1.1 vorgestellte Fehlertyphierarchie erlaubt von den spezifischen Fehlertypen zu abstrahieren und diese durch die allgemeineren Fehlertypen zu ersetzen. (2) Die hierarchische Struktur der Strukturmodelle erlaubt ebenfalls eine Abstraktion. So kann eine unterlagerte Komponentenstruktur durch eine abstrahierte Fehlerpropagierung der umfassenden Komponente ersetzt werden. Die Umsetzungen der Abstraktionen werden in [GTS04] beschrieben.

4.3 Quantitative Analyse

Während in der qualitativen Analyse nur das prinzipielle Auftreten einer Gefahr betrachtet wird, wird bei der quantitativen Analyse die Wahrscheinlichkeit einer

Gefahr sowie das resultierende Risiko betrachtet.

Die Gefahrenwahrscheinlichkeit wird auf der bei der Top-Down Analyse erstellten booleschen Formel $\psi^{\exists}$ (s. Abschnitt 4.2.2) berechnet. Diese Formel besteht nur noch aus den Ereignisvariablen e_i. Wenn für jede dieser Ereignisvariablen eine Wahrscheinlichkeit für den Eintritt dieses Ereignisses bekannt ist, kann unter der Voraussetzung, dass die Ereignisse voneinander unabhängig sind, analog zu quantitativen Analysen bei der Fehlerbaumanalyse die Wahrscheinlichkeit des Eintritts der Gefahr berechnet werden.

Die Berechnung wird direkt auf der BDD-Repräsentation von $\psi^{\exists}$ durchgeführt [CM93, GT06]. Die Funktion probability berechnet rekursiv, startend mit dem Wurzelknoten des BDDs, die Wahrscheinlichkeit der Formel.

$$probability(\text{false}) = 0 \tag{4.13}$$

$$probability(\text{true}) = 1 \tag{4.14}$$

$$\begin{aligned} probability(e_i) = {} & p(e_i) \cdot probability(e_i.high) \\ & + (1 - p(e_i)) \cdot probability(e_i.low) \end{aligned} \tag{4.15}$$

Die Wahrscheinlichkeit der Terminalknoten true und false sind 1 bzw. 0. Bei diesen Knoten bricht die Rekursion auch ab. Die Wahrscheinlichkeit des Knotens für eine Ereignisvariable wird auf Basis der Wahrscheinlichkeit für diese Ereignisvariable ($p(e_i)$) sowie den bereits berechneten Wahrscheinlichkeiten für die über die high- und low-Kanten erreichbaren Knoten berechnet. Da die Funktion probability bei dieser Berechnung nur einmal für jeden Knoten aufgerufen werden muss, wenn bereits erfolgte Berechnungen für einen Knoten gespeichert werden, wird die Berechnung in linearer Zeit bzgl. der Anzahl der Knoten im BDD durchgeführt.

Abbildung 4.16 zeigt die Berechnung der Wahrscheinlichkeit auf dem BDD der Formel $e_1 \wedge (e_3 \vee \neg e_3 \neg e_2)$. Da für die beiden Terminalknoten die Wahrscheinlichkeiten nach den Formeln 4.13 und 4.14 definiert sind, wird im ersten Schritt die Wahrscheinlichkeit des Knotens e_1 berechnet. Diese Wahrscheinlichkeit beträgt nach Formel 4.15 0.3. Danach werden die Wahrscheinlichkeiten für den Knoten e_2 und nachfolgend für den Knoten e_3 berechnet. Damit ist die Wurzel des BDDs erreicht und die Wahrscheinlichkeit für den Eintritt der Gefahr ψ beträgt somit $probability(\psi) = 0.24$.

Das Risiko wird vereinfacht als Gefahrenwahrscheinlichkeit multipliziert mit dem Schaden des zugehörigen Unfalls berechnet [TCS98]:

$$risk(\psi) = severity(\psi) \cdot probability(\psi) \tag{4.16}$$

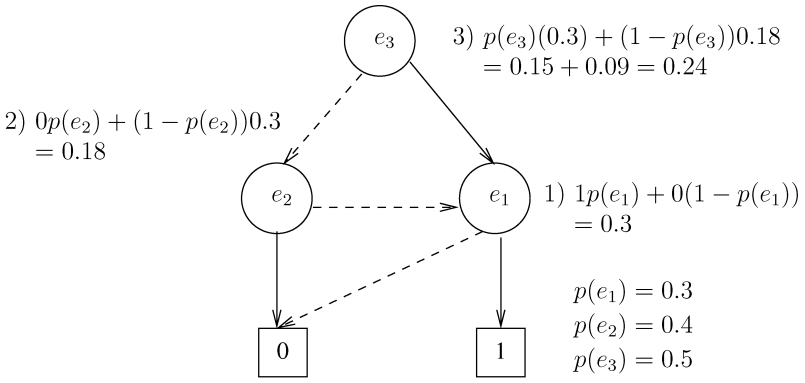


Abbildung 4.16: Wahrscheinlichkeitsberechnung auf einem BDD [GT06]

4.4 Ansatzpunkte zur Verbesserung der Sicherheit

Die Analysen der Sicherheit aus den vorherigen Abschnitten ermöglichen die Entscheidung, ob Anforderungen bzgl. der Gefahrenwahrscheinlichkeit bzw. des Risikos eingehalten werden. Falls die Anforderungen nicht eingehalten werden, ist eine wichtige Information, welche Stellen in der Architektur im Besonderen zur Nichteinhaltung der Anforderungen beitragen bzw. welche Stellen die erfolgversprechendsten Ansatzpunkte für eine Verbesserung der Sicherheit darstellen.

Die Gefahrenwahrscheinlichkeit beruht ursächlich zum einen auf den Wahrscheinlichkeiten der einzelnen Basisereignisse sowie auf deren unterschiedlichen Typen. Daraus ergeben sich die beiden Ansatzpunkte für die Verbesserung der Sicherheit: (1) Identifikation der Basisereignisse, deren Verminderung der Wahrscheinlichkeit die größte Verbesserung der Gefahrenwahrscheinlichkeit zur Folge hat und (2) Identifikation der Stellen an denen die Anwendung von Techniken zur Umwandlung des Fehlertyps (z.B. die Erkennung von Wertefehlern und daraus folgender Ausfall durch ein Self-Checking Pair [Sto96]) die größten Auswirkungen auf die Gefahrenwahrscheinlichkeit hat. Beide Punkte sind in der Terminologie zur Gefahrenbehandlung von Leveson [Lev95] der Gefahrenreduktion zuzuordnen, also der Verringerung der Wahrscheinlichkeit des Eintretens einer Gefahr.

4.4.1 Reduzierung der Ereigniswahrscheinlichkeit

In Abschnitt 4.3 wurde erläutert, wie die Top-Down Analyse die Gefahrenwahrscheinlichkeit unter Nutzung der Wahrscheinlichkeiten der Basisereignisse berechnet. Diese Berechnung wird nun zur Bestimmung von einem oder mehreren möglichen Ansatzpunkten für die Reduzierung der Ereigniswahrscheinlichkeit angepasst. Heuristisch werden die Wahrscheinlichkeiten der Basisereignisse verän-

dert. Durch einen zusätzlichen Optimierungsschritt werden dann die Basisereignisse bestimmt, deren heuristische Veränderungen in der größten Verbesserung der Gefahrenwahrscheinlichkeit resultieren.

Anstelle der Wahrscheinlichkeit $p(e_i)$ für ein Basisereignis e_i wird die Funktion $p_\alpha(e_i)$ genutzt. Die Funktion $p_\alpha(e_i)$ beschreibt eine mit dem festen Faktor α geringere Wahrscheinlichkeit des Basisereignisses e_i, wenn die Variable $a_i \in \{0, 1\}$ den Wert 1 hat. Die Variablen a_i werden genutzt, um die Verringerung der Wahrscheinlichkeit für einzelne Basisereignisse an- bzw. auszuschalten. Formel 4.17 stellt diesen Zusammenhang mittels einer Fallunterscheidung dar.

$$p_\alpha(e_i) = \begin{cases} p(e_i) & \text{für} \quad a_i = 0 \\ \alpha \cdot p(e_i) & \text{für} \quad a_i = 1 \end{cases} \tag{4.17}$$

Wenn nun die quantitative Analyse aus Abschnitt 4.3 mit dieser Formel genutzt wird, ergibt sich eine Formel, die hinsichtlich der Variablen a_i bzgl. der Wahrscheinlichkeit (Formel 4.18) bzw. des Risikos (Formel 4.19) einer Menge von Gefahren Ψ minimiert werden kann. Ergebnis sind die Belegungen der Variablen a_i, die beschreiben, an welcher Stelle die Verringerung der Wahrscheinlichkeit eines Basisereignisses mit dem Faktor α die größte Verringerung der Summe der Gefahrenwahrscheinlichkeiten bzw. der Risiken bringt.

$$min \sum_{\psi_i \in \Psi} probability(\psi_i) \tag{4.18}$$

$$min \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i) \tag{4.19}$$

Das Optimierungsproblem kann noch durch eine zusätzliche Bedingung für die Belegungen der Variablen a_i erweitert werden. Formel 4.20 beschreibt, dass die Anzahl der auf 1 gesetzten a_i auf den Wert max_a beschränkt wird. Der Sonderfall $max_a = 1$ beschreibt die Suche nach dem (einen) Basisereignis, dessen Verringerung der Wahrscheinlichkeit, die Summe der Gefahrenwahrscheinlichkeiten bzw. der Risiken am meisten vermindert.

$$\sum_{i=1...n} a_i \leq max_a \tag{4.20}$$

Falls der für die Minimierung verwendete Optimierer keine Fallunterscheidung unterstützt, kann statt Formel 4.17 Formel 4.21 für $p_\alpha(e_i)$ genutzt werden.

$$p_\alpha(e_i) = p(e_i) \cdot (1 - a_i) + \alpha \cdot p(e_i) \cdot a_i \tag{4.21}$$

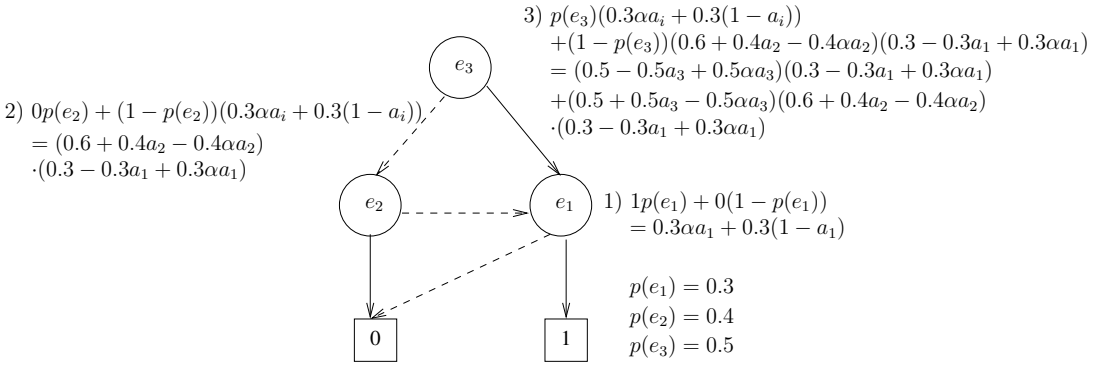


Abbildung 4.17: Wahrscheinlichkeitsberechnung mit der Funktion $p_\alpha(e_i)$

Abbildung 4.17 zeigt die Nutzung von $p_\alpha(e_i)$ in Bezug auf das Beispiel aus Abbildung 4.16. Für den Parameter $\alpha = 0.01$ ergibt sich das Optimierungsproblem aus Formel 4.22. Die Lösung dieses Optimierungsproblems, also eine Belegung der Variablen a_1, a_2, a_3 ergibt dann den Fehlerzustand dessen Wahrscheinlichkeitsverringerung um den Faktor $\alpha = 0.01$ den größten Effekt auf die Verringerung der Gefahrenwahrscheinlichkeit hat.

$$\begin{aligned} min: \quad & (0.5 - 0.495 * a_3) * (0.3 - 0.297 * a_1) \\ & + (0.5 + 0.495 * a_3) * (0.6 + 0.396 * a_2) * (0.3 - 0.297 * a_1) \end{aligned} \tag{4.22}$$

Die Belegung $a_1 = 1, a_2 = 0, a_3 = 0$ mit $max_a = 1$ ergibt mit 0.00240 die geringste Wahrscheinlichkeit. In diesem Beispiel ist es also sinnvoll zu versuchen, die Wahrscheinlichkeit des Fehlerzustands e_1 zu verringern. Dies kann zum Beispiel durch die Anwendung von Fehlertoleranztechniken, wie z.B. redundante Systemelemente, umgesetzt werden. In [TBG04, TSG04, TG05, Tic06, THMD08] wurden Techniken zur Spezifikation von Fehlertoleranztechniken beschrieben, mit denen die Struktur und Fehlerpropagierung eines Systems um Fehlertoleranztechniken erweitert wird.

4.4.2 Änderung der Fehlertypen

Der Effekt des Einsatzes von Techniken zur Veränderung des Fehlertyps kann auf der Komponentenstruktur simuliert und dann mit Hilfe der quantitativen Analyse berechnet werden. Hierzu wird jeder Konnektor zwischen zwei Komponenten durch eine spezielle Analysekomponente ersetzt (s. Abbildung 4.18). Diese Analysekomponente besitzt eine Fehlerpropagierung, die den zu ändernden

Fehlertyp in einen anderen Fehlertyp, z.B. Crash, transformiert. Dies geschieht nicht immer, sondern nur in Zusammenhang mit einem Flag, das diese spezielle Fehlerpropagierung einschaltet; ansonsten hat diese Analysekomponente die triviale Fehlerpropagierung – ein eingehender Fehler wird mit dem selben Fehlertyp direkt wieder ausgegeben.

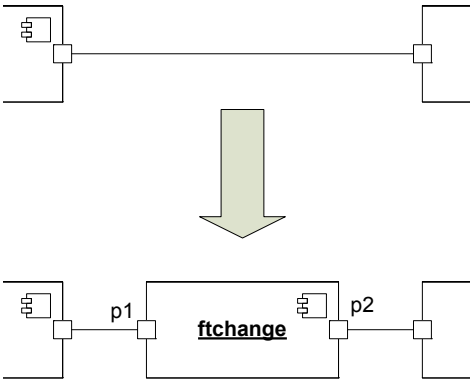


Abbildung 4.18: Vorgehen zur Analyse der Effekte durch Änderung der Fehlertypen

Die in Abbildung 4.18 abgebildete Komponente ftchange ersetzt den oberen Konnektor. Diese Komponente besitzt die in Formel 4.23 dargestellte Fehlerpropagierung, die in Abhängigkeit von der Belegung der Variablen $fc_i \in \{0, 1\}$ den Fehlertyp ändert oder nicht.

$$\psi_{ftchange} : \begin{cases} f_{ftchange,p1,i,t} \Leftrightarrow f_{ftchange,p2,o,t}, & \text{für} \quad fc_i = 0 \\ f_{ftchange,p2,i,t} \Leftrightarrow f_{ftchange,p1,o,t} & \\ f_{ftchange,p1,i,t} \Leftrightarrow f_{ftchange,p2,o,crash}, & \text{für} \quad fc_i = 1 \\ f_{ftchange,p2,i,t} \Leftrightarrow f_{ftchange,p1,o,crash} & \end{cases} \tag{4.23}$$

Wenn nun die quantitative Analyse auf der resultierenden Komponentenstruktur durchgeführt wird, ist das Ergebnis eine Formel berechnet aus den Wahrscheinlichkeiten der Basisereignisse $p(e_i)$ und den Fehlertypänderungsvariablen fc_i. Die resultierende Formel wird dann minimiert, um die Belegung der Variablen fc_i zu bestimmen, welche die Summe der Gefahrenwahrscheinlichkeiten (s. Formel 4.24) bzw. der Risiken (s. Formel 4.25) minimiert.

$$min \sum_{\psi_i \in \Psi} probability(\psi_i) \tag{4.24}$$

$$min \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i) \tag{4.25}$$

Das Optimierungsproblem kann noch durch eine zusätzliche Bedingung für die Belegungen der Variablen fc_i erweitert werden. Formel 4.26 beschreibt eine Bedingung, welche die Anzahl der auf 1 gesetzten fc_i, also die Anzahl der eingesetzten Komponenten zur Fehlertypänderung, auf den Wert max_{fc} beschränkt.

$$\sum_{i=1...n} fc_i \leq max_{fc} \tag{4.26}$$

4.5 Zusammenfassung

In diesem Kapitel wurde aufbauend auf den Komponenteninstanzstrukturen aus Kapitel 3 eine komponentenbasierte Gefahrenanalyse vorgestellt. Auf Basis von Verhaltensmodellen der einzelnen Komponententypen, die vom funktionalen Verhalten abstrahieren und nur das Fehlerpropagierungsverhalten enthalten, ergibt sich das Fehlerpropagierungsverhalten des Gesamtsystems durch die Komponenteninstanzstruktur.

Dieses Fehlerpropagierungsverhalten erlaubt in Zusammenhang mit der Gefahrendefinition qualitative und quantitative Analysen. Mit zwei qualitativen Analysen wird zum einen bestimmt, welche Gefahren von einer fest ausgewählten Menge an Basisereignissen verursacht werden. Zum anderen werden für eine fest ausgewählte Gefahr die verschiedenen Kombinationen der Fehlerzustände bestimmt, die zu dieser Gefahr führen. Die quantitative Analyse berechnet auf Basis der (statistisch unabhängigen) Wahrscheinlichkeiten der Fehlerzustände die Gefahrenwahrscheinlichkeit und das Risiko.

Für die Bestimmung, welche Stellen in der Komponenteninstanzstruktur im Besonderen zur Nichteinhaltung der Anforderungen beitragen bzw. welche Stellen die erfolgversprechendsten Ansatzpunkte für eine Verbesserung der Sicherheit mit Hilfe von Fehlertoleranztechniken darstellen, wurden des Weiteren zwei Heuristiken vorgestellt.

5 Gefahrenanalyse von selbstoptimierenden Systemen

Selbstoptimierende Systeme ändern ihr Verhalten zur Laufzeit. Dies kann durch Änderung von Parametern als auch durch Strukturanpassung geschehen [FGK+04]. Eine Verhaltensanpassung kann natürlich Auswirkungen auf die Sicherheit des Systems haben, z.B. können sich das Auftreten bzw. die Wahrscheinlichkeiten von Gefahren ändern. Die Verhaltensanpassung, insbesondere die Strukturanpassung, zur Laufzeit muss also bei der Gefahrenanalyse mitbetrachtet werden.

Verwandte Arbeiten im Bereich der Gefahrenanalyse betrachten keine rekonfigurierbare Systeme [FM93, Wal05, KLM03, Gru03b, GKP05, KGF07, PM99, ORS06] oder unterstützen nicht eine regelbasierte Modellierung der Rekonfiguration [GOR06], wie sie die in Kapitel 3 vorgestellten Komponentenstorydiagramme ermöglichen.

Die in Kapitel 4 vorgestellte Gefahrenanalyse basiert auf der Struktur des Systems, in Form der mit der in Kapitel 3 vorgestellten Modellierungssprache spezifizierten Komponenteninstanzstruktur. Da selbstoptimierende Systeme die Struktur anpassen, um ihr Verhalten anzupassen, wird die vorgestellte Gefahrenanalyse für diese Art von Systemen erweitert.

Um dies zu erreichen, muss die Gefahrenanalyse in Bezug auf sich ändernde Strukturen, die zur Laufzeit in unterschiedlichen Konfigurationen resultieren, angepasst werden. Abbildung 5.1 zeigt eine Erweiterung des Übersichtsbilds 4.1 aus Kapitel 4 zur komponentenbasierten Gefahrenanalyse. Die Gefahrenanalyse für selbstoptimierende Systeme bestimmt also automatisch auf Basis der Fehlerpropagierung der verschiedenen Konfigurationen die Kombinationen der Fehlerzustände sowie die zugehörigen Konfigurationen, die zur Gefahr führen.

Neben der reinen Fehlerpropagierung in verschiedenen Konfigurationen ist es im Zusammenhang mit selbstoptimierenden Systemen z.B. wichtig zu wissen, in welcher Konfiguration die Gefahrenwahrscheinlichkeit am höchsten ist. Dies stellt dann eine obere Schranke dar, egal wie sich das System zur Laufzeit verhält.

Die einzelnen Konfigurationen können in der Mechatronic UML entweder mittels Hybriden Rekonfigurationscharts (s. Abschnitt 2.3.4) oder mit den in dieser Arbeit beschriebenen Komponentenstorydiagrammen (s. Abschnitt 3.2)

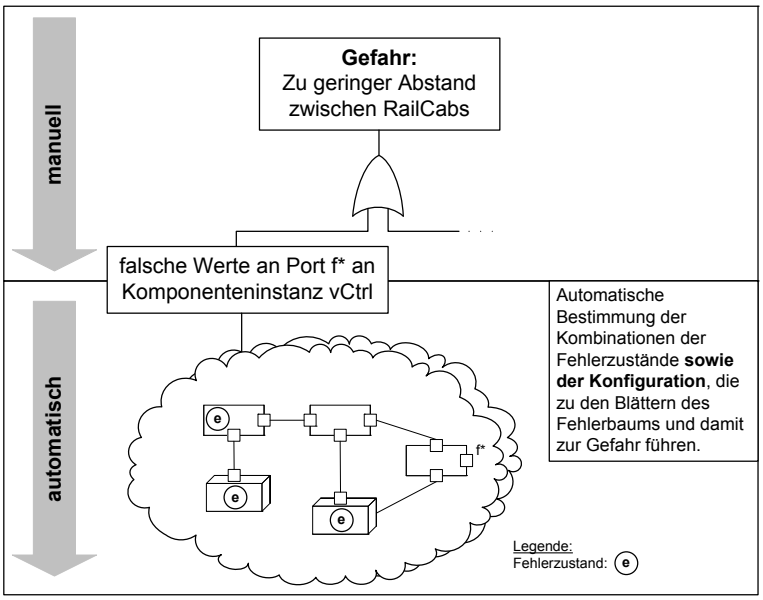


Abbildung 5.1: Erweiterung der komponentenbasierten Gefahrenanalyse für selbstoptimierende Systeme

modelliert werden.

Im nächsten Abschnitt wird die für die Gefahrenanalyse von selbstoptimierenden Systemen notwendige Bestimmung der erreichbaren Konfigurationen beschrieben. In Abschnitt 5.2 werden die in Kapitel 4 vorgestellten Analysemodelle für die Fehlerpropagierung hinsichtlich der Unterstützung für unterschiedliche Konfigurationen erweitert. Abschnitt 5.3 beschreibt die qualitative Gefahrenanalyse für selbstoptimierende Systeme während Abschnitt 5.4 eine Darstellung quantitativer Analysen beinhaltet. Das Kapitel schließt mit einer Zusammenfassung.

5.1 Erreichbare Konfigurationen

Eine Gefahrenanalyse für selbstoptimierende Systeme kann nicht einfach alle auf Basis der Strukturmodelle möglichen Konfigurationen betrachten, sondern darf nur die Konfigurationen der Analyse zugrunde legen, die zur Laufzeit auf Grund des modellierten Verhaltens erreicht werden.

Würde die Gefahrenanalyse die Analyse nicht auf erreichbare Konfigurationen beschränken, könnte der Fall eintreten, dass eine Konfiguration als unsicher, im Sinne einer höheren Gefahrenwahrscheinlichkeit als in den Anforderungen gefor-

dert, betrachtet wird, obwohl diese Konfiguration zur Laufzeit nie erreicht wird. Dies könnte dann in unnötigen Entwurfsänderungen resultieren, die durch eine Gefahrenanalyse, die nur die erreichbaren Konfigurationen betrachtet, vermieden werden.

Zuerst werden in Abschnitt 5.1.1 für Komponenteninstanzstrukturen mit Hybriden Rekonfigurationscharts die erreichbaren Zustände unter geschickter Ausnutzung der Verfeinerungsbeziehungen zwischen Echtzeitkoordinationsmustern und den Hybriden Rekonfigurationscharts der Mechatronic UML beschrieben. Nachfolgend wird in Abschnitt 5.1.2 auf die erreichbaren Konfigurationen von Komponentenstorydiagrammen eingegangen.

5.1.1 Hybride Rekonfigurationscharts

Mit Hybriden Rekonfigurationscharts werden das diskrete Echtzeitverhalten sowie die zustandsabhängigen unterlagerten Komponentenstrukturen selbstoptimierender Komponenten modelliert. Bei einer Gefahrenanalyse müssen alle Zustände bzw. Zustandskombinationen der beteiligten Komponenten betrachtet werden, da diese Zustandskombinationen jeweils einer Komponentenstruktur entsprechen, die bei der Gefahrenanalyse betrachtet wird.

Die Modellierung des Echtzeitverhaltens in der Mechatronic UML basiert auf einem kompositionalen Ansatz (s. Abschnitt 2.3.2). Das Echtzeitverhalten für einzelne Koordinationsszenarien wird mit Echtzeitkoordinationsmustern und Real-Time Statecharts modelliert und verifiziert. Nachfolgend werden die einzelnen Komponenten als Verfeinerung der Muster entwickelt. Dieser kompositionale Ansatz ermöglicht die Verifikation großer Systeme, da die Muster sowie die Komponenten einzeln verifiziert werden können. Er kann nun auch für die Gefahrenanalyse ausgenutzt werden, um nicht den Zustandsraum des Gesamtsystems für die Gefahrenanalyse aufzubauen, sondern nur die der Muster und der einzelnen Komponenten. Aufbauend auf den Zustandsräumen der Muster und Komponenten wird dann mit Hilfe der Verfeinerung inkrementell der Zustandsraum beschränkt auf die Konfigurationen aufgebaut. In jedem dieser inkrementellen Schritte werden nicht mehr benötigte Informationen aus dem aufzubauenden Zustandsraum entfernt. Insgesamt ergibt sich dadurch eine effizientere Berechnung der Konfigurationen des Systems als beim Aufbau des kompletten Zustandsraums des Gesamtsystems. Der auf die Konfigurationen eingeschränkte Zustandsraum kann allerdings immer noch sehr groß werden, so dass der Ansatz auf Systeme mittlerer Größe angewandt werden kann.

5.1.1.1 Beispiel

Das RailCab-Beispiel zur Konvoifahrt aus den vorherigen beiden Kapiteln wird zur Erläuterung der Gefahrenanalyse für selbstoptimierende Systeme aufgegriffen. Das Beispiel wird um die Optimierung des Komforts durch eine Übertragung von Streckendaten als zweiten Aspekt ergänzt. Dies wird realisiert durch eine Kommunikation mit einer Abschnittskontrolle. Diese Streckendaten ermöglichen einem Fahrzeug die Beschleunigung nach links/rechts sowie in Z-Richtung (oben/unten) durch die aktive Federung und Neigung zu regeln und so für den Fahrgast einen höchstmöglichen Komfort zu erreichen. Abbildung 5.2 zeigt obiges Szenario des RailCab-Systems mit zwei Fahrzeugen und einer Abschnittskontrolle.

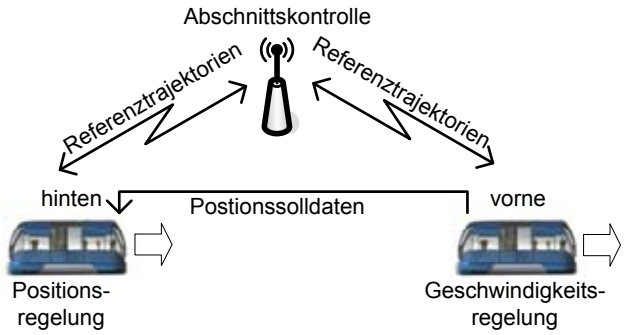


Abbildung 5.2: Laufendes Beispiel

Konvoiorganisation Das Verhalten der Fahrzeuge für die Konvoiorganisation ist (beschränkt auf 2 Fahrzeuge) mittels Echtzeitkoordinationsmustern sowie Hybriden Rekonfigurationscharts modelliert [GTB+03]. Im Muster wird das Kommunikationsprotokoll für das Bilden und das Auflösen eines Konvois beschrieben. Das Hybride Rekonfigurationschart für ein RailCab verfeinert die Rollen des Musters und beinhaltet im Synchronisationschart die Rekonfiguration der unterlagerten Komponentenstruktur. Wenn das Fahrzeug alleine oder als führendes Fahrzeug in einem Konvoi fährt, wird eine Geschwindigkeitsregelung eingesetzt. Als nicht führendes Fahrzeug in einem Konvoi wird eine erweiterte Abstandsregelung eingesetzt, bei der das führende Fahrzeug jedem nachfolgenden Fahrzeug im Konvoi periodisch Sollpositionen zuschickt [HVB+05, GHH+06, HTS+08a].

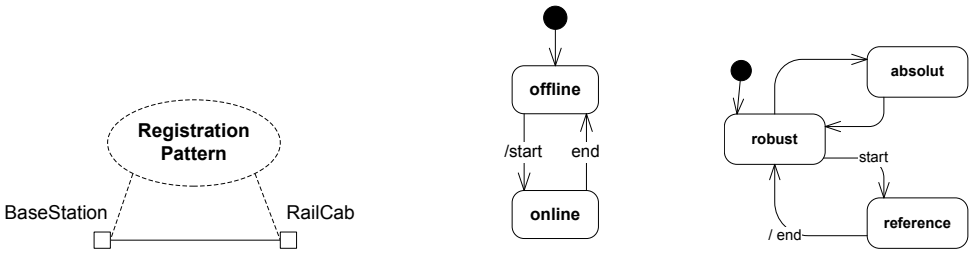
Regelung der Fahrzeugdynamik Um einen höchstmöglichen Komfort zu erreichen, werden von der Abschnittskontrolle so genannte Referenztrajektorien an

die Fahrzeuge geschickt, die es ermöglichen die aktive Federung optimal zu regeln [GBSO04, TMV06]. Ein RailCab kann unterschiedliche Regler nutzen, je nachdem welche Daten bzw. Sensoren zur Verfügung stehen.

5.1.1.2 Modelle

Die Struktur und das Verhalten wird mit zwei Echtzeitkoordinationsmustern sowie zwei Komponententypen und zugehörigen Hybriden Rekonfigurationscharts modelliert.

Registration Muster Abbildung 5.3 zeigt Struktur und Verhalten des Echtzeitkoordinationsmusters für die Kommunikation zwischen Fahrzeug und Abschnittskontrolle (BaseStation). Das Senden der Referenztrajektorien kann in der Abschnittskontrolle an- und ausgeschaltet werden. Die RailCab Rolle besitzt unterschiedliche Zustände, je nachdem ob von der Abschnittskontrolle Referenztrajektorien zur Verfügung stehen bzw. welche Daten von den Sensoren korrekt gelesen werden können.



(a) Musterstruktur (b) BaseStation Rolle (c) RailCab Rolle

Abbildung 5.3: Registration Koordinationsmuster

Konvoi Muster Abbildung 5.4 zeigt das Konvoi Echtzeitkoordinationsmuster [GTB+03]. Das Muster besteht aus den zwei Rollen FrontRole und RearRole. Erstere beschreibt das Verhalten des vorderen Fahrzeugs in einem Konvoi zweier Fahrzeuge; letztere beschreibt das Verhalten des hinteren Fahrzeugs. Die Rollen beschreiben das Koordinationsverhalten bzgl. der Bildung und des Auflösens eines Konvois durch den Austausch von Nachrichten. Die Entscheidung, ob ein Konvoi gebildet wird oder nicht, ist nicht im Verhalten enthalten, sondern wird bei der Verfeinerung des Rollenverhaltens für die Komponenten ergänzt.

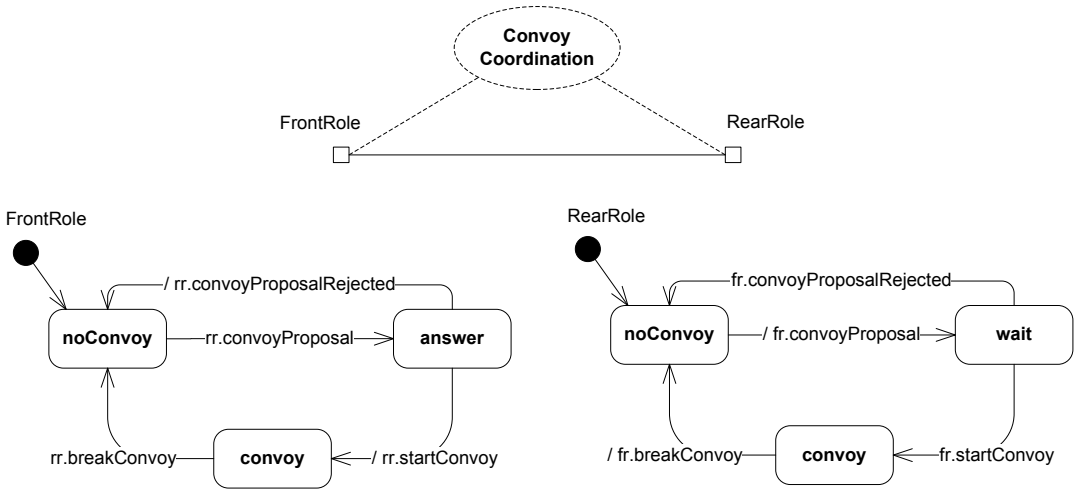
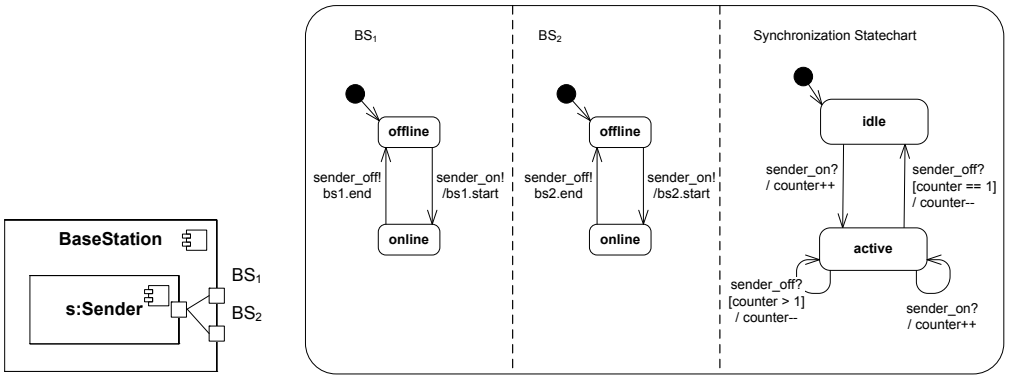


Abbildung 5.4: Konvoi Koordinationsmuster

BaseStation Komponententyp Auf Basis der oben dargestellten Echtzeitkoordinationsmuster zeigt Abbildung 5.5 eine Instanz des Komponententyps BaseStation sowie das spezifizierte Hybride Rekonfigurationschart. Der Komponententyp BaseStation verfeinert die Rolle BaseStation des Musters Registration zweimal zu den Ports BS_1 und BS_2, um zwei Fahrzeuge mit Referenztrajektorien zu versorgen. Diese Ports sind mit der unterlagerten Sender Komponenteninstanz verbunden, welche die Daten verschickt. Das Synchronisationschart der BaseStation Komponente besteht aus zwei Zuständen idle und active, zwischen denen, je nachdem ob Daten verschickt werden sollen, gewechselt werden. Im Zustand active ist die unterlagerte Sender Komponente aktiv, im Zustand idle nicht.

RailCab Komponententyp Abbildung 5.6 zeigt die Struktur des RailCab Komponententyps. Der Komponententyp verfeinert die RearRole und die FrontRole des Koordinationsmusters für die Konvoiorganisation zu den Ports RR' und FR'. Des Weiteren verfeinert er die RailCab Rolle des Registration Musters. Zusätzlich besitzt der Komponententyp mehrere unterlagerte Komponenten für die Regelung der Beschleunigung in Fahrtrichtung sowie des Aufbaus links und rechts sowie in Z-Richtung (oben/unten).

Diese unterlagerten Komponenten sind nicht immer aktiv. Abbildung 5.7 zeigt das Hybride Rekonfigurationschart für den RailCab Komponententyp. Neben den verfeinerten Rollen beinhaltet es zwei Synchronisationscharts sCC und sBS. Ersteres beschreibt die Synchronisation für die Ports RR' und FR', so dass analog zu [GTB+03] ein Fahrzeug nur entweder vorne oder hinten in einem Konvoi fährt. Im



(a) BaseStation Komponententyp

(b) BaseStation Statechart

Abbildung 5.5: Der BaseStation Komponententyp ist für das Versenden der Referenztrajektorien zuständig.

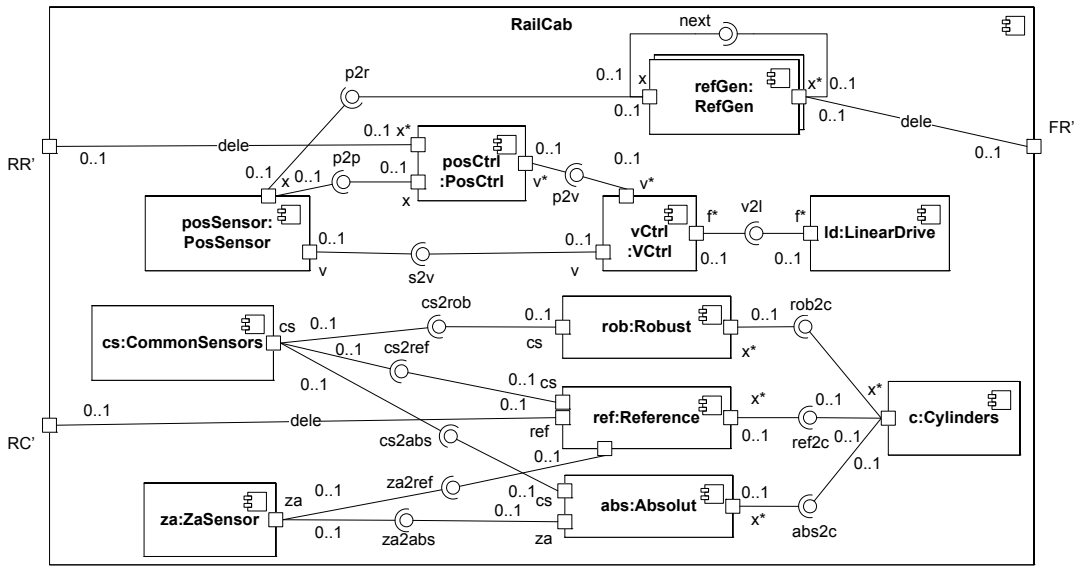


Abbildung 5.6: RailCab Komponententyp

Zustand noConvoy wird nur der Geschwindigkeitsregler, Positionssensor und der Linearmotor genutzt. Die Zustände convoyRear und intermediate2 haben zusätzlich noch den Positionsregler, um als letztes Fahrzeug in einem Konvoi mitfahren zu können. Die Zustände convoyFront und intermediate für ein im Konvoi führendes Fahrzeug enthalten neben dem Geschwindigkeitsregler, dem Positionssensor und

dem Linearmotor zusätzlich eine RefGen Komponenteninstanz zur periodischen Berechnung der Sollpositionen für das nachfolgende Fahrzeug.

Die Zustände robust, absolute und reference im Synchronisationschart sBS enthalten die unterlagerten Komponenten mit den gleichen Namen sowie die gemeinsamen Sensoren (CommonSensors) und die Zylinder. Die Zustände absolute und reference enthalten zusätzlich die ZaSensor Komponente. Der Eingangsport RC', über den die Referenztrajektorie empfangen wird, ist im Zustand reference aktiv.

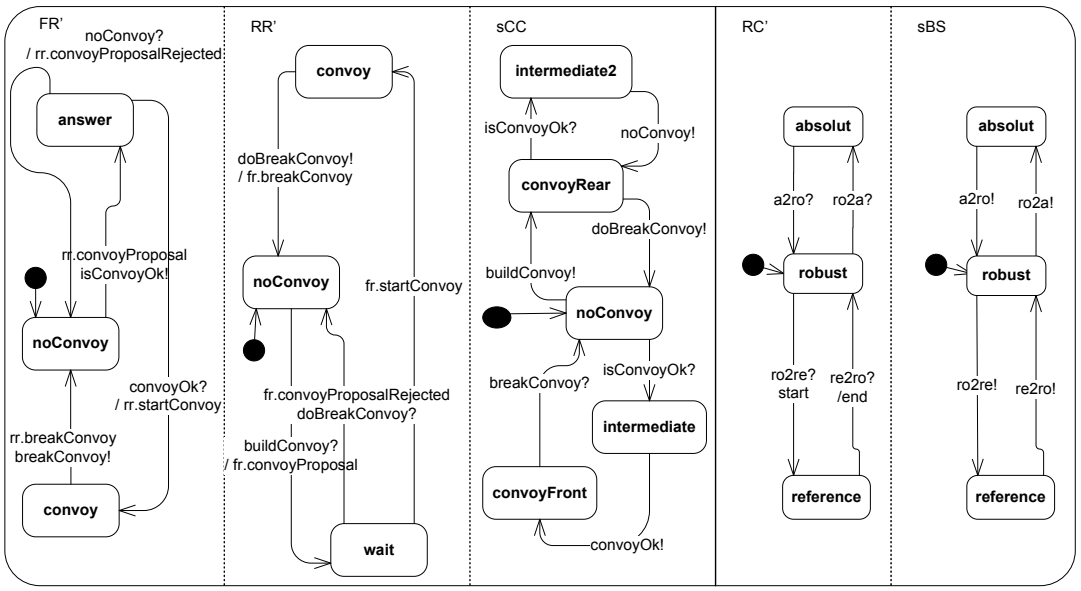


Abbildung 5.7: Verhalten des RailCab Komponententyps

5.1.1.3 Inkrementelle Erreichbarkeitsanalyse der Konfigurationen

Um eine Gefahrenanalyse für das laufende Beispiel durchzuführen, müssen die verschiedenen erreichbaren Konfigurationen bestimmt werden. Abbildung 5.8 zeigt die Komponenteninstanzstruktur für das laufende Beispiel. Zwei RailCab Komponenteninstanzen kommunizieren mit einer BaseStation auf Basis des durch das Registration Muster spezifizierten Protokolls. Zusätzlich läuft die Konvoibildung zwischen den Fahrzeugen über das Konvoi Muster.

Im Verhalten einer Komponente wird durch den kompositionalen Ansatz der Mechatronic UML (s. Abschnitt 2.3) zwischen Synchronisationscharts, welche die unterlagerte Konfiguration der Komponenten in einem Zustand modellieren,

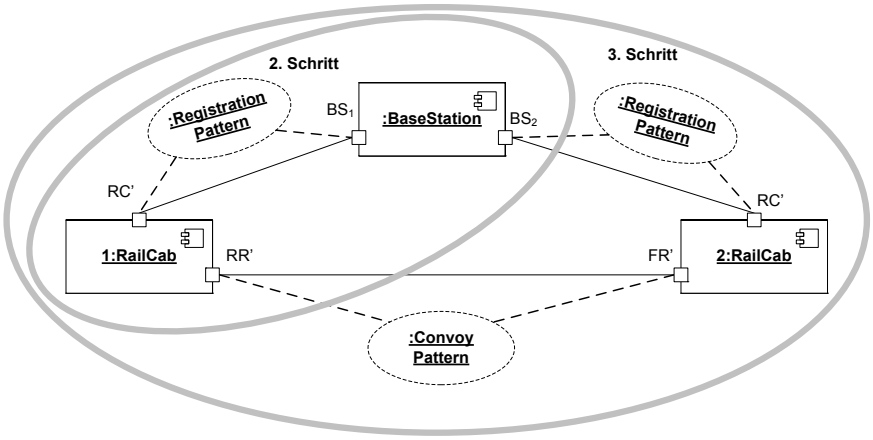


Abbildung 5.8: Komponentenstruktur des Beispiels und die inkrementellen Schritte zur Berechnung der erreichbaren Konfigurationen

und Port Statecharts, die die Kommunikation mit anderen Komponenteninstanzen spezifizieren und Verfeinerungen der Musterrollen sind, unterschieden. Für die Analyse der erreichbaren Konfigurationen sind nur die Zustandskombinationen aller Synchronisationscharts der beteiligten Komponenten relevant, da sie die erreichbaren Konfigurationen definieren. Die Zustände der Port Statecharts im erreichbaren Zustandsraum sind für die Konfigurationen irrelevant. Die Port Statecharts werden daher nur gebraucht, um die Zustandskombinationen der Synchronisationscharts zu berechnen, die über ein Koordinationsmuster miteinander verbunden sind. Folglich ist die grundlegende Idee für die Erreichbarkeitsanalyse der Konfigurationen, dass schrittweise die Zustandskombinationen der Systemkomponenten aufgebaut werden und so früh wie möglich Informationen über die Zustände der Port Statecharts aus dem Zustandsraum entfernt werden, das heißt, nachdem alle Komponenteninstanzen, die durch ein Muster verbunden wurden, verarbeitet wurden.

Die Erreichbarkeitsanalyse der Konfigurationen wird wie in Abbildung 5.8 abgebildet schrittweise durchgeführt. Für jeden Komponententyp und jedes Echtzeitkoordinationsmuster werden die erreichbaren Zustände einzeln berechnet (z.B. mit NuSMV [CCG+02]). Dann werden die erreichbaren Zustände zweier Komponenten mit dem Kreuzprodukt kombiniert unter der Einschränkung, dass die resultierenden Zustandskombinationen der Zustände der Port Statecharts in den erreichbaren Zustandskombinationen der zugehörigen Musterrollen enthalten sind. Hierbei wird die Verfeinerungsrelation zwischen Zuständen der Rollen Statecharts und der Port Statecharts aus [GTB+03] ausgenutzt. Nach diesem Schritt werden die Zustandsinformationen über die Port Statecharts, die bereits

verarbeitet wurden und daher nicht mehr benötigt werden, aus den erreichbaren Zuständen entfernt. Die resultierende Menge an erreichbaren Zustandskombinationen wird dann schrittweise analog mit den erreichbaren Zuständen der restlichen Komponenteninstanzen kombiniert, bis alle Komponenteninstanzen verarbeitet wurden.

Für das laufende Beispiel in Abbildung 5.8 werden zuerst die erreichbaren Zustände der Komponententypen BaseStation und RailCab sowie der Registration und Convoy Echtzeitkoordinationsmuster berechnet. Danach werden im zweiten Schritt die erreichbaren Zustandskombinationen der Komponenteninstanzen :BaseStation und 1:RailCab unter Ausnutzung der Verfeinerungsbeziehung des kompositionalen Ansatzes, wie in Definition 5.3 dargestellt, berechnet.

Definition 5.1 *Die erreichbaren Zustände der parallelen Ausführung zweier Komponenteninstanzen C und D mit den Rollenstatecharts $S_1^C, \ldots, S_k^C$ und $S_1^D, \ldots, S_l^D$ sowie den Synchronisationsstatecharts S_s^C und S_s^D auf Basis der transitiv-reflexiven Hülle t^* ($t \in T_{C\|D}$, die Transitionsrelation der parallelen Ausführung von C und D [GST+03]) ist definiert als*

$$\begin{aligned} RS_{C\|D} = &\{(s_1^c, \ldots, s_k^c, s_s^c, s_1^d, \ldots, s_l^d, s_s^d) \in \{RS_C \times RS_D\} | \\ &(s_{1,0}^c, \ldots, s_{k,0}^c, s_{s,0}^c, s_{1,0}^d, \ldots, s_{k,0}^d, s_{s,0}^d) t^* (s_1^c, \ldots, s_k^c, s_s^c, s_1^d, \ldots, s_l^d, s_s^d)\} \end{aligned}$$

Definition 5.2 *Die Abbildung $map_r : S \rightarrow S$ gibt für jeden Zustand eines Port Statecharts den verfeinerten Zustand aus dem Rollen Statechart basierend auf der Verfeinerungsbeziehung $\sqsubseteq$ aus [GTB+03, Gie03] zurück.*

Definition 5.3 *Die Menge $RS'_{C\|D}$ kann von den erreichbaren Zuständen der Komponenten C und D (RS_C und RS_D) unter Ausnutzung der erreichbaren Zustände eines Echtzeitkoordinationsmuster P (RS_P) berechnet werden, ohne dass der komplette Erreichbarkeitsgraph von $C\|D$ wie in Definition 5.1 aufgebaut werden muss:*

$$\begin{aligned} RS'_{C\|D} = &\{(s_1^c, \ldots, s_k^c, s_s^c, s_1^d, \ldots, s_l^d, s_s^d) | \\ &(s_1^c, \ldots, s_k^c, s_s^c) \in RS_C \wedge (s_1^d, \ldots, s_l^d, s_s^d) \in RS_D \wedge \\ &(map_r(s_1^c), \ldots, map_r(s_k^c), map_r(s_1^d), \ldots, map_r(s_l^d)) \in RS_P\} \end{aligned}$$

Die strukturelle Verfeinerungsbeziehung $\sqsubseteq$ aus [GTB+03] garantiert, dass $RS_{C\|D} \subseteq RS'_{C\|D}$ gilt. Es gilt nicht die Gleichheit der Mengen, da die Verfeinerungsbeziehung erlaubt, dass ein Zustand im Rollen Statechart in mehrere Zustände im Port Statechart verfeinert wird. Da RS' inkrementell berechnet

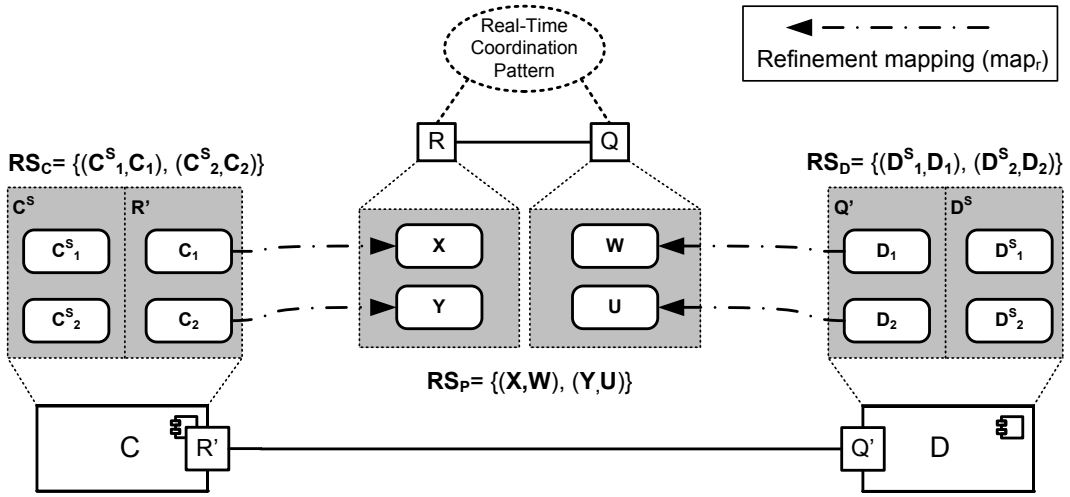


Abbildung 5.9: Beispiel zur Berechnung der erreichbaren Konfigurationen mit $RS'_{C\|D}$

werden kann, wird $RS'_{C\|D}$ anstatt von $RS_{C\|D}$ in den inkrementellen Schritten der Erreichbarkeitsanalyse genutzt.

Abbildung 5.9 zeigt an Hand eines stark vereinfachten Beispiels, wie mit Hilfe von Definition 5.3 die erreichbaren Zustandskombinationen von $C\|D$ berechnet werden.

In der Mitte der Abbildung ist ein Echtzeitkoordinationsmuster mit den zwei Rollen R und Q dargestellt. Für jede dieser Rollen (sowie für den Konnektor, welcher hier aus Gründen der Übersichtlichkeit nicht dargestellt wird) gibt es ein Real-Time Statechart mit Zuständen X und Y sowie W und U. Mittels einer Erreichbarkeitsanalyse auf diesen beiden Real-Time Statecharts werden nun die erreichbaren Zustandskombinationen RS_P, in diesem Beispiel (X,W) und (Y,U), berechnet.

Die links und rechts in der Abbildung dargestellten Komponenteninstanzen C und D setzen das Echtzeitkoordinationsmuster um und verfeinern dabei die Rollen R bzw. Q. Des Weiteren besitzen beide Komponenteninstanzen Hybride Rekonfigurationscharts als Verhaltensmodelle bestehend aus jeweils zwei parallelen Statecharts (C^S und R' sowie D^S und Q'). Das innere parallele Statechart (R' bzw. Q') ist die Verfeinerung der Musterrolle; das äußere ist das Synchronisationschart, in dessen Zuständen die unterlagerten Komponentenstrukturen eingebettet sind. Ein Wechsel der Zustände dieser Synchronisationscharts resultiert also in einer Rekonfiguration zur Laufzeit. Auf Basis der Verfeinerungsrelation aus [GTB+03, GST+03] existiert für die Zustände der verfeinerten Rollenstatecharts

ein Mapping zu den Zuständen der Rollenstatecharts: $map_r : C_1 \mapsto X, C_2 \mapsto Y, D_1 \mapsto W, D_2 \mapsto U$. Des Weiteren werden mit Hilfe einer Erreichbarkeitsanalyse für die einzelnen Komponenten ebenfalls alle möglichen Zustandskombinationen berechnet. Dies sind im Beispiel $RS_C = \{(C_1^S, C_1), (C_2^S, C_2)\}$ für die Komponente C und $RS_D = \{(D_1^S, D_1), (D_2^S, D_2)\}$ für die Komponente D. Nach Definition 5.3 gilt $RS'_{C\,D} = \{(C_1^S, C_1, D_1, D_1^S), (C_2^S, C_2, D_2, D_2^S)\}$.

Wenn nun die erreichbaren Zustände der Komponenteninstanzen :BaseStation und 1:RailCab nach obigem Beispiel berechnet wurden, können die Zustandsinformationen bzgl. der Zustände der Port Statecharts RC' und BS_1 aus der Menge der erreichbaren Zustände entfernt werden, da sie bereits nach obigem Muster verarbeitet wurden und für die erreichbaren Konfigurationen nicht relevant sind.

Im nächsten Schritt werden die bisher berechneten Zustandskombinationen mit den erreichbaren Zuständen der Komponenteninstanz 2:RailCab analog zu oben kombiniert. Tabelle 5.10 zeigt die erreichbaren Zustandskombinationen für das laufende Beispiel (aus Übersichtsgründen nur in Bezug auf die Kommunikation mit der Basestation).

1:RailCab	2:RailCab	:BaseStation
reference	reference	active
reference	robust	active
reference	absolut	active
robust	reference	active
robust	robust	active
robust	absolut	active
absolut	absolut	active
absolut	reference	active
absolut	robust	active
robust	absolut	idle
absolut	robust	idle
robust	robust	idle
absolut	absolut	idle

Tabelle 5.1: Erreichbare Zustandskombinationen der Synchronisationscharts der Komponenteninstanzen 1:RailCab, 2:RailCab, und :BaseStation in Bezug auf die Kommunikation mit der Basestation

Abbildung 5.10 zeigt zwei der Konfigurationen, die sich aus der Menge der erreichbaren Zustandskombinationen der Synchronisationscharts aus Tabelle 5.1 ergeben.

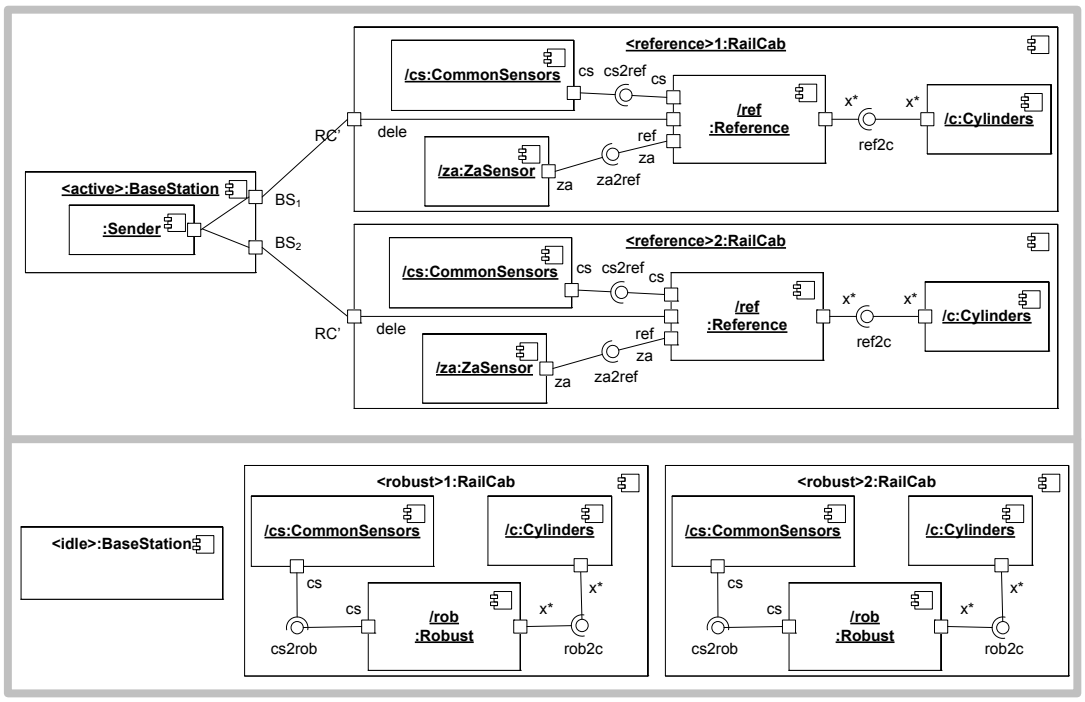


Abbildung 5.10: Zwei ausgewählte erreichbare Konfigurationen für den Registry-Teil des Beispiels.

5.1.2 Komponentenstorydiagramme

Im Gegensatz zu den oben beschriebenen Hybriden Rekonfigurationscharts nutzen Komponentenstorydiagramme nicht Zustandsmaschinen zur Beschreibung von unterschiedlichen Konfigurationen einer Komponente. Stattdessen werden Regeln für einen Komponententyp spezifiziert, welche die einem Typen unterlagerte Komponenteninstanzstruktur erzeugen oder transformieren. Durch die Anwendung dieser Regeln ergeben sich ausgehend von einem Startzustand die resultierenden Konfigurationen.

In diesem Abschnitt wird eine Technik beschrieben, die *alle* möglichen Komponenteninstanzstrukturen bestimmt, die aus der Ausführung der Komponentenstorydiagramme resultieren. Die grundlegende Idee ist die Nutzung von GROOVE [Ren04, RKS06]. GROOVE ist ein Werkzeug zur Erreichbarkeitsanalyse von Graphtransformationssystemen. Auf Basis eines Startgraphen und einer Menge von Graphtransformationen werden die durch die Anwendung der Graphtransformationen erreichbaren Zustände berechnet. Eine Instanz des Komponententyps der Systemebene ist Startgraph für die Ausführung der Komponentenstorydiagramme. GROOVE wird nun genutzt, um ausgehend von diesem Startgraphen

alle erreichbaren Komponenteninstanzstrukturen zu berechnen.

Komponentenstorydiagramme werden auf Basis der in Abschnitt 3.1 definierten Komponententypen modelliert. Diese Komponententypen erlauben die Spezifikationen von `0..1` und `0..*` Kardinalitäten für Ports, Komponenten und Konnektoren. Für eine Erreichbarkeitsanalyse ist die Kardinalität `0..*` problematisch, da diese zu unendlichen Zustandsräumen führt.

Charakteristika mechatronischer Systeme sind harte Echtzeiteigenschaften und begrenzter Speicherplatz. Bei der Anwendung von Komponentenstorydiagrammen in einem konkreten mechatronischen System müssen daher feste Obergrenzen für die Anzahl an Instanzen von Ports, Komponenten und Konnektoren angegeben werden, um diesen spezifischen Eigenschaften mechatronischer Systeme zu genügen. Diese festen Obergrenzen werden dann auch bei der Erreichbarkeitsanalyse mit GROOVE eingehalten.

In [Rei07] wurden Übersetzungsregeln von Storydiagrammen in GROOVE Graphtransformationen vorgestellt. Diese unterstützen neben einzelnen Storypattern auch Sequenzen sowie Schleifen und Entscheidungen im Kontrollfluss.

In Abschnitt 3.2.10 wurde eine Übersetzung der Komponentenstorydiagramme auf Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen beschrieben. Es liegt daher nahe, GROOVE zu nutzen, um mittels der aus den Komponentenstorydiagramme generierten Storydiagramme alle erreichbaren Zustände, also Komponenteninstanzstrukturen, zu berechnen.

Neben den in Abschnitt 3.2.6 beschriebenen Aktivitäten mit Programmtext der Zielplattform, die GROOVE naturgemäß nicht ausführen kann, unterstützt der Ansatz aus [Rei07] iterierte Storypatterns und Methodenaufrufe nicht.

Um dennoch GROOVE anwenden zu können, werden nur Komponentenstorydiagramme für die Erreichbarkeitsanalyse unterstützt, die keine Aktivitäten mit Programmtext beinhalten. Methodenaufrufe werden direkt auf die bereits existierende Unterstützung von Sequenzen von Storypatterns übersetzt.

Iterierte Storypatterns werden auf normale Schleifen und zusätzliche Aktivitäten übersetzt, um dann mit dem Ansatz aus [Rei07] nach GROOVE übersetzt zu werden. Iterierte Storypatterns werden nicht nur einmal sondern in einer Schleife für alle möglichen Bindungen der Objekte ausgeführt. Wenn man ein iteriertes Storypattern lediglich durch eine Schleife im Kontrollfluss ersetzt, kann es passieren, dass nicht alle verschiedenen Bindungen in der Schleife betrachtet werden, sondern z.B. immer die gleiche. Das Storypattern im Schleifenkopf muss also angepasst werden, um die bereits betrachteten Bindungen zu speichern.

Dies geschieht mit Hilfe von *Markierungsobjekten*. Ein Markierungsobjekt wird zum Storypattern im Schleifenkopf als negatives Objekt hinzugefügt; ein weiteres wird als zu erzeugendes Objekt hinzugefügt. Beide Markierungsobjekte haben Links zu allen anderen Objekten im Storypattern des Schleifenkopfs. Wenn nun

die Schleife ausgeführt wird, wird eine Belegung gesucht, die noch mit keinem Markierungsobjekt verbunden ist, also noch nicht in der Schleife betrachtet wurde. Wenn eine solche gefunden wurde, wird das Markierungsobjekt erzeugt und dann der Schleifenrumpf ausgeführt.

Wenn keine Belegung ohne Markierungsobjekt gefunden wird, sind alle möglichen Belegungen bereits verarbeitet und die Schleife wird verlassen. Nach dem Ende der Schleife werden in einer zusätzlichen Schleife alle Markierungsobjekte gelöscht. Um dies zu ermöglichen, gibt es ein festes Objekt, welches die Markierungsobjekte speichert.

Für den Fall, dass iterierte Storypattern geschachtelt sind, erhalten die Markierungsobjekte bei der Überprüfung und Erzeugung einen eindeutigen Wert für das id-Attribut.

Abbildung 5.11 zeigt das iterierte Storypattern A. In diesem Storypattern wird versucht, ausgehend vom this Objekt die Objekte x:X und y:Y zu binden. Für jede gefundene Bindung werden die über die Transition [each time] erreichbaren Aktivitäten ausgeführt. Nachdem alle Bindungen verarbeitet wurden, wird die Aktivität über die Transition [end] verlassen.

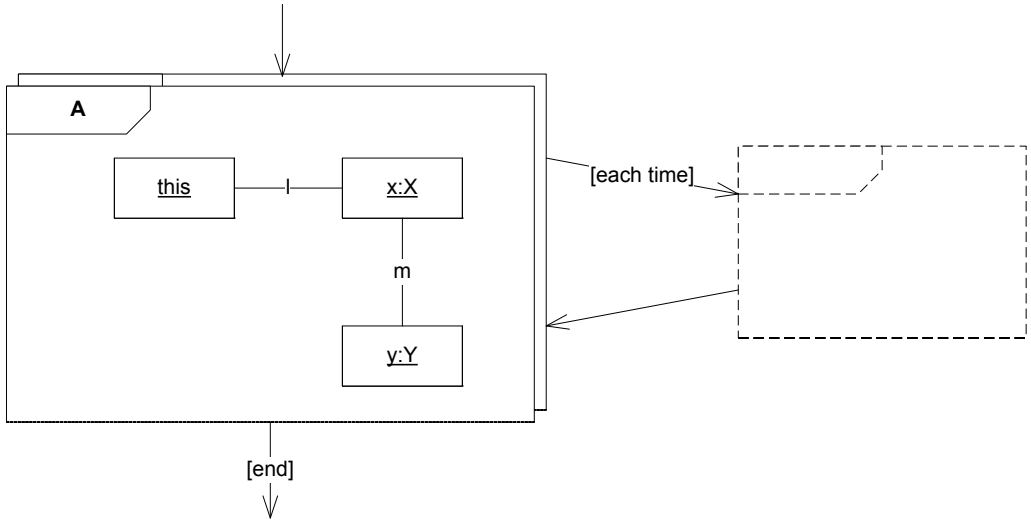


Abbildung 5.11: Beispiel für ein iteriertes Storypattern

Das iterierte Storypattern aus Abbildung 5.11 wird, wie in Abbildung 5.12 gezeigt, in Schleifen durch die oben skizzierte Nutzung von Markierungsknoten übersetzt. Das iterierte Storypattern wird in ein normales Storypattern übersetzt. Weiterhin werden die oben dargestellten Markierungsobjekte mit zugehörigen Links hinzugefügt. Die Bedingung der Transition [each time] wird durch [success] und die Bedingung der Transition [end] wird durch [failure] ersetzt. Die

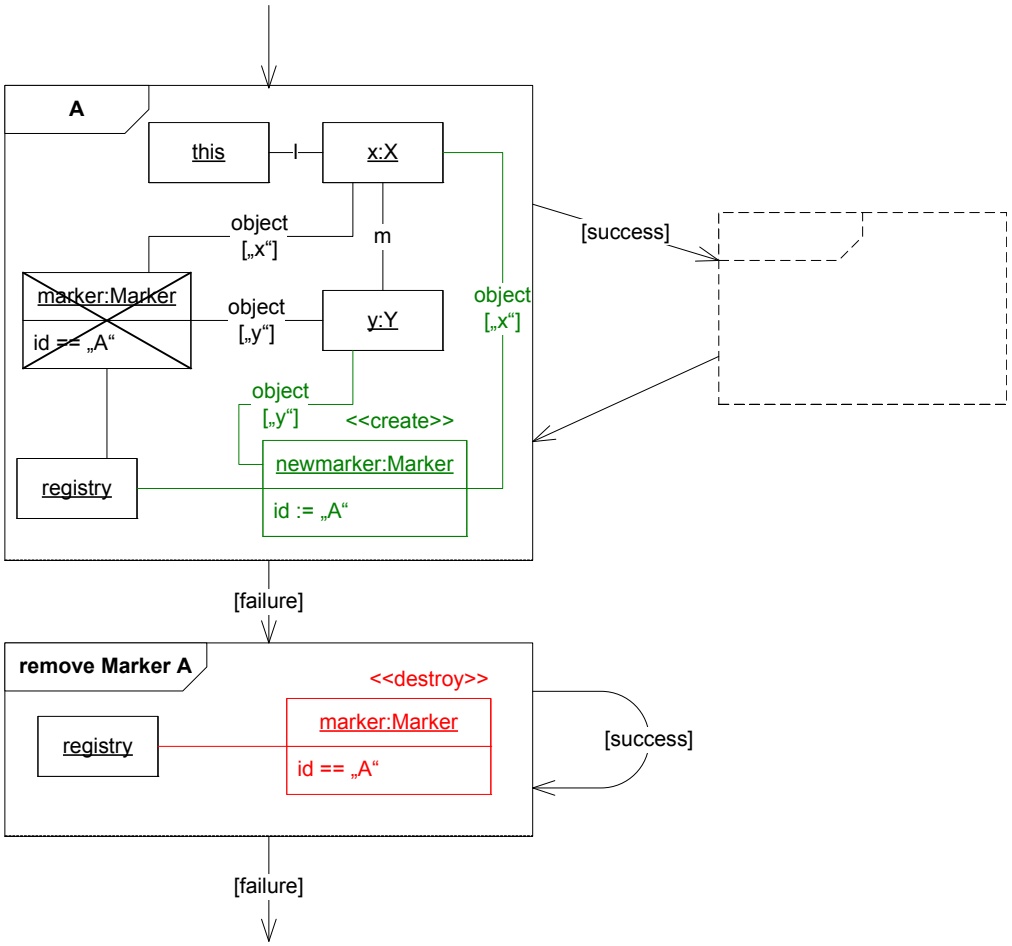


Abbildung 5.12: Übersetzung in Schleifen

Markierungsobjekte garantieren nun, dass für jede Belegung die weiteren Aktivitäten ausgeführt werden. Nach der Verarbeitung aller Belegungen werden in der, in einer Schleife ausgeführten, Aktivität remove Marker A alle erzeugten Markierungsobjekte entfernt.

In iterierten Storypatterns und in, über die Transition [each time] erreichbaren, weiteren Storypatterns erzeugte Objekte werden ebenfalls mit Markierungen versehen. Diese Markierungen werden dann genutzt, um bei der Bindung diese im iterierten Storypattern erzeugte Objekte zu ignorieren, um die gewählte Pre-Select Semantik für iterierte Komponentenstorypatterns umzusetzen.

5.2 Analysemodell

Nachdem nun die erreichbaren Konfigurationen der selbstoptimierenden Systeme bestimmt worden sind, werden im Folgenden die in Kapitel 4 beschriebenen Techniken zur Gefahrenanalyse um die Betrachtung von Konfigurationen erweitert. Diese Erweiterung der Gefahrenanalyse betrachtet alle erreichbaren Konfigurationen. Die Skalierbarkeit ist daher eingeschränkt. Eine mögliche Technik zur Verbesserung der Skalierbarkeit wird im Ausblick dieser Arbeit (Abschnitt 8.1) skizziert.

Eine Gefahr ist im Beispiel dieses Kapitels eine falsche Position des Aufbaus, die in Kurven zu einem Entgleisen führen kann. Dieser Zustand des Systems wird im Fehlerbaum aus Abbildung 5.13 dargestellt.

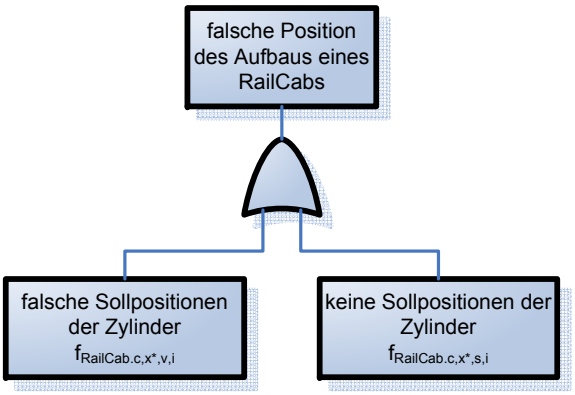


Abbildung 5.13: Fehlerbaum für die Gefahr einer falschen Position des RailCab-Aufbaus

Eine quantitative Top-Down Gefahrenanalyse für die Gefahrendefinition aus Abbildung 5.13 ergibt für die zwei Konfigurationen aus Abbildung 5.10 die Gefahrenwahrscheinlichkeiten aus Tabelle 5.2 bei angenommen Wahrscheinlichkeiten der Fehlerzustände von 0.0001. Diese beiden Konfigurationen sind die schlechteste bzw. beste Konfiguration in Bezug auf die Gefahrenwahrscheinlichkeit.

1:RailCab	2:RailCab	:BaseStation	Hazard probability
reference	reference	active	0.00059985
robust	robust	idle	0.00019999

Tabelle 5.2: Gefahrenwahrscheinlichkeiten für zwei ausgewählte Konfigurationen

Anstelle einer simplen Analyse aller einzelnen Konfigurationen werden die in Abschnitt 4.2.2 beschriebenen Analysemodelle für den Fall von selbstoptimie-

renden Systemen erweitert, so dass alle Konfigurationen auf einmal analysiert werden. Dies ermöglicht, die beste oder schlechteste Konfiguration bzgl. der Gefahrenwahrscheinlichkeit zu bestimmen, ohne alle Konfigurationen strukturell aufbauen und einzeln analysieren zu müssen.

Grundlage für die Analyse von verschiedenen Konfigurationen einer Komponente ist deren Abbildung auf boolesche Variablen. So können mit einer booleschen Variable c_1 zwei Konfigurationen (Konfiguration 1: $c_1 = \text{true}$ bzw. Konfiguration 2: $c_1 = \text{false}$) dargestellt werden. Allgemein können n Konfigurationen mit $\lceil log_2 n \rceil$ booleschen Variablen abgebildet werden (s. Tabelle 5.3 für die Abbildung von 6 Konfigurationen).

	c_1	c_2	c_3
Konf. 1	false	false	false
Konf. 2	false	false	true
Konf. 3	false	true	false
Konf. 4	false	true	true
Konf. 5	true	false	false
Konf. 6	true	false	true

Tabelle 5.3: Belegung der Konfigurationsvariablen für 6 Konfigurationen

Da die Analyse boolesche Variablen für die Modellierung der Fehlerpropagierung nutzt, können die Konfigurationsvariablen c_i in das Modell der Fehlerpropagierung integriert werden, um die unterschiedlichen Fehlerpropagierungen der Konfigurationen je nach Wert der Variable c_i umzuschalten. Formel 5.1 zeigt die Fehlerpropagierungen $\psi_{s,1}$ und $\psi_{s,2}$ zweier Architekturkonfigurationen. Diese Abbildung der Konfigurationen fügt sich nahtlos in die bisherigen Fehlerpropagierungsmodelle ein.

$$(c_1 \wedge \psi_{s,1}) \vee (\neg c_1 \wedge \psi_{s,2}) \tag{5.1}$$

Nicht alle Konfigurationen sind auf Basis der Erreichbarkeitsanalysen möglich. Diese müssen von der Analyse ausgeschlossen werden. Der Ausschluss einer Konfiguration aus der Analyse wird durch das Hinzufügen einer booleschen Bedingung über die Konfigurationsvariablen in das Analysemodell umgesetzt. Wenn zu einem Port in einer Konfiguration keine Portinstanz existiert, muss der Wert der zugehörigen Ausfallvariable auf false festgesetzt werden, da ja kein Ausfall an dieser Portinstanz auftreten kann. Dies wird durch eine Biimplikation der Variable mit false erreicht.

Wenn die Fehlerpropagierungsmodelle auf diese Art mit den Konfigurationsvariablen kombiniert worden sind, werden die in Kapitel 4 vorgestellten qualitativen

und quantitativen Analysetechniken analog angewendet.

5.3 Qualitative Analyse

In Bezug auf die qualitative Analyse ist vor allem das Auftreten einer Gefahr wichtig. Auf Basis der BDD-Repräsentation lassen sich die folgenden Fragen beantworten:

- Tritt eine Gefahrensituation in irgendeiner Konfiguration auf?
- In welchen Konfigurationen tritt eine Gefahr auf?

Der BDD des Gesamtsystems aus den (erweiterten) Fehlerpropagierungsformeln der einzelnen Komponenten bzw. Konfigurationen wird, wie in Abschnitt 4.2.2 erläutert, mit der Gefahrendefinition kombiniert, um das Auftreten einer Gefahr zu analysieren.

Wenn der aus Fehlerpropagierung und Gefahrendefinition entstandene BDD nur noch aus dem 0-Terminalknoten besteht, gibt es keine Belegung der Variablen, welche die durch den BDD abgebildete boolesche Formel wahr macht. Es gibt also keine Konfiguration, in der die Gefahr eintritt.

Wenn der BDD nicht nur aus dem 0-Terminalknoten besteht, zeigt die Bestimmung der Implikanten aus Abschnitt 4.2.2.2 nicht nur welche Basisereignisse eintreten müssen, sondern auch durch die Belegung der Konfigurationsvariablen in den jeweiligen Implikanten in welcher Konfiguration diese Gefahr auftritt.

5.4 Quantitative Analyse

Neben der qualitativen Analyse, ob eine Gefahr auftritt und in welcher Konfiguration, ist, wie in Abschnitt 4.2.2 für nicht selbstoptimierende Systeme beschrieben, auch die Gefahrenwahrscheinlichkeit für selbstoptimierende Systeme wichtig. Diese quantitativen Analysen werden auf der oben dargestellten BDD-Repräsentation der Fehlerpropagierung des Systems inklusive der Konfigurationsvariablen durchgeführt.

Nach Entfernung der Ausfallvariablen besteht der BDD nur aus Konfigurationsvariablen c_i und Ereignisvariablen e_i. Die in Abschnitt 4.3 dargestellte Technik zur Berechnung der Wahrscheinlichkeit wird um eine spezielle Behandlung der Konfigurationsvariablen erweitert. Wenn bei der rekursiven Berechnung der Gefahrenwahrscheinlichkeit der Algorithmus auf eine Konfigurationsvariable c_i trifft, wird nicht ein neuer Wert berechnet, sondern die folgende Formel nach oben propagiert:

$$probability(c_i) = \begin{cases} probability(c_i.low) & \text{für} \quad c_i = 0 \\ probability(c_i.high) & \text{für} \quad c_i = 1 \end{cases} \tag{5.2}$$

Formel 5.2 besagt, dass wenn c_i den Wert 1 (für wahr) bzw. 0 (für falsch) annimmt, der Wert des high-Knotens bzw. des low-Knotens genutzt werden soll. Diese Anpassung resultiert in der Propagierung von Formeln und nicht Wahrscheinlichkeiten, wie bei der quantitativen Gefahrenanalyse für Systeme ohne Rekonfiguration.

$$\begin{aligned} probability(c_i) = c_i \cdot probability(c_i.high) \\ + (1 - c_i) \cdot probability(c_i.low) \end{aligned} \tag{5.3}$$

Formel 5.3 stellt die Formel zur einfacheren Nutzung in einem Optimierungswerkzeug nicht als Fallunterscheidung sondern als Multiplikation mit der Variablen c_i dar.

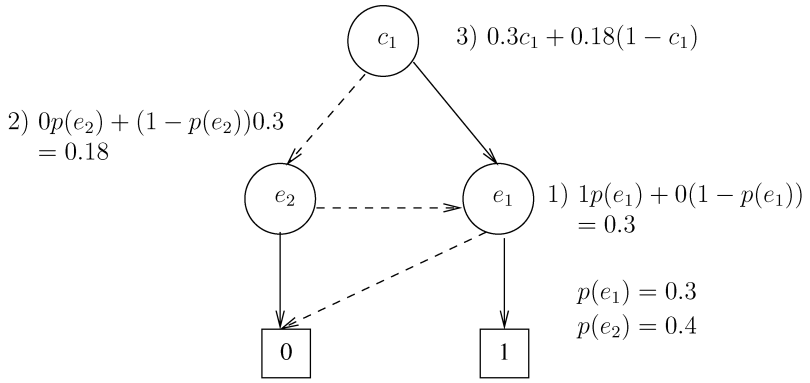


Abbildung 5.14: Wahrscheinlichkeitsberechnung auf einem BDD mit Konfigurationsvariablen [GT06]

Abbildung 5.14 zeigt eine Abwandlung des Beispiels aus Abbildung 4.16. Hier wurde die Ereignisvariable e_3 durch die Konfigurationsvariable c_1 ersetzt. Die Schritte 1 und 2 sind identisch zur Berechnung ohne Varianten. In Schritt 3 wird für den Knoten c_1 Formel 5.2 angewendet. Der gesamte BDD wird dementsprechend in die Formel $0.3c_1 + 0.18(1 - c_1)$ umgesetzt. Diese Formel lässt sich nun hinsichtlich unterschiedlicher Ziele untersuchen.

5.4.1 Schlechteste Konfiguration

Bei der Entwicklung sicherheitskritischer Systeme ist es wichtig zu wissen, welche Konfiguration die schlechteste bzgl. der Gefahrenwahrscheinlichkeit bzw. des

Risikos ist. Das Ergebnis stellt dann eine obere Schranke für die Gefahrenwahrscheinlichkeit bzw. das Risiko dar. Wenn diese obere Schranke niedriger ist als die Anforderungen, können alle Konfigurationen eingesetzt werden, ohne alle Konfigurationen einzeln zu analysieren.

$$max(probability(\psi)) \tag{5.4}$$
$$max(severity(\psi) \cdot probability(\psi)) \tag{5.5}$$

Wenn man die aus dem obigen BDD mit den Konfigurationsvariablen unter Nutzung von Formel 5.2 entstehende Formel maximiert (Formel 5.4), erhält man die Belegung der Konfigurationsvariablen, also die Konfiguration, mit der höchsten Wahrscheinlichkeit der Gefahr ψ. Formel 5.5 erweitert diese Maximierung um die Betrachtung des Risikos durch die Multiplikation der Gefahrenwahrscheinlichkeit mit dem potentiellen Schaden.

$$max \sum_{\psi_i \in \Psi} probability(\psi_i) \tag{5.6}$$
$$max \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i) \tag{5.7}$$

Für eine Menge von Gefahren Ψ kann es interessant sein, die Konfiguration zu identifizieren, für die die Summe der Gefahrenwahrscheinlichkeiten bzw. Risiken am höchsten ist. Dies wird mit den Formeln 5.6 bzw. 5.7 berechnet.

5.4.2 Beste Konfiguration

Analog zu oben wird die beste Konfiguration (also die Belegung der Variablen c_i) bzgl. der Gefahrenwahrscheinlichkeit und des Risikos mit Hilfe der folgenden Zielfunktionen berechnet:

$$min(probability(\psi)) \tag{5.8}$$
$$min(severity(\psi) \cdot probability(\psi)) \tag{5.9}$$
$$min \sum_{\psi_i \in \Psi} probability(\psi_i) \tag{5.10}$$
$$min \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i) \tag{5.11}$$

5.4.3 Erlaubte Konfigurationen

Wenn die schlechteste Konfiguration eine höhere Gefahrenwahrscheinlichkeit bzw. eine höheres Risiko hat, als durch die Anforderungen vorgegeben, ist eine Analyse der erlaubten Konfigurationen sinnvoll. Erlaubte Konfigurationen sind die Konfigurationen, die die geforderte Gefahrenwahrscheinlichkeit bzw. das geforderte Risiko einhalten.

Algorithmus 4 : List<Configuration> computeAllowedConfigurations (List<Gefahr> gefahren)

```
   begin
       List<Configurations> forbidden ← ∅;
       List<Configurations> oldForbidden ← ∅;
       firstIteration ← true;
       while oldForbidden ≠ forbidden ∨ firstIteration = true do
           oldForbidden ← forbidden ;
7          violatingConfigurations ← maxProbability (forbidden,gefahren);
8          forbidden ← forbidden ∪ violatingConfigurations;
           firstIteration ← false;
       return forbidden;
   end
```

Auf Basis der oben dargestellten Analysemodelle kann diese Fragestellung durch eine iterative Analyse (s. Algorithmus 4) beantwortet werden. Bei dieser Analyse werden in jedem Iterationsschritt Konfigurationen der einzelnen Komponenten identifiziert, die zu einer Verletzung der Anforderungen führen (Zeile 7). Bei der Berechnung der Gefahrenwahrscheinlichkeit werden die Zielfunktionen 5.4 und 5.5 genutzt. Diese Konfigurationen werden dann bei der nächsten Analyse ausgeschlossen (Zeile 8). Wichtig dabei ist, dass für alle Komponenteninstanzen eines Komponententyps diese Konfiguration ausgeschlossen wird. Diese iterative Analyse endet, wenn entweder alle Konfigurationen ausgeschlossen wurden oder die Gefahrenanalyse ergibt, dass alle Anforderungen erfüllt wurden.

Auf Basis der Ergebnisse dieser Analyse kann dann das entworfene Systeme angepasst werden, indem z.B. die ausgeschlossenen Konfigurationen der Komponenten geändert oder angepasst werden.

5.5 Zusammenfassung

In diesem Kapitel wurde die in Kapitel 4 eingeführte komponentenbasierte Gefahrenanalyse in Bezug auf die Unterstützung von Rekonfiguration erweitert. In einem ersten Schritt wurden alle erreichbaren Konfigurationen bestimmt.

Die um Konfigurationen erweiterten Analysetechniken ermöglichen die Bestimmung der Konfigurationen, in denen Kombinationen von Fehlerzuständen zum Auftreten von Gefahren führen. Des Weiteren wurden die quantitativen Analysen erweitert, so dass die besten, die schlechtesten und alle erlaubten Konfigurationen in Bezug auf die Gefahrenwahrscheinlichkeit und das Risiko berechnet werden.

6 Werkzeugunterstützung

In den vorhergehenden Kapiteln wurde eine Gefahrenanalyse für selbstoptimierende Systeme vorgestellt. Sie wurde prototypisch in Form mehrerer Plugins für das Werkzeug Fujaba4Eclipse implementiert. In Abschnitt 6.1 werden auf Basis des in den Kapiteln 3 bis 5 verwendeten Beispiels die graphische Benutzerschnittstelle sowie die verschiedenen Schritte bei der Modellierung und Gefahrenanalyse erläutert. Darauf folgt eine Beschreibung der Architektur des Softwareprototyps. Die Ergebnisse der mit Hilfe des Prototyps durchgeführten ersten Experimente hinsichtlich der Skalierbarkeit der Gefahrenanalyse werden in Abschnitt 6.3 präsentiert.

6.1 Benutzerschnittstelle

6.1.1 Modellierung

Im Folgenden wird die Benutzerschnittstelle des Prototyps für die in Kapitel 3 präsentierten Modellierungssprache für Komponentenstrukturen und die darauf aufbauenden Komponentenstorydiagramme vorgestellt. Die präsentierten Ergebnisse zeigen eine Vorversion des in der vorliegenden Arbeit genutzten Beispiels und stammen aus [Hol08].

Abbildung 6.1 zeigt den hierarchischen Komponententyp für ein RailCab aus Abbildung 3.7 modelliert in Fujaba4Eclipse.

Die Abbildungen 6.2 und 6.3 zeigen die Komponentenstorydiagramme für die Rekonfiguration des Koordinator-RailCabs, wenn (1) ein weiteres Fahrzeug in den Konvoi eintritt sowie (2) wenn ein Fahrzeug den Konvoi verlässt.

Das Komponentenstorydiagramm RailCab::InsertPosCalc modelliert die Rekonfiguration eines Koordinator-RailCabs, wenn ein neues Fahrzeug dem Konvoi beitritt. Parameter für das Komponentenstorydiagramm sind der Name der zu erzeugenden Komponenteninstanz, die Position des neuen Fahrzeugs im Konvoi (position) sowie die Portinstanz (posParam) über die das Kooordinatorfahrzeug dem hinzugekommenen RailCab die Sollpositionen sendet.

In den ersten beiden Aktivitäten wird die Stelle in der Kette von Positionsberechnungskomponenten gesucht, an der die neue Komponenteninstanz erzeugt

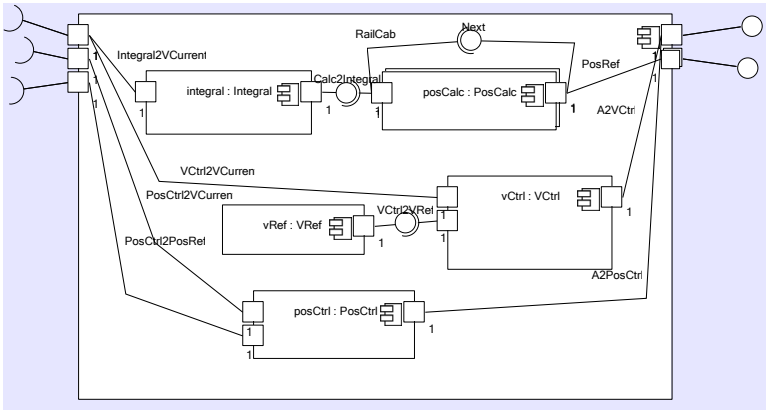


Abbildung 6.1: Hierarchischer Komponententyp RailCab

werden soll. Diese Suche beginnt bei der Integral-Komponenteninstanz, die immer einmal existiert. Danach werden in einer Schleife die in einer Kette verbundenen Positionsberechnungskomponenten traversiert, bis die Stelle auf Basis des Parameters position gefunden wird, wo die neue Komponenteninstanz erzeugt werden soll. Die Erzeugung wird in verschiedenen Aktivitäten des Komponentenstorydiagramms durchgeführt, je nachdem, ob die neue Aktivität am Anfang, in der Mitte oder am Ende der Kette erzeugt wird.

Das Komponentenstorydiagramm RailCab::RemovePosCalc modelliert das Entfernen einer Positionsberechnungskomponente im Koordinator-RailCab, wenn ein Fahrzeug den Konvoi verlässt. Dies stellt die komplementäre Funktion zu RailCab::InsertPosCalc dar und ist analog modelliert.

Abbildung 6.4 zeigt die Komponenteninstanzstruktur des Koordinator-RailCabs sowie von fünf weiteren Fahrzeugen, die mit dem oben gezeigten RailCab::InsertPosCalc sowie weiteren (s. [Hol08]) Komponentenstorydiagrammen erzeugt wurden.

Das Fahrzeug an Position 4 verlässt diesen Konvoi. Das Koordinator-Fahrzeug rekonfiguriert nun seine Komponenteninstanzstruktur durch den Aufruf des Komponentenstorydiagramms RailCab::RemovePosCalc. Dieses entfernt die Positionsberechnungskomponente für das Fahrzeug, welches den Konvoi verlässt, und verbindet die übrigen Komponenten neu, so dass sich die Komponenteninstanzstruktur aus Abbildung 6.5 ergibt.

6.1.2 Gefahrenanalyse

Zusätzlich zu den vorgestellten Modellen für die Komponenteninstanzstrukturen müssen vor der Ausführung der Gefahrenanalyse die abstrakten Fehlerpropagie-

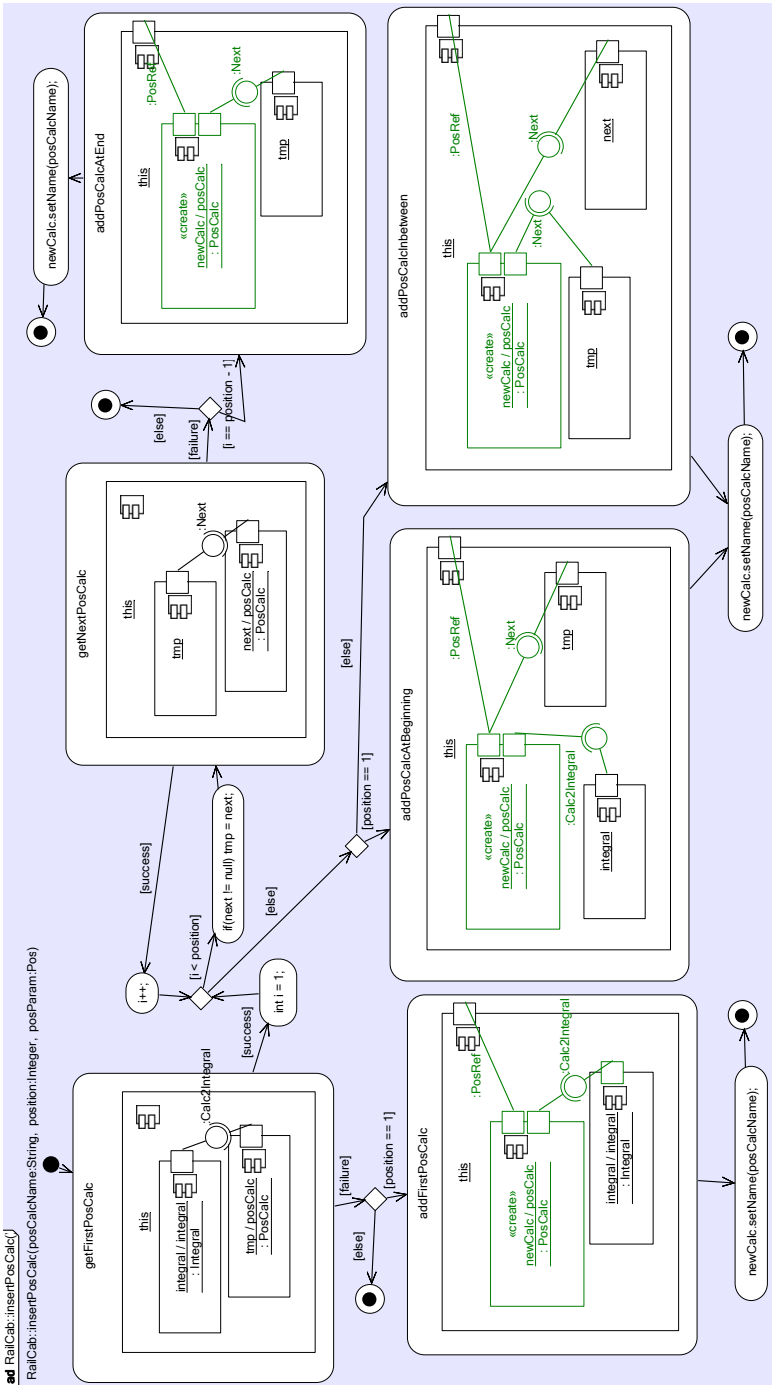


Abbildung 6.2: Komponentenstorydiagramm RailCab::insertPosCalc

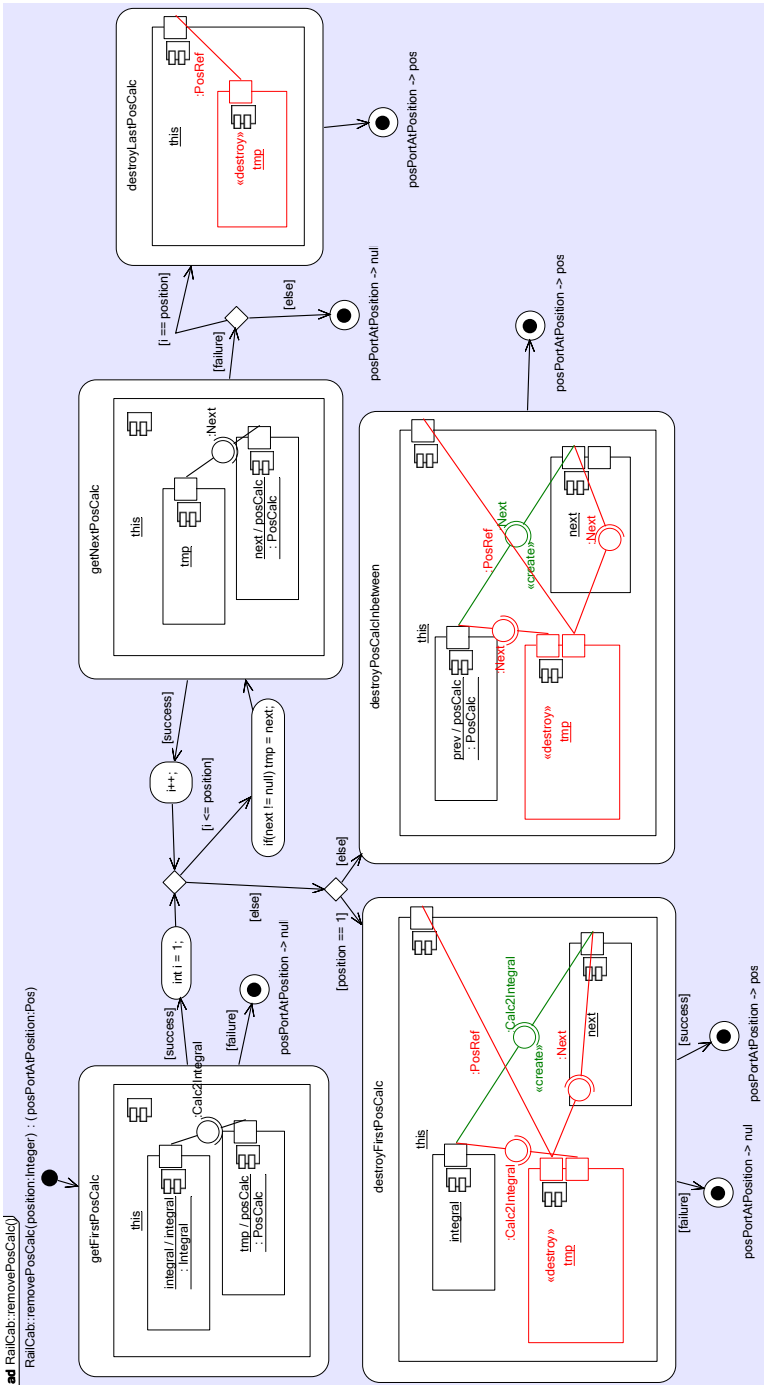


Abbildung 6.3: Komponentenstorydiagramm RailCab::removePosCalc

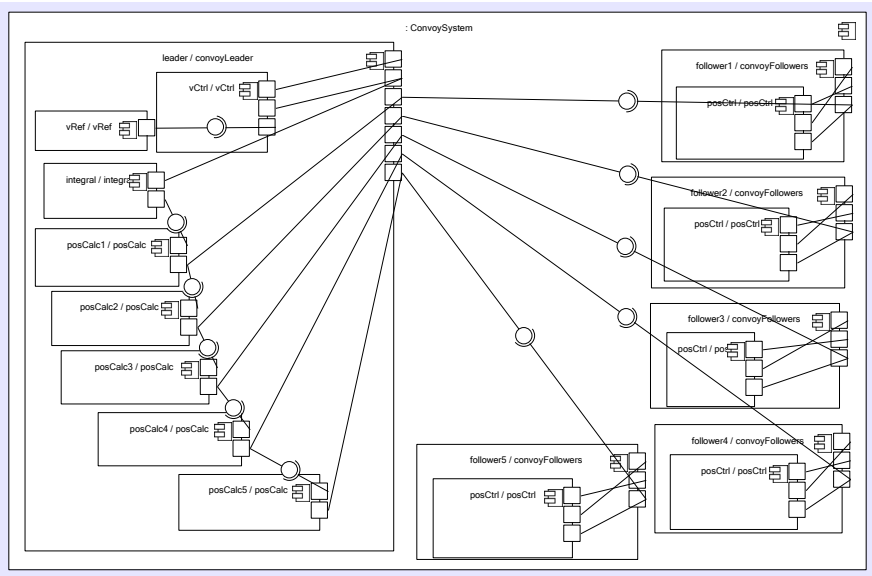


Abbildung 6.4: Komponenteninstanzstruktur nach Konvoi-Initialisierung mit fünf folgenden RailCabs

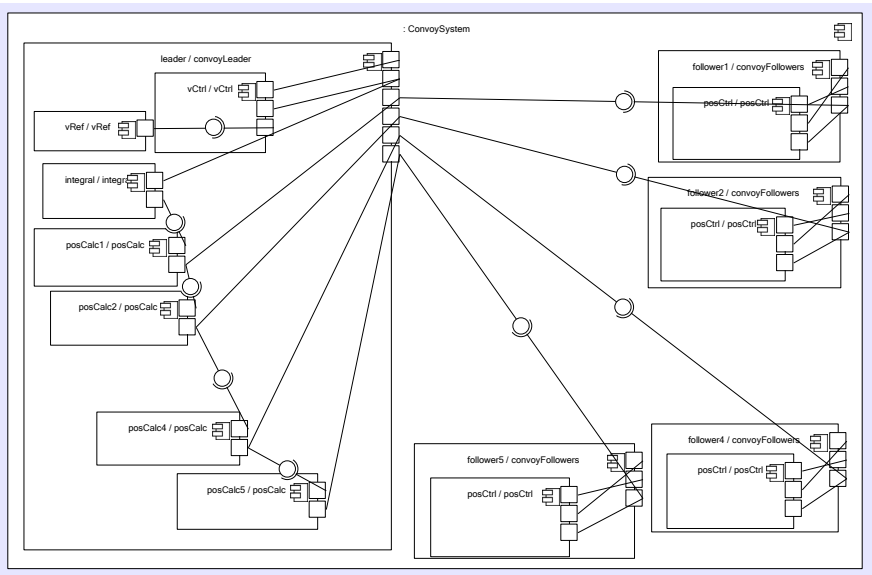


Abbildung 6.5: Komponenteninstanzstruktur nach dem Entfernen des RailCabs an Position 3

rungen für die einzelnen Komponententypen modelliert werden. Die im Folgenden gezeigten Ergebnisse stammen aus [THMD08].

Abbildung 6.6 zeigt im Werkzeug Fujaba4Eclipse das Modell einer Beispielfehlertyphierarchie bestehend aus den Fehlertypen Value, Crash und Timing, die eine Verfeinerung des allgemeinen Fehlers vom Typ Failure sind.

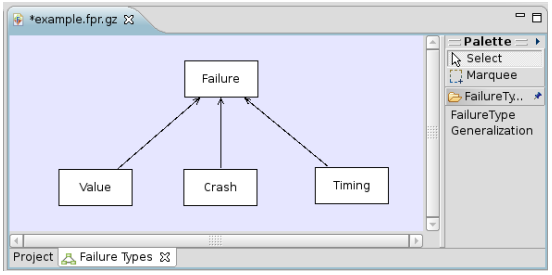


Abbildung 6.6: Beispielfehlertyphierarchie

Basierend auf der definierten Fehlertyphierarchie wird das abstrakte Fehlerpropagierungsverhalten der Komponententypen definiert. Abbildung 6.7 stellt die Fehlerpropagierung für den Komponententyp ConvoyOperator dar. In der unteren Hälfte des Fensters werden links die bereits spezifizierten Fehlerpropagierungsformeln aufgelistet. In der Mitte wird eine Fehlerpropagierung bzgl. falscher Werte (Fehlertyp Value) gezeigt. Diese Fehlerpropagierung verbindet mit booleschen Operatoren die auf der rechten Seite dargestellten spezifizierten Ausfällen an den Ports sowie die ebenfalls dort dargestellten Basisereignisse.

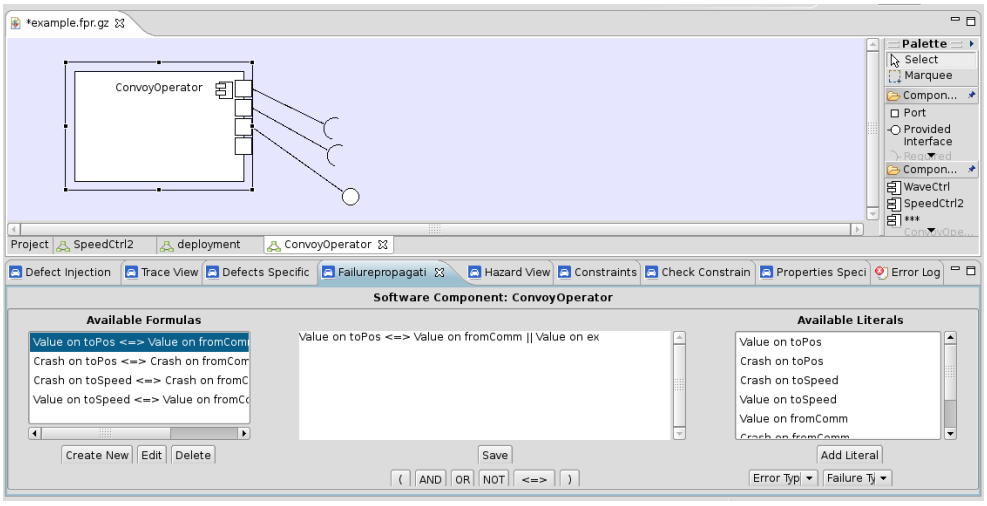


Abbildung 6.7: Fehlerpropagierung eines Komponententyps

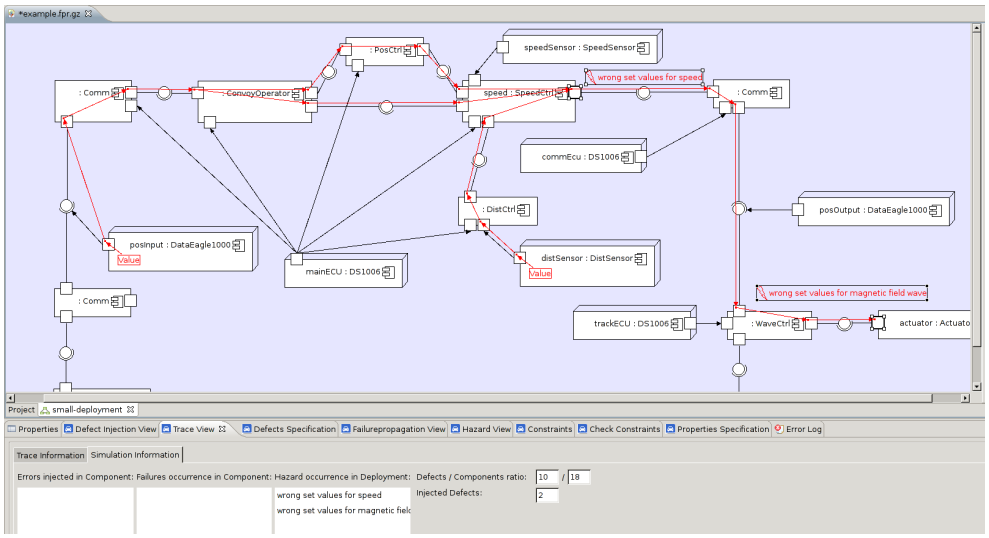


Abbildung 6.8: Simulation der Fehlerpropagierung

Die Bottom-Up Analyse berechnet auf Basis einer vom Anwender ausgewählten festen Menge von Basisereignissen, welche Gefahren eintreten. Hierbei werden die Fehlerpropagierungen für die einzelnen Komponenteninstanzen in einer Konfiguration angewandt, so dass sich eine Fehlerpropagierung ausgehend von den ausgewählten Basisereignissen über die Struktur ergibt. Eine solche Fehlerpropagierung wird in Abbildung 6.8 dargestellt. Zwei Basisereignisse vom Typ Value werden in die Komponenteninstanz posInput:DataEagle1000 und in die Komponenteninstanz distSensor:DistSensor injiziert. Ersteres modelliert eine fehlerhafte Übertragung der Daten von einem RailCab zum anderen; das zweite Basisereignis repräsentiert fehlerhafte Sensordaten vom Abstandssensor. Durch die Simulation der Fehlerpropagierungen ergibt sich, dass zwei Gefahrensituationen entstehen können. Zum einen kann sich eine falsche Geschwindigkeit des RailCabs ergeben; zum anderen resultieren die Basisereignisse in falschen Werten für die Magnetfelder des Linearmotors. Beide Situationen können in Unfällen resultieren.

Die Top-Down Analyse kann nun für die Gefahren genutzt werden, um alle Basisereignisse zu bestimmen, die zu einer Gefahrensituation führen. Abbildung 6.9 zeigt diese Ergebnisse der qualitativen Top-Down Analyse.

Für diese Basisereignisse werden nun Wahrscheinlichkeiten (z. B. auf Basis von Herstellerangaben oder historischen Daten) angegeben; die Gefahrenwahrscheinlichkeit wird dann auf Basis dieser Wahrscheinlichkeiten berechnet (s. Abbildung 6.10).

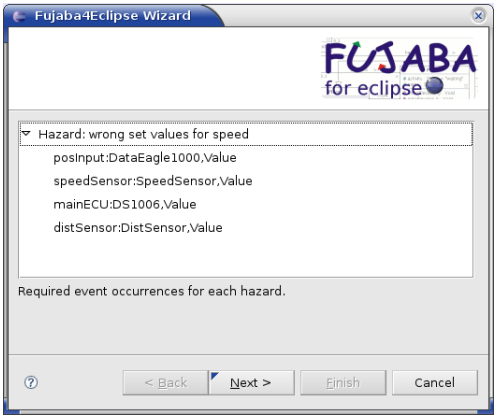
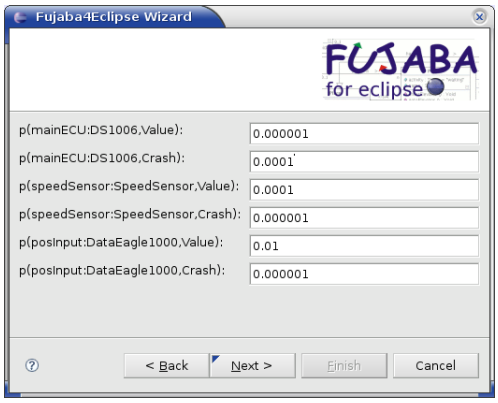
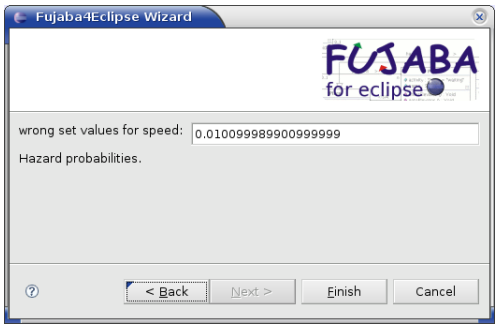


Abbildung 6.9: Basisereignisse einer Gefahr



(a) Eingabe der Wahrscheinlichkeiten der Basisereignisse



(b) Berechnete Gefahrenwahrscheinlichkeit

Abbildung 6.10: Quantitative Gefahrenanalyse

6.2 Softwarearchitektur

Fujaba4Eclipse ist eine Portierung von Fujaba auf die Entwicklungsumgebung Eclipse. Eclipse basiert auf einer komponentenbasierten Architektur. Ein kleiner Kern wird dabei durch einzelne Komponenten, Plugins genannt, erweitert. Fujaba4Eclipse ist ein solches Plugin, das ebenfalls durch Plugins erweitert werden kann. Die MECHATRONIC UML wurde als Plugins für Fujaba und Fujaba4Eclipse realisiert [BGH+05, BGH+07].

Abbildung 6.11 zeigt eine Übersicht über die Architektur des entwickelten Werkzeugs. Die dargestellten Plugins enthalten weitere Plugins, die in Tabelle 6.1 weiter aufgeschlüsselt sind. Die Plugins Fujaba4Eclipse und TGG werden

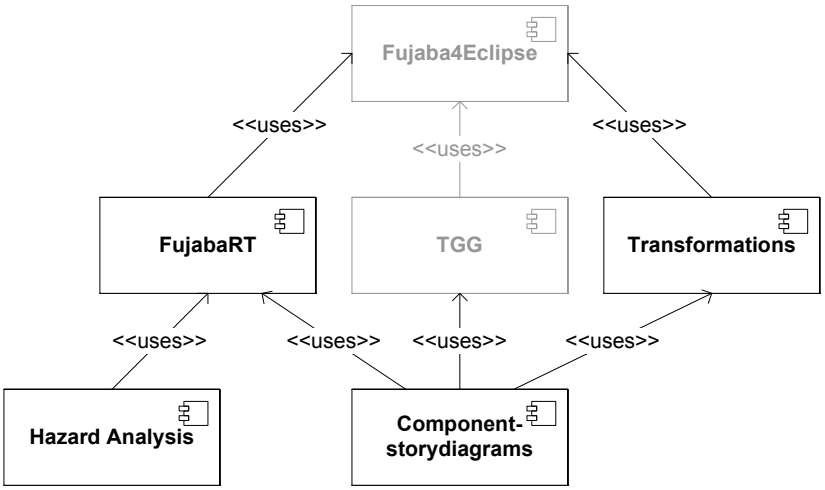


Abbildung 6.11: Übersicht der Architektur des entwickelten Werkzeugs

im Gegensatz zu den übrigen Plugins nur genutzt. Die restlichen Plugins wurden erweitert bzw. neu entwickelt. Fujaba4Eclipse enthält den Kern von Fujaba sowie die Unterstützung für Story Driven Modeling in Form von Klassendiagrammen, Storydiagrammen und Quelltextgenerierung. Das Plugin TGG enthält den Editor für Triple-Graph-Grammatiken und die zugehörige Quelltextgenerierung sowie die Ausführungsunterstützung für den aus den TGG-Regeln generierten Quelltext.

Die Mechatronic UML wurde im Plugin FujabaRT umgesetzt. Dies beinhaltet die komponentenbasierte Strukturmodellierung, die Real-Time Statecharts und die Hybriden Rekonfigurationscharts. Hier wurde die in Abschnitt 3.1 vorgestellte Erweiterung der Strukturspezifikationssprache umgesetzt.

Das Plugin Componentstorydiagrams beinhaltet die Implementierung der Komponentenstorydiagramme [Hol08, HT08] aus Abschnitt 3.2. Des Weiteren enthält dieses Plugin die Quelltextgenerierung unter Nutzung der Plugins TGG und Transformations. Letzteres enthält die in Anhang A vorgestellte Erweiterung der Storydiagramme als Zielsprache für die Übersetzung der Komponentenstorydiagramme. Die Gefahrenanalyse aus den Kapiteln 4 und 5 wurde im Plugin Hazard Analysis implementiert [Sch07].

Der überwiegende Teil der in dieser Arbeit vorgestellten Konzepte wurde prototypisch in den genannten Plugins umgesetzt. Die Implementierung der restlichen Konzepte dauert derzeit noch an.

[1]Dieses Plugin beinhaltet den aus den TGGs generierten Quelltext für die Übersetzung der Komponentenstorydiagramme in Storydiagramme auf dem Metamodell der Komponenteninstanzstrukturen (s. Abschnitt 3.2.10)

Plugin	Klassen	LOC
Hazard Analysis		
analysis	85	10.025
defects	59	5.784
deployment	63	5.820
formulas	67	5.220
hardware	18	1772
simulation.failurepropagation	15	4.394
simulation.failurepropagation.trace	50	8.252
Transformations		
specification	139	14.169
generation	47	6.062
execution	26	3.583
transformationExecution	3	226
Componentstorydiagrams		
UMLRT2	288	79.365
mapping[1]	74	60.093
mapping.codegen	11	1.325
Summe	1045	205.690

Tabelle 6.1: Größe der entwickelten bzw. erweiterten Plugins für Fujaba4Eclipse

6.3 Experimentelle Ergebnisse

Mit der vorgestellten Werkzeugunterstützung wurden erste Experimente hinsichtlich der Skalierbarkeit der Gefahrenanalyse durchgeführt [Sch07]. Abbildung 6.12 zeigt eine abstrakte Beispielkomponentenstruktur, die aus n Komponenteninstanzen des Komponententyps A besteht, die in einer Kette verbunden sind. Die Fehlerpropagierung dieses Komponententyps besteht aus zwei Basisereignissen, die mit Und sowie Oder mit einem eingehenden Ausfall verknüpft sind. Die Kette startet mit einer Instanz des Komponententyps Processor. Zur Betrachtung von zyklischen Modellen kann die erste Komponenteninstanz der Kette mit der letzten verbunden werden. Die Anzahl der Instanzen von A kann nun zur Prüfung der Skalierbarkeit der entwickelten Gefahrenanalyse variiert werden.

Tabelle 6.2 aus [Sch07] zeigt die Größe der BDDs, welche die Fehlerpropagierung des Modells enthalten, sowie die Berechnungszeiten in Sekunden auf einem Intel Pentium 4 mit 2GHz mit 512MB Hauptspeicher:

n: Anzahl der Komponenteninstanzen

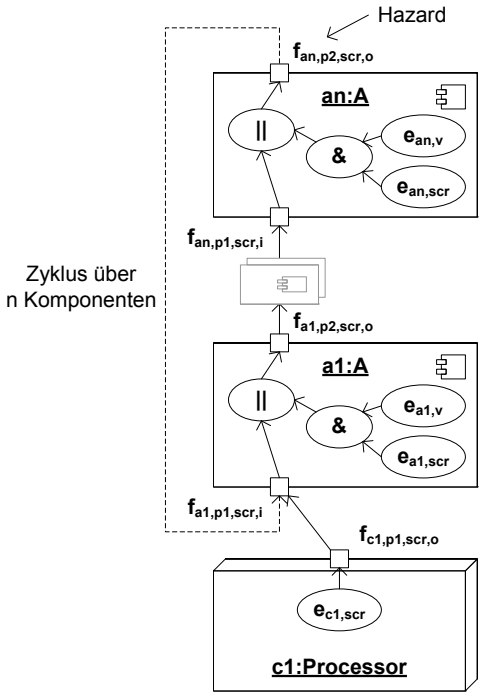


Abbildung 6.12: Skalierungsbeispiel A

B: Anzahl der BDDs für die Fehlerpropagierung der einzelnen Komponenteninstanzen

K: Anzahl der gesamten BDD-Knoten in B

EK: Anzahl der BDD-Knoten für die Fehlerpropagierung des Gesamtsystems

EvK: Anzahl der BDD-Knoten, welche die Basisereignisse repräsentieren.

Aufbau: Zeit in Sekunden für den Aufbau der Fehlerpropagierung des Gesamtsystems

Exist: Zeit in Sekunden für die Entfernung der Ausfallvariablen der Fehlerpropagierung

Quant: Zeit in Sekunden für die Berechnung der Gefahrenwahrscheinlichkeit

$\sum$: Gesamtzeit in Sekunden

Die Ergebnisse zeigen, dass für das Skalierungsbeispiel aus 6.12 die Größe der BDDs überschaubar bleibt und die Zeit bis zur Berechnung auch für Komponentenstrukturen mit 100 Instanzen unter einer Sekunde bleibt. Allerdings ist

deutlich eine exponentielle Steigerung der Berechnungszeit, vor allem beim Vergleich der Berechnungszeit von 50 und 100 Komponenten, zu sehen, so dass die Skalierbarkeit eingeschränkt ist.

n	**B**	**K**	**EK**	**EvK**	**Aufbau**	**Exist**	**Quant**	$\sum$
1	2	13	12	3	0,012	0,001	0,002	0,015
10	21	103	93	21	0,014	0,005	0,005	0,024
50	101	503	453	101	0,060	0,023	0,012	0,095
100	201	1003	903	201	0,420	0,160	0,030	0,610

Tabelle 6.2: Größe der BDDs und Berechnungszeiten für nicht-zyklische Modelle

Tabelle 6.3 zeigt die Ergebnisse für die zyklische Variante des Skalierungsbeispiels. Zyklus bezeichnet hier die Zeit in Sekunden, die für die Entfernung der Ausfallvariablen unter Nutzung des Ansatzes aus [Rau03, Sch07] für zyklische Modelle benötigt wird. Die Ergebnisse sind vergleichbar mit dem nicht-zyklischen Modell.

n	**B**	**K**	**EK**	**EvK**	**Aufbau**	**Zyklus**	**Quant**	$\sum$
1	3	15	9	3	0,015	0,004	0,002	0,021
10	21	105	63	21	0,020	0,016	0,004	0,040
50	101	505	303	101	0,080	0,030	0,011	0,121
100	201	1005	630	201	0,440	0,120	0,028	0,588

Tabelle 6.3: Größe der BDDs und Berechnungszeiten für zyklische Modelle

Abbildung 6.13 zeigt ein abstraktes Beispiel für eine Komponente B mit zwei Konfigurationen mit verschiedenen Basisereignissen.

Die Ergebnisse aus Tabelle 6.4 aus [Sch07] zeigen, dass die BDDs durch die zusätzlichen BDD-Variablen (Spalte **VK**) für die Abbildung der unterschiedlichen Konfigurationen größer werden. Der Aufbau der BDDs dauert allerdings vergleichbar lange. Als Optimierer wurde der MINLP-Optimierer Bonmin [BBC$^+$05] genutzt; dessen Laufzeit ist in der Spalte **Opt** eingetragen.

6.4 Zusammenfassung

In diesem Kapitel wurde die Werkzeugunterstützung der in den Kapiteln 3 bis 4 vorgestellten Modellierungssprachen und Gefahrenanalyse gezeigt. Zu Anfang wurde die entwickelte Benutzerschnittstelle erläutert. Danach wurde die Architektur der Implementierung der Benutzerschnittstelle sowie der Analysetechniken

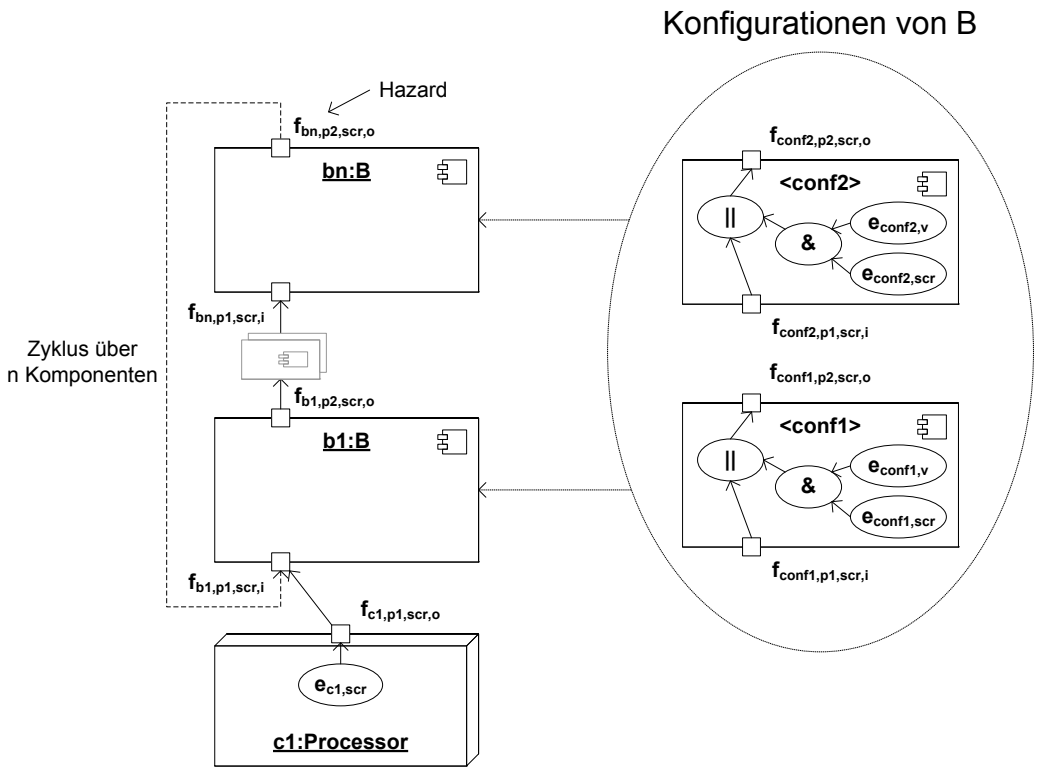


Abbildung 6.13: Skalierungsbeispiel B mit Konfigurationen

n	B	K	EK	EvK	VK	Aufbau	Zyklus	Opt	$\sum$
1	5	26	26	5	1	0,009	0,008	0,076	0,093
10	23	179	197	41	10	0,028	0,024	0,106	0,158
50	103	859	957	201	50	0,200	0,090	0,137	0,427
100	203	1709	1907	401	100	0,590	0,110	0,198	0,898

Tabelle 6.4: Größe der BDDs und Berechnungszeiten für Modelle mit Rekonfiguration

vorgestellt. Auf Grundlage der Werkzeugunterstützung wurde anhand zweier Experimente mit generischen Beispielen untersucht, ob die Gefahrenanalyse auch für Systeme mit einer größeren Anzahl an Komponenten (auch mit verschiedenen Konfigurationen) skaliert.

7 Verwandte Arbeiten

Im Folgenden wird die in dieser Arbeit beschriebene Gefahrenanalyse selbstoptimierender Systeme mit verwandten Arbeiten verglichen. Die Gefahrenanalyse besteht aus einer Modellierungssprache für rekonfigurierbare Architekturen und einer auf den Modellen aufbauenden Analysetechnik. Dementsprechend werden im ersten Abschnitt verwandte Modellierungstechniken für die Rekonfiguration von selbstoptimierenden Systemen vorgestellt. Der zweite Abschnitt behandelt verwandte Arbeiten für die Analysetechnik.

7.1 Graphtransformationen für komponentenbasierte Softwarearchitekturen

Graphtransformationen [Roz97, Bus02] werden oft zur Spezifikation der Rekonfiguration selbstoptimierender Systeme eingesetzt. Der Fokus bei der Auswahl für diesen Vergleich lag auf Ansätzen, in denen Graphtransformationen mit einer Architekturbeschreibungssprache, wie der hier entwickelte Ansatz, integriert sind.

7.1.1 Taentzer

Taentzer et al. stellen in [TGM00] verteilte Graphtransformationen vor. Diese Graphtransformationen ermöglichen sowohl die Spezifikation von lokalen Datenänderungen der einzelnen Teilsysteme des verteilten Systems als auch die Propagierung der Änderungen im System. Des Weiteren ermöglicht der Ansatz die Spezifikation von Änderungen der Systemstruktur durch Hinzufügen und Löschen eines Teilsystems. Der Ansatz unterstützt keine hierarchischen Systeme und keine Ports für die Komponenten. Die explizite Spezifikation eines Kontrollflusses bezüglich der Reihenfolge der Anwendung einzelner Regeln wird nicht angeboten.

7.1.2 Kacem

Eine UML-basierte Spezifikation von Softwarearchitekturen und deren Rekonfiguration mittels Graphtransformationen wird von Kacem et al. in [KKJD06] vorgestellt. Für die Struktursicht erweitern die Autoren die UML 2.0 Komponentendiagramme um ein Profil. Diese Erweiterungen betreffen die Definition eines Typmodells für die Komponentenstrukturen mit Konnektoren, Kardinalitäten, sowie allgemeine OCL-Constraints. Des Weiteren wird die Rekonfiguration modelliert. Eine Rekonfiguration besteht aus einer graphischen Notation auf Basis von Graphtransformationen sowie weiteren OCL Constraints, die Bedingungen für die Ausführung der Rekonfiguration spezifizieren. Die gewählte visuelle Syntax führt bei größeren Rekonfigurationsspezifikationen zu unübersichtlichen Darstellungen. Sequenzen, Entscheidungen und Schleifen können nicht explizit modelliert werden. Eine formale Semantik für die Rekonfigurationssprache wird nicht definiert. Die Typkonformität der Komponenteninstanzstrukturen nach Ausführung einer Rekonfiguration wird nicht betrachtet.

7.1.3 COOL

Die von Grunske vorgestellte Architekturbeschreibungssprache COOL [Gru03c] beinhaltet neben der Spezifikation der Architektur auch eine Spezifikationssprache für qualitätsverbessernde Transformationen der Architektur, wie zum Beispiel N-Modular-Redundancy und Watchdog. Im Gegensatz zu den in dieser Arbeit vorgestellten Komponentenstorydiagrammen steht also nicht die Rekonfiguration einer Komponenteninstanzstruktur zur Laufzeit sondern die Transformation zur Entwurfszeit zur Verbesserung der nicht-funktionalen Qualitätsanforderungen im Fokus. COOL basiert als formales Modell auf typisierten, hierarchischen Hypergraphen zur Spezifikation der Architekturen und Hypergraphtransformationen [Hab92] zur Spezifikation der qualitätsverbessernden Transformationen. Die Transformationen werden graphisch auf Basis eines angepassten Komponenteninstanzdiagramms dargestellt. Zusammengesetzte Operationen wie Sequenzen, Bedingungen und Schleifen von Transformationen werden nicht unterstützt.

7.1.4 Métayer

Daniel Le Métayer stellt in [Mét98, FM98] einen Ansatz vor, der Graphgrammatiken zur Spezifikation von Systemarchitekturen sowie deren Evolution mittels Graphersetzungsregeln mit einer an CSP [Hoa85] angelehnten Sprache zur Definition des Verhaltens der Komponenten kombiniert. Eine Graphgrammatik

beschreibt hierbei Architekturstile, wie z.B. Pipe-/Filter, Client-Server. Durch die Anwendung der Produktionsregeln der Graphgrammatik wird die initiale Komponenteninstanzstruktur erzeugt. Die Komponenteninstanzstruktur wird zur Laufzeit durch eine explizite Entität, den Koordinator, rekonfiguriert. Dieser besteht aus einer Menge von Graphersetzungsregeln, die unter gewissen Bedingungen, z.B. bestimmte Werte öffentlicher Attribute der Komponenteninstanzen, ausgeführt werden. Für den Koordinator können nur einzelne Graphersetzungsregeln zur Rekonfiguration spezifiziert werden. Sequenzen, Entscheidungen und Schleifen sind nur mit einem Umweg über die Attribute der Komponenteninstanzen und derem Verhalten umzusetzen. Im Unterschied zu den in dieser Arbeit vorgestellten Komponentenstorydiagrammen besitzt die von Métayer definierte Sprache keine visuelle Syntax. Des Weiteren unterstützt sie keine Ports sondern nur Komponenteninstanzen und Kanten.

7.1.5 Wermelinger

Wermelinger et al. stellen in [WF99, WF02, WLF01] einen zu Métayer ähnlichen Ansatz vor. Das Verhalten der Komponenten wird dabei mit der Sprache CommUnity [LF99] spezifiziert. Eine Konfiguration besteht aus den Komponenteninstanzen, die mit Konnektoren verbunden sind, deren Verhalten ebenfalls in CommUnity beschrieben wird. Die textuelle Sprache zu Modellierung der Rekonfiguration enthält neben den grundlegenden Operationen wie Erzeugen und Löschen von Elementen einer Konfiguration eine OCL ähnliche Sprache zur Definition der Anwendungsstelle einer Rekonfiguration sowie Attributzuweisungen. Auf Basis dieser grundlegenden Operationen besteht die Möglichkeit Sequenzen, Entscheidungen und Schleifen zu spezifizieren.

Die vorgestellte Sprache ist ausdrucksstark. Sie nutzt allerdings keine visuelle Syntax. Die textuelle Darstellung ist vor allem bei größeren Rekonfigurationsspezifikationen nachteilig gegenüber einer visuellen Syntax. Des Weiteren unterstützt die Architekturbeschreibung sowie die Rekonfiguration keine Ports von Komponenten.

7.1.6 Mechatronic UML Ansätze

Für die Mechatronic UML wurden bereits einige Ansätze [Krä06, GS04, BGS05a, Sch06, BBG+06, HHG08, Hir08] für die regelbasierte Spezifikation der Rekonfiguration entwickelt. Diese werden im Folgenden genauer vorgestellt.

7.1.6.1 Krämer

In [Krä06] wurde ein Ansatz zur regelbasierten Modellierung der Rekonfiguration vorgestellt. Der Ansatz basiert auf Storydiagrammen, die wie in dieser Arbeit auf die graphische Syntax der Komponenteninstanzstrukturen angepasst wurden. In den einzelnen Aktivitäten der Diagramme sind Graphtransformationen eingebettet, in denen Komponenteninstanzen und Portinstanzen sowie Kompositions- und Delegationskonnektoren erzeugt und gelöscht werden können. Des Weiteren wird in der Arbeit beschrieben, wie für solche Modelle Quelltext generiert wird, der unter harten Echtzeitrestriktionen ausgeführt wird. Im Unterschied zu Komponentenstorydiagrammen erlauben die Verhaltensdiagramme von Krämer nicht die Modellierung von Schleifen. Ebenfalls ist die Strukturmodellierung und damit ebenso die Rekonfiguration auf eine fixe Anzahl von Ports beschränkt, die nur aktiviert und deaktiviert werden können. Es existiert kein Typmodell, so dass keine starke Typsicherheit für die Verhaltensdiagramme gewährleistet ist.

7.1.6.2 Schilling

Schilling et al. beschreiben in [GS04, BGS05a, Sch06, BBG+06] einen Ansatz für die Rekonfiguration von nicht-hierarchischen Komponenteninstanzstrukturen. Dieser Ansatz betrachtet als Anwendungsbeispiel die Instantiierung von Echtzeitkoordinationsmustern zwischen Komponenteninstanzen. Die vorgestellte Lösung basiert auf einer regelbasierten Spezifikation der Transformation der Komponenteninstanzstrukturen und Echtzeitkoordinationsmusterinstanzen. Dies bedeutet, dass in gewissen Situationen Echtzeitkoordinationsmuster instantiiert und zerstört werden. Als Formalismen werden Klassendiagramme zur Strukturdefinition, Objektdiagramme für Instanzstrukturen sowie Storypattern zur regelbasierten Beschreibung der Transformation der Strukturen genutzt. Der Ansatz unterstützt bei der Modellierung nur nicht-hierarchische Klassendiagramme. Es können also keine Ports und Rekonfigurationen von Komponentenhierarchien modelliert werden. Des Weiteren ist die Modellierung auf einzelne Änderungsoperationen beschränkt, da nur Storypatterns und keine Storydiagramme unterstützt werden.

7.1.6.3 Hirsch

Der Ansatz von Schilling et al. beschränkt sich auf Echtzeitkoordinationsmuster mit einer festen und beim Entwurf bereits bekannten Anzahl an Rollen. Hirsch et al. [HHG08, Hir08] betrachten parametrisierte Koordinationsmuster mit einer variablen Anzahl an Rollen. Während die Rollen bisher eine implizite Kardinalität von 0..1 hatten, wurden von Hirsch et al. Multirollen eingeführt, die

eine beliebige andere Kardinalität haben können. Die Instantiierung und Zerstörung der Ports als Instanzen der Rollen wird ebenfalls wie bei Schilling et al. mit Graphtransformationen beschrieben. Die Graphtransformationen werden als Seiteneffekte in den Real-Time Statecharts, als Verhaltensspezifikationen der Rollen, aufgerufen. Der Ansatz unterstützt nur einzelne Änderungsoperationen. Sequenzen, Entscheidungen und Schleifen könnten in den Real-Time Statecharts spezifiziert werden. Der Ansatz unterstützt Ports und nutzt eine visuelle Syntax. Typisierung wird nicht betrachtet.

7.2 Gefahrenanalyse

Nach der Beschreibung verwandter Modellierungssprachen für rekonfigurierbare Architekturen werden in einer kurzen Übersicht verwandte Arbeiten vorgestellt, die Konzepte aus der komponentenbasierten Architekturmodellierung auf eine Gefahrenanalyse anwenden. Danach werden zwei Ansätze genauer vorgestellt, die es teilweise ermöglichen Gefahrenanalysen auf rekonfigurierbaren Systemen durchzuführen.

7.2.1 Komponentenbasierte Gefahrenanalysen

Im Bereich der Gefahrenanalysen wurde in den letzten Jahren die Modularisierung durch komponentenbasierte Analysen ebenfalls als notwendig erkannt [FM93, Wal05, KLM03, Gru03b, PM99]. Alle diese Ansätze nutzen wiederverwendbare Module für einzelne Teile des Systems, die deren Verhalten im Fehlerfall analog zur Fehlerpathologie aus [Lap92] spezifizieren. Diese Module werden dann je nach System zusammengesetzt und hinsichtlich der Gefahren analysiert. Die Komponentenstruktur des Systems wird im Gegensatz zu [FM93, KLM03, PM99] nur in [Wal05, Gru03b] ausgenutzt.

Eine Gefahrenanalyse für selbstoptimierende Systeme muss diese spezifischen Eigenschaften wie common-mode Fehler und zyklische Modelle unterstützen. Während common-mode Fehler alle genannten Ansätze korrekt behandeln, werden zyklischen Strukturmodelle neben dem hier vorgestellten Ansatz nur von [FM93, Wal05, PM99] unterstützt.

Alle diese Ansätze setzen eine statische Struktur des Systems voraus. Sie sind daher nicht für selbstoptimierende Systeme mit Rekonfiguration geeignet.

7.2.2 State Event Fault Trees

State Event Fault Trees (SEFTs) [KG04, GKP05, KGF07] ermöglichen für eine Komponente nicht nur eine Fehlerpropagierung zu beschreiben sondern zustandsbasiertes Verhalten mit Zeit. Für eine Komponente wird dabei ein Zustandsdiagramm spezifiziert. Die Zustände sind mit Transitionen verbunden. Eine Transition kann nun je nach Typ der Transition schalten, wenn eine gewisse Zeit seit dem Eintreten in den Quellzustand vergangen ist, eine stochastisch verteilte Zeit verstrichen ist oder ein Ereignis, wie zum Beispiel ein Fehler, eingetreten ist. Diese Ereignisse können neben Fehlern allerdings auch andere Ereignisse wie Nachrichtenaustausch sein.

Eine Gefahrenanalyse mit SEFTs basiert auf einer Spezifikation der (hierarchischen) Systemarchitektur. Auf Basis dieser Systemarchitektur werden die einzelnen SEFTs miteinander kombiniert. Die Semantik der SEFTs ist durch eine Übersetzung auf Deterministische Stochastische Petri Netze (DSPNs) [MC87] definiert. Die Modelle können damit mit passenden Werkzeugen wie TimeNET [ZFGH00] qualitativ und quantitativ analysiert werden. Die Modelle dürfen laut [GKP05] allerdings nicht zu groß werden, da sonst die Analyse nicht mehr skaliert.

Mit SEFTs werden wiederholte Basisereignisse korrekt betrachtet. Zyklen sind mit einer expliziten Zeitverzögerung erlaubt. Die Systemstruktur ist statisch, da keine Rekonfiguration betrachtet wird. Mit Einschränkungen könnte es möglich sein, die zustandsbasierte Modellierung der SEFTs für die Spezifikation von Rekonfiguration zu nutzen. Dies wird von den Autoren allerdings nicht betrachtet.

7.2.3 Deductive Cause-Consequence Analysis

Die *Deductive Cause-Consequence Analysis (DCCA)* [ORS06] nutzt im Unterschied zu allen übrigen Arbeiten keine extra spezifizierten Fehlerverhaltensmodelle für die Komponenten des Systems sondern das bereits spezifizierte Verhalten. Das Verhalten wird mit Automaten modelliert und manuell um Fehlerautomaten erweitert. Diese modellieren die unterschiedlichen Fehlerzustände der Komponente wie einen temporären oder permanenten Ausfall. Die Gefahren werden mit Hilfe von CTL [Eme90] formalisiert.

Auf Basis dieses formalen Modells wird dann mit Modelchecking verifiziert, ob zum einen die Gefahren eintreten können und zum anderen welche Fehlerzustände dazu eintreten müssen. Durch die Nutzung der Verhaltensmodelle ergibt sich eine sehr genaue und detaillierte Analyse. Bei großen Modellen könnten sich allerdings durch das Modelchecking Skalierungsprobleme ergeben.

DCCA wurde in [GOR06] für die Analyse rekonfigurierender Systeme erwei-

tert. Hierbei wird die Rekonfiguration des Systems durch einen speziellen, Rekonfigurator genannten, endlichen Automaten spezifiziert. Auf Basis dieses Modells werden dann für rekonfigurierende Systeme die Fehlerzustände, die in einer Gefahr resultieren und nicht durch eine Rekonfiguration behoben werden können, berechnet. Der Ansatz unterstützt sowohl qualitative als auch quantitative Analysen.

Im Vergleich zu dem hier entwickelten Ansatz unterstützt der Ansatz allerdings nicht die regelbasierte Spezifikation der Rekonfiguration der Komponentenstruktur.

7.3 Zusammenfassung

Die Modellierung rekonfigurierbarer Architekturen wird durch eine Reihe von Ansätzen auf Basis von Graphtransformationen ermöglicht. Alle Ansätze unterstützen die Spezifikation einzelner Rekonfigurationen. Sequenzen von Rekonfigurationen sowie die Betrachtung von Typisierung und Typkonformität werden nur von einigen Ansätzen unterstützt. Nur ein Ansatz [Krä06] betrachtet explizit die Domäne selbstoptimierender Systeme, in denen die Rekonfiguration unter harten Echtzeitkriterien ausgeführt werden muss. Allerdings erlaubt dieser Ansatz keine Schleifen in der Spezifikation der Rekonfiguration und garantiert nicht die Typkonformität der durch die Ausführung der Rekonfigurationen entstehenden Konfigurationen.

Es existieren einige verwandte komponentenbasierte Gefahrenanalysen auf Basis einer Fehlerpropagierung, die zum vorliegenden Ansatz vergleichbare Fähigkeiten haben. Diese unterstützen allerdings nur statische Komponentenstrukturen. Der DCCA-Ansatz [ORS06, GOR06] betrachtet als einziger auch rekonfigurierbare Systeme. Durch die Nutzung von Modelchecking auf den Verhaltensmodellen verfolgt er allerdings einen anderen Ansatz als der hier vorgestellte. Die Rekonfiguration wird des Weiteren auch nicht durch ein Regelsystem sondern nur durch eine Aufzählung der unterschiedlichen Zustände modelliert.

Insgesamt lässt sich feststellen, dass kein verwandter Ansatz existiert, der eine Gefahrenanalyse selbstoptimierender Systeme mit einer regelbasierten Modellierung der Strukturrekonfiguration verbindet.

8 Zusammenfassung und Ausblick

Selbstoptimierung ist ein Konzept, welches ermöglicht, mechatronische Systeme mit einer inhärenten Teilintelligenz auszustatten [GKP08]. Diese Systeme reagieren autonom auf Änderungen in der Umgebung. Dies geschieht durch eine Anpassung der Ziele und nachfolgend einer Änderung des Verhaltens [FGK+04].

Damit selbstoptimierende Systeme auch in sicherheitskritischen Umgebungen eingesetzt werden können, muss die Sicherheit überprüft werden. Fehler im selbstoptimierenden System können die Sicherheit beeinträchtigen. Fehler müssen daher erkannt und ihre Auswirkungen auf die Sicherheit analysiert werden, falls sie nicht behoben werden können. Bisher wurde für die Mechatronic UML die Erkennung systematischer Fehler, z.B. falsch modellierte Verhaltensmodelle, mit Hilfe formaler Verifikationstechniken betrachtet [GTB+03, GBSO04, BBG+06]. Zufällige Fehler, wie zum Beispiel fehlerhafte Werte von Sensoren, wurden bisher nicht betrachtet.

Im Rahmen dieser Arbeit wurde daher eine Gefahrenanalyse vorgestellt, die die bisherigen formalen Verifikationstechniken hinsichtlich der Betrachtung zufälliger Fehler ergänzt. Die Gefahrenanalyse basiert auf einer Betrachtung der Propagierung der Fehler in der Systemstruktur als Ursachen für Gefahren. Ziel bei der Entwicklung der Gefahrenanalyse war, die spezifischen Eigenschaften selbstoptimierender Systeme, wie zyklische Strukturen, (regelbasierte) Rekonfiguration und common-mode Fehler, geeignet zu unterstützen. Auf Basis der Gefahrenanalyse wurden Heuristiken entwickelt, die Stellen zur Verbesserung der Systemstruktur durch Fehlertoleranztechniken identifizieren. Verwandte Ansätze, wie [FM93, Wal05, KLM03, Gru03b, GKP05, KGF07, PM99, ORS06, GOR06] betrachten manche dieser Eigenschaften. Ein Ansatz auf Basis einer Fehlerpropagierung der Komponenten, der alle diese Eigenschaften unterstützt, existierte allerdings noch nicht.

Ein besonderer Fokus bei der Gefahrenanalyse lag auf der geeigneten Modellierung der Rekonfiguration selbstoptimierender Systeme. In dieser Arbeit wurde ein regelbasierter Ansatz für die Modellierung der Rekonfiguration verfolgt. Die Regeln beschreiben, wie eine Konfiguration in eine andere Konfiguration transformiert wird. Die Modellierungssprache wendet Graphtransformationen auf komponentenbasierte Strukturen an. Die Syntax ist dabei graphisch und an Komponentendiagramme angelehnt, so dass sich die Modellie-

rung der Rekonfiguration sehr gut mit der Strukturmodellierung integriert. Neben einzelnen Rekonfigurationen können auch Sequenzen, Entscheidungen und Schleifen spezifiziert werden. Die Sprache ist weiterhin so definiert, dass ein kontextsensitiv syntaxgesteuerter Editor umgesetzt werden konnte, der garantiert, dass bei der Ausführung der Rekonfigurationen nur typkonforme Konfigurationen entstehen. Der vorgestellte Ansatz geht damit über frühere Arbeiten [Krä06, Sch06, Hir08] im Rahmen der MECHATRONIC UML sowie verwandte Arbeiten [TGM00, KKJD06, Gru03c, Mét98, WF99, WF02] hinaus.

Die vorgestellten Konzepte wurden prototypisch als Plugins für das Entwicklungswerkzeug Fujaba4Eclipse umgesetzt.

8.1 Ausblick

Auf Basis der Ergebnisse der Gefahrenanalyse sowie der beschriebenen Heuristiken für die Ansatzpunkte für eine Verbesserung der Sicherheit können an ausgewählten Stellen der Komponenteninstanzstruktur Fehlertoleranztechniken eingesetzt werden. Es wurden bereits strukturelle Spezifikationen von Fehlertoleranzmustern auf Basis von Graphtransformationen entwickelt [TBG04, Tic06, THMD08], die eine automatisierte Anwendung der Fehlertoleranztechniken ermöglichen. Vor allem bei der Einführung von redundanten Softwarekomponenteninstanzen muss auch deren Verteilungsstruktur auf die Ausführungshardware korrekt betrachtet werden. In [Pau05b] wurden diesbezügliche Einschränkungen formuliert und umgesetzt. Bei der Anwendung der Fehlertoleranzmuster wird die Komponentenstruktur angepasst. Somit ergibt sich eine Änderung der Fehlerpropagierung in der Systemstruktur und damit neue Ergebnisse bei einer wiederholten Gefahrenanalyse.

Die Herausforderung bei der Integration der Fehlertoleranzmuster in die Konzepte der vorliegenden Arbeit liegt darin, dass bei der Anwendung nicht einfach wie bisher in [TBG04, Tic06, THMD08] nur die Komponenteninstanzstruktur geändert werden kann. Die Komponenteninstanzstruktur muss konform zur Definition der Komponententypen sein, die also dann auch angepasst werden muss. Die spezifizierten Komponentenstorydiagramme müssen ebenfalls angepasst werden, da diese die Instanzstruktur erzeugen und anpassen und somit auch um die Anwendung des Fehlertoleranzmusters ergänzt werden müssen.

Neben einer Anpassung der Systemstruktur zur Entwurfszeit für die Anwendung von Fehlertoleranz können alternativ auch Rekonfigurationen zur Laufzeit durchgeführt werden. Dies wird als *selbstheilend* [Shi05] bezeichnet. Die Gefahrenanalyse betrachtet derzeit nur die einzelnen Konfigurationen und nicht die Übergänge zwischen diesen. Dies bedeutet, dass Rekonfigurationen, die wie bei

selbstheilenden Systemen als Reaktion auf Fehler durchgeführt werden, nicht betrachtet werden. Hier bietet es sich an, die Ansätze aus [TG03a, TG03a, TG04, TSG04, TGSP05, TG05, DDK+07] geeignet zu integrieren.

Die Fehlerpropagierungsmodelle der Gefahrenanalyse betrachten derzeit keine Zeit. Deshalb besteht die Möglichkeit, dass die Analyse Gefahren ermittelt, die unter Berücksichtigung der zeitlichen Beschränkungen, denen die Fehlerpropagierung unterliegt, nicht auftreten können (false positives). Des Weiteren ist möglich, dass Gefahren nicht ermittelt werden, die erst unter den Echtzeiteigenschaften des konkreten Systems auftreten (false negatives). Dies kann im Besonderen im Zusammenwirken mit der Rekonfiguration des Systems auftreten. Die Erweiterung der Fehlerpropagierungsmodelle um Zeit, wie sie zum Beispiel zum Zweck der Online-Diagnose in [AKB04] umgesetzt wurde, wäre also ein interessanter Aspekt für weiterführende Forschung.

Für eine bessere Skalierbarkeit der Gefahrenanalyse bei großen Modellen und vor allem bei der Rekonfiguration ist zu prüfen, ob eine Dekomposition des Modells möglich ist. Das heißt, es werden Teile des Modells analysiert und später bei der Analyse des Gesamtsystems als Black Box wiederverwendet. Dieser Black Box wird dazu die berechnete Fehlerwahrscheinlichkeit und die Fehlerpropagierung des Teilmodells zugeordnet. Diese Dekomposition ermöglicht eine Abstraktion des Modells, was seine Komplexität für die Analyse verringert. Dabei muss die Abschätzung der Fehlerwahrscheinlichkeit und -propagierung für ein Teilmodell pessimistisch sein. Andernfalls kann es vorkommen, dass sich die Gefahrenwahrscheinlichkeit des Gesamtsystems nach einer Verfeinerung als größer herausstellt, als sie zuvor berechnet wurde, und somit geforderte Schranken für die Gefahrenwahrscheinlichkeit nicht mehr eingehalten werden. Dadurch wäre das System weniger sicher, als durch die Analyse gezeigt.

Die Fehlerpropagierung muss derzeit manuell vom Entwickler spezifiziert werden. Fehler bei dieser Spezifikation können dazu führen, dass die Fehlerpropagierungsmodelle keine korrekte Abstraktion des Verhaltens einer Komponente im Fehlerfall darstellen. Es ist zu untersuchen, ob das Verfahren aus [ORS06, GOR06] genutzt werden kann, um das Fehlerpropagierungsverhalten per Modelchecking automatisch zu bestimmen. Hierzu müsste das Verhalten einer Komponente, um interne Fehlerzustände sowie eingehende und ausgehende Ausfälle, wie in [ORS06] beschrieben, erweitert werden. Auf diesem erweiterten Modell könnte dann überprüft werden, ob temporallogische Formeln, die mögliche Fehlerpropagierungen beschreiben, gültig sind.

Eine Alternative hierzu ist der Einsatz von Simulationen der Modelle hinsichtlich des Fehlerverhaltens in Kombination mit Techniken des Data Minings [BSN07], um automatisiert das Fehlerpropagierungsmodell zu generieren.

In [Fra06] wurden für die Entwicklung der Prinziplösung mechatronischer Sys-

teme in den frühen Phasen mehrere Modellierungssprachen vorgeschlagen, mit denen verschiedene Aspekte wie Struktur, Anforderungen und Verhalten spezifiziert werden können. Die vorgeschlagene Gefahrenanalyse kann prinzipiell bereits auf der Prinziplösung durchgeführt werden, in dem für die Komponenten der Wirkstruktur Fehlerpropagierungsmodelle und Gefahren spezifiziert werden. So kann bereits in den frühen Phasen, wenn noch keine detaillierten Verhaltensmodelle existieren, bereits die Sicherheit des entworfenen Systems überprüft werden.

Da die Sicherheit während der ganzen Entwicklung kontinuierlich mitbetrachtet werden muss [IEC, Sto96, Lev95], ist folgerichtig eine Integration der Gefahrenanalyse auf der Prinziplösung mit dem Übergang der Modelle der Prinziplösung in die domänenspezifischen Modellen, wie in [GFG+05, GGS+07] beschrieben, durchzuführen.

Für parametrisierte Koordinationsmuster [HHG08, Hir08] wurden Erzeugungs- und Reduktionsregeln, ähnlich den in dieser Arbeit beschriebenen Komponentenstorypattern, definiert, deren Aufruf als Seiteneffekt in die Real-Time Statecharts der Musterrollen integriert wurden. Diese Integration könnte eine Basis für eine Kombination der Komponentenstorydiagramme mit dem Echtzeitverhalten der Komponenten, in Form von Real-Time Statecharts, sein.

Literaturverzeichnis

Eigene Veröffentlichungen

[BGH+05] 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, May 2005, S. 670–671

[BGH+07] Burmester, Sven ; Giese, Holger ; Henkler, Stefan ; Hirsch, Martin ; Tichy, Matthias ; Gambuzza, Alfonso ; Münch, Eckehard ; Vöcking, Henner: Tool Support for Developing Advanced Mechatronic Systems: Integrating the Fujaba Real-Time Tool Suite with CAMeL-View. In: *Proc. of the 29th International Conference on Software Engineering (ICSE), Minneapolis, Minnesota, USA*, IEEE Computer Society Press, May 2007, S. 801–804

[BGN+03] Burmester, Sven ; Giese, Holger ; Niere, Jörg ; Tichy, Matthias ; Wadsack, Jörg P. ; Wagner, Robert ; Wendehals, Lothar ; Zündorf, Albert: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In: *Proc. of the Workshop on Tool-Integration in System Development (TIS), Helsinki, Finland, Satellite Event of the joint Conferences ESEC/FSE 2003*, 2003, S. 51–56

[BGN+04] Burmester, Sven ; Giese, Holger ; Niere, Jörg ; Tichy, Matthias ; Wadsack, Jörg P. ; Wagner, Robert ; Wendehals, Lothar ; Zündorf, Albert: Tool Integration at the Meta-Model Level within the FUJABA Tool Suite. In: *International Journal on Software Tools for Technology Transfer (STTT)* 6 (2004), August, Nr. 3, S. 203–218

[BGST05] Burmester, Sven ; Giese, Holger ; Seibel, Andreas ; Tichy, Matthias: Worst-Case Execution Time Optimization of Story Pat-

terns for Hard Real-Time Systems. In: *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, 2005, S. 71–78

[BGT05] Burmester, Sven ; Giese, Holger ; Tichy, Matthias: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In: Assmann, Uwe (Hrsg.) ; Rensink, Arend (Hrsg.) ; Aksit, Mehmet (Hrsg.): *Model Driven Architecture: Foundations and Applications* Bd. 3599, Springer Verlag, August 2005 (Lecture Notes in Computer Science), S. 47–61

[DDK+07] Danne, Christoph ; Dück, Viktor ; Klöpper, Benjamin ; Brinkmann, Jürgen ; Tichy, Matthias: Considering Runtime Restrictions in Self-Healing Distributed Systems. In: *Proceedings of the IEEE 21st International Conference on Advanced Information Networking and Applications (AINA-07), Niagara Falls, Canada*, IEEE Computer Society Press, May 2007

[DHK+08] Dell'Aere, Allessandro ; Hirsch, Martin ; Klöpper, Benjamin ; Koster, Markus ; Krupp, Alexander ; Müller, Thomas ; Oberthür, Simon ; Romaus, Christoph ; Schmidt, Alexander ; Sondermann-Wölke, Christoph ; Tichy, Matthias ; Vöcking, Henner: *Verlässlichkeit selbstoptimierender Systeme.* HNI-Verlag, 2008

[FGM+07] Frank, Ursula ; Giese, Holger ; Müller, Thomas ; Oberthür, Simon ; Romaus, Christoph ; Tichy, Matthias ; Vöcking, Henner: Potenziale und Risiken der Selbstoptimierung für die Verlässlichkeit mechatronischer Systeme. In: *Proc. of the Fifth Paderborner Workshop Entwurf mechatronischer Systeme*, HNI-Verlagsschriftenreihe, March 2007

[GFG+05] Gausemeier, Jürgen ; Frank, Ursula ; Giese, Holger ; Klein, Florian ; Schmidt, Andreas ; Steffen, Daniel ; Tichy, Matthias: A Design Methodology for Self-Optimizing Systems. In: Verkehr Braunschweig e.V., Gesamtzentrum für (Hrsg.): *Contributions to the 6th Braunschweig conference of Automation, Assistance and Embedded Real Time Platforms for Transportation - Air-planes, Vehicles, Trains - (AAET2005)* Bd. II, GZVB, February 2005. – ISBN 3–937655–04–2, S. 456–479

[GGS+07] Gausemeier, Jürgen ; Giese, Holger ; Schäfer, Wilhelm ; Axenath, Björn ; Frank, Ursula ; Henkler, Stefan ; Pook, Sebastian

; Tichy, Matthias: Towards the Design of Self-Optimizing Mechatronic Systems: Consistency between Domain-Spanning and Domain-Specific Models. In: *Proc. of the 16th International Conference on Engineering Design (ICED), Paris, France*, 2007

[GHH+06] Giese, Holger ; Henkler, Stefan ; Hirsch, Martin ; Tichy, Matthias ; Vöcking, Henner: Modellbasierte Entwicklung vernetzter, mechatronischer Systeme am Beispiel der Konvoifahrt autonom agierender Schienenfahrzeuge. In: Gausemeier, Jürgen (Hrsg.) ; Rammig, Franz J. (Hrsg.) ; Schäfer, Wilhelm (Hrsg.) ; Trächtler, Ansgar (Hrsg.) ; Wallaschek, Jürgen (Hrsg.): *4. Paderborner Workshop Entwurf mechatronischer Systeme, 27.-28. September, 2006, Paderborn, Deutschland* Bd. 189, 2006 (HNI-Verlagsschriftenreihe), S. 457–473

[GHH+08] 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.* IGI Global, Hershey, PA, 2008

[GST+03] Giese, Holger ; Schilling, Daniela ; Tichy, Matthias ; Burmester, Sven ; Schäfer, Wilhelm ; Flake, Stephan: Towards the Compositional Verification of Real-Time UML Designs / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Deutschland, July 2003 (tr-ri-03-241). – Forschungsbericht. – 1–47 S.

[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)

[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. – ISBN 1–58113–743–5, S. 38–47

[GTS04] Giese, Holger ; Tichy, Matthias ; Schilling, Daniela: Compositional Hazard Analysis of UML Components and Deployment Models. In: *Proc. of the 23rd International Conference on Computer Safety, Reliability and Security (SAFECOMP), Potsdam, Germany* Bd. 3219, Springer Verlag, September 2004 (Lecture Notes in Computer Science)

[HT08] Holtmann, Jörg ; Tichy, Matthias: Component Story Diagrams in Fujaba4Eclipse. In: *Proc. of the 6th International Fujaba Days 2008, Dresden, Germany*, 2008, S. 44–47

[HTS+08a] Henke, Christian ; Tichy, Matthias ; Schneider, Tobias ; 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

[HTS+08b] Henke, Christian ; Tichy, Matthias ; Schneider, Tobias ; Böcker, Joachim ; Schäfer, Wilhelm: System Architecture and Risk Management for Autonomous Railway Convoys. In: *Proceedings of the 2nd Annual IEEE International Systems Conference, Montreal, Canada*, 2008

[TBG04] Tichy, Matthias ; Becker, Basil ; Giese, Holger: Component Templates for Dependable Real-Time Systems. In: Schürr, Andy (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany* Bd. tr-ri-04-253, University of Paderborn, September 2004 (Technical Report), S. 27–30

[TG03a] Tichy, Matthias ; Giese, Holger: An Architecture for Configurable Dependability of Application Services. In: Lemos, Rogério de (Hrsg.) ; Gacek, Christina (Hrsg.) ; Romanowsky, Alexander (Hrsg.): *Proc. of the Workshop on Software Architectures for Dependable Systems (WADS) (International Conference on Software Engineering 2003 Workshop 7), Portland, USA*, 2003

[TG04] Tichy, Matthias ; Giese, Holger: A Self-Optimizing Run-Time Architecture for Configurable Dependability of Services. In: Lemos, Rogério de (Hrsg.) ; Gacek, Cristina (Hrsg.) ; Romanovsky, Alexander (Hrsg.): *Architecting Dependable Systems II* Bd. 3069. Springer Verlag, 2004, S. 25–51

[TG05] Tichy, Matthias ; Giese, Holger: Extending Fault Tolerance Patterns by Visual Degradation Rules. In: *Proc. of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, 2005, S. 67–74

[TGS06] Tichy, Matthias ; Giese, Holger ; Seibel, Andreas: Story Diagrams in Real-Time Software. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2006 (Technical Report), S. 15–22

[TGSP05] Tichy, Matthias ; Giese, Holger ; Schilling, Daniela ; Pauls, Wladimir: Computing Optimal Self-Repair Actions: Damage Minimization versus Repair Time. In: Lemos, Rogério de (Hrsg.) ; Romanovsky, Alexander (Hrsg.): *Proc. of the ICSE 2005 Workshop on Architecting Dependable Systems, St. Louis, Missouri, USA*, ACM Press, May 2005, S. 1–6

[TH07] Tichy, Matthias ; Henkler, Stefan: Towards a Transformation Language for Component Structures. In: *Proc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, 2007

[THHO08] Tichy, Matthias ; Henkler, Stefan ; Holtmann, Jörg ; Oberthür, Simon: Component Story Diagrams: A Transformation Language for Component Structures in Mechatronic Systems. In: *Postproc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, 2008

[THMD08] Tichy, Matthias ; Henkler, Stefan ; Meyer, Matthias ; Detten, Markus von: Safety of Component-Based Systems: Analysis and Improvement using Fujaba4Eclipse. In: *Companion Proceedings of the 30th International Conference on Software Engineering (ICSE), Leipzig, Germany*, 2008

[Tic06] Tichy, Matthias: Pattern-Based Synthesis of Fault-Tolerant Embedded Systems. In: *Proc. of the Doctoral Symposium of the Fourteenth ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE), Portland, Oregon, USA*, ACM Press, November 2006, S. 13–18

[TMG06] Tichy, Matthias ; Meyer, Matthias ; Giese, Holger: On Semantic Issues in Story Diagrams. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, University of Paderborn, September 2006 (Technical Report), S. 10–14

[TSG04] Tichy, Matthias ; Schilling, Daniela ; Giese, Holger: Design of Self-Managing Dependable Systems with UML and Fault Tolerance Patterns. In: *Proc. of the Workshop on Self-Managed Systems (WOSS) 2004, FSE 2004 Workshop, Newport Beach, USA*, 2004

Literatur

[AD94] Alur, Rajeev ; Dill, David L.: A Theory of Timed Automata. In: *Theoretical Computer Science* 126 (1994), S. 183–235

[AKB04] Abdelwahed, Sherif ; Karsai, Gabor ; Biswas, Gautam: System diagnosis using hybrid failure propagation graphs. In: *The 15th International Workshop on Principles of Diagnosis*, 2004

[Ake78] Akers, Sheldon B.: Binary Decision Diagrams. In: *IEEE Trans. Computers* 27 (1978), Nr. 6, S. 509–516

[ALRL04] Avizienis, Algirdas ; Laprie, Jean-Claude ; Randell, Brian ; Landwehr, Carl: Basic Concepts and Taxonomy of Dependable and Secure Computing. In: *IEEE Transactions on Dependable and Secure Computing* 01 (2004), Nr. 1, S. 11–33. – ISSN 1545–5971

[BBC+05] Bonami, P. ; Biegler, L.T. ; Conn, A.R. ; Cornuejols, G. ; Grossmann, I.E. ; Laird, C.D. ; Lee, J. ; Lodi, A. ; Margot, F. ; N.Sawaya ; Waechter, A.: An Algorithmic Framework for Convex Mixed Integer Nonlinear Programs. 2005 (IBM Research Report RC23771). – Forschungsbericht

[BBD+02] Behrmann, Gerd ; Bengtsson, Johan ; David, Alexandre ; Larsen, Kim G. ; Pettersson, Paul ; Yi., Wang: Uppaal Implementation Secrets. In: *Proc. of 7th International Symposium on Formal Techniques in Real-Time and Fault Tolerant Systems*, 2002

[BBD+07] Bentler, Sebastian ; Brügger, Stefan ; Dorociak, Rafal ; Hoffmann, Florian ; Hoffmann, Thomas ; Holtmann, Jörg

; Janzen, Waldemar ; Knoop, Markus ; Lüken, Frank ; Oberhoff, Andreas ; Polat, Sirvan ; Reinert, Rafal ; Detten, Markus von ; Werner, Christoph: *Abschlussdokumentation der Projektgruppe Automotive Software-Engineering*. 2007

[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

[BCDW04] Bradbury, Jeremy S. ; Cordy, James R. ; Dingel, Jürgen ; Wermelinger, Michel: A survey of self-management in dynamic software architecture specifications. In: Garlan, David (Hrsg.) ; Kramer, Jeff (Hrsg.) ; Wolf, Alexander L. (Hrsg.): *Proceedings of the 1st ACM SIGSOFT Workshop on Self-Managed Systems, WOSS 2004, Newport Beach, California, USA, October 31 - November 1, 2004*, ACM, 2004. – ISBN 1–58113–989–6, S. 28–33

[BCM+04] Böckle, Günter ; Clements, Paul ; McGregor, John D. ; Muthig, Dirk ; Schmid, Klaus: Calculating ROI for Software Product Lines. In: *IEEE Software* 21 (2004), May/June, Nr. 3, S. 23–31

[BG05] Burmester, Sven ; Giese, Holger: Visual Integration of UML 2.0 and Block Diagrams for Flexible Reconfiguration in Mechatronic UML. In: *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), Dallas, Texas, USA*, IEEE Computer Society Press, September 2005, S. 109–116

[BGS05a] Becker, Basil ; Giese, Holger ; Schilling, Daniela: A Plugin for Checking Inductive Invariants when Modeling with Class Diagrams and Story Patterns. In: *Proc. of the 3rd International Fujaba Days 2005, Paderborn, Germany*, 2005, S. 1–4

[BGS05b] 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 (Lecture Notes in Computer Science), S. 25–40

[BLM02] Bianco, Vieri D. ; Lavazza, Luigi ; Mauri, Marco: Formalization of UML Statecharts for Real-Time Software Modeling. In: *Proceedings of the 6th International Conference on Integrated Design and Process Technology, IDPT2002*, 2002

[Bol90] Bolton, Martin: *Digital systems design with programmable logic.* Addison-Wesley, 1990

[Bra04] Bradbury, Jeremy: Organizing Definitions and Formalisms for Dynamic Software Architectures / Queen's University, Kingston, Ontario, Canada. 2004 (2004-447). – Forschungsbericht

[Bry86] Bryant, Randal E.: Graph-Based Algorithms for Boolean Function Manipulation. In: *IEEE Trans. Computers* 35 (1986), Nr. 8, S. 677–691

[Bry92] Bryant, Randal E.: Symbolic Boolean manipulation with ordered binary-decision diagrams. In: *ACM Computing Surveys* 24 (1992), September, Nr. 3, S. 293 – 318

[BSDB00] Bradley, David A. ; Seward, Derek ; Dawson, David ; Burge, Stuart: *Mechatronics and the Design of Intelligent Machines and Systems.* Nelson Thornes, 2000

[BSN07] Balzer, Heinrich ; Stein, Benno ; Niggemann, Oliver: Diagnose in verteilten automotiven Systemen. In: *Proc. of the Fifth Paderborner Workshop Entwurf mechatronischer Systeme* Bd. 210, HNI-Verlagsschriftenreihe, March 2007, S. 243–254

[BSVMH84] Brayton, Robert K. ; Sangiovanni-Vincentelli, Alberto L. ; McMullen, Curtis T. ; Hachtel, Gary D.: *Logic Minimization Algorithms for VLSI Synthesis.* Kluwer Academic Publishers, 1984

[Bur02] Burmester, Sven: *Generierung von Java Real-Time Code für zeitbehaftete UML Modelle*, University of Paderborn, Department of Computer Science, Paderborn, Germany, Diplomarbeit, September 2002

[Bur06] Burmester, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems.* Berlin, Deutschland, Universität Paderborn, Dissertation, 2006

[Bus02] Busatto, Giorgio: *An abstract model of hierarchical graphs and hierarchical graph transformation*, Universität Paderborn, Dissertation, 2002

[CCG+02] Cimatti, A. ; Clarke, E. ; Giunchiglia, E. ; Giunchiglia, F. ; Pistore, M. ; Roveri, M. ; Sebastiani, R. ; Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In: *Proc. of the International Conference on Computer-Aided Verification (CAV 2002)* Bd. 2404. Copenhagen, Denmark : Springer Verlag, July 2002 (Lecture Notes in Computer Science)

[CM93] Coudert, Olivier ; Madre, Jean C.: Fault tree analysis: 10^{20} prime implicants and beyond. In: *Proceedings of the Annual Reliability and Maintainability Symposium*. Atlanta, GA , USA : IEEE Press, January 1993, S. 240–245

[CSRL01] Cormen, Thomas H. ; Stein, Clifford ; Rivest, Ronald L. ; Leiserson, Charles E.: *Introduction to Algorithms*. McGraw-Hill Higher Education, 2001

[DB98] Drechsler, Rolf ; Becker, Bernd: *Binary Decision Diagrams - Theory and Implementation*. Springer Verlag, 1998

[Dib02] Dibble, Peter C.: *Real-Time Java Platform Programming*. Prentice Hall, 2002

[Dit00] Ditze, Carsten: *Towards Operating System Synthesis*. University of Paderborn, Germany, 2000 (HNI-Verlagsschriftenreihe 157)

[DMY02] David, Alexandre ; Möller, Oliver ; Yi, Wang: Formal Verification of UML Statecharts with Real-Time Extensions. In: Kutsche, Ralf-Detler (Hrsg.) ; Weber, Herbert (Hrsg.): *Proceedings of 5th International Conference on Fundamental Approaches to Software Engineering (FASE 2002)*. Grenoble, France : Springer Verlag, 2002 (Lecture Notes in Computer Science 2306), S. 218–232

[Eme90] Emerson, E. A.: Temporal and Modal Logic. In: *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*. Elsevier Science Publishers B.V., 1990, S. 995–1072

[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*. first. Paderborn, Germany : Bonifatius GmbH, 2004 (HNI-Verlagsschriftenreihe Band 155). – ISBN 3–935433–64–6

[FM93] Fenelon, Peter ; McDermid, John A.: An integrated tool set for software safety analysis. In: *Journal of Systems and Software* 21 (1993), Nr. 3, S. 279–290

[FM98] Fradet, Pascal ; Métayer, Daniel L.: Structured Gamma. In: *Science of Computer Programing* 31 (1998), Nr. 2-3, S. 263–289

[FMNP94] Fenelon, P. ; McDermid, J. A. ; Nicolson, M. ; Pumfrey, D. J.: Towards integrated safety analysis and design. In: *ACM SIGAPP Applied Computing Review* 2 (1994), Nr. 1, S. 21–32

[FNT98] Fischer, T. ; Niere, J. ; Torunski, L.: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, July 1998

[FNTZ98] Fischer, T. ; Niere, J. ; Torunski, L. ; Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, G. (Hrsg.) ; G.Rozenberg (Hrsg.): *Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer Verlag, 1998 (LNCS 1764)

[Fra06] Frank, Ursula: *Spezifikationstechnik zur Beschreibung der Prinziplösung selbstoptimierender Systeme*, Universität Paderborn, Heinz Nixdorf Institut, Rechnerintegrierte Produktion, Dissertation, 2006

[GB03] Giese, Holger ; Burmester, Sven: Real-Time Statechart Semantics / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Germany, June 2003 (tr-ri-03-239). – Forschungsbericht. – 1–32 S.

[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

[GHJV94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

[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

[GKP05] Grunske, Lars ; Kaiser, Bernhard ; Papadopoulos, Yiannis: Model-Driven Safety Evaluation with State-Event-Based Component Failure Annotations. In: Heineman, George T. (Hrsg.) ; Crnkovic, Ivica (Hrsg.) ; Schmidt, Heinz W. (Hrsg.) ; Stafford, Judith A. (Hrsg.) ; Szyperski, Clemens A. (Hrsg.) ; Wallnau, Kurt C. (Hrsg.): *Component-Based Software Engineering, 8th International Symposium, CBSE 2005, St. Louis, MO, USA, May 14-15, 2005, Proceedings* Bd. 3489, Springer Verlag, 2005 (Lecture Notes in Computer Science), S. 33–48

[GKP08] Gausemeier, Jürgen ; Kahl, Sascha ; Pook, Sebastian: From Mechatronics to Self-Optimizing Systems. In: Gausemeier, Jürgen (Hrsg.) ; Rammig, Franz J. (Hrsg.) ; Schäfer, Wilhelm (Hrsg.): *Self-Optimizing Mechatronic Systems – Design the Future. Proc. of the 7th International Heinz Nixdorf Symposium, Paderborn, Germany* Bd. 223, HNI-Verlag, February 2008 (HNI-Verlagsschriftenreihe)

[GMW00] Garlan, David ; Monroe, Robert T. ; Wile, David: Acme: Architectural Description of Component-Based Systems. In: Leavens, Gary T. (Hrsg.) ; Sitaraman, Murali (Hrsg.): *Foundations*

of Component-Based Systems. Cambridge University Press, 2000, S. 47–68

[GOR06] Güdemann, Matthias ; Ortmeier, Frank ; Reif, Wolfgang: Safety and Dependability Analysis of Self-Adaptive Systems. In: *Proccedings of the 2nd International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2006)*, 2006

[Gri03] Grimm, Klaus: Software technology in an automotive company: major challenges. In: *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2003. – ISBN 0–7695–1877–X, S. 498–503

[Gru03b] Grunske, Lars: Annotation of Component Specifications with Modular Analysis Models for Safety Properties. In: Overhage, Sven (Hrsg.) ; Turowski, Klaus (Hrsg.): *Proc. of the 1st Int. Workshop on Component Engineering Methodology, Erfurt, Germany*, 2003

[Gru03c] Grunske, Lars: Transformational Patterns for the Improvement of Safety. In: *Proc. of the The Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 03)*, Microsoft Buisness Press, 2003

[GS04] Giese, Holger ; Schilling, Daniela: Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models / University of Paderborn. Paderborn, Germany, December 2004 (tr-ri-04-252). – Forschungsbericht

[Hab92] Habel, Annegret: *Lecture Notes in Computer Science*. Bd. 643: *Hyperedge Replacement: Grammars and Languages*. Springer, 1992

[Har87] Harel, David: STATECHARTS: A Visual Formalism for complex systems. In: *Science of Computer Programming* 3 (1987), Nr. 8, S. 231–274

[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, May 2008, S. 33–40

[Hir04] Hirsch, Martin: *Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL*, University of Paderborn, Diplomarbeit, June 2004

[Hir08] Hirsch, Martin: *Modell-basierte Verifikation von vernetzten mechatronischen Systemen.* Paderborn, Deutschland, Universität Paderborn, Dissertation, 2008

[HKK04] Hardung, Bernd ; Kölzow, Thorsten ; Krüger, Andreas: Reuse of software in distributed embedded automotive systems. In: *EMSOFT '04: Proceedings of the 4th ACM international conference on Embedded software.* New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–860–1, S. 203–210

[HLS00] Holsti, Niklas ; Långbacka, Thomas ; Saaarinen, Sami: Worst-Case Execution Time Analysis for Digital Signal Processors. In: *Proc. of the European Signal Processing Conference 2000 (EUSIPCO 2000)*, 2000

[Hoa85] Hoare, C. A. R.: *Communicating Sequential Processes.* Prentice-Hall International, 1985 (Series in Computer Science)

[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

[Hol08] Holtmann, Jörg: *Graphtransformationen für komponentenbasierte Softwarearchitekturen.* Paderborn, Deutschland, Universität Paderborn, Diplomarbeit, 2008

[Hon98] Honekamp, U.: *IPANEMA - Verteilte Echtzeit-Informationsverarbeitung in mechatronischen Systemen.* Düsseldorf, Universität Paderborn, Dissertation, 1998

[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, Hyogo, Japan*, 2005

[IEC] International Electronical Commission (Hrsg.): *IEC 1508: Functional Safety: Safety-Related Systems (Draft).* Geneva, Switzerland: International Electronical Commission, `http://www.12207.com/safety.htm`

[KBB+02] Knauber, Peter ; Bermejo, Jesus ; Böckle, Günter ; Prado Leite, Julio C. ; Linden, Frank van d. ; Northrop, Linda ; Stark, Michael ; Weiss, David M.: Quantifying Product Line Benefits. In: *Proc. of the 4th International Workshop on Product Family Engineering*, Springer Verlag, 2002, S. 155–163

[KG04] Kaiser, Bernhard ; Gramlich, C.: State-Event-Fault-Trees - A Safety Analysis Model for Software Controlled Systems. In: Heisel, Maritta (Hrsg.) ; Liggesmeyer, Peter (Hrsg.) ; Wittmann, Stefan (Hrsg.): *Computer Safety, Reliability, and Security, 23rd International Conference, SAFECOMP 2004, Potsdam, Germany, September 21-24, 2004, Proceedings* Bd. 3219, Springer Verlag, 2004 (Lecture Notes in Computer Science), S. 195–209

[KGF07] Kaiser, Bernhard ; Gramlich, Catharina ; ; Förster, Marc: State/event fault trees – A safety analysis model for software-controlled systems. In: *Reliability Engineering & System Safety* 92 (2007), Nr. 11, S. 1521–1537

[KKJD06] Kacem, Mohamed H. ; Kacem, Ahmed H. ; Jmaiel, Mohamed ; Drira, Khalil: Describing dynamic software architectures using an extended UML model. In: *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing.* New York, NY, USA : ACM Press, 2006, S. 1245–1249

[Kle99] Klein, T.: *Rekonstruktion von UML Aktivitäts- und Kollaborationsdiagrammen aus Java-Quelltexten*, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, October 1999

[Kle08] Klein, Florian: *A Model-Driven Approach to Multi-Agent-Design.* Paderborn, Deutschland, Universität Paderborn, Dissertation, 2008

[KLM03] Kaiser, Bernhard ; Liggesmeyer, Peter ; Maeckel, Oliver: A New Component Concept for Fault Trees. In: *Proceedings of the*

8th National Workshop on Safety Critical Systems and Software (SCS 2003), Canberra, Australia. 9-10th October 2003 Bd. 33, 2003 (Research and Practice in Information Technology)

[KNNZ99] Köhler, Hans J. ; Nickel, Ulrich A. ; Niere, Jörg ; Zündorf, Albert: Using UML as a visual programming language / University of Paderborn. Paderborn, Germany, August 1999 (tr-ri-99-205). – Forschungsbericht

[Krä06] Krämer, Helmer: *Laufzeitanpassungen von Softwarestrukturen für mechatronische Systeme.* Paderborn, Deutschland, Universität Paderborn, Diplomarbeit, 2006

[KW07] Kindler, Ekkart ; Wagner, Robert: Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios / Software Engineering Group, Department of Computer Science, University of Paderborn. 2007 (tr-ri-07-284). – Forschungsbericht. – 75 S.

[Lap92] Laprie, Jean C. (Hrsg.): *Dependable computing and fault tolerant systems.* Bd. 5: *Dependability : basic concepts and terminology in English, French, German, Italian and Japanese [IFIP WG 10.4, Dependable Computing and Fault Tolerance].* Wien : Springer Verlag, 1992. – ISBN 3–211–82296–8 or 0–387–82296–8

[LE91] Löwe, Michael ; Ehrig, Hartmut: Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. In: *Proceedings of the 16rd International Workshop on Graph-Theoretic Concepts in Computer Science.* London, UK : Springer-Verlag, 1991. – ISBN 3–540–53832–1, S. 338–353

[Lev95] Leveson, Nancy G.: *Safeware: System Safety and Computers.* Addison-Wesley, 1995

[LF99] Lopes, Antónia ; Fiadeiro, José Luiz: Using Explicit State to Describe Architectures. In: Finance, Jean-Pierre (Hrsg.): *Fundamental Approaches to Software Engineering, Second Internationsl Conference, FASE'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings* Bd. 1577, Springer, 1999 (Lecture Notes in Computer Science), S. 144–160

[LL73] Liu, C. L. ; Layland, James W.: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. In: *Journal of the ACM* 20 (1973), Nr. 1, S. 46–61

[LQV01] Lavazza, Luigi ; Quaroni, Garbiele ; Venturelli, Matteo: Combining UML and formal notations for modelling real-time systems. In: Gruhn, Volker (Hrsg.): *Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundation of Software Engineering (FSE-9), Vienna, Austria, September 10-14*, ACM Press, 2001, S. 196–206

[MBC+95] Marsan, M. A. ; Balbo, G. ; Conte, G. ; Donatelli, S. ; Franceschinis, G.: *Modelling with Generalized Stochastic Petri Nets*. John Wiley and Sons, Inc., 1995

[MC87] Marsan, M. A. ; Chiola, G.: On Petri Nets with deterministic and exponentially distributed firing times. In: *Proc. of the European Workshop on Applications and Theory of Petri Netrs 1986* Bd. 266, Springer Verlag, 1987 (Lecture Notes in Computer Science), S. 132–145

[McC56] McCluskey, Edward J.: Minimization of Boolean Functions. In: *Bell System Technical Journal* 35 (1956)

[MDK94] Magee, Jeff ; Dulay, Naranker ; Kramer, Jeff: Regis: A Constructive Development Environment for Distributed Programs. In: *Distributed Systems Engineering Journal* 1 (1994), September, Nr. 5, S. 304–312

[Mét98] Métayer, Daniel L.: Describing Software Architecture Styles Using Graph Grammars. In: *IEEE Transactions on Software Engineering* 24 (1998), Nr. 7, S. 521–533

[Mey97] Meyer, Bertrand: *Object-Oriented Software Construction*. Prentice Hall, 1997 `THIS/OOOSE_SHO.pdf`. – ISBN 0–13–629155–4. – 2nd edition

[MP94] McDermid, J.A. ; Pumfrey, D.J.: A Development of Hazard Analysis to aid Software Design. In: *Proceedings of the Ninth Annual Conference on Computer Assurance (COMPASS94)*. Gaithersburg, MD, USA, 1994, S. 17–25

[OB04] Oberthür, Simon ; Böke, Carsten: Flexible Resource Management – A framework for self-optimizing real-time systems. In: Kleinjohann, Bernd (Hrsg.) ; Gao, Guang R. (Hrsg.) ; Kopetz, Hermann (Hrsg.) ; Kleinjohann, Lisa (Hrsg.) ; Rettberg, Achim (Hrsg.): *Proceedings of IFIP Working Conference on Distributed and Parallel Embedded Systems (DIPES'04)*, Kluwer Academic Publishers, 23 - 26 August 2004

[Obj01] Object Management Group (Hrsg.): *Model Driven Architecture (MDA) Edited by Joaquin Miller and Jishnu Mukerji.* Object Management Group, 2001

[Obj07] Object Management Group (Hrsg.): *UML 2.1.2 Superstructure Specification.* Object Management Group, November 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

[OMG03] OMG, Object Management Group: *UML 2.0 OCL Final Adopted Specification.* OMG Document ptc/03-10-14. – ftp://-ftp.omg.org/pub/docs/ptc/03-10-14.pdf

[Opp06] Oppermann, Patrick: *Parsing and Analysing UML Text language in CASE-Tools*, University of Kassel, Kassel, Germany, Studienarbeit, October 2006

[OQ55] Orman Quine, William van: A Way to Simplify Truth Functions. In: *The American Mathematical Monthly* 62 (1955)

[ORS06] Ortmeier, Frank ; Reif, Wolfgang ; Schellhorn, Gerhard: Deductive Cause-Consequence Analysis. In: *Proceedings of the 16th IFAC World Congress*, Elsevier Science Publishers B.V., June 2006

[Pau05b] Pauls, Wladimir: *Verteilungsmodellierung für zuverlässige, komponentenbasierte Software in eingebetteten Systemen mit UML*, University of Paderborn, Department of Computer Science,Paderborn, Germany, Diplomarbeit, March 2005

[Pel01] Peled, Doron A.: *Software reliability methods*. New York : Springer Verlag, 2001 (Texts in computer science). – ISBN 0–387–95106–7

[PM99] Papadopoulos, Yiannis ; McDermid, John A.: Hierarchically Performed Hazard Origin and Propagation Studies. In: Felici, Massimo (Hrsg.) ; Kanoun, Karama (Hrsg.) ; Pasquini, Alberto (Hrsg.): *Computer Safety, Reliability and Security, 18th International Conference, SAFECOMP'99, Toulouse, France, September, 1999, Proceedings* Bd. 1698, Springer, 1999 (Lecture Notes in Computer Science), S. 139–152

[PZ97] Pratt, Terence ; Zelkowitz, Marvin: *Programmiersprachen: Design und Implementierung*. Prentice Hall, 1997

[Rau03] Rauzy, Antoine: A new methodology to handle Boolean models with loops. In: *IEEE Transactions on Reliability* 52 (2003), March, S. 96– 105. – ISSN 0018–9529

[RD97] Rauzy, Antoine ; Dutuit, Yves: Exact and Truncated Computations of Prime Implicants of Coherent and non-Coherent Fault Trees within Aralia. In: *Reliability Engineering and System Safety* 58 (1997), Nr. 2, S. 127–144

[Rei85] Reisig, W.: *Petri nets - An introduction*. Springer Verlag, 1985 (EATCS Monographs on Theoretical Computer Science)

[Rei07] Reineke, Peter: *Model checking von Story-Diagrammen mittels GROOVE*, University of Paderborn, Germany, Studienarbeit, 2007

[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), S. 479–485

[Ric08] Richtermeier, Manuel: *Worst Case Execution Time Berechnung von Story Diagrammen für mechatronische Systeme*, Institut für Informatik, Fachgebiet Softwaretechnik, Universität Paderborn, Studienarbeit, 2008

[RKS06] Rensink, Arend ; Kastenberg, Harmen ; Staijen, Tom: *User Manual for the GROOVE Tool Set*. Department of Computer Science, University of Twente, 2006

[Roz97] Rozenberg, G. (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations*. World Scientific Pub Co, 1997. – ISBN 9810228848. – Volume 1

[Rud84] Rudell, Richard L.: Multiple-Value Logic Minimization for PLA Synthesis / University of California at Berkeley, USA. 1984 (M86/65). – Forschungsbericht

[Ruf01] Ruf, Jürgen: RAVEN: Real-Time Analyzing and Verification Environment. In: *Journal on Universal Computer Science (J.UCS), Springer* 7 (2001), February, Nr. 1, S. 89–104

[SB96] Spuri, Marco ; Buttazzo, Giorgio C.: Scheduling Aperiodic Tasks in Dynamic Priority Systems. In: *Real-Time Systems* 10 (1996), Nr. 2, S. 179–210

[Sch94] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: *Proc. of the* 20^{th} *International Workshop on Graph-Theoretic Concepts in Computer Science*. Herrschin, Germany : Spinger Verlag, June 1994

[Sch02] Schmid, Klaus: A comprehensive product line scoping approach and its validation. In: *ICSE '02: Proc. of the 24th International Conference on Software Engineering*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–472–X, S. 593–603

[Sch06] Schilling, Daniela: *Kompositionale Softwareverifikation mechatronischer Systeme*. Paderborn, Deutschland, Universität Paderborn, Dissertation, 2006

[Sch07] Scharberth, Stefan: *Ein Werkzeug für die komponenten-basierte Gefahrenanalyse mit Entwurfsoptimierung und Variantenauswahl*. Paderborn, Deutschland, Universität Paderborn, Diplomarbeit, 2007

[Sed08] Seda, Milos: Heuristic Set-Covering-Based Postprocessing for Improving the Quine-McCluskey Method. In: *International Journal of Computational Intelligence (IJCI)* 4 (2008), Nr. 2, S. 139–143

[Sei05] Seibel, Andreas: *Story Diagramme für eingebettete Echtzeitsysteme*, University of Paderborn, Department of Computer Science, Paderborn, Germany, Studienarbeit, 2005

[SGW94] Selic, Bran ; Gullekson, Garth ; Ward, Paul: *Real-Time Object-Oriented Modeling*. John Wiley and Sons, Inc., 1994. – ISBN 0471–59917–4

[Shi05] Shin, Michael E.: Self-healing components in robust software architecture for concurrent and distributed systems. In: *Science of Computer Programming* 57 (2005), July, S. 27–44

[SK94] Slonneger, Kenneth ; Kurtz, Barry L.: *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1994

[Sto96] Storey, Neil: *Safety-Critical Computer Systems*. Addison-Wesley, 1996

[SW07] Schäfer, Wilhelm ; Wehrheim, Heike: The Challenges of Building Advanced Mechatronic Systems. In: *FOSE '07: 2007 Future of Software Engineering*, IEEE Computer Society, 2007, S. 72–84

[SWZ95] Schürr, A. ; Winter, A.J. ; Zündorf, Albert: Graph Grammar Engineering with PROGRES. In: Schäfer, Wilhelm (Hrsg.): *Proc. of European Software Engineering Conference (ESEC/FSE)*, Springer Verlag, 1995 (LNCS 989)

[SZ04] Schäuffele, Jörg ; Zurawka, Thomas: *Automotive Software Engineering*. Vieweg, 2004. – ISBN 3528110406

[Szy98] Szyperski, Clemens: *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998

[TCS98] Thurner, Erwin M. ; Cin, Mario D. ; Schneeweiss, Winfrid G.: Verläßlichkeitsbewertung komplexer Systeme. In: *Informatik Spektrum* 21 (1998), Nr. 6, S. 318–327

[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

[TH02] Thiel, Steffen ; Hein, Andreas: Modeling and Using Product Line Variability in Automotive Systems. In: *IEEE Software* 19 (2002), July/August, Nr. 4, S. 66–72. http://dx.doi.org/http://doi.ieeecomputersociety.org/10.1109/MS.2002.1020289. – DOI http://doi.ieeecomputersociety.org/10.1109/MS.2002.1020289

[TMV06] Trächtler, Ansgar ; Münch, Eckehard ; Vöcking, Henner: Iterative Learning and Self-Optimization Techniques for the Innovative Railcab-System. In: *Proc. of the 32nd Annual Conference of the IEEE Industrial Electronics Society – IECON'06, Paris, France*, IEEE, 2006

[VDI04] *VDI guideline 2206 - Design methodology for mechatronic systems.* : *VDI guideline 2206 - Design methodology for mechatronic systems*, 2004

[VGRH91] Veseley, W. E. ; Goldberg, F. F. ; Roberts, N. H. ; Haasl, D. F.: *Fault Tree Handbook.* Systems and Reliability Resarch, Office of Nuclear Regulatory Resedarch, U.S. Nuclear Regulatory Commission, Washington, D.C., USA, 1991 (NUREG-0492). http://www.nrc.gov/reading-rm/doc-collections/nuregs/staff/sr0492/

[Wal05] Wallace, Malcolm: Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In: *Electronic Notes in Theoretical Computer Science* 141 (2005), Nr. 3, S. 53–71

[WF99] Wermelinger, Michel ; Fiadeiro, José Luiz: Algebraic Software Architecture Reconfiguration. In: Nierstrasz, Oscar (Hrsg.) ; Lemoine, Michel (Hrsg.): *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings* Bd. 1687, Springer, 1999 (Lecture Notes in Computer Science). – ISBN 3–540–66538–2, S. 393–409

[WF02] Wermelinger, Michel ; Fiadeiro, José Luiz: A graph transformation approach to software architecture reconfiguration. In: *Sci. Comput. Program.* 44 (2002), Nr. 2, S. 133–155

[WGN03] Wagner, Robert ; Giese, Holger ; Nickel, Ulrich: A Plug-In for Flexible and Incremental Consistency Management. In: *Proc.*

of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA, Blekinge Institute of Technology, San Francisco, October 2003 (Technical Report)

[WLF01] Wermelinger, Michel ; Lopes, Antónia ; Fiadeiro, José L.: A graph based architectural (Re)configuration language. In: *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM, 2001. – ISBN 1–58113–390–1, S. 21–32

[ZFGH00] Zimmermann, Armin ; Freiheit, Jörn ; German, Reinhard ; Hommel, Günter: Petri Net Modelling and Performability Evaluation with TimeNET 3.0. In: Haverkort, Boudewijn R. (Hrsg.) ; Bohnenkamp, Henrik C. (Hrsg.) ; Smith, Connie U. (Hrsg.): *Computer Performance Evaluation: Modelling Techniques and Tools, 11th International Conference, TOOLS 2000, Schaumburg, IL, USA, March 27-31, 2000, Proceedings* Bd. 1786, Springer, 2000 (Lecture Notes in Computer Science), S. 188–202

[Zün02] Zündorf, Albert: *Rigorous Object Oriented Software Development - draft- Version 0.3*. University of Paderborn, 2002

Abbildungsverzeichnis

Liste der Algorithmen

A Erweiterte Storydiagramme

Im Rahmen der Projektgruppe „Automotive Software-Engineering“ [BBD+07] wurden Storydiagramme hinsichtlich der Modellierung von Refactorings von Software erweitert. Ziel dieser Arbeiten war es, Storydiagramme um syntaktische Konstrukte zu erweitern, die es ermöglichen, komplexe Refactorings in kleinere Teile zu unterteilen. Des Weiteren ermöglichen diese erweiterten Storydiagramme im Vergleich zu herkömmlichen Storydiagrammen mehrere Rückgabewerte.

Diese Erweiterung der Storydiagramme wurde als Zielsprache für die Übersetzung der Komponentenstorydiagramme gewählt, da sie eine 1-zu-1 Übersetzung der Syntaxelemente der Komponentenstorydiagramme ermöglichen (vor allem bzgl. der Aufrufe von anderen Komponentenstorydiagrammen). Hierdurch konnte die Übersetzung der Komponentenstorydiagramme überwiegend mit Triple-Graph-Grammatiken formalisiert werden.

A.1 Syntax

Abbildung A.1 zeigt ein erweitertes Storydiagramm A bestehend aus Start-Aktivität, erweitertem Storypattern und Stop-Aktivität. Die Signatur des erweiterten Storydiagramms besteht aus dem Namen A, den Parametern param1 vom Typ PT1 und param2 vom Typ PT2 sowie den Rückgabeparametern result1 vom Typ RT1 und result2 vom Typ RT2.

Im erweiterten Storypattern act wird neben der üblichen Spezifikation von linker (LHS) und rechter (RHS) Regelseite einer Graphtransformation in der Syntax der Storypatterns der Aufruf eines anderen erweiterten Storydiagramms durch einen *Transformation Call*-Knoten modelliert. Der Aufruf wird nur bei erfolgreichem Binden der LHS und nach der Erstellung der RHS ausgeführt. Argumente werden über spezielle Kanten (Stereotyp ≪argument≫) zwischen Objekten des Storypatterns und dem Transformation Call übergeben. Rückgabewerte können ebenfalls über spezielle Kanten (Stereotyp ≪result≫) an Objekte des Storypatterns gebunden werden. Diese Objekte sind Teil der RHS und können in nachfolgenden Storypatterns genutzt werden.

An der Stop-Aktivität werden die Argumente für die Rückgabeparameter gesetzt. Im Beispiel wird das Objekt rs1 für den Rückgabeparameter result1 und

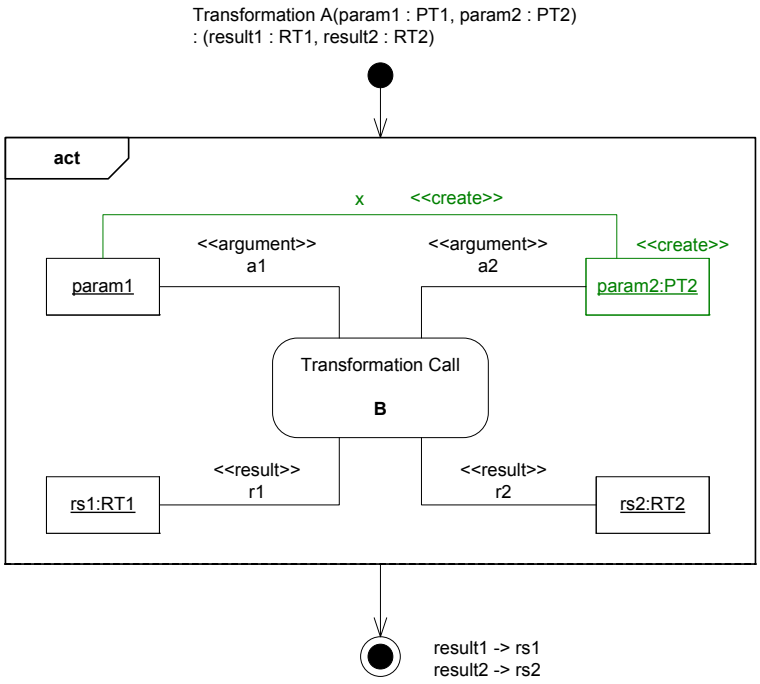


Abbildung A.1: Beispiel für ein erweitertes Storydiagramm

das Objekt rs2 für den Rückgabeparameter result2 gesetzt.

A.2 Formale Semantik

Die Semantik der erweiterten Storydiagramme wird durch eine Übersetzung auf herkömmliche Storydiagramme formalisiert. Die Übersetzung basiert darauf, dass für jedes spezifizierte erweiterte Storydiagramm eine Klasse generiert wird, die von der abstrakten Klasse AbstractTransformation erbt (s. Abbildung A.2). Diese Klasse stellt die Datenstrukturen für die Übergabe von Argumenten und Rückgabewerten über die qualifizierten Assoziationen arguments und results zur Verfügung. Schlüssel der qualifizierten Assoziationen sind die Namen der Parameter und Rückgabeparameter.

Die generierte Klasse wird danach um zwei herkömmliche Storydiagramme checkPrecondition() und execute() in Form von Methoden der Klasse erweitert. Das erste Storydiagramm enthält Storypattern zur Überprüfung, ob alle Argumente für die Parameter korrekt übergeben wurden. Des Weiteren können in einem erweiterten Storydiagramm Aktivitäten als Vorbedingung markiert werden (Stereotyp ≪precondition≫). Die erfolgreiche Ausführung dieser Aktivitäten wird

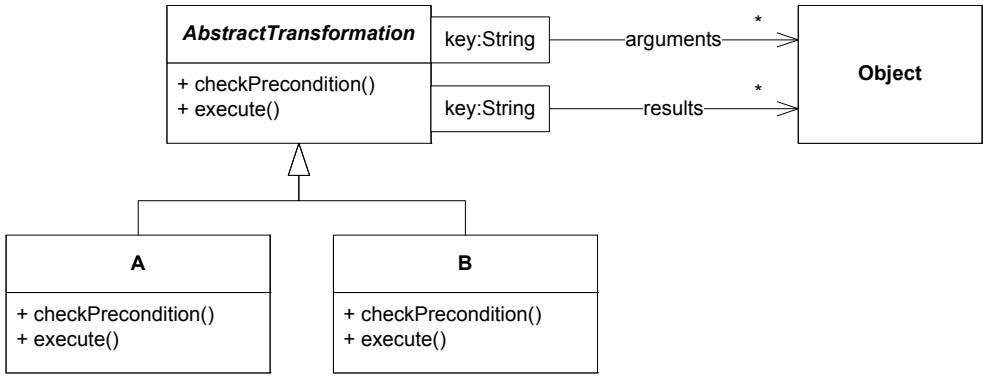


Abbildung A.2: Klassendiagramm für die generierten Storydiagramme

ebenfalls im checkPrecondition() Storydiagramm überprüft.

Das Storydiagramm execute() beinhaltet zusätzlich zur Ausführung des erweiterten Storydiagramms auch die Verarbeitung der Argumente und Rückgabewerte. Abbildung A.3 zeigt das aus dem erweiterten Storydiagramm A generierte herkömmliche Storydiagramm. In der ersten Aktivität werden die Argumente über die Assoziation arguments gebunden. In der generierten Aktivität StoryPattern act wird der Graphtransformationsanteil des Storypatterns act des erweiterten Storydiagramms A ausgeführt. Danach wird in der nächsten Aktivität das erweiterte Storydiagramm B mit den passenden Argumenten aufgerufen. In den nächsten Aktivitäten werden die Rückgabewerte von B gebunden und nachfolgend die Rückgabewerte von A in die Datenstruktur results übertragen.

Weitere Details der Syntax und Semantik der erweiterten Storydiagramme finden sich in [BBD+07].

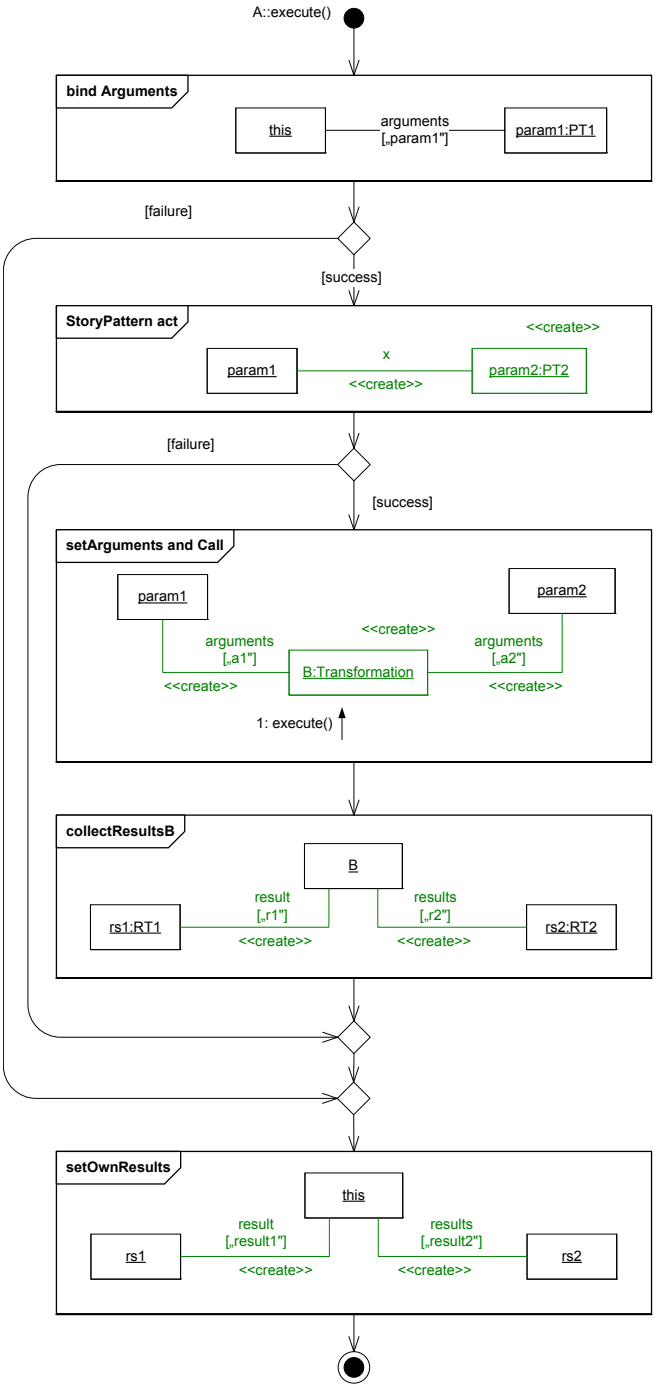


Abbildung A.3: Übersetzung der erweiterten Storydiagramme auf herkömmliche Storydiagramme.

B Gefahrenanalyse für Systeme mit Varianten und Produktlinien

Selbstoptimierende Systeme ändern ihre Struktur zur Laufzeit, um ihr Verhalten hinsichtlich geänderter Ziele anzupassen. Die in Kapitel 5 präsentierten Analysemodelle und -techniken können auch für herkömmliche Systeme, die in verschiedenen strukturellen Varianten entworfen werden, genutzt werden. Im Unterschied zu selbstoptimierenden Systemen werden die Varianten nicht zur Laufzeit umgeschaltet, sondern beim Entwurf bzw. Bau der Systeme festgelegt. Hierfür werden im Folgenden verschiedene weitere Analysetechniken auf Basis der obigen Analysemodelle präsentiert, die bei der Auswahl der geeigneten Varianten genutzt werden können.

B.1 Beste Variante

Wenn bei der Systementwicklung mehrere Systemarchitekturvarianten erarbeitet wurden, ist die Betrachtung der Gefahrenwahrscheinlichkeiten der unterschiedlichen Varianten ein Teilaspekt für die Entscheidung für eine Variante. Hierbei ist vor allem die beste Variante hinsichtlich der Gefahrenwahrscheinlichkeit interessant. Allerdings muss eine Variante, die für eine Gefahr optimal ist, für eine andere Gefahr nicht unbedingt optimal sein. Es ist also die Variante zu bestimmen, die für eine Menge von Gefahren optimal ist.

Ein weiterer zu beachtender Aspekt ist, dass bei der Entwicklung sicherheitskritischer Systeme für die verschiedenen Gefahren Obergrenzen für die Gefahrenwahrscheinlichkeit definiert werden. Diese müssen bei der Auswahl der besten Variante eingehalten werden.

Für die Berechnung der optimalen Variante werden daher die folgenden Bedingungen bezüglich der Wahrscheinlichkeiten der Gefahren $probability(\psi_i)$ sowie allgemein der maximalen Höhe der Gefahrenwahrscheinlichkeit $probability^{max}(\psi_i)$ bei der Bestimmung der besten Variante für eine Menge von Gefahren Ψ betrachtet:

$$\forall_{\psi_i \in \Psi} probability(\psi_i) \leq probability^{max}(\psi_i) \tag{B.1}$$

Des Weiteren ist neben der Wahrscheinlichkeit auch die Schwere (severity) der Gefahr relevant. Das Produkt von Wahrscheinlichkeit und Schwere einer Gefahr, das Risiko (risk) [IEC] ist hier passender. Daher wird bei der Berechnung der besten Variante für eine Menge von Gefahren Ψ folgende Zielfunktion minimiert:

$$min \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i)$$

B.2 Produktfamilien

Die Bestimmung der besten Variante ist vor allem interessant, wenn nur diese Variante nachher eingesetzt werden soll. Falls allerdings mehrere Varianten im Rahmen von Produktfamilien [KBB+02, Sch02, BCM+04, TH02] eingesetzt werden sollen, ist es für die Sicherheit auch wichtig zu wissen, welche die schlechteste Variante in Bezug auf die Gefahrenwahrscheinlichkeit bzw. Risiko ist.

Dies lässt sich analog zu obiger Bestimmung der besten Variante mit Hilfe einer geänderten Zielfunktion beantworten:

$$max \sum_{\psi_i \in \Psi} severity(\psi_i) \cdot probability(\psi_i)$$

Die Zielfunktion maximiert die Summe der Gefahrenwahrscheinlichkeiten multipliziert mit der Schwere der Gefahr. Das Resultat ist die Variante, die (unter Einhaltung aller Obergrenzen) die schlechteste Variante ist in Bezug auf die Summe der Risiken.

B.2.1 Schlechteste Variantenkombination

Weiterhin ist wichtig zu wissen, welche Kombination unterschiedlicher Varianten die schlechteste bzgl. der Gefahrenwahrscheinlichkeit bzw. des Risikos ist. Diese Fragestellung ist wichtig, da beim (gleichzeitigen) Einsatz verschiedener Varianten diese in Kombination an einer Gefahrensituation beteiligt sein können. Das Ergebnis stellt dann eine obere Schranke für die Gefahrenwahrscheinlichkeit bzw. das Risiko dar. Wenn diese obere Schranke dann niedriger ist als die Anforderungen, können alle Varianten eingesetzt werden.

Während für einzelne Komponententypen in den bisherigen Analysen genau eine Variante ausgewählt wurde, also alle Instanzen einer Komponente von der gleichen Variante sind (im folgenden *globale* Variante genannt), wird für die Analyse der schlechtesten Variantenkombination für jede Instanz einer Komponente einzeln entschieden, welche Variante diese Komponenteninstanz ist (im folgenden als *lokale* Variante bezeichnet).

Eine Belegung der Variantenvariablen c_i beschreibt hierbei nur noch die Auswahl einer Variante für eine Komponenteninstanz und nicht wie vorher für alle Instanzen eines Komponententyps. Der BDD für die Fehlerpropagierung inkl. der Variantenvariablen kann nun analog zu den bisherigen Analysen aufgebaut und analysiert werden.

B.3 Erlaubte Varianten

Des Weiteren ist es wichtig zu wissen, welche Varianten nicht eingesetzt werden dürfen, da sie für mindestens eine Gefahr eine höhere Wahrscheinlichkeit haben als erlaubt.

Dies lässt sich durch Berechnung aller Gefahrenwahrscheinlichkeiten für jede Variante überprüfen. Diese Berechnung kann direkt auf den aus Variantenvariablen und Ereignisvariablen bestehenden BDDs für die einzelnen Gefahren durchgeführt werden. Da diese BDDs für die Berechnung jeder Variante genutzt werden können, lässt sich diese Berechnung in linearer Zeit bzgl. der Anzahl der Varianten, der Anzahl der Knoten im BDD und der Anzahl der Gefahren durchführen.

Algorithmus 5 : boolean checkMaxForVariants (Variant variant, List<Gefahr> gefahren)

```
begin
    foreach gefahr ∈ gefahren do
        if probability (variant,gefahr) > max (gefahr) then
            return false;
    return true;
end
```

Algorithmus 5 prüft für eine Variante, ob diese die Anforderungen für alle Gefahren einhält. Algorithmus 6 nutzt dies, um alle Varianten zu berechnen, die jeweils alle Anforderungen einhalten. Dieses Ergebnis wird dann genutzt, um alle nicht erlaubten Varianten zu berechnen, indem die erlaubten Varianten aus der Menge aller Varianten entfernt werden.

B.4 Auswahl der robustesten Variante

Die für die Basisereignisse bei der Analyse genutzten Wahrscheinlichkeiten können durchaus fehlerhaft sein. In diesem Zusammenhang ist die Variante inter-

Algorithmus 6 : List<Variant> computeAllowedVariants (List<Variant> allVariants, List<Gefahr> gefahren)

```
begin
    List<Variant> allowedVariants ← ∅;
    foreach variant ∈ allVariants do
        if checkMaxForVariants (variant, gefahren) = true then
            allowedVariants ← allowedVariants ∪ variant;
    return allowedVariants;
end
```

essant, die bzgl. der Fehler in den Wahrscheinlichkeiten der Basisereignisse am robustesten ist unter der Vorrausetzung, dass die Anforderungen noch erfüllt werden. Die Robustheit bzgl. der Fehler in den Wahrscheinlichkeiten bedeutet, dass diese Variante die größte Abweichung der Wahrscheinlichkeiten der Basisereignisse erlaubt.

Um diese Variante zu bestimmen, wird bei der Erstellung der Formel für den BDD anstelle der Wahrscheinlichkeit $probability(e_i)$ Formel B.2 genutzt:

$$probability(e_i) + \beta(e_i)\delta \tag{B.2}$$

Der Ausdruck $\beta(e_i)\delta$ beschreibt hierbei die Abweichung der Wahrscheinlichkeit. Die Abweichung $\delta \in \mathbb{R}^+$ wird durch die Multiplikation mit dem angenommen Fehler der Wahrscheinlichkeit des Basisereignisses $probability(e_i)$ ($\beta(e_i) \in \mathbb{R}^+$) passend zur Größenordnung der Wahrscheinlichkeit $probability(e_i)$ skaliert. Es empfiehlt sich daher passende Werte für $\beta(e_i)$ zu wählen ([GT06]). Abbildung B.1 stellt diesen Zusammenhang dar.

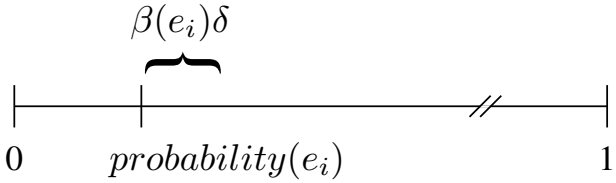


Abbildung B.1: Anpassung der Wahrscheinlichkeitswerte

Wenn nun unter Berücksichtigung der Anforderungen δ maximiert wird, ergibt sich die Variante, die die größten Abweichung zur Wahrscheinlichkeit der Basisereignisse erlaubt und dennoch noch sicher bzgl. der maximalen Gefahrenwahrscheinlichkeit ist.