UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft
Fakultät für Elektrotechnik, Informatik, Mathematik

Diplomarbeit
Studiengang Informatik (DPO4)

Graphtransformationen für komponentenbasierte Softwarearchitekturen

vorgelegt von
Jörg Holtmann

vorgelegt bei:
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Gregor Engels

Betreuer: Matthias Tichy

April 2008

Danksagung
Diese Diplomarbeit entstand im Rahmen meines Informatikstudiums an der Universität Paderborn. Mein besonderer Dank gilt Matthias Tichy aka mtt für die hervorragende Rundum-Betreuung dieser Arbeit sowie für die Beeinflussung meines weiteren Werdegangs. Des Weiteren bedanke ich mich bei Volker Fortströer, Christoph Oberhokamp und Axel Tenschert für das Korrekturlesen und an die Leute aus dem Poolraum für die kleinen Ablenkungen zwischendurch.

Paderborn, April 2008

Jörg Holtmann

Erklärung
Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen sind, habe ich als Zitat kenntlich gemacht.

Paderborn, 12. April 2008

Jörg Holtmann

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Moderne Softwaresysteme bestehen aus diversen Subsystemen oder Komponenten, welche miteinander kommunizieren, um ein bestimmtes Verhalten zu erreichen. Die Anordnung und Komposition dieser Komponenten und deren Interaktionsbeziehungen untereinander machen die Struktur eines Softwaresystems aus, welche auch als *Softwarearchitektur* [SG96] bezeichnet wird.

Die steigende Komplexität sowohl bei der Entwicklung als auch im laufenden Betrieb von Softwaresystemen stellt neue Anforderungen an die Spezifikation von Softwarearchitekturen. So sind Entscheidungen bereits auf der obersten architektonischen Designebene zu treffen, um im resultierenden Softwaresystem Ziele wie z. B. eine gute Skalierbarkeit oder eine hohe Erweiterbarkeit zu erreichen. Um solche Designentscheidungen zu beschreiben, werden *Architekturbeschreibungssprachen* [SG96] eingesetzt. Solche Sprachen ermöglichen die Beschreibung, Modellierung und Strukturierung einer Softwarearchitektur mit formalen und graphischen Artefakten.

Ein aktueller Trend im Software Engineering ist die Entwicklung von *selbstoptimierenden Systemen* [FGK+04]. Ein selbstoptimierendes Softwaresystem passt sein Verhalten im laufenden Betrieb an sich verändernde Gegebenheiten der Umwelt an. Diese Verhaltensanpassung kann durch eine Anpassung der Systemparameter oder aber durch eine Strukturanpassung vorgenommen werden.

Als fortlaufendes Beispiel für ein selbstoptimierendes Softwaresystem wird im Folgenden die Neue Bahntechnik Paderborn[1] betrachtet. Dies stellt ein neuartiges Verkehrssystem dar, welches u. a. aus autonomen, führerlosen Schienenfahrzeugen namens RailCabs besteht. Diese RailCabs können Konvois bilden, um den Luftwiderstand zu reduzieren und somit Energie zu sparen. In dem stark vereinfachten Beispiel für diese Arbeit wird angenommen, dass im Konvoibetrieb das führende RailCab seine eigene Geschwindigkeit und somit die des gesamten Konvois kontrolliert. Zu diesem Zweck sendet es fortlaufend Referenzpositionen an jedes der nachfolgenden RailCabs, welche ihre Position an diese Referenzpositionen anpassen. Abb. 1.1 skizziert diese Kommunikationsstruktur.

Zur Realisierung dieser Kommunikation wird im führenden RailCab eine Softwarekomponente pro Nachfolger zur Berechnung der jeweiligen Referenzposition aus der aktuellen Geschwindigkeit eingesetzt. Da RailCabs während der Fahrt einem Konvoi beitreten oder einen Konvoi verlassen können, muss die zu Grunde liegende Softwarearchitektur zur Laufzeit das Hinzufügen und Entfernen von Softwarekomponenten sowie die daher benötigte Umstrukturierung vorhandener Interaktionsbeziehungen erlauben. Somit liegt also ein selbstoptimierendes Softwaresystem vor, welches sein Verhalten mittels strukturellen Veränderungen anpasst.

[1] `http://www.railcab.de`

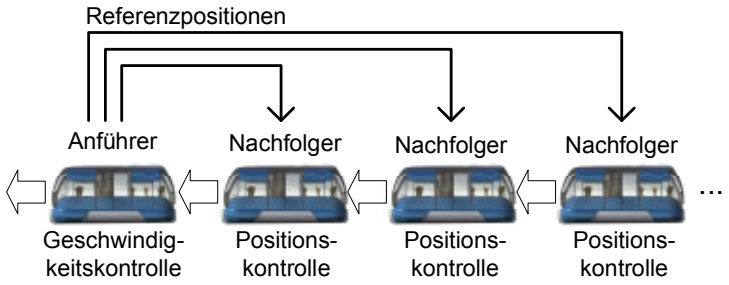


Abbildung 1.1: Kommunikation in einem RailCab-Konvoi [THHO08]

Das beschriebene System kann allerdings nicht alleine durch den Einsatz von Softwarekomponenten funktionieren. So ermittelt die Software z. B. Informationen über den Systemzustand mit Hilfe von Sensoren und steuert das extern sichtbare Verhalten über Aktoren. Sensoren und Aktoren greifen dabei auf die physikalische Ebene des Systems zu, welche aus mechanischen und elektrischen Bauelementen besteht. Ein solches System, welches Entwicklungen der unterschiedlichen Forschungsdisziplinen Maschinenbau, Elektrotechnik und Informatik vereint, wird als *mechatronisches System* bezeichnet.

Wie auch an das RailCab-System werden an mechatronische Systeme oft Echtzeit- oder sicherheitskritische Anforderungen gestellt. Des Weiteren erschweren häufig kontinuierliche Signale aus der Sensorik den Wechsel zwischen diskreten Zuständen, und die Ressourcen zur Ausführung der Software sind meist beschränkt. Für eine ganzheitliche Modellierung eines komponentenbasierten, mechatronischen Systems, welche auch die o. g. Aspekte anspricht, wurde im Fachgebiet Softwaretechnik der Universität Paderborn die Sprache Mechatronic UML [Bur05, Sch06] entwickelt. Diese Sprache basiert auf der Modellierungssprache UML [Obj07b], dem de facto Industriestandard zur Modellierung von Struktur und Verhalten von Softwaresystemen, welche in ihrer Grundform nicht genügend Ausdrucksfähigkeit zur Modellierung aller domänenspezifischer Aspekte mechatronischer Systeme besitzt. Die technische Umsetzung von Mechatronic UML ist über eine Reihe von Plugins für das CASE[2]-Werkzeug Fujaba[3] in Form der Fujaba Real-Time Tool Suite[4] erhältlich.

1.1 Problemstellung

In Mechatronic UML werden UML Komponentendiagramme zur strukturellen Modellierung eines Systems eingesetzt. Dabei wird zwischen Komponententypen und -instanzen unterschieden. Für Komponententypen werden strukturelle Elemente wie vorhandene Ports und Schnittstellen zur Kommunikation mit der Umwelt sowie Verhaltensbeschreibungen spezifiziert. In einem konkreten System werden Komponenteninstanzen mit diesen Eigenschaften angeordnet und entsprechend ihrer Schnittstellenspezifikation

[2] **C**omputer-**A**ided **S**oftware **E**ngineering
[3] **F**rom **U**ML to **J**ava **A**nd **B**ack **A**gain, `http://www.fujaba.de`
[4] `http://www.fujaba.de/projects/realtime`

untereinander verbunden. Abb. 1.2 zeigt die Komponenteninstanzstruktur eines RailCab-Konvois mit drei Teilnehmern in der Syntax von MECHATRONIC UML, welche die Möglichkeit weiterer folgender RailCabs andeutet. Eine solche Struktur bestehend aus einer bestimmten Anzahl untereinander verbundener Komponenteninstanzen zu einem gewissen Zeitpunkt wird auch *Konfiguration* genannt.

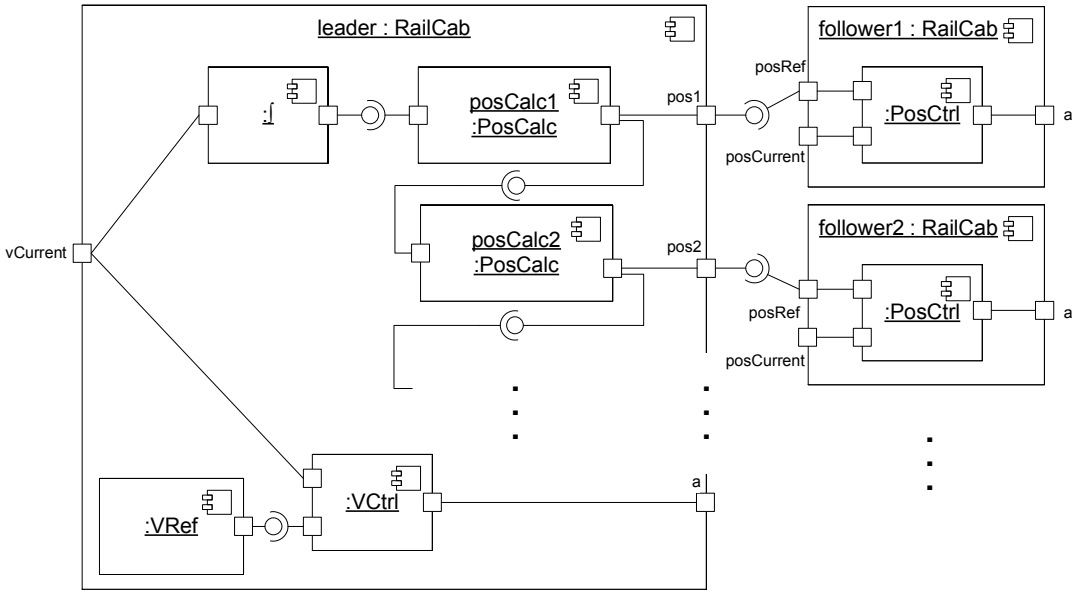


Abbildung 1.2: Konfiguration eines RailCab-Konvois in MECHATRONIC UML [THHO08]

Die Software des führenden RailCabs wird durch die Komponenteninstanz leader:RailCab repräsentiert, welche aus weiteren eingebetteten Komponenteninstanzen für die Geschwindigkeitskontrolle sowie für die Berechnung der Referenzpositionen der nachfolgenden RailCabs komponiert ist. Die Portinstanz a stellt einen Interaktionspunkt zur Kontrolle des Antriebs bereit. Über die Portinstanz vCurrent wird die aktuelle Geschwindigkeit des RailCabs geliefert, während :VRef die geforderte Referenzgeschwindigkeit zur Verfügung stellt. Die Komponenteninstanz :VCtrl vergleicht diese beiden Werte und variiert die Beschleunigung über a, um die aktuelle Geschwindigkeit der Referenzgeschwindigkeit anzupassen. Die Komponenteninstanz :$\int$ ermittelt aus der aktuellen Geschwindigkeit die aktuelle Position des führenden RailCabs auf der Bahnstrecke. Für jedes folgende RailCab wird im führenden RailCab eine Komponenteninstanz :PosCalc verwendet, welche die Referenzposition des nachfolgenden RailCabs berechnet und diese an sie übermittelt. In einer simplen Implementierung bekommen die Komponenteninstanzen :PosCalc als Eingabe die Position des vorausgehenden RailCabs, fügen dessen Länge zuzüglich eines Sicherheitsabstandes hinzu und berechnen daraus die Referenzposition des nachfolgenden RailCabs. Im Konvoi folgende RailCabs sind mit einer Komponenteninstanz :PosCtrl ausgestattet, welche die Referenzposition und die aktuelle Position posCurrent vergleicht und dementsprechend den Antrieb ansteuert.

Der vorgestellte Ansatz zur strukturellen Modellierung von Systemen erlaubt lediglich

Konfigurationsmengen einer festgelegten Größe. So ist es nicht möglich, Komponententypen zu definieren, deren Instanzen prinzipiell beliebig viele eingebettete Komponenteninstanzen mit sich führen können. Des Weiteren ist aus diesem Grund der vorhandene Mechanismus zur Strukturanpassung lediglich für kleine Konfigurationsmengen ausgelegt. Basierend auf der fixen Komponententypdefinition muss jede einzelne, zur Laufzeit gewünschte Konfiguration modelliert werden. Neben der Inpraktikabilität dieser Spezifikation von Strukturanpassungen für große Konfigurationsmengen birgt dies zusätzlich den Nachteil, dass alle Konfigurationen zum Zeitpunkt der Modellierung bekannt sein müssen.

Dagegen erfordert das vorgestellte Beispiel des RailCab-Konvois offensichtlich den Ansatz einer flexibleren Komponententypdefinition. Um z. B. einen Konvoi beliebiger Größe zu erlauben, sollte z. B. die Definition eines Komponententypen RailCab möglich sein, dessen Instanzen aus n eingebetteten :PosCalc-Instanzen bestehen können. Eine solche variable Typdefinition würde eine prinzipiell unendlich große Konfigurationsmenge zulassen. Daher wird eine Strukturanpassung benötigt, welche anstelle der Modellierung jeder einzelnen Konfiguration die Differenzen zwischen ihnen beschreibt.

1.2 Lösungsansatz

Ziel dieser Diplomarbeit ist die Beseitigung der beschriebenen Einschränkungen durch neue Konzepte. Im ersten Schritt soll eine variable Typdefinition der in MECHATRONIC UML eingesetzten Komponenten zur Modellierung der Struktur eines Systems ermöglicht werden. Zu diesem Zweck soll das Konzept der UML Kompositionsstrukturen in die Komponentendiagramme der MECHATRONIC UML integriert werden. Wie Komponenten verfügen auch die strukturierten Klassen der UML Kompositionsstrukturen über Ports und Schnittstellen sowie eingebettete, kompositional abhängige Elemente, welche untereinander mit Konnektoren verbunden sind. Die Verwendung von Rollen und Multiplizitäten bei der Modellierung der internen Kompositionsstruktur einer strukturierten Klasse kann zur Laufzeit auch mehrere eingebettete Instanzen desselben Klassifizierers innerhalb einer übergeordneten Instanz erlauben. Durch die Einführung dieses Modellierungsansatzes in die MECHATRONIC UML Komponententypen soll die geforderte variable Typdefinition erreicht werden, welche die Größe der möglichen Konfigurationsmengen nicht mehr beschränkt und somit die Wiederverwendbarkeit von Komponententypen steigert.

Diese variablen Komponententypen sind Voraussetzung für eine Strukturanpassung, welche Differenzen zwischen Konfigurationen durch Transformationen beschreibt. Hierzu soll basierend auf Graphtransformationen eine neue Sprache namens *Komponentenstorydiagramme* [THHO08] entwickelt werden, welche die initiale Konfiguration eines Systems über alle Komponentenhierarchien hinweg instanziiert und vorhandene Konfigurationen strukturell verändert. Grundlage dieser Sprache ist der in FUJABA eingesetzte Formalismus der Storydiagramme [FNTZ98], welche Graphtransformationen auf typisierten Objektstrukturen mit den Kontrollstrukturen der Programmiersprachen kombinieren. Anstatt Objekten und Links werden bei der Ausführung von Komponentenstorydiagrammen

Komponenten- und Konnektorinstanzen erstellt und zerstört. Die Spezifikation der Transformationen erfolgt dabei auf den variablen Komponententypen. Zwecks einer verständlichen graphischen Repräsentation und einer geringen visuellen Komplexität wird die Modellierung in der konkreten Syntax der MECHATRONIC UML Komponentendiagramme durchgeführt. Gleichzeitig werden die Kontrollstrukturen – wie Sequenzen, Verzweigungen oder Schleifen – sowie die graphischen Notationen der konventionellen Storydiagramme übernommen. Komponentenstorydiagramme werden für die konkrete Ausführung auf eine Variante von Storydiagrammen auf den zu Grunde liegenden Objektstrukturen der Komponenten abgebildet. Die Ausführung erfolgt dabei zur Entwurfszeit.

Mittels der Erweiterung der Komponententypdefinition und der Entwicklung einer neuen Graphtransformationssprache für Komponenteninstanzen werden unendlich große Konfigurationsmengen erlaubt und die Praktikabilität der Modellierung der Strukturanpassung für große Konfigurationsmengen erhöht. Die einzelnen Konfigurationen müssen nicht mehr zum Modellierungszeitpunkt bekannt sein, sondern können sich während der Ausführung der Transformationen entwickeln. Des Weiteren können auf Komponententypebene programmiersprachliche Konstrukte wie Mengen oder Listen eingeführt werden, deren Elemente und Elementbeziehungen durch Operationen in Form von Komponentenstorydiagrammen verändert werden. Die beschriebenen Konzepte sollen prototypisch in Fujaba4Eclipse[5] implementiert werden.

1.3 Struktur der Arbeit

Kapitel 2 stellt zunächst die für das tiefere Verständnis dieser Arbeit essentiellen Grundlagen vor, während Kapitel 3 verwandte Arbeiten zum Thema vergleicht. Nach dieser Einführung in die Domäne fasst Kapitel 4 die benötigten Anforderungen an diese Arbeit aus den Grundlagen zu MECHATRONIC UML und den verwandten Arbeiten detaillierter zusammen und grenzt sie von weiterführenden Themen ab, welche in zukünftigen Arbeiten basierend auf den neuen Konzepten umgesetzt werden können. Kapitel 5 stellt die Integration der UML Kompositionsstrukturen in die Komponententypdefinition von MECHATRONIC UML vor, während Kapitel 6 die neue Transformationssprache einführt. Dazu gehört eine informale syntaktische und semantische Beschreibung sowie die Abbildung auf den Zielformalismus. Kapitel 7 führt die Kernpunkte der technischen Umsetzung und die Ergebnisse der Evaluierung an. Das letzte Kapitel fasst die Ergebnisse dieser Diplomarbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten, welche durch die neu entwickelten Ansätze ermöglicht werden.

[5] Eclipse-Portierung von FUJABA, `http://www.fujaba.de/projects/eclipse`

2 Grundlagen

Dieses Kapitel stellt die Grundlagen vor, auf denen diese Arbeit basiert. Der folgende Abschnitt skizziert zunächst die Grundkonzepte von Graphtransformationen und stellt anschließend die konkreten Formalismen für Graphtransformationen vor, welche in dieser Arbeit Anwendung finden oder auf denen sie basiert. Abschnitt 2.2 gibt einen umfassenderen Einblick in die Begriffe der Softwarearchitekturen. Abschnitt 2.3 stellt neben den Architekturen und Konzepten der MECHATRONIC UML eine detaillierte Analyse der Einschränkungen bzgl. der Modellierung von Struktur und Strukturanpassung vor. Der letzte Abschnitt führt das Konzept der UML Kompositionsstrukturen ein.

2.1 Graphtransformationen

Graphtransformationen stammen aus der Theorie der *Graphgrammatiken* [Roz97], welche eine intuitive Beschreibung von Graphmanipulationen erlauben. Solche Beschreibungen lassen sich graphisch notieren und basieren auf formalen Grundlagen. Mit entsprechendem Toolsupport lassen sich diese Manipulationen z. B. auf Datenstrukturen anwenden. Eine Graphgrammatik besteht aus einem Startgraph und einer Menge von *Produktionsregeln (Graphtransformationsregeln, Graphersetzungsregeln)*, welche auf den Startgraph angewendet werden können. Durch die Anwendung der Produktionsregeln entstehen Folgegraphen, auf die wiederum die Produktionsregeln der Grammatik angewendet werden können. Somit beschreibt eine Graphgrammatik mit dem Startgraph und den Produktionsregeln eine Menge von Graphen, die aus dem Startgraph gebildet werden können.

Die Grundidee einer Graphtransformationsregel p basiert auf der Modellierung zweier Graphen L und R, einer so genannten *linken Seite (left-hand side, LHS, Vorbedingung)* und einer *rechten Seite (right-hand side, RHS, Nachbedingung)*. Die linke Seite beschreibt dabei die Graphstruktur vor der Ausführung der Regel, während die rechte Seite die Struktur nach der Ausführung beschreibt. In einem gegebenen *Wirtsgraph* G wird mittels eines Graphhomomorphismus m nach einer *Anwendungsstelle* von L gesucht (die LHS wird *gebunden*) und – falls erfolgreich – durch R ersetzt, so dass G in einen neuen Graphen H transformiert wird. Eine solche *direkte Ableitung* wird mit $G \overset{p,m}{\Longrightarrow} H$ bezeichnet. Ebenso lassen sich über eine *Co-Produktion* p^* die unveränderten Elemente von G und H zueinander in Verbindung setzen, und ebenso lässt sich die RHS über einen Graphhomomorphismus m^* auf H abbilden. Abb. 2.1 verdeutlicht dies.

Für die Ausführung von Graphtransformationen gibt es unter den algebraischen Ansätzen die Ansätze *Double-Pushout (DPO)* [EPS73] und *Single-Pushout (SPO)* [Löw93]. Neben der eigentlichen Ausführungsweise einer Regel unterscheiden sich die Ansätze vor allem in der Behandlung zweier Spezialfälle. Zum einen können zwei verschiedene Knoten

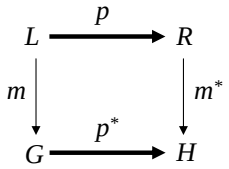


Abbildung 2.1: Schematische Darstellung der direkten Ableitung $G \overset{p,m}{\Longrightarrow} H$

der LHS auf denselben Knoten im Wirtsgraphen abgebildet werden, da kein Isomorphismus für diese Abbildung verlangt wird. Wenn z. B. einer der Knoten in der LHS gelöscht und der andere Knoten erhalten werden soll, gibt es in diesem Fall einen Konflikt für die Abbildungsfunktion. DPO verlangt für jeden zu löschenden Knoten der Regel, dass dieser im Wirtsgraph eindeutig identifizierbar ist. Der SPO-Ansatz dagegen bevorzugt eine Löschung gegenüber der Erhaltung von Knoten, d. h. dass in dem beschriebenen Fall der doppelt gebundene Knoten gelöscht wird. Zum anderen kann die Löschung eines mit anderen Kanten inzidenten Knoten zu einem ungültigen Graph führen, falls die Kanten nicht auch gelöscht werden. Dies würde zu einer Kante ohne Start- bzw. Zielknoten (*dangling edge*) im Graph führen. Im DPO-Ansatz wird bei der Löschung eines Knotens mit inzidenten Kanten verlangt, dass auch die Kanten explizit gelöscht werden, während SPO alle inzidenten Kanten implizit löscht. Mit SPO lassen sich also (evtl. auch unbeabsichtigte) Effekte modellieren, welche im DPO-Ansatz nicht möglich sind. Somit ist SPO ausdrucksstärker, während DPO restriktiver ist.

Um ungewollte Seiteneffekte bzgl. des Löschens inzidenter Kanten eines zu löschenden Knoten zu vermeiden und um die Graphtransformationsregeln ausdrucksstärker zu machen, können sie im SPO-Ansatz *Anwendungsbedingungen (application conditions)* besitzen. Die *negative Anwendungsbedingung (negative application condition, NAC)* besitzt *Einschränkungen (constraints)*, die sich auf jeweils einen Knoten der Regel beziehen. Eine LHS mit negativer Anwendungsbedingung kann nur dann erfolgreich gebunden werden, wenn im Wirtsgraph kein Knoten existiert, den die einzelnen Einschränkungen ausschließen. Andere Anwendungsbedingungen können z. B. den SPO-Ansatz um Restriktionen des DPO-Ansatzes erweitern.

Programmierte Graphersetzungssysteme, welche die Modellierung sowie die konkrete Ausführung von Graphtransformationen erlauben, benutzen zusätzliche Erweiterungen für die zu Grunde liegenden Graphen sowie für die Graphtransformationen selber. Das Prinzip der typisierten Graphen erweitert die Semantik von Graphen um eine strukturerhaltende Abbildung auf einen Typgraphen. Dieses Prinzip macht den Einsatz von Graphtransformationen vor allem für Datenstrukturen der OOP sinnvoll, da Typgraphen als Klassendiagramme und Instanzgraphen als Objektdiagramme fungieren können. Mittels Typgraphen lassen sich z. B. Kardinalitätsbeschränkungen für inzidente Kantentypen eines Knotentyps modellieren oder definieren, welche Knotentypen adjazent sein dürfen. Attributierte Graphen können Knoten um Attribute bereichern. Attribute sind dabei Labels mit einem Alphabet aus vordefinierten Datentypen. In Kombination mit typisierten Graphen können somit Knoten Attributbedingungen oder Attributzuweisun-

gen zugeordnet werden, welche über den entsprechende Knotentyp klassifiziert sind. Um Graphtransformationen im Sinne einer visuellen, höheren Programmiersprache benutzen zu können, bereichern solche Systeme die Graphtransformationsregeln oft um Kontrollstrukturen wie Verzweigungen oder Schleifen. Siehe hierzu auch [CEER96, Zün01]. Mit solchen Erweiterungen werden Graphtransformationen in der modellbasierten Softwareentwicklung und -analyse verwendet.

2.1.1 Storydiagramme

Storydiagramme [FNTZ98, Zün01, FNT98] stellen einen konkreten, UML-basierten Formalismus der Graphtransformationen dar, welcher speziell für FUJABA entwickelt wurde. Ein Storydiagramm ist ein adaptiertes UML 1 Aktivitätsdiagramm [BRJ99], somit basieren die nachfolgend benutzten Begrifflichkeiten und Semantiken aus der gängigen Literatur über Storydiagramme ebenfalls auf der UML 1[1].

Storydiagramme bestehen aus Aktivitäten, welche reinen programmiersprachlichen Code (*Statement-Aktivitäten*) oder *Storypatterns* enthalten können. Storypatterns enthalten die eigentlichen Graphtransformationsregeln. Ein Storypattern ist eine Variante von UML Objekt- und Kollaborationsdiagrammen[2] [BRJ99], deren Elemente über Klassendiagramme typisiert sind. Knoten oder *Objektvariablen* eines Storypatterns sind Repräsentanten für konkrete Objekte im Wirtsgraphen, welche eindeutig einer Klasse zugeordnet sind. Die verbindenden Kanten repräsentieren die Objektbeziehungen (*Links*), die über Assoziationen klassifiziert sind, welche die korrespondierenden Klassen verbinden. Abb. 2.2 zeigt ein Beispiel für ein Storydiagramm. Das Storydiagramm ist auf dem Komponentenmetamodell der MECHATRONIC UML spezifiziert und fügt einem vorhandenen Konvoisystem eine Komponenteninstanz :RailCab inkl. der entsprechenden Port- und Konnektorinstanzen hinzu.

Da Storydiagramme Varianten von Aktivitätsdiagrammen sind, können mit ihnen Kontrollstrukturen wie Sequenzen, Verzweigungen und Schleifen gebildet werden. Eine Parallelisierung von Aktivitäten ist nicht möglich. Zum einen erlauben diese Kontrollstrukturen eine Dekomposition der einzelnen Storypatterns in überschaubare Teilregeln anstelle einer großen Regel. Beispielsweise wird in Abb. 2.2 im Storypattern getLeader zunächst die führende Komponenteninstanz :RailCab anhand der Portinstanz :VCurrent identifiziert und gebunden, während im Storypattern createFollower die eigentliche Instanziierung der :RailCab-Komponenteninstanz für das folgende RailCabs stattfindet. Zum anderen wird durch die Kontrollstrukturen die Ausdrucksfähigkeit der Storydiagramme erhöht. Den Transitionen können *Bedingungen (Guards)* mit in der Zielprogrammiersprache formulierten booleschen Ausdrücken zugeordnet werden, welche den Kontrollfluss nach einem Verzweigungsknoten bestimmen. Des Weiteren gibt es die speziellen Guards success und failure für ausgehende Transitionen eines Storypatterns, welche eine Reaktion auf die erfolgreiche bzw. erfolglose Anwendung des Storypatterns erlauben. Eine spezielle Form der

[1] UML 1 Aktivitätsdiagramme basieren auf UML 1 Zustandsautomaten, während die Semantik von UML 2 Aktivitätsdiagrammen [Obj07b] durch ein Tokenkonzept wie das der Petrinetze beschrieben wird.

[2] In der UML 2 bekannt als Kommunikationsdiagramm [Obj07b].

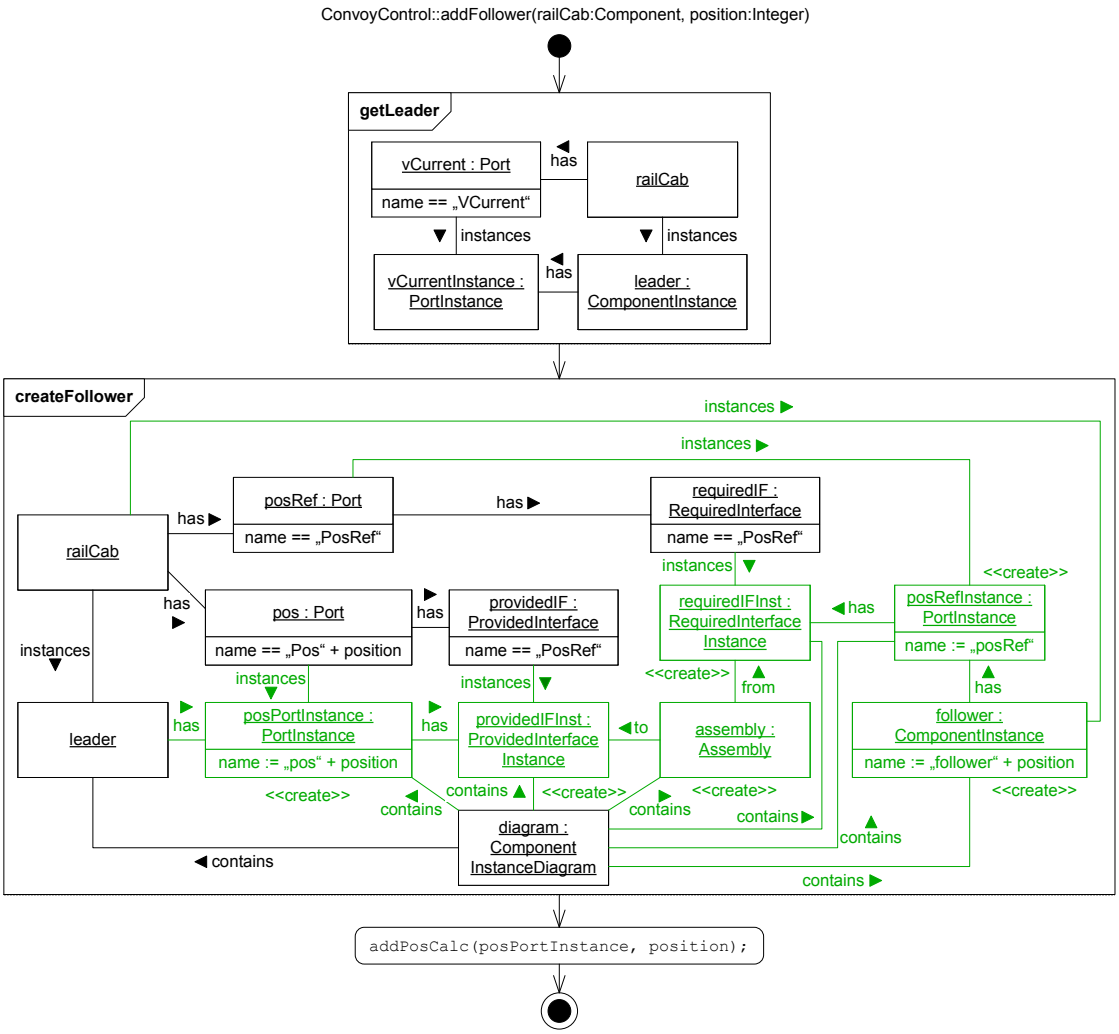


Abbildung 2.2: Storydiagramm zum Hinzufügen einer Komponenteninstanz :RailCab zu einem Konvoisystem

Schleife ist das forEach-Storypattern. Ein solches Storypattern wird sooft ausgeführt wie die LHS auf den Wirtsgraph angewendet werden kann. Spezielle Guards für die ausgehenden Transitionen des forEach-Storypatterns sind each time und end, die entsprechenden Transitionen werden nach jedem Schleifendurchlauf bzw. nach Terminierung der Schleife durchlaufen.

Zwecks einer einfachen Modellierung und Lesbarkeit wird sowohl die LHS als auch die RHS der Graphtransformationsregel im selben Storypattern definiert, indem Objektvariablen und Links *Modifizierer* in Form von Stereotypen zugeordnet werden können. Elemente ohne Modifizierer müssen für eine erfolgreiche Anwendung des Storypatterns existieren und bleiben nach der Anwendung erhalten. Objektvariablen und Links mit ≪destroy≫-Modifizierern (rot dargestellt) müssen ebenfalls für eine erfolgreiche Anwendung des Storypatterns gebunden werden, werden aber bei der Ausführung zerstört. Für Objektvariablen und Links mit ≪create≫-Modifizierern (grün dargestellt) werden Objekte und Objektbeziehungen im Wirtsgraph instanziiert. Die LHS eines Storypatterns definiert sich somit durch die Objektvariablen und Links ohne Modifizierer und mit ≪destroy≫-Modifizierer, während sich die RHS durch die Objektvariablen und Links ohne Modifizierer und mit ≪create≫-Modifizierern definiert.

Storydiagramme werden eingebettet in Methodenrümpfe einer Klasse, und können somit über die Signatur der Methode Parameter in Form von primitiven Datentypen oder Objektreferenzen entgegennehmen und Rückgabewerte liefern. In Abb. 2.2 gehört das Storydiagramm addFollower zur Klasse ConvoyControl. Über die Parameter ist Zugriff auf die Referenz des Objekts railCab:Component, welches den RailCab-Komponententyp darstellt, und einen Positionswert des Typs Integer möglich. In der letzten Aktivität des Storydiagramms, einer Statement-Aktivität, wird die Methode addPosCalc derselben Klasse aufgerufen, welche die :PosCalc-Komponenteninstanz erzeugen und mit der Portinstanz posPortInstance an der richtigen Position verbinden soll. Zu diesem Zweck werden posPortInstance und position als Argumente übergeben.

Man unterscheidet zwischen gebundenen und ungebundenen Objektvariablen. Ein Objekt wird innerhalb eines Storydiagramms einmal an eine Objektvariable gebunden und bleibt an diese gebunden bis zur Terminierung des Storydiagramms, bis zur Zuweisung eines anderen Objekts oder bis zur Objektzerstörung. Gebundene Objektvariablen werden in Storypatterns ohne Typinformation dargestellt, da an anderer Stelle bereits Objekte über einen Typ an sie gebunden wurden. In einem Storypattern gebundene Objektvariablen können in nachfolgenden Storypatterns wiederverwendet werden. In Abb. 2.2 wird z. B. die Objektvariable leader:ComponentInstance in getLeader gebunden und in createFollower als gebundene Variable wiederverwendet. Übergebene Parameter sind im gesamten Storypattern als gebundene Objektvariablen verfügbar, beispielsweise die Objektvariable railCab.

In Storypatterns sind Attributbedingungen sowie Attributzuweisungen möglich. Im Storypattern getLeader wird z. B. das Objekt vCurrent:Port anhand seines Namens „VCurrent“ gebunden. Dieses Objekt repräsentiert den Porttyp VCurrent. Beispiele für Attributzuweisungen finden sich im Storypattern createFollower, z. B. wird dem Attribut name des Objekts posPortInstance:PortInstance der Wert „pos“+position zugewiesen.

Für die Objektvariablen und die verbindenden Links der Storypatterns sowie für die

Storypatterns selber können Anwendungsbedingungen festgelegt werden. Ein *negatives Objekt* stellt ein Objekt dar, welches für eine erfolgreiche Anwendung des Storypatterns nicht im Wirtsgraph vorhanden sein darf. Negative Objektvariablen werden durchgestrichen dargestellt. *Optionale Objekte* dürfen, müssen aber nicht im Wirtsgraph vorhanden sein, um ein Storypattern erfolgreich anzuwenden. Optionale Objektvariablen werden durch einen gestrichelten Rahmen gekennzeichnet. Die beschriebenen Anwendungsbedingungen lassen sich auch einzelnen Links zuordnen und mit jeweils sinnvollen Modifizierern kombinieren. Des Weiteren gibt es die Möglichkeit, ein *Constraint* in Form eines in der Zielprogrammiersprache formulierten booleschen Ausdrucks für das gesamte Storypattern zu definieren. Das Storypattern wird nur dann ausgeführt, wenn der boolesche Ausdruck positiv ausgewertet wird. Ein *Mengenobjekt* fasst mehrere Objekte gleichen Typs zusammen. Eine solche Objektmenge lässt sich komplett löschen und ermöglicht massenhafte Erzeugungen oder Löschungen neuer bzw. vorhandener Objektbeziehungen zu den einzelnen Elementen. Mengen werden mit einem zusätzlichen angedeuteten Rahmen dargestellt.

Bezüglich dangling edges verfahren Storypatterns nach dem SPO-Ansatz, so werden zu einem zu zerstörenden Objekt inzidente Links implizit gelöscht. Des Weiteren werden zu einem zu instanziierenden Objekt im Storypattern modellierte, inzidente Links implizit erstellt. Z. B. wird im Storypattern createFollower den einzelnen Links kein Modifizierer zugeordnet, lediglich den Objektvariablen. Konflikte für die Abbildungsfunktion treten bei Storypatterns nicht auf, da Isomorphismus für die LHS gefordert wird, d. h. dass jeder Knoten der LHS auf genau einen Knoten im Wirtsgraph abgebildet wird. Eine weitere Besonderheit ist die Ersetzung von Links, welche typisiert sind über eine Assoziation mit einer oberen Multiplizitätsgrenze von 1 auf mindestens einer Seite. Wenn ein zu einem solchen Link inzidentes Objekt instanziiert werden soll, wird ein mit einem anderen, zu erhaltenden Objekt inzidenter Link der gleichen Assoziation durch den neuen Link ersetzt, damit die resultierende Objektstruktur valide gegenüber dem korrespondierenden Klassendiagramm ist. Des Weiteren wird bei der Bindung von Objekten die Klassenhierarchie berücksichtigt. So werden auch Objekte einer erbenden Klasse an eine Objektvariable eines bestimmten Typs gebunden.

Storypatterns können auf Korrektheit hin verifiziert werden. Mit dem Graphmodelchecker GROOVE[3], welcher eine Importfunktion für Storypatterns besitzt, lässt sich z. B. von einem initialen Startgraph aus eine Erreichbarkeitsanalyse bzgl. struktureller Eigenschaften des Graphen durchführen [KR06a]. Die strukturellen Eigenschaften werden dabei als LHS einer Graphtransformationsregel modelliert. Mit der Sicherheitseigenschaft (safety property) wird überprüft, dass der modellierte Subgraph in keinem erreichbaren Zustand des zu Grunde liegenden Graphen auftritt. Mit der Lebendigkeitseigenschaft (liveness property) kann verifiziert werden, dass der modellierte Subgraph von jedem Zustand des Gesamtgraphen aus erreichbar ist.

Graphmodelchecker wie GROOVE benötigen einen Startgraphen für die Erreichbarkeitsanalyse, des Weiteren muss sichergestellt werden dass der generierte Zustandsraum endlich ist. Letzteres ist in mechatronischen Systemen nur selten gegeben. Aus diesen

[3] **GR**aphs for **O**bject-**O**riented **VE**rification

Gründen wird in [Sch06] ein Verifikationsansatz für Storypatterns vorgestellt, welcher ohne initialen Startgraphen und ohne Zustandsraumgenerierung auskommt. Die Verifikation arbeitet stattdessen induktiv, d. h. es wird gezeigt, dass die Regelanwendung auf einen korrekten Graphen wiederum in einem korrekten Graphen resultiert. Dies wird simuliert, indem für einen inkorrekten Graphen bzw. für ein *unerwünschtes Muster* alle anwendbaren Transformationen rückwärts ausgeführt werden. Kann mit der Rückwärtstransformation ein korrekter Ausgangsgraph erzeugt werden, so gibt es inkorrekte Graphtransformationen. Da der in dieser Arbeit vorgestellte Graphtransformationsansatz ebenfalls die Domäne der mechatronischen Systeme abdeckt und Graphmodelchecker somit unpraktikabel sind, könnten die vorgestellten Strukturtransformationen in zukünftigen Arbeiten mit einem ähnlichen Verifikationsansatz auf Korrektheit überprüft werden.

Fujaba unterstützt für Storydiagramme eine Codegenerierung für Java und weitere objektorientierte Sprachen. Da Storydiagramme Methoden definieren, wird für jedes Storydiagramm der jeweilige Methodencode in den Code der zu Grunde liegenden Klasse generiert.

2.1.2 Transformationsdiagramme

Transformationsdiagramme [Mey06] sind eine Erweiterung von Storydiagrammen und wurden ursprünglich für das musterbasierte Reengineering von Softwaresystemen im Rahmen der Projektgruppe Automotives Software Engineering[4] entwickelt. Sie werden als Zielformalismus für die Abbildung der in dieser Arbeit vorgestellten Komponentenstorydiagramme benutzt. Transformationsdiagramme erweitern Storydiagramme u. a. um die nahtlose Integration von direkten Aufrufen weiterer Transformationsdiagramme innerhalb eines umgebenden Transformationsdiagramms. Durch ein weiter entwickeltes Konzept für Parameter und Rückgabewerte können gebundene Objekte der aufrufenden Transformation auf einfache Weise als Parameterargumente an die aufgerufene Transformation übergeben, und Rückgabewerte der aufgerufenen Transformation in Form von Objektvariablen innerhalb der aufrufenden Transformation verwendet werden. In konventionellen Storydiagrammen ist dies lediglich über Umwege durch Statement-Aktivitäten möglich. Daher wird eine Dekomposition der Regeln ermöglicht, wodurch die Wiederverwendbarkeit steigt.

Abb. 2.3 zeigt ein solches Transformationsdiagramm, mit dem die Komponenteninstanz eines nachfolgenden RailCabs aus einem Konvoisystem entfernt werden soll. In diesem Beispiel wird die Portinstanz, mit der die zu löschende Komponenteninstanz :RailCab verbunden ist, nicht direkt über den Namen gebunden, sondern innerhalb des (hier nicht dargestellten) Transformationsdiagramms RemovePosCalc. Im Storypattern getPosPortInstance wird das Transformationsdiagramm RemovePosCalc über eine spezielle Aktivität namens *Transformationsaufruf (Transformation Call)* aufgerufen. Dabei wird das Argument leader:ComponentInstance übergeben, was durch den Link mit ≪argument≫-Stereotyp dargestellt wird. Argumente primitiver Datentypen werden direkt innerhalb des Transformationsaufrufs annotiert, siehe position. In RemovePosCalc wird die entspre-

[4] PG ASE, `http://wwwcs.uni-paderborn.de/cs/ag-schaefer-static/Lehre/PG/ASE/`

chende :PosCalc-Instanz gelöscht und die ursprünglich mit ihr verbundene Portinstanz über die Rückgabedeklaration posRetDecl zurückgegeben. Deren Rückgabewert wird über den Link mit dem Stereotyp ≪result≫ an posPortInstance gebunden. Konnte eine solche Portinstanz erfolgreich ermittelt werden, wird diese in destroyFollower zusammen mit dem verbindenden Konnektor und der Komponenteninstanz :RailCab gelöscht.

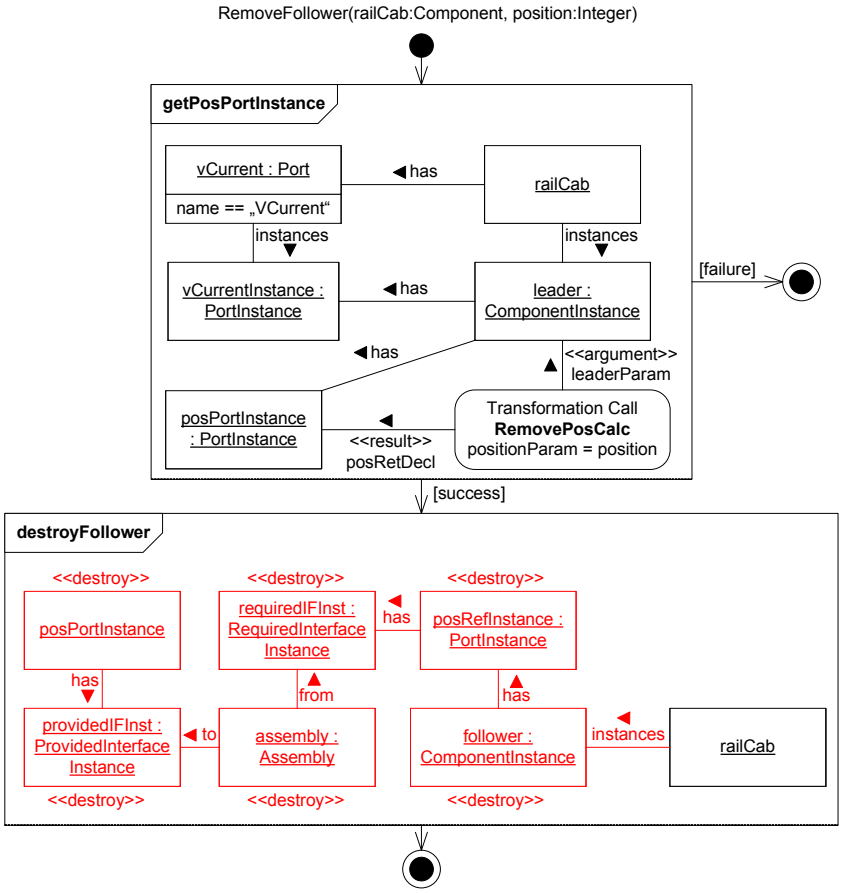


Abbildung 2.3: Transformationsdiagramm zum Löschen einer Komponenteninstanz :RailCab aus einem Konvoisystem

Der im letzten Unterabschnitt beschriebene Verifikationsansatz für einzelne Storypatterns zur Vermeidung unerwünschter resultierender Graphen [Sch06] wird in [Det08] für komplette Transformationsdiagramme adaptiert. Transformation Calls werden von diesem Ansatz allerdings nicht berücksichtigt. Des Weiteren beschäftigen sich aktuelle Forschungsarbeiten mit weiteren Verifikationsmethoden für Transformationsdiagramme. So ist es z. B. wünschenswert, neben verbotenen Mustern auch *zu erhaltende Muster* zu spezifizieren, welche nach der Ausführung eines Transformationsdiagramms noch erhalten sein müssen [Det08].

Auch für Transformationsdiagramme bietet Fujaba eine Codegenerierung, im Gegen-

satz zu konventionellen Storydiagrammen wird allerdings für jedes Transformationsdiagramm eine eigene Klasse generiert.

2.1.3 Triple-Graph-Grammatiken

Mit konventionellen Graphtransformationen werden i. A. Transformationen auf derselben Graphklasse spezifiziert. Für Transformationen zwischen Graphen verschiedener Graphklassen bzw. Metamodelle sind diese nur bedingt geeignet. Transformationen in die Rückrichtung müssten beispielsweise gesondert modelliert werden. Auch Konsistenzüberprüfungen sowie eine inkrementelle Übertragung von Veränderungen eines Graphen in einen anderen sind nicht möglich. Aus dieser Motivation heraus wurde der Formalismus der *Triple-Graph-Grammatiken (TGGs)* entwickelt [Sch95].

TGGs ermöglichen eine deklarative, graphbasierte Modelltransformation eines Quellgraphen in einen Zielgraphen und vice versa, wobei die Klassen von Quelle und Ziel unterschiedlich sein können. Neben dem kompletten Durchlauf einer solchen Transformation sind auch bidirektionale Modellsynchronisationen sowie Konsistenzüberprüfungen zwischen zwei verschiedenen Graphen möglich.

TGGs bestehen aus Produktionsregeln für drei verschiedenen Arten von Graphen: Einen linken und einen rechten Graphen sowie einen Korrespondenzgraphen. Der linke und rechte Graph sind Transformations-/Synchronisationsquelle bzw. -ziel, der Korrespondenzgraph stellt zum einen die Beziehung zwischen den Knoten des linken und rechten Graphen her und zum anderen die Beziehung zwischen den verschiedenen Produktionsregeln. Eine Produktionsregel einer TGG, im folgenden TGG-Regel genannt, besteht aus Tripeln von einzelnen Produktionsregeln für den jeweiligen Graphtyp. Ein solches Tripel modelliert somit drei verschiedene Produktionen, welche mit konventionellen Graphgrammatiken einzeln beschrieben werden müssten. TGGs erlauben lediglich monotone Produktionen, d. h. dass die LHS einer TGG-Regel (und damit die jeweiligen linken Seiten aller drei enthaltenen Produktionen) auch in der RHS vorhanden sein muss und somit keine Knoten und Kanten gelöscht werden.

TGG-Regeln werden in dieser Arbeit für das Mapping von Komponentenstorydiagrammen auf Transformationsdiagramme benutzt. Zur Modellierung der TGG-Regeln wird das Fujaba-Plugin TGGEditor, zur Ausführung das Plugin MoTE[5] [Wag06] verwendet. TGG-Regeln dieser Plugins benutzen die Syntax der Storydiagramme und werden übersetzt in Tripel von Storydiagrammen, aus denen letztendlich ausführbarer Code generiert wird.

2.2 Softwarearchitekturen

Nach Shaw und Garlan [SG96] wird durch die steigende Komplexität und Größe von Softwaresystemen die Architektur des Gesamtsystems zu einem wichtigeren Thema als die Wahl von Algorithmen und Datenstrukturen für einzelne Berechnungen. Eine *Softwarearchitektur* repräsentiert ein Softwaresystem mit Komponenten, aus denen das System

[5] **Mo**del **T**ransformation **E**ngine, `http://www.fujaba.de/projects/tgg`

aufgebaut ist, mit Interaktionen zwischen diesen Komponenten sowie mit Mustern, nach denen diese Elemente angeordnet werden. Ein solches System kann wiederum eine Komponente eines größeren Systems sein, somit ergibt sich eine hierarchische, kompositionale Struktur.

Softwarearchitekturen lassen sich mit *Architekturbeschreibungssprachen (architecture description languages, ADLs)* spezifizieren. Zwecks einer einheitlichen und formalen Notation bieten ADLs Konstrukte zur deklarativen Modellierung von Softwarearchitekturen. Nach Shaw und Garlan sollten die Möglichkeiten dieser Konstrukte u. a. die Beschreibung von Systemen als hierarchische Komposition aus unabhängigen Komponenten und Interaktionen in Form von abstrakten Rollen, die dynamische Strukturanpassung von Systemen und deren Analyse umfassen. Für eine weiterführende Diskussion bzgl. der Definition und Charakteristiken von ADLs sei auf [MT00] verwiesen. Abschnitt 3.2 stellt einige ADLs mit Bezug auf den in dieser Arbeit vorgestellten Ansatz vor.

In der komponentenbasierten Softwareentwicklung wird mit der Modularisierung und Dekomposition eines Softwaresystems in Komponenten sowie mit der Benutzung existierender Komponenten die Wiederverwendbarkeit von Software gesteigert, um der steigenden Komplexität zu begegnen. *Softwarekomponenten* sind geschlossene Einheiten mit festgelegten Abhängigkeiten zu ihrer Umgebung, welche ihre Implementierung kapseln und Schnittstellen bereitstellen, über welche auf die Dienste der Komponente zugegriffen kann [Szy98]. Komponenten können unabhängig voneinander entwickelt und in verschiedenen Systemen oder von anderen Komponenten benutzt werden. Der ausschließliche Zugriff auf Komponenten über Schnittstellen ermöglicht ihren modularen Austausch. Komponenten können aus weiteren Komponenten zusammengesetzt sein. Interaktionen zwischen Komponenten werden in ADLs mit *Konnektoren* beschrieben, welche eine abstrakte Repräsentation der Abhängigkeiten von Komponenten zu den angebotenen Diensten anderer Komponenten sind und somit Kommunikationsbeziehungen darstellen. Während Szyperski nicht zwischen einem Komponententyp und einer Komponenteninstanz unterscheidet, da nach ihm eine Komponente höchstens einmal in einem System verwendet wird, müssen ADLs nach Shaw und Garlan [SG96] oder Medvidovic und Taylor [MT00] die Unterscheidung von Typen und Instanzen für Komponenten und Konnektoren unterstützen.

Die erlaubten Komponentengefüge innerhalb von Systemen bzw. übergeordneten Komponenten werden mit Mustern beschrieben. Ein solches Muster wird *Architekturstil* genannt. Shaw und Garlan vergleichen Architekturstile mit den strukturellen Entwurfsmustern (Design Patterns) [GHJV95] auf der höher liegenden architektonischen Entwurfsebene, während Buschmann et al. sie mit ihren Architekturmustern [BMR+96] vergleichen. Architekturstile beschreiben gleichartige Familien von Architekturen und gemeinsame Eigenschaften, die für alle Mitglieder eines Stils gelten. Ein bekannte Beispiel für einen Architekturstil nach Shaw und Garlan ist „Pipes and Filters“ [SG96]. Architekturstile können beispielsweise mithilfe eines Typsystems, in Form einer Sprache durch grammatikalische Produktionen oder durch eine Menge von Axiomen und Inferenzregeln konkret definiert werden [Gar95].

2.3 Mechatronic UML

Mechatronische Systeme können in ihren Ressourcen stark eingeschränkt sein und werden oft in sicherheitskritischen, verteilten Echtzeitumgebungen eingesetzt. Daher wird für eine Spezifikation eines mechatronischen Systems eine ADL benötigt, welche diesen Kriterien genügt. Auch die UML und ihre Varianten SysML oder UML-RT können zur Systemmodellierung herangezogen werden (siehe Abschnitt 2.4). Die Echtzeitkonzepte dieser Sprachen sind aber sehr eingeschränkt, dagegen bietet das UML Profile for Schedulability, Performance, and Time (SPT-Profil) [Obj05] sehr ausgefeilte Möglichkeiten zur Modellierung von Zeit- und Ressourcenanforderungen. Allerdings lässt sich nicht spezifizieren, wie diese Anforderungen in Verhaltensbeschreibungen oder in eine komplexe Softwarearchitektur integriert werden können. Des Weiteren ist man im Bereich der mechatronischen Systeme mit hybriden Systemen konfrontiert, welche diskrete Kontrollmodi mit kontinuierlichen Signalen vereinen. Diese lassen sich z. B. mit SysML modellieren, welche allerdings nicht über die Möglichkeiten zur Modellierung von Strukturanpassungen wie die ADLs verfügt. Da die UML und ihre Derivate lediglich Modellierungssprachen sind, fehlen des Weiteren Konzepte für die Analyse der modellierten Systeme. Andere, existierende Modellierungsansätze für hybride Systeme oder Echtzeitsysteme integrieren nicht ganzheitlich die Modellierung von Struktur und Verhalten, Verifikation und Implementierung.

Um für einen solchen ganzheitlichen Ansatz die weit verbreiteten Notationen der UML wiederzuverwenden und sie auf die Domäne der mechatronischen Systeme zu zuschneiden, wurde im Fachgebiet Softwaretechnik der Universität Paderborn die UML-basierte Sprache Mechatronic UML [Bur05, Sch06] zur Modellierung komponentenbasierter, mechatronischer Systeme entwickelt. Der Fokus liegt auf der Koordination von verteilten Systemen, auf der Modellierung von Echtzeitverhalten sowie der Verifikation und Analyse von sicherheitskritischen Aspekten. Auch die Mechatronic UML kann in ihrer speziellen Domäne als ADL verstanden werden.

2.3.1 Struktur

Komponenten sind die Grundbausteine der Architekturen von Mechatronic UML. Zur Kommunikation mit der Umwelt besitzen Komponenten *Ports*, auf welche über *Schnittstellen (interfaces)* zugegriffen werden kann. Man unterscheidet zwischen einer *angebotenen Schnittstelle (provided interface)*, welche Dienste der Komponente zur Verfügung stellt, und einer *benötigten Schnittstelle (required interface)*, welche eine Abhängigkeit zur Umwelt ist und von Diensten anderer Komponenten bedient werden muss. Zur Modellierung werden UML Komponentendiagramme [Obj07b] verwendet. In Mechatronic UML wird generell zwischen Typen und Instanzen unterschieden. So werden für *Komponententypen* allgemeine Eigenschaften festgelegt, welche für alle *Komponenteninstanzen* des Typs gelten. Analog gibt es *Porttypen* und *Portinstanzen* sowie *Schnittstellentypen* und *Schnittstelleninstanzen*.

Abb. 2.4 stellt Komponentendiagramme für einige der Komponententypen dar, deren Instanzen im RailCab-Konvoi aus der Einleitung verwendet wurden. Der Komponen-

tentyp PosCalc besitzt zwei Porttypen PosRef und Pos, welche mit einer benötigten bzw. zwei angebotenen Schnittstellen des Typs PosRef ausgestattet sind. Schnittstellen werden mit der so genannten „Lollipop“-Notation visualisiert. So werden benötigte Schnittstellen durch mit Ports verbundene Halbkreise, angebotene Schnittstellen durch mit Ports verbundene Kreise repräsentiert. PosCalc berechnet Referenzpositionen eines nachfolgenden RailCabs aus der aktuellen Position des vorausgehenden RailCabs. Über die benötigte Schnittstelle PosRef wird die Position des vorausgehenden Shuttles erwartet, die korrespondierenden angebotenen Schnittstellen von PosCalc und $\int$ bieten einen Zugriff auf diese Daten.

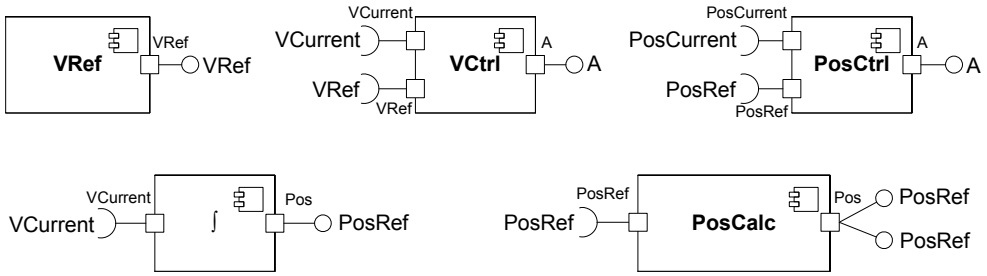


Abbildung 2.4: Komponentendiagramme für einfache Komponententypen

Um die Wiederverwendbarkeit von Komponententypen zu erhöhen, werden Komponententypen aus weiteren Komponententypen gebildet. Somit unterscheidet man Komponententypen in *einfache* und *hierarchische Komponententypen* [THHO08]. Einfache Komponententypen sind atomare Komponententypen, welche sich nicht in weitere Komponententypen zerlegen lassen, vgl. Abb. 2.4. Hierarchische Komponententypen im bisherigen Ansatz von MECHATRONIC UML bestehen aus Instanzen weiterer einfacher oder auch hierarchischer Komponententypen. Wird ein solcher hierarchischer Komponententyp instanziiert, wird die eingebettete Komponenteninstanzstruktur des Typen direkt übernommen, d. h. die (implizite) Multiplizität der eingebetteten Elemente beträgt exakt 1.

Abb. 2.5 zeigt den hierarchischen Komponententypen RailCab, in denen Instanzen der einfachen Komponententypen aus Abb. 2.4 gemäß ihrer Schnittstellenkompatibilitäten über *Konnektoren* verbunden werden. *Kompositionskonnektoren (assembly connectors)* verbinden eine angebotene und eine benötigte Schnittstelle gleichen Namens auf derselben Hierarchieebene, visualisiert durch die vereinigten Schnittstellen in Lollipop-Notation. Ein Beispiel hierfür ist der Kompositionskonnektor zwischen :VRef und :VCtrl. *Delegationskonnektoren (delegation connectors)* verbinden eine Schnittstelle einer eingebetteter Komponenteninstanz mit einer Schnittstelle gleichen Typs und gleichen Namens des umgebenden Komponententyps, z. B. die Konnektoren zwischen den :PosCalc-Instanzen und den Porttypen Pos1/Pos2. Aufrufe einer Schnittstelle der umgebenden Komponente werden über diesen Konnektor an die eingebetteten Komponenten delegiert, und vice versa. Informationen fließen nach der UML von benötigten Schnittstellen zu angebotenen Schnittstellen [Obj07b]. Eine Typisierung von Konnektoren wie die von Komponenten, Ports und Schnittstellen, gibt es in MECHATRONIC UML nicht.

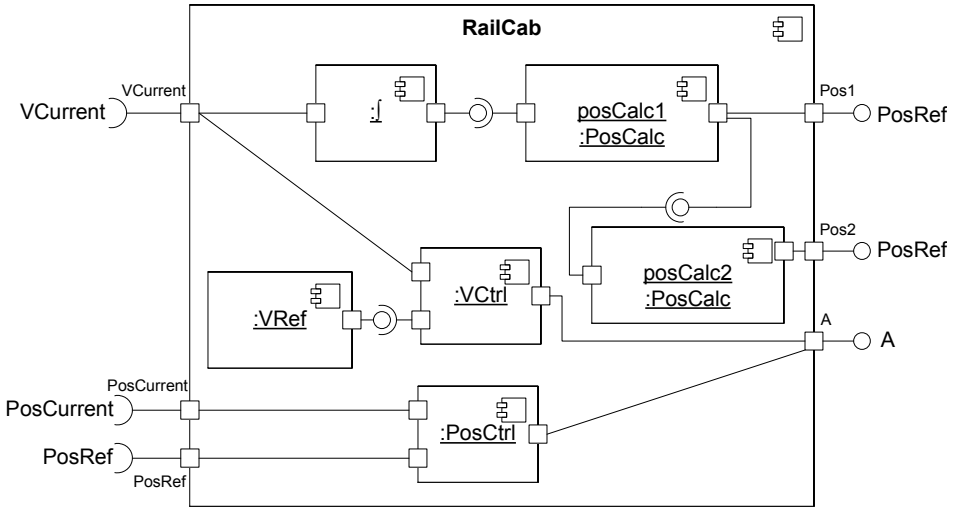


Abbildung 2.5: Komponentendiagramm für hierarchischen Komponententyp RailCab

Die Modellierung eines Architekturstils für ein gesamtes System ist in MECHATRONIC UML nicht möglich, da man keine Typdefinition für die oberste Systemebene anlegen kann. Stattdessen werden Komponenten direkt instanziiert und gemäß ihrer Schnittstellenabhängigkeiten untereinander verbunden. Eine konkrete Komponenteninstanzstruktur für ein (Teil-)System zu einem bestimmten Zeitpunkt, bestehend aus einer festen Anzahl von untereinander verbundenen Komponenten-, Port-, Schnittstellen- und Konnektorinstanzen, wird in Anlehnung an [FGK+04, Bur05] in dieser Arbeit als *Konfiguration* bezeichnet.

2.3.2 Diskretes Verhalten

Ein *diskretes System* wechselt zwischen verschiedenen diskreten Kontrollmodi. Ein RailCab kann z. B. entweder den Konvoi leiten oder im Konvoi folgen, Zustände dazwischen gibt es nicht. Bisher wurde lediglich die strukturelle Modellierung von Komponententypen betrachtet, für die Komponententypen bzw. deren Instanzen muss auch ein solches diskretes Verhalten spezifiziert werden können.

Statecharts [Har87] bzw. deren UML-Varianten der Verhaltens- und Protokollzustandsautomaten [Obj07b] bieten ein übersichtliches und zugleich mächtiges Modell für diskrete Zustände. Ein Zeitverhalten ist allerdings lediglich rudimentär definiert, für die Modellierung harter Echtzeitsysteme mit z. B. *maximalen Ausführungszeiten (worst-case execution times, WCETs)* eignen sich UML Zustandsautomaten in ihrer ursprünglichen Form somit nicht [BGS05].

Daher wurde das Konzept der *Realtime-Statecharts (RTSCs)* [BGS05, BG03, GB03, Bur05, Sch06] entwickelt. In RTSCs finden die After- und When-Konstrukte herkömmlicher UML Zustandsautomaten keine Anwendung, dafür werden RTSCs um Uhren erweitert und um Konstrukte, welche diese Uhren auswerten. Zustände besitzen zusätzlich Zeitinvarianten, WCETs für alle Methoden und ein Periodenintervall für die do()-Methode.

Zeitinvarianten spezifizieren dabei, wann ein Zustand spätestens verlassen werden muss, WCETs wie lange die Ausführung einer Methode maximal dauern darf und das Periodenintervall wie häufig die do()-Methode ausgeführt werden soll. Des Weiteren können Uhren beim Betreten oder Verlassen des Zustandes zurückgesetzt werden. Transitionen werden um Zeitguards, Prioritäten, Deadlines und WCETs für Seiteneffekte erweitert. Ein Zeitguard ist ein zusätzlicher Guard, welcher Uhren auswertet. Prioritäten verwalten mehrere, zum gleichen Zeitpunkt aktivierte Transitionen zur Vermeidung von Nichtdeterminismus. Deadlines geben mittels eines Zeitintervalls an, wann die Transition frühestens und wann spätestens schalten darf. Dabei ist zu beachten, dass Transitionen eines RTSCs im Gegensatz zu den konventionellen Statecharts nicht in Nullzeit schalten. Zusätzlich werden Transitionen um einen Mechanismus angereichert, der eine Synchronisation mit einer weiteren Transition in einem parallelen Zustand erlaubt.

Aus RTSCs lässt sich Code für Real-Time Java [GB00] und C++ generieren, welcher die geforderten Echtzeiteigenschaften auf geeigneten Echtzeitbetriebssystemen umsetzt.

2.3.3 Kommunikation

Mit *Echtzeit-Koordinationsmustern (Real-Time Coordination Patterns)* [GTB+03, Bur05, Sch06] kann die Kommunikation zwischen verteilten Komponenten spezifiziert werden. Im hier angeführten Beispiel kann ein solches Echtzeit-Koordinationsmuster das Ein- oder Austreten eines RailCabs in den Konvoi modellieren. Die Muster werden zunächst zwischen miteinander kommunizierenden *Rollen* definiert. Rollen abstrahieren von konkreten Komponenten und stellen einen Platzhalter für einen externen Kommunikationspartner dar. Sie sind über einen Konnektor miteinander verbunden. Das Verhalten der Rollen als auch des verbindenden Konnektors wird über RTSCs spezifiziert. Um eine Interaktion zwischen den Rollen zu gewährleisten, kommunizieren Rollen-RTSCs miteinander über Signale. Das Konnektor-RTSC entscheidet, welche Signale von welcher Quelle zu welchem Ziel gelangen müssen, liefert diese aus und modelliert Charakteristiken bzgl. der Dienstgüte (Quality of Service, QoS).

Für die einzelnen Rollen können Invarianten modelliert werden, die eine Rolle erfüllen muss. Für das gesamte Koordinationsmuster wird eine Musterbedingung angegeben, welche Anforderungen bzgl. des Verhaltens der Rollen und des Konnektors sowie der Rolleninvarianten erhebt. Mit Modelchecking kann dann verifiziert werden, dass das Verhalten des gesamten Koordinationsmusters korrekt ist. Die Trennung von Mustern und konkreten Komponententypen fördert die Wiederverwendbarkeit der Muster, so kann das Verhalten verschiedener Muster zunächst unabhängig von konkreten Komponententypen spezifiziert und verifiziert werden.

Soll der RailCab-Komponententyp die Rolle eines zuvor spezifizierten Koordinationsmusters einnehmen, so muss das Verhalten des Komponententyps das Verhalten der Rolle verfeinern. Verfeinerung einer Rolle bedeutet für RTSCs, dass kein nach außen sichtbares Verhalten hinzugefügt wird und dass das Verhalten der Rolle erhalten bleibt. Die Verifikation des Musters wird dabei zunächst basierend auf den Rollen und anschließend auf den Komponententypen durchgeführt, welche das Verhalten der Rollen verfeinern. Dies impliziert die Korrektheit des Systems für alle Konfigurationen, unabhängig von der tat-

sächlichen Anzahl möglicher Komponenteninstanzen [GTB+03]. Für einen Konvoi mit einer großen Anzahl an RailCabs wird somit die Korrektheit der Konvoikoordination garantiert ohne eine ressourcenbelastende Verifikation aller miteinander kommunizierenden Komponenteninstanzen.

2.3.4 Gefahrenanalyse

Zur Entwurfszeit können komponentenbasierte Gefahrenanalysen auf einem System ausgeführt werden [GTS04, GT06]. Zu diesem Zweck können *Hazards* für ein System spezifiziert werden. Ein Hazard ist eine Situation, in der Gefahr für Menschen oder die Umgebung besteht [Sto96] und tritt nach der von [GTS04, GT06] vorgeschlagenen Analysemethode bei einer bestimmten Kombination von manifestierten Fehlern in den Komponenteninstanzen einer Konfiguration auf. Fehler können direkt in einer Komponenteninstanz entstehen oder als Folgefehler über Konnektoren propagiert werden und sich somit an Portinstanzen mit benötigten Schnittstellen manifestieren. Um dies zu modellieren, werden Komponententypen mit Fehlerpropagierungsregeln basierend auf boolescher Logik versehen. Eine Fehlerpropagierung für einen Komponententypen besteht aus einer Menge von ein- und ausgehenden Folgefehlern, direkt in der Komponente entstandenen Fehlern und deren Abhängigkeiten untereinander. Um einen Fehlerfall im System zu simulieren, werden zur Entwurfszeit in Komponenteninstanzen Fehler injiziert, welche sich basierend auf den kompositionalen Fehlerpropagierungsregeln im System fortpflanzen. Auf dem System werden anschließend Gefahrenanalysen durchgeführt, und im Bedarfsfall wird die Konfiguration verändert.

2.3.5 Hybrides Verhalten

Wie bereits erwähnt, ist man in der Mechatronik oft mit *hybriden Systemen* konfrontiert. Ein hybrides System vereint diskrete und *kontinuierliche Systeme*. Ein kontinuierliches System verarbeitet kontinuierliche Signale, z. B. die aktuelle Geschwindigkeit oder Beschleunigung des RailCabs. Bei einem Wechsel zwischen den diskreten Zuständen des RailCabs müssen diese Signale kontinuierlich angepasst werden, beispielsweise darf die Betätigung des Antriebs des führenden RailCabs nicht aufgrund eines Zustandswechsels schlagartig ausgesetzt werden.

Für die Modellierung des Verhaltens und der Struktur von hybriden Systemen wurden *hybride Komponenten*[6] und *hybride Rekonfigurationscharts* [GBSO04, BGO04, BGT05, Bur05, Sch06] eingeführt. Hybride Rekonfigurationscharts sind eine Erweiterung von RTSCs und ergänzen diese um *Überblendtransitionen (fading transitions)*. Diese sind mit einer Überblendfunktion ausgestattet, welche die Anpassung von kontinuierlichen Signalen während eines Zustandswechsels vornimmt. Des Weiteren ermöglichen hybride Rekonfigurationscharts die Strukturanpassung von Komponenteninstanzen, dies wird im folgenden Unterabschnitt vorgestellt.

[6] Man beachte, dass das laufende Beispiel dieser Arbeit lediglich aus diskreten Komponenten besteht, um eine Konzentration auf den strukturellen Aspekt zu ermöglichen. In der Realität ist RailCab ein hybrider Komponententyp.

2.3.6 Strukturanpassung

Softwaresysteme können zur Laufzeit ihre Struktur verändern, um ihr Verhalten anzupassen. Komponenten oder Objekte kommen hinzu und verschwinden, und Kommunikationsbeziehungen zwischen existierenden Komponenten oder Objekten werden verändert. Eine solche *Verhaltensanpassung (behavior adaptation)*, welche die Anordnung und Beziehungen von Elementen verändert, wird *Strukturanpassung (structural adaptation)* genannt [FGK+04].

Strukturanpassungen werden im bisherigen Ansatz von MECHATRONIC UML mit hybriden Rekonfigurationscharts durchgeführt. Mit ihnen lassen sich Komponenten-, Port- und Konnektorinstanzen aktivieren bzw. deaktivieren und somit unterschiedliche Konfigurationen modellieren. Abb. 2.6 zeigt ein stark vereinfachtes hybrides Rekonfigurationschart ohne Überblendtransitionen und Echtzeitbeschränkungen, welches die möglichen Konfigurationen des RailCab-Komponententypen definiert. Im Zustand Follower wird die Komponenteninstanz :PosCtrl eingesetzt, welche die Referenzposition posRef mit der aktuellen Position posCurrent vergleicht und basierend auf der Differenz die Beschleunigung a variiert. Im Zustand Leader-NoFollower wird die Beschleunigung von der Komponenteninstanz :VCtrl kontrolliert, welche die aktuelle Geschwindigkeit vCurrent und die Referenzgeschwindigkeit, bereit gestellt von :VRef, auswertet. Für jedes nachfolgende RailCab wird im führenden RailCab eine Komponenteninstanz :PosCalc verwendet, um die jeweilige Referenzposition zu berechnen. Dies wird in den Zuständen Leader-1Follower und Leader-2Followers spezifiziert.

2.3.7 Einschränkungen

Der vorgestellte Modellierungsansatz von MECHATRONIC UML für die Struktur eines Systems hat aus der Modellierungsperspektive zwei Nachteile [THHO08].

Zum einen ist es nicht möglich, einen hierarchischen Komponententyp mit beliebig vielen eingebetteten Komponenteninstanzen zu modellieren. Wie in Abb. 1.2 aus der Einleitung angedeutet, soll im laufenden Beispiel ein Konvoi mit einer beliebigen Anzahl von RailCabs gebildet werden.[7] Dies erfordert für n im Konvoi folgende RailCabs nicht nur n Komponenteninstanzen :RailCab für die nachfolgenden RailCabs, sondern auch n eingebettete Komponenteninstanzen :PosCalc und n entsprechend verbundene Portinstanzen für leader:RailCab. Das laufende Beispiel erfordert also eine unendliche Menge möglicher Konfigurationen. Der hierarchische Komponententyp aus Abb. 2.5 dagegen erlaubt lediglich maximal zwei eingebettete Komponenteninstanzen :PosCalc. Das hybride Rekonfigurationschart in Abb. 2.6 zählt alle sinnvollen Konfigurationen auf, wobei die Konfigurationsmenge offensichtlich endlich ist.

Zum anderen ist die Modellierung von Strukturanpassungen in Form von hybriden Rekonfigurationscharts nur praktikabel bei einer kleinen Menge möglicher Konfigurationen, da jede mögliche Konfiguration einzeln modelliert werden muss. Für einen Komponententyp RailCab mit z. B. 100 eingebetteten Komponenteninstanzen :PosCalc müssen im

[7] In einem realen Echtzeit-System wird diese beliebige Anzahl auf einen konkreten maximalen Wert fixiert, um eine vorhersagbare WCET für Strukturanpassungen zu garantieren.

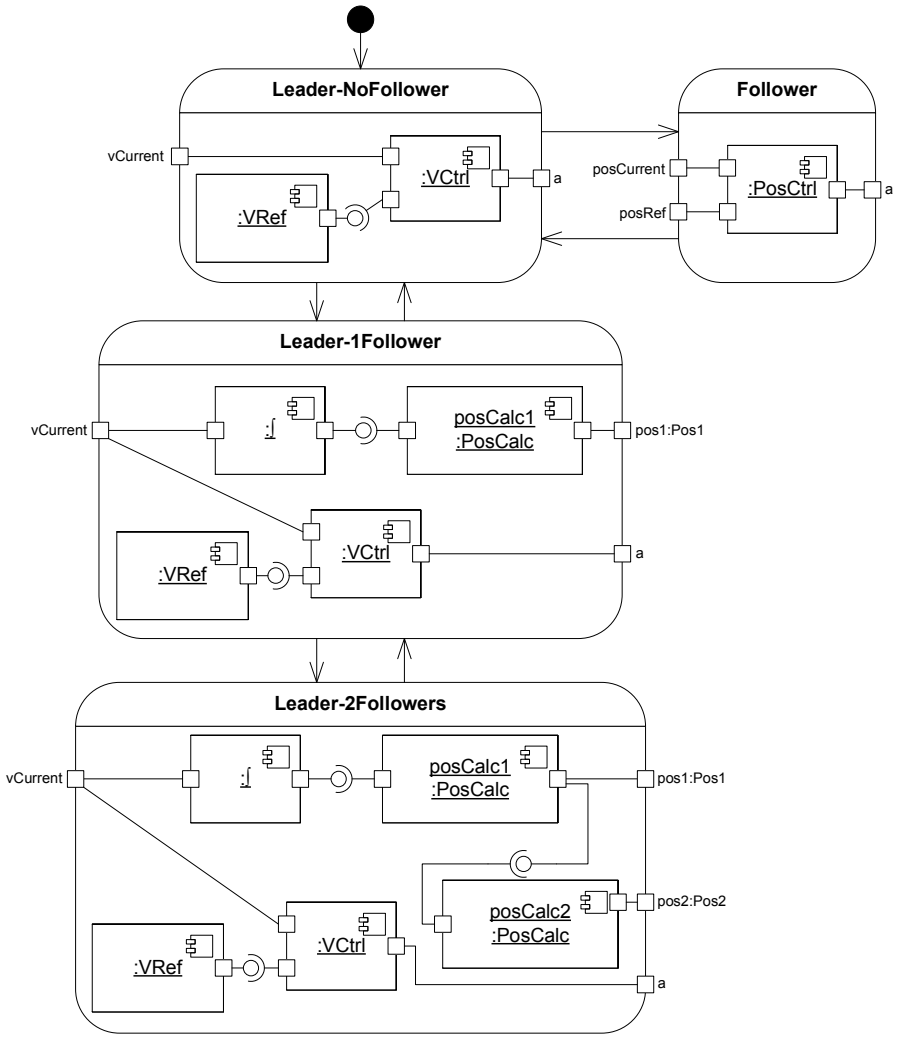


Abbildung 2.6: Vereinfachtes hybrides Rekonfigurationschart

korrespondierenden hybriden Rekonfigurationschart alle eingebetteten Komponenteninstanzsituationen aufgelistet werden.

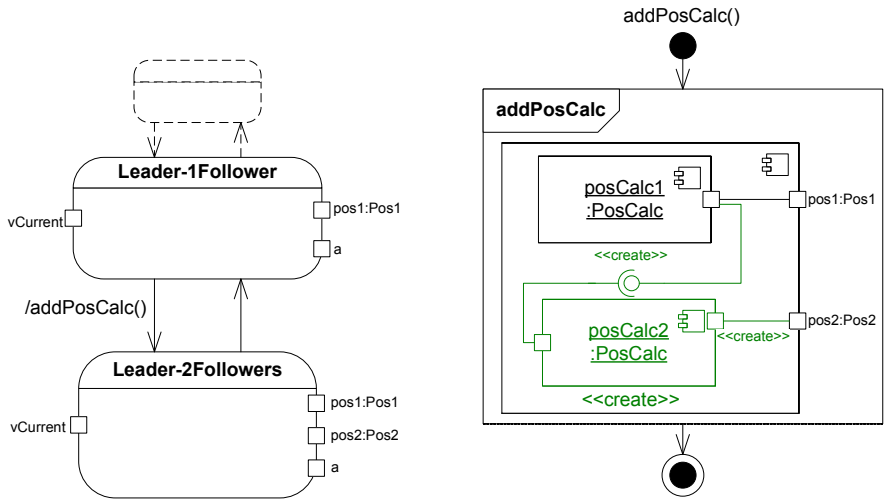
Ein dritter, in [BG05, Bur05] angeführter Nachteil ist, dass alle möglichen Konfigurationen bereits zum Zeitpunkt der Modellierung bekannt sein müssen.

Eine Möglichkeit zur teilweisen Eliminierung dieser Nachteile besteht im Ansatz, die Veränderungen zwischen den einzelnen Konfigurationen zu beschreiben, anstatt wie die hybriden Rekonfigurationscharts jede einzelne aufzuzählen. Um diese Veränderungen modellbasiert zu spezifizieren, können Graphtransformationen eingesetzt werden. In MECHATRONIC UML werden bereits Graphtransformationen in Form von Storydiagrammen (siehe Unterabschnitt 2.1.1) verwendet. Diese operieren allerdings lediglich auf der internen Datenstruktur der Komponenten und werden als Seiteneffekte von Transitionen in RTSCs und in hybriden Rekonfigurationscharts aufgerufen. Somit spezifizieren sie zusammen mit den RTSCs und hybriden Rekonfigurationscharts das interne, reaktive Verhalten einer Komponente [Bur05, Sch06]. Wie im Beispiel von Abschnitt 2.1.1 zu sehen, können Storydiagramme auch zur Veränderung von Konfigurationen verwendet werden. Diese berücksichtigen allerdings nicht die Komponententypdefinition, so können auf Metamodellebene beliebig Komponenteninstanzen ohne Korrespondenz zu einem Typ oder sogar neue Komponententypen erstellt werden. Außerdem ist die Modellierung komplexerer Transformationen unpraktikabel, da solche Storydiagramme auf der abstrakten Syntax der Komponentendiagramme spezifiziert werden müssen. Somit wird eine Kenntnis der abstrakten Syntax vorausgesetzt, und eine hohe Komplexität ist gegeben.

Ein konkreter Ansatz, welcher die Idee von Veränderungen zwischen Konfigurationen aufgreift, besteht in Graphtransformationen auf der eingebetteten Instanzstruktur von Komponenten zur Laufzeit [Krä06b]. Die vorgestellten Strukturtransformationen werden unter Einhaltung von Echtzeitbeschränkungen mittels *Verhaltensanpassungen* modelliert. Verhaltensanpassungen sind Varianten von Storydiagrammen, welche auf der konkreten Syntax der Komponentendiagramme von MECHATRONIC UML spezifiziert werden. Sie ermöglichen die Instanziierung/Zerstörung von eingebetteten Komponenten-/Portinstanzen und Konnektoren von Komponenten, siehe Abb. 2.7(b). Die Modellierung auf der konkreten Syntax des Komponentenmodells reduziert die visuelle Komplexität. Verhaltensanpassungen werden übersetzt in konventionelle Storydiagramme. Mit *Verhaltensdiagrammen*, einer Variante von hybriden Rekonfigurationscharts, werden die Verhaltensanpassungen über Seiteneffekte von Transitionen aufgerufen. Des Weiteren können sie Portinstanzen von Komponenteninstanzen aktivieren bzw. deaktivieren. Dies wird durch Abb. 2.7(a) skizziert.

Der Ansatz unterliegt allerdings einigen Einschränkungen. Zum einen liegt der Fokus der Arbeit auf der Einhaltung des Echtzeitverhaltens. Da eine vorhersagbare Ausführungszeit garantiert werden muss, lassen sich keine Schleifen oder Anwendungsbedingungen[8] in Verhaltensanpassungen modellieren. Zum anderen wird keine Änderung an den fixen Komponententypstrukturen vorgenommen, so dass nach wie vor im laufenden Beispiel die maximale Anzahl der verfügbaren eingebetteten :PosCalc-Komponenteninstanzen und Porttypen die maximale Größe des Konvois bestimmen.

[8] Anwendungsbedingungen resultieren auf Ebene des programmiersprachlichen Codes in Schleifen.



(a) Verhaltensdiagramm (b) Verhaltensanpassung

Abbildung 2.7: Kreierung einer neuen Komponenteninstanz :PosCalc mit Verhaltensdiagramm und Verhaltensanpassung

Genauer gesagt wird die implizite Multiplizität von 1 der eingebetteten Komponenteninstanzen nicht betrachtet. Dies bedeutet dass beliebig viele eingebettete Instanzen erstellt werden dürfen, was nicht durch den umgebenden hierarchischen Komponententyp erlaubt ist. Weiterhin lässt sich durch dieses Problem die in Abb. 2.7(b) dargestellte Verhaltensanpassung nicht über einen Seiteneffekt weiterer Transitionen aufrufen, da explizit die Portinstanzen :Pos1 und :Pos2 gebunden werden. Sofern es einen Porttyp Pos3 geben würde, müsste für eine korrespondierende Strukturtransformation eine neue Verhaltensanpassung modelliert werden. Letztendlich lassen sich Portinstanzen von hierarchischen Komponenteninstanzen lediglich über die Verhaltensdiagramme aktivieren oder deaktivieren, eine dynamische Erzeugung ist nicht möglich. Dies ist eine ernst zunehmende Einschränkung, da alle möglichen Portkonfigurationen einer Komponente weiterhin einzeln modelliert werden müssen, siehe Abb. 2.7(a).

In [BG05] wird ein erster Vorschlag zur Modellierung von *Rekonfigurationsregeln (reconfiguration rules)* gemacht. Diese Rekonfigurationsregeln beschreiben, basierend auf Graphtransformationen, ebenfalls die Veränderungen zwischen einzelnen Konfigurationen, besitzen aber nicht die Einschränkungen der Verhaltensanpassungen aus [Krä06b]. Die Problematik der starren Komponententypstrukturen wird allerdings nicht berücksichtigt. Die Idee der Rekonfigurationsregeln wird in [THHO08] aufgegriffen und weiter entwickelt. Es werden *variable, hierarchische Komponententypen* vorgestellt, welche das Konzept der UML Kompositionsstrukturen [Obj07b] zur Definition variabler Komponententypstrukturen integrieren. Diese erlauben z. B. für eine Komponenteninstanz :RailCab eine beliebige Anzahl eingebetteter :PosCalc-Komponenteninstanzen. Mit *Komponentenstorydiagrammen (Component Story Diagrams)* werden Graphtransformationen zur Instanziierung einer initialen Systemkonfiguration und zur Rekonfiguration einer vor-

handenen Systemkonfiguration basierend auf der konkreten Syntax der Komponentendiagramme modelliert.

Durch die variablen Komponententypstrukturen können Komponentenstorydiagramme typbasiert modelliert werden. Es werden Veränderungen zwischen Konfigurationen beschrieben, welche (prinzipiell unendlich) große Konfigurationsmengen ermöglichen und die Praktikabilität der Modellierung von Strukturanpassungen erhöhen. Des Weiteren müssen nicht während des Modellierungszeitpunktes alle Konfigurationen bekannt sein. Die Verwendung der konkreten Syntax der Komponentendiagramme ermöglicht in Verbindung mit den Kontrollstrukturen[9] und Anwendungsbedingungen der Storydiagramme auch komplexe Strukturtransformationen.

Der Ansatz wird aufgrund der beschriebenen Vorteile im Rahmen dieser Diplomarbeit weiterentwickelt und zum Großteil umgesetzt.

2.4 UML Kompositionsstrukturen

Aufgrund der großen industriellen Akzeptanz der Mehrzweck-Modellierungssprache UML wurde gezeigt, dass auch sie durch Erweiterungen als ADL eingesetzt werden kann [MT00]. Diese Versuche basierten noch auf der Version 1. Mit der UML 2 wurden neue Diagrammarten eingeführt, welche die UML weiter in Richtung der Modellierungsmöglichkeiten von ADLs treiben [ICG+04].

Einer dieser neuen Diagrammarten liegt das Konzept der *Kompositionsstrukturen (Composite Structures)* [Obj07b] zur Modellierung der internen Struktur von Klassen zu Grunde. *Strukturierte Klassen (Structured Classes)* der UML Kompositionsstrukturen sind aus miteinander kommunizierenden *Properties* aufgebaut und besitzen – ähnlich Komponenten – *Ports* und *Schnittstellen (Interfaces)* für die Kommunikation mit der Umwelt. Ein Property spielt eine klassifizierende Rolle für eine Menge von Instanzen zur Laufzeit, d. h. dass Instanzen einer strukturierten Klasse eine Menge eingebetteter Instanzen besitzen können, welche über die Klasse des Property's typisiert sind. Ein Property stellt also eine Referenz auf eine Klasse oder einen Repräsentanten für eine Klasse dar. Properties besitzen Multiplizität, welche die Anzahl möglicher Instanzen des Property's zur Laufzeit bestimmt.

Konnektoren (Connectors) verbinden zwei oder mehr Properties, falls sie kompatible Schnittstellen besitzen. Ein Konnektor stellt eine Kommunikationsbeziehung zwischen den Properties dar, deren Semantik nicht weiter festgelegt ist. Die Kommunikationsbeziehung kann z. B. ein simpler Pointer oder auch eine komplexe Netzwerkverbindung sein. Der Konnektor kann über eine Assoziation zwischen den korrespondierenden Klassen der Properties typisiert sein. Die Konnektoren besitzen Quell- und Zielmultiplizitäten, mit denen ausgedrückt wird wie viele Instanzen der jeweiligen Properties miteinander

[9] Für Strukturtransformationen mit z. B. Schleifen können WCETs ermittelt werden, indem man die maximale Anzahl möglicher Komponenteninstanzen einschränkt [THHO08, TGS06]. Dies würde, geeignete Formalismen vorausgesetzt, auch eine Ausführung zur Laufzeit und unter Echtzeitbedingungen ermöglichen.

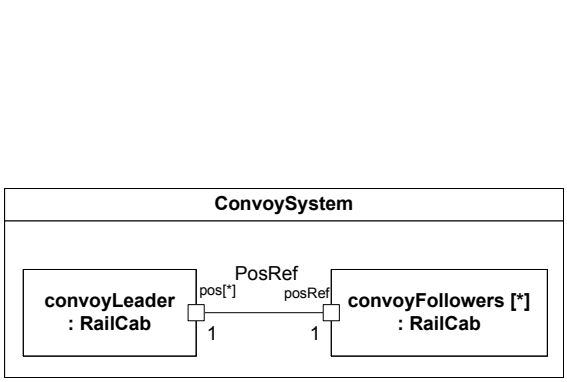
verbunden werden können. Auch die Ports der Properties besitzen Multiplizitäten, um verschiedene Portinstanzen einer Property-Instanz ausdrücken zu können.

Ist die Beziehung zwischen einer strukturierten Klasse und einem eingebetteten Property kompositionaler Natur, so wird das Property *Part* genannt. Instanzen von Parts werden automatisch zerstört, wenn die umgebende Instanz der strukturierten Klasse zerstört wird.

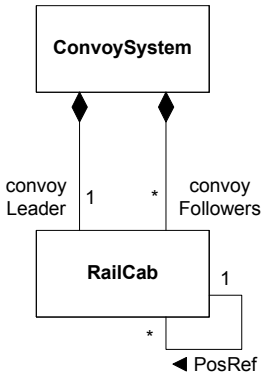
In Abb. 2.8 wird die Klasse RailCab mit einer angebotenen sowie einer benötigten Schnittstelle PosRef dargestellt[10]. Die Schnittstellen sind den Ports posRef bzw. pos zugeordnet. Der Port posRef besitzt Multiplizität 1, während pos die Multiplizität $*$ besitzt, d. h. es kann pro :RailCab-Instanz eine Instanz des Ports posRef und beliebig viele Instanzen des Ports pos geben.

Abbildung 2.8: Klasse RailCab mit Ports und Schnittstellen

Abb. 2.9(a) zeigt ein Kompositionsstrukturdiagramm für die Klasse ConvoySystem, Abb. 2.9(b) visualisiert die gleiche Klassenkonstellation als Klassendiagramm. ConvoySystem enthält zwei Parts der Klasse RailCab, welche die Rollen convoyLeader bzw. convoyFollowers spielen. convoyLeader hat eine implizite Multiplizität von 1, während vom Part convoyFollowers beliebig viele Instanzen existieren können, visualisiert durch die Annotation [*]. Die Quell- und Zielmultiplizität 1 des Konnektors PosRef, welcher die entsprechenden angebotenen und benötigten Schnittstellen verbindet, beschreibt, dass jede Portinstanz pos mit genau einer Portinstanz posRef verbunden wird.



(a) Kompositionsstrukturdiagramm



(b) Klassendiagramm

Abbildung 2.9: Klasse ConvoySystem als konventionelles Klassendiagramm und als Kompositionsstrukturdiagramm

[10]Man beachte, dass RailCab eine beispielhafte Klasse ist und nicht mit der RailCab-Komponente aus dem vorherigen Abschnitt zu verwechseln ist.

In Abb. 2.4 ist eine Instanz der Klasse ConvoySystem zu sehen, welche die convoyLeader-Instanz mit zwei verbundenen Instanzen des Parts convoyFollowers enthält. Die Korrespondenz der eingebetteten Instanzen zu ihren Parts wird durch ein folgendes „/“ zuzüglich des Rollennamens dargestellt. Zur Vervollständigung der Typinformation kann optional die Klassenzugehörigkeit in der bekannten Schreibweise ausgedrückt werden.

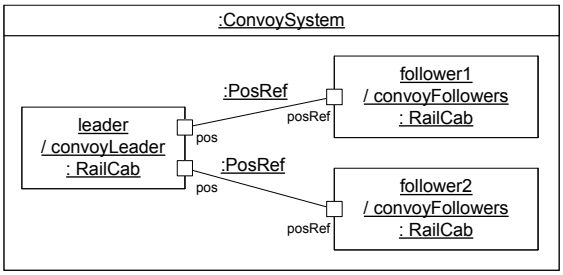


Abbildung 2.10: Instanz der Klasse ConvoySystem

Mit strukturierten Klassen lassen sich speziellere Strukturbeschränkungen ausdrücken als mit den Kompositionen der Klassendiagramme. So gilt die Kommunikationsbeziehung zwischen convoyLeader und convoyFollowers lediglich für Instanzen innerhalb von ConvoySystem-Instanzen, während die Assoziation PosRef aus Abb. 2.9(b) für alle RailCab-Instanzen anwendbar ist.

Die speziell für die Systemmodellierung entwickelte UML-Variante SysML [Obj07a] verwendet u. a. ein weiter entwickeltes Konzept der Composite Structures. Blöcke, eine Ableitung von strukturierten Klassen, können diversifiziertere Varianten von Properties enthalten, die über andere Blöcke typisiert sind. Auch die Modellierungssprache ROOM[11] [SGW94] verfügt über das ähnliche Konzept der *replizierten Referenzen*. Replizierte Referenzen sind, ähnlich wie Properties, Referenzen auf typisierende *Akteur-Klassen (Actor Classes)* mit einem Replikationsfaktor. Dieser Replikationsfaktor gibt die maximale Anzahl der möglichen Instanzen zur Laufzeit an, wie die Multiplizitäten der UML. Auf die gleiche Weise lassen sich Ports replizieren und mit replizierten Referenzen verbinden. In UML-RT [Sel98] wird dieses Actor-Konzept über so genannte *Capsules* in die UML 1.1 übernommen. Capsules werden in UML-RT ebenso wie Ports mit einem Multiplizitätsfaktor versehen.

In dieser Diplomarbeit wird das Konzept der UML Kompositionsstrukturen in die Komponentendiagramme der Mechatronic UML integriert, um variable Komponentenstrukturen zu ermöglichen.

[11] **R**eal-Time **O**bject-**O**riented **M**odeling

3 Verwandte Arbeiten

Die in dieser Arbeit vorgestellten Graphtransformationen beschreiben Veränderungen der eingebetteten Kompositionsstrukturen von Komponenten. Komponenten können aus weiteren Komponenten aufgebaut sein, wodurch sich eine Hierarchie ergibt. Diese Hierarchien müssen bei der Konzeption der Graphtransformationen berücksichtigt werden. Der folgende Abschnitt stellt einige Ansätze für hierarchische Graphtransformationen vor, deren Konzepte interessant für diese Arbeit sind. Der zweite Abschnitt gibt einen Überblick über ADLs, welche explizit Konstrukte zur Modellierung variabler Typstrukturen und/oder zur Modellierung von Strukturanpassungen unterstützen. Einige dieser ADLs beschreiben ebenfalls hierarchische Strukturen und Strukturanpassungen.

3.1 Hierarchische Graphtransformationen

Im Software Engineering begegnet man Komplexität mit Modularisierung und Dekomposition. Ein weiteres wichtiges Thema ist die Datenkapselung (information hiding). Gerade Komponenten, wie sie in der MECHATRONIC UML benutzt werden, verkörpern diese Prinzipien. Wie bereits beschrieben, bilden Komponenten eine hierarchische Struktur.

Ab einer gewissen Größe und Komplexität verlieren auch Graphen und Graphtransformationen ihre Intuitivität und Übersichtlichkeit. Daher gibt es ebenfalls Ansätze zur Modularisierung und Dekomposition von Graphen, welche die Graphelemente des zu Grunde liegenden Graphen zu logischen Einheiten namens *Packages* gruppieren und diese Einheiten in einer Hierarchie organisieren [Bus02, DHP02]. Solche Graphsysteme werden *hierarchische Graphen* genannt, auf eine genaue Definition wird aufgrund der vielen unterschiedlichen Ansätze verzichtet. Während die Ansätze klassischer Graphtransformationen seit über 30 Jahren erforscht werden und allgemein anerkannt sind, sind Graphtransformationen auf hierarchischen Graphen – so genannte *hierarchische Graphtransformationen* – noch ein forschungsnahes Thema. Hierarchische Graphtransformationen werden i. A. abgebildet auf klassische Graphtransformationen für flache Graphen.

Busatto [Bus02] untersucht verschiedene Ansätze für hierarchische Graphen und Graphtransformationen und entwickelt daraus ein allgemeines Modell, welches die untersuchten Arbeiten abdeckt. Des Weiteren werden Aspekte wie Typisierung, Kapselung und Aggregation von hierarchischen Graphen betrachtet. Die Art der Aggregation beschreibt die Zuordnung der Elemente des zu Grunde liegenden Graphen zu den Packages, in manchen Ansätzen können Knoten lediglich zu einem Package zugeordnet werden, in anderen dagegen können sie von mehreren Packages geteilt werden. Entsprechende Kombinationen gibt es auch für die verbindenden Kanten, hier unterscheidet man zwischen Kanten, welche die Grenzen der Packages überschreiten dürfen (*boundary-crossing edges*) und Kanten, die nur Knoten innerhalb eines Packages verbinden dürfen. Der Spezialfall

der *modularen Dekomposition* lässt sich für die Modellierung von komponentenbasierten Softwarearchitekturen anwenden: Packages enthalten Knoten auf kompositionale Weise – d. h. ein Knoten ist genau einem Package zugeordnet – und grenzüberschreitende Kanten sind verboten.

Ein hierarchischer Graph nach [Bus02] besteht aus der Package-Hierarchie, dem zu Grunde liegenden Graph und deren Zuordnung, was mit drei unterschiedlichen flachen Graphen repräsentiert wird. Eine hierarchische Graphtransformation wird überführt in eine getrennte, gleichzeitige Graphtransformation auf allen drei Graphen, nach der Ausführung und Zusammenführung wird die Konsistenz des hierarchischen Graphen überprüft.

Drewes et al. [DHP02] beschreiben Graphtransformationen auf hierarchischen Hypergraphen. Diese besitzen spezielle Hyperkanten (*Frames*), welche wiederum hierarchische Hypergraphen enthalten können. Frames sind eine Analogie zu den Packages. Graphtransformationen auf diesen Graphen binden den Inhalt von Frames in Form von *Variablen*, um die Regeln generisch zu halten[1]. Um Frames austauschen zu können, werden grenzüberschreitende Kanten verboten. Letztendlich werden die hierarchischen Graphtransformationen auf flache Graphtransformationen abgebildet, indem jeder Frame durch seinen Inhalt ersetzt wird. Frames werden in der Arbeit auch als Komponentengraphen bezeichnet. Der Ansatz wird in der ADL COOL [Gru04] (siehe auch nächster Abschnitt) praktisch angewendet, um Strukturtransformationen auf einer hierarchischen, komponentenbasierten Softwarearchitektur durchzuführen.

Taentzer et al. [TGM00] stellen eine zweistufige hierarchische Graphtransformation (*verteilte Graphtransformation*) innerhalb der ADL-Domäne vor. Ein *verteilter Graph* ist zusammengesetzt aus der *Netzwerkebene* und der tiefer liegenden *lokalen Ebene*. Die Netzwerkebene besteht aus *Komponentenknoten* und verbindenden *Kommunikationskanten*. Die lokale Ebene stellt die interne Struktur der Komponentenknoten als separaten *Komponentengraph* dar. Auch die Kommunikationskanten der Netzwerkebene haben Repräsentationen auf der lokalen Ebene. Die Komponentengraphen der lokalen Ebene können zusammen als der zu Grunde liegende Graph und die Komponentenknoten der Netzwerkebene als Package-Knoten betrachtet werden. Verteilte Graphtransformationsregeln sind aufgeteilt in flache Graphtransformationsregeln für die Netzwerk- und die lokale Ebene. Zunächst müssen die Netzwerkregeln ausgeführt werden, da sie die obere Ebene darstellen und z. B. neue Komponentenknoten erstellen können. Anschließend können die lokalen Transformationen für die erhaltenen und neu erstellten Komponentenknoten ausgeführt werden. Für die lokalen Graphen gibt es keine grenzüberschreitenden Kanten.

Engels und Heckel [EH00] beschreiben hierarchische Graphtransformation explizit unter dem Aspekt der Modellierung von Hardware- und Softwarearchitekturen sowie deren Evolution. Miteinander verbundene Hardwarekomponenten bilden die oberste Hierarchieebene. Sie besitzen Softwarekomponenten, welche wiederum aus einzelnen Objekten bestehen. Auf diese Weise bildet sich eine dreistufige hierarchische Struktur. Diese Struktur

[1] Ansonsten müsste in der LHS und RHS der Regel ebenfalls der Inhalt der Frames und somit die LHS des kompletten Graphen über alle Hierarchien hinweg modelliert werden, um die Transformation auszuführen. Dies würde die Vorteile von hierarchischen Transformationen aufheben.

wird auf einen flachen Graphen abgebildet, indem kompositionale Beziehungen mittels *Aggregationskanten* aufgelöst werden. Auf diese Weise kann die hierarchische Struktur mittels konventioneller Graphtransformationen verändert werden. Grenzüberschreitende Kanten sind in diesem Ansatz explizit erlaubt, so können Objekte verschiedener Software- und Hardwarekomponenten miteinander kommunizieren. Dies wird durch die spezielle Art der Abbildung auf flache Graphtransformationen ermöglicht. Neben Objekten werden auch Komponenten auf Typgraphen abgebildet und dementsprechend die Knoten der Graphtransformationen klassifiziert.

Für einen Überblick weiterer Ansätze zu hierarchischen Graphtransformationen sei auf [Bus02] verwiesen. Für diese Arbeit wird ein Graphtransformationsansatz benötigt, welcher verwandt ist mit den vorgestellten hierarchischen Graphtransformationen. Der Fokus liegt hierbei auf Graphtransformationen auf der eingebetteten Kompositionsstruktur von Komponenten der MECHATRONIC UML. Eine solche Strukturanpassung kann wiederum Graphtransformationen auf Komponenteninstanzen in tiefer liegenden Hierarchieebenen anstoßen. Komponenten können somit als Packages betrachtet werden. Eingebettete Elemente einer Komponente dürfen keine grenzüberschreitenden Kanten besitzen, da eine Transformation auf einem solchen eingebetteten Element ungewollte Seiteneffekte haben kann. Außerdem widersprechen grenzüberschreitende Kanten dem Paradigma der Kapselung von Komponenten. Da der zu Grunde liegende Graph eine flache Objektstruktur in Form einer Instanz des Komponentenmetamodells von MECHATRONIC UML ist, sind die komponentenbasierten, hierarchischen Transformationen auf flache Graphtransformationen abzubilden.

3.2 Architekturbeschreibungssprachen

Bradbury et al. [BCDW04] untersuchen verschiedene ADLs im Hinblick auf ihre Fähigkeiten zur Selbstadaptivität. Es werden verschiedene Kriterien vorgestellt, welche die Ausdrucksfähigkeit und Skalierbarkeit einer ADL bzgl. der Modellierung von selbstadaptierenden Systemen beschreiben.

Abb. 3.1 skizziert den Ablauf der vier Basisschritte einer dynamischen Strukturanpassung zur Laufzeit. Die Initiierung der Änderung kann intern oder extern – z. B. durch einen Benutzer – erfolgen. Nach der Initiierung erfolgt die Selektion und danach die tatsächliche Ausführung der durchzuführenden Strukturanpassung. Der zu Grunde liegende Formalismus bestimmt die Implementierung der Ausführung. Mögliche Schleifen in der Rekonfigurationsoperation oder Operationsaufrufe können zur Selektion weiterer Strukturanpassungen führen. Nach der Ausführung der Rekonfiguration ist die resultierende Architektur nach bestimmten Maßstäben zu bewerten. Nach erfolgter Strukturanpassung können weitere Änderungsoperationen durchgeführt werden.

In dieser Arbeit wird eine Transformationssprache vorgestellt, welche zur Entwurfszeit ausgeführt wird. Formalismen für die Initiierung, Selektion und Bewertung werden nicht entwickelt, da sie lediglich zur Laufzeit Relevanz haben. Während der Entwurfszeit sind Kriterien bzgl. der Implementierung relevant, da sie die Möglichkeiten eines Systems zur Modifikation seiner Konfigurationen bestimmen. Ein Kriterium, welches die Aus-

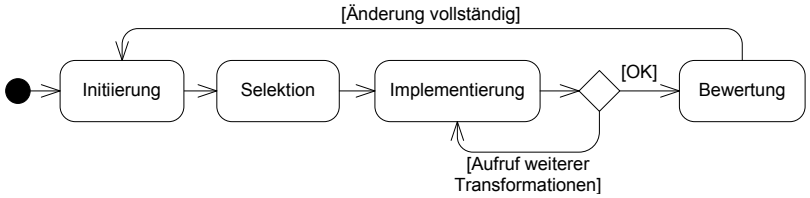


Abbildung 3.1: Transformationsprozess in selbstadaptiven Systemen [BCDW04]

drucksfähigkeit einer ADL betrifft, bezieht sich auf die unterstützten *Änderungsarten*. Die Änderungsarten werden unterteilt in *Basis-Rekonfigurationsoperationen*, *kompositionale Operationen* und *Operationskonstrukte*. Basis-Rekonfigurationsoperationen umfassen die trivialen Möglichkeiten des Hinzufügens und Entfernens von Komponenten und Konnektoren, während kompositionale Operationen das Hinzufügen und Entfernen ganzer Subsysteme oder Gruppen von Architekturelementen unterstützen[2]. Operationskonstrukte beschreiben die aus den Programmiersprachen bekannten Kontrollflussstrukturen Sequenz, Verzweigung und Iteration. Während die meisten der vorgestellten ADLs die Basis- und kompositionalen Operationen unterstützen, stellen nur wenige Sprachen alle Operationkonstrukte zur Verfügung.

Im Folgenden werden exemplarisch einige der von Bradbury et al. untersuchten Sprachen sowie eine weitere Sprache vorgestellt. Diese ADLs haben Ähnlichkeiten zu MECHATRONIC UML oder besitzen für die umzusetzende Transformationssprache interessante Konzepte.

Métayer [Mét96] stellt einen Formalismus vor, welcher die Definition eines Architekturstils und dessen Evolution mittels Graphen und Graphtransformationen beschreibt. Ein Architekturstil wird mittels einer kontextfreien Graphgrammatik beschrieben, wobei Knoten Komponententypen und Kanten Konnektortypen repräsentieren. Den einzelnen Knoten werden Beschreibungen in einer CSP[3]-ähnlichen Notation zugeordnet, um das interne Verhalten einer Komponenteninstanz zu definieren. Durch die Grammatik wird eine initiale Konfiguration generiert. Rekonfigurationen werden in Form von Graphtransformationsregeln in einem zentralen *Koordinator* hinterlegt, welcher die Transformationen für das gesamte System verwaltet. Um sicherzustellen, dass die Ausführung jeder Transformation wieder in einer validen Konfiguration bzgl. des Architekturstils resultiert, wird ein Algorithmus zur Überprüfung jeder Regel des Koordinators eingesetzt, ähnlich der statischen Typüberprüfung für Programmiersprachen. Den Regeln können Anwendungsbedingungen zugeordnet werden, um das Verhalten des Systems besser zu kontrollieren. Der Ansatz verfügt weder über Operationskonstrukte (abgesehen von Bedingungen) noch über eine Hierarchisierung von Komponenten. Durch die Verwendung der kontextfreien Grammatik kann ausgedrückt werden, wie viel Instanzen eines Komponententyps vorhanden sein und welche Komponententypen über welche Konnektortypen verbunden werden dürfen, was einer variablen Komponententypdefinition gleichkommt.

[2] Kompositionale Operationen implizieren in diesem Kontext nicht zwangsläufig hierarchische Operationen im Sinne des letzten Abschnitts.

[3] **C**ommunicating **S**equential **P**rocesses [Hoa78]

CommUnity [WF99] ist eine Modellierungssprache für parallele Programme und ein Beispiel für die Kombination von Graphtransformationen mit programmiersprachlichen Kontrollstrukturen. Architekturstile werden mit typisierten Graphen beschrieben, DPO-Graphtransformationen modellieren Strukturanpassungen. Ein neuerer Ansatz [WLF01] lagert die Graphtransformationen in Kommandos aus, welche in Sequenzen, Verzweigungen und Schleifen eingebettet und aus kompositionalen Kommandos aufgerufen werden können. Eine graphische Repräsentation dieser Konstrukte existiert nicht.

Darwin wurde ursprünglich für die verteilte Entwicklungsumgebung Regis als Konfigurationssprache entwickelt [MDK94]. Mit der Einführung der Prozessalgebra π-Kalkül zur Beschreibung der operativen Semantik und Strukturanpassung [MDEK95] sowie der Kombination mit der Modellierungssprache Alloy zur Modellierung von variablen Typstrukturen [GMK02] hat sich Darwin zu einer deklarativen, universellen ADL weiterentwickelt. Während mit den bereits vorgestellten ADLs keine hierarchischen, wiederverwendbaren Komponenten im Sinne der UML Komponenten oder Kompositionsstrukturen modelliert werden können, stellt Darwin bereits vor der UML-Einführung Komponenten, Ports, angebotene/benötigte Kommunikationsobjekte (ähnlich Schnittstellen), variable Typstrukturen sowie eine ähnliche graphische Notation zur Modellierung bereit. Das Verhalten von Basiskomponententypen wird mit einer Programmiersprache beschrieben, kompositionale Komponententypen können aus weiteren kompositionalen sowie Basiskomponententypen aufgebaut sein. Somit ist Hierarchisierung und Wiederverwendbarkeit gegeben. Ein erheblicher Nachteil in frühen Versionen von Darwin bestand darin, dass die dynamische Strukturanpassung eines Systems lediglich die Instanziierung neuer Komponenten und deren Kommunikationsverbindungen (Konnektoren) unterstützt, eine Restrukturierung vorhandener Verbindungen war nicht möglich. Durch die Kombination mit Alloy wurde auch dieser Mangel behoben. Operationskonstrukte werden allerdings nicht von Darwin unterstützt.

Für eine Vorstellung weiterer ADLs für selbstadaptive Systeme sei auf [BCDW04] verwiesen. Während Bradbury et al. ADLs unter dem Gesichtspunkt der Selbstadaptivität untersuchen, liegt der Fokus anderer Studien [Cle96, MT00] auf Ausdrucksfähigkeit, Toolsupport und Möglichkeiten zur Modellierung von Komponenten/Konnektoren sowie Kompositionalität.

Keiner der bisher vorgestellten Ansätze betrachtet eine Analyse von Architekturen hinsichtlich Sicherheit oder Qualität, wie es z. B. in gewisser Weise bei Mechatronic UML möglich ist. Dieses Thema wird dagegen in BALANCE [Gru04] behandelt, einem Werkzeug zur Verbesserung von Qualitätseigenschaften eines hierarchisch aufgebauten, softwareintensiven technischen Systems in der Entwurfsphase. Konfigurationen solcher Systeme lassen sich hinsichtlich Qualitätsmerkmalen analysieren. Mit der ADL COOL werden Komponententypen modelliert und eine konkrete Konfiguration abgeleitet, als Datenmodell werden hierarchische Hypergraphen verwendet. Mittels Operatoren basierend auf Graphtransformationen auf diesen Graphen [DHP02] (siehe letzter Abschnitt) lässt sich im Anschluss an die Analyse die Konfiguration transformieren, um deren Qualitätseigenschaften verbessern. Eine Modellierung variabler Komponententypstrukturen oder die regelbasierte, initiale Instanziierung von Komponenteninstanzstrukturen ist nicht möglich.

Die in dieser Arbeit umzusetzende Transformationssprache benötigt Merkmale von allen vorgestellten ADLs. Tabelle 3.1 listet die jeweiligen relevanten Merkmale auf. Der Fokus liegt dabei auf der Strukturanpassung zur Entwurfszeit. Um große Konfigurationsmengen zu ermöglichen, müssen Typstrukturen variabel sein. Bei der Typdefinition sowie der Spezifikation der Strukturanpassung sind Hierarchien zu berücksichtigen, welche durch den kompositionalen Aufbau der Komponenten entstehen. Der Formalismus soll möglichst ausdrucksstark sein und neben Basis- und kompositionalen Operationen auch über Operationskonstrukte verfügen. Letztendlich sollen die Transformationen eine geeignete visuelle Darstellung auf Basis der graphischen Repräsentation von Komponenten der MECHATRONIC UML besitzen.

	Variable Typstrukturen	**Hierarchien**	**Operations-konstrukte**	**Geeignete Visualisierung**
Métayer	X	-	(-)	X
COMMUNITY	?	?	X	-
Darwin	X	X	-	X
COOL	-	X	-	X

Tabelle 3.1: Übersicht über die Merkmale der vorgestellten ADLs

4 Anforderungen und Zielabgrenzung

In dieser Diplomarbeit soll eine neue, auf Storydiagrammen basierende Graphtransformationssprache namens *Komponentenstorydiagramme* [THHO08] vorgestellt werden. Diese Sprache soll Strukturanpassungen für MECHATRONIC UML durch die Beschreibung von Veränderungen zwischen einzelnen Konfigurationen ermöglichen, anstelle einer Modellierung von einzelnen Konfigurationen wie in hybriden Rekonfigurationscharts.

Die Beschreibung von Veränderungen zwischen Konfigurationen macht lediglich bei großen Konfigurationsmengen Sinn. Die unflexible Komponententypdefinition im bisherigen Ansatz der MECHATRONIC UML erlaubt nur eingebettete Elemente mit einer impliziten Multiplizität von 1, wodurch auf Instanzebene direkte Abbilder der Komponententypen erstellt werden. Somit ist die Größe der Konfigurationsmengen beschränkt. Eine Variation der Konfigurationen ist lediglich über eine Aktivierung und Deaktivierung einzelner Bereiche möglich, von denen jeder einzelne Bereich explizit modelliert werden muss. Somit ist diese Art der Strukturanpassung aus praktischen Gründen lediglich für kleinere Konfigurationsmengen anwendbar. Es gibt bereits einen Ansatz [Krä06b], welcher Transformationen zwischen Konfigurationen beschreibt. Dieser erkennt allerdings nicht die Notwendigkeit einer Änderung der unflexiblen Komponententypdefinition. Daher lassen sich Konfigurationen generieren, welche nicht typkonform sind. Um große Konfigurationsmengen bei gleichzeitiger Typkonformität zu gewährleisten, ist die bisherige Komponententypdefinition durch die Integration des Konzeptes der UML Kompositionsstrukturen abzuändern.

Der Hauptvorteil von Graphtransformationen ist die Kombination einer formalen, präzisen Beschreibung mit einer visuellen, intuitiven Repräsentation. Um letzteres auch für Komponentenstorydiagramme zu gewährleisten, soll deren Modellierung die konkrete Syntax der Komponentendiagramme unterstützen. Eine Modellierung auf der abstrakten Syntax würde, wie in den Unterabschnitten 2.1.1 und 2.1.2 beispielhaft gezeigt, die Kenntnis des Komponentenmetamodells erfordern. Des Weiteren führt dies zu einer hohen Komplexität sowie zu einer erschwerten Sicherstellung der Typkonformität der generierten Konfigurationen. Um die Modellierung in der konkreten Syntax zu gewährleisten, ist eine geeignete graphische Repräsentation bereitzustellen.

Bei der Modellierung von Komponentenstorydiagrammen muss sichergestellt werden, dass allen Elementen der Transformationssprache, welche strukturelle Elemente der MECHATRONIC UML repräsentieren, Typinformationen des zu Grunde liegenden Komponentenmodells zugeordnet werden. Dadurch kann eine Form der statischen Typisierung auf Ebene der Transformationssprache erzwungen werden.

Bezüglich der graphischen Repräsentation und den Typinformationen müssen außerdem die zu Grunde liegenden Komponententyphierarchien bei der Spezifikation und Ausführung von Komponentenstorydiagrammen berücksichtigt werden. Komponenten-

storydiagramme werden auf Transformationsdiagramme in der abstrakten Syntax des zu Grunde liegenden Komponentenmodells abgebildet. Durch diese Abbildung wird die Semantik der Komponentenstorydiagramme festgelegt, und der vorhandene Mechanismus zur konkreten Ausführung von Transformationsdiagrammen kann wiederverwendet werden.

Die zu entwickelnde Transformationssprache soll eine möglichst hohe Ausdrucksfähigkeit besitzen. Dies soll zum einen durch die Integration von Operationskonstrukten wie Sequenzen, Verzweigungen und Schleifen für die Spezifizierung von Strukturtransformationen, wie in Abschnitt 3.2 beschrieben, erfolgen. Konventionelle Storydiagramme bieten solche Operationskonstrukte in Form von Kontrollstrukturen der Aktivitätsdiagramme. Da Komponentenstorydiagramme auf den klassischen Storydiagrammen basieren, können deren Kontrollstrukturen wiederverwendet werden. Zum anderen können, unter Einschränkungen (siehe Abschnitt 6.5.5), auch die negativen und optionalen Anwendungsbedingungen der Storydiagramme in Komponentenstorydiagrammen verwendet werden.

Zielabgrenzung

Die Ausführung von Komponentenstorydiagrammen wird in dieser Arbeit lediglich zur Entwurfszeit betrachtet. Für eine Ausführung zur Laufzeit wird ein geeigneter Formalismus benötigt, welcher die Initiierung und Selektion der Strukturanpassungen vornimmt, vgl. Abschnitt 3.2. In dieser Arbeit übernimmt der Benutzer die Auswahl und die Ansteuerung der Transformationen.

Der Einsatz von Komponentenstorydiagrammen zur Laufzeit erfordert Einschränkungen, falls Echtzeitcharakteristika berücksichtigt werden müssen. In diesem Fall muss eine Berechnung von WCETs ermöglicht werden. In dieser Arbeit wird der Fokus auf die Modellierung variabler Komponententypen mit unendlich vielen möglichen Konfigurationen und die Modellierung einer möglichst ausdrucksstarken Transformationssprache gelegt, welche z. B. Iterationen unterstützt. WCETs können auch für Komponentenstorydiagramme ermittelt werden, indem die maximalen Multiplizitäten der variablen Komponententypstrukturen auf konkrete Werte fixiert werden [THHO08, TGS06]. Zusätzlich sind weitere Eigenschaften der Sprache bzgl. der Vorhersagbarkeit des Echtzeitverhaltens zu untersuchen, was allerdings Thema zukünftiger Arbeiten ist.

Die Einführung von variablen Komponentenstrukturen erfordert eine Adaption der in Unterabschnitt 2.3.3 vorgestellten Echtzeit-Koordinationsmuster. Die Muster wurden ursprünglich für eine fixe Anzahl teilnehmender Rollen definiert. Die in dieser Arbeit vorgestellten variablen Komponententypstrukturen erlauben dagegen eine beliebige Anzahl an Komponenteninstanzen, welche diese Rollen verfeinern können. Die Variabilität der Komponentenstrukturen beeinflusst des Weiteren die Rollensynchronisation. So müssen zum einen Rollen mit beliebig vielen anderen Rollen kommunizieren und Konnektor-RTSCs die Auslieferung zwischen beliebig vielen Rollen spezifizieren können. Zum anderen können durch Strukturanpassungen neue Komponenteninstanzen und somit neue Rollen zur Laufzeit hinzukommen sowie vorhandene entfernt werden, welche von einem Koordinationsmuster verwaltet werden müssen. Diese Thematiken werden aufgrund ihrer

Komplexität in dieser Arbeit nicht behandelt. Hirsch et al. [HHG08] stellen *dynamische Kollaborationen* vor, welche die o. g. Aspekte mit *Multi-Rollen* bereits umsetzen und eine erste Idee für die Verifikation von sicherheitskritischen Eigenschaften geben. Zukünftige Arbeiten könnten letzteres Thema auf eine adäquatere Weise untersuchen sowie den gesamten Ansatz mit Hilfe der neu gewonnenen Dynamik noch generischer gestalten.

5 Integration variabler Komponententypstrukturen in Mechatronic UML

Graphtransformationen zur Beschreibung von Veränderungen zwischen Konfigurationen sind lediglich bei großen Konfigurationsmengen sinnvoll. Die bisherigen, starren Komponententypstrukturen der MECHATRONIC UML geben keine Multiplizität für eingebettete Elemente vor und implizieren somit kleine Konfigurationsmengen. Im Prinzip werden auf Instanzebene direkte Abbilder der Komponententypen erstellt. Mittels hybriden Rekonfigurationscharts lassen sich nach der Instanziierung bestimmte Bereiche von Komponenteninstanzen aktivieren bzw. deaktivieren.

Um die Menge möglicher Konfigurationen zu vergrößern und um eine typgerechte Modellierung und Ausführung von Komponentenstorydiagrammen zu ermöglichen, wird in diesem Kapitel die vorgeschlagene Integration des Konzepts der UML Kompositionsstrukturen in die MECHATRONIC UML Komponenten beschrieben. Der Hauptgrund für die Flexibilität der UML Kompositionsstrukturen ist die Verwendung von Multiplizitäten für Ports und Parts in der Strukturbeschreibung, welche die Anzahl möglicher Instanzen zur Laufzeit bestimmen. Wie in [THHO08] vorgeschlagen, werden daher ebenfalls Parts und Multiplizitäten in MECHATRONIC UML eingeführt.

5.1 Typmodellierung

Wie in Unterabschnitt 2.3.1 bereits beschrieben, werden die strukturellen Bausteine der MECHATRONIC UML über Komponententypen festgelegt. Komponenten sind abgeschlossene Einheiten, die ihren Zustand und ihre Implementierung kapseln. Ein Port stellt einen Interaktionspunkt zwischen einer Komponente und seiner Umwelt dar. Schnittstellen spezifizieren die Natur der Interaktionen, die über den Port erfolgen können. Eine benötigte Schnittstelle charakterisiert die Anfragen, die eine Komponente an ihre Umwelt stellen kann, während eine angebotene Schnittstelle die Anfragen charakterisiert, welche die Umwelt an eine Komponente stellen kann [Obj07b].

5.1.1 Erweiterte Komponententypen

Die in diesem Abschnitt beschriebenen Eigenschaften von Komponententypen gelten sowohl für einfache als auch hierarchische Komponententypen. Komponenten-, Port- und Schnittstellentypen werden zunächst wie in Unterabschnitt 2.3.1 beschrieben definiert. Komponenten- und Porttypen haben einen Namen, der sie eindeutig in ihrem Kontext

definiert, d. h. zwei Ports unterschiedlicher Komponenten können den gleichen Namen besitzen. Schnittstellen repräsentieren eine Interface-Klasse im zu Grunde liegenden Datenmodell der Komponente und übernehmen den Namen der Interface-Klasse.

Porttypen werden um Multiplizitäten erweitert, welche die Anzahl ihrer möglichen Instanzen zur Laufzeit bestimmt. Ein Porttyp mit der Multiplizität $*$ wird *Multiport* genannt. Porttypen mit Multiplizität 1 stellen die Ports im bisherigen Ansatz von MECHATRONIC UML dar. Multiports werden durch ein angedeutetes, doppeltes Rechteck dargestellt, Abb. 5.1 zeigt diese Notation.

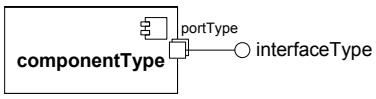


Abbildung 5.1: Multiport

RailCab-Beispiel Abb. 5.2 zeigt die einfachen Komponententypen für das laufende Beispiel. Bis auf eine zusätzliche benötigte Schnittstelle PosRef für die PosCalc-Komponente haben sich gegenüber Unterabschnitt 2.3.1 keine Änderungen ergeben. Die zusätzliche Schnittstelle wird aufgrund einer Veränderung der eingebetteten Struktur des Komponententypen RailCab benötigt, welche im nächsten Unterabschnitt vorgestellt wird.

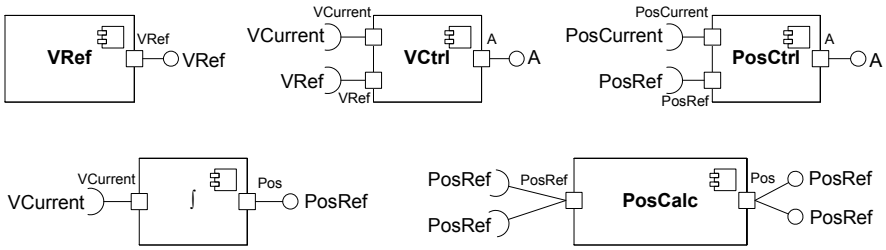


Abbildung 5.2: Einfache Komponententypen nach neuem Ansatz

Multiports können gleichartige Porttypen des bisherigen Ansatzes zusammenfassen, wie z. B. die Ports Pos1 und Pos2 des Komponententyps RailCab aus Unterabschnitt 2.3.1. Auf diese Weise ist der Komponententyp RailCab nicht mehr auf eine fixe Anzahl Ports beschränkt. Abb. 5.3 zeigt die veränderten Porttypen von RailCab ohne Interna. Der Multiport Pos steht nun für eine beliebige Menge gleichartiger Portinstanzen.

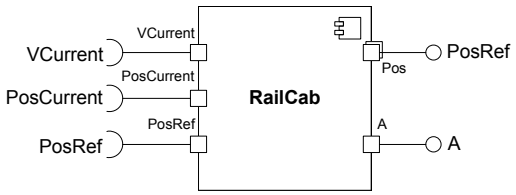


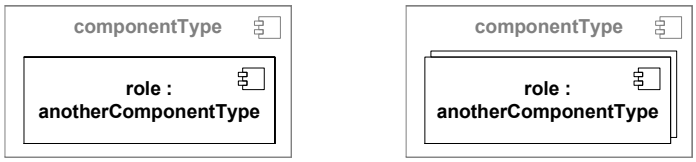
Abbildung 5.3: Komponententyp RailCab mit Multiport Pos

5.1.2 Hierarchische, variable Komponententypen

Hierarchische Komponententypen im alten Ansatz von MECHATRONIC UML besitzen eine mangelnde Flexibilität, da direkt Instanzen anderer Typen eingebettet werden, siehe Abb. 2.5 in Unterabschnitt 2.3.1. Jede dieser Instanzen besitzt implizit eine Multiplizität von 1, somit werden auf Instanzebene Abbilder der Komponententypen erstellt. Variationen sind lediglich möglich, indem ausgewählte Elemente aktiviert oder deaktiviert werden.

Im neuen Ansatz werden die eingebetteten Instanzen von hierarchischen Komponententypen durch Parts ersetzt (vgl. Abschnitt 2.4). Parts sind Rollen oder Repräsentanten für einen Typ und stehen für eine Menge von Instanzen, die eine Instanz des übergeordneten hierarchischen Komponententyps über eine kompositionale Beziehung enthalten kann. Kompositional bedeutet, dass eingebettete Instanzen nur zu der übergeordneten Instanz gehören und in keinem anderen Kontext verwendet werden. Somit impliziert eine Zerstörung der übergeordneten Instanz die Zerstörung aller Part-Instanzen.

Die zentralen Elemente sind *Komponenten-Parts*, ihnen wird ein Rollenname sowie eine Multiplizität zugeordnet. Besitzt ein Komponenten-Part die Multiplizität $*$, so wird von einem *Multipart* gesprochen. Multiparts werden, ähnlich wie Multiports, durch einen zweiten angedeuteten Rahmen visualisiert. Eingebettete Komponenten-Parts werden fett und nicht unterstrichen beschriftet. Abb. 5.4 zeigt die beiden schematischen Darstellungen für Komponenten-Parts mit den Multiplizitäten 0..1 und $*$.



(a) Part (b) Multipart

Abbildung 5.4: Komponenten-Parts

Analog definiert sind *Port-Parts* und *Interface-Parts*, deren Multiplizitäten von den zu Grunde liegenden Typen übernommen werden (siehe vorheriger Unterabschnitt). Sie werden korrespondierend zu den Ports und Schnittstellen des zu Grunde liegenden Komponententypen beim Anlegen eines neuen Komponenten-Parts erstellt. Abb. 5.5 stellt das Schema von Port- und Interface-Parts dar.

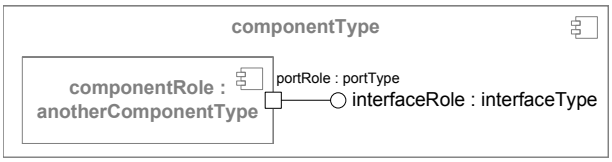


Abbildung 5.5: Port-Part und Interface-Part

Anstelle der bisher verwendeten Konnektorinstanzen werden *Konnektortypen* mit Na-

men eingesetzt. Konnektortypen werden zwar kompositional in hierarchischen Komponententypen eingebettet. Sie stellen allerdings keine Rolle für einen bereits definierten Typen dar, sondern werden direkt im umgebenden Komponententypen definiert. Somit sind sie keine Parts im eigentlichen Sinne, sondern Typen.

Man unterscheidet nach wie vor zwischen Kompositions- und Delegationskonnektoren. Ein *Kompositionskonnektortyp* verbindet ein Interface-Part eines angebotenen Schnittstellentypen und ein Interface-Part eines benötigten Schnittstellentypen, sofern die beiden Schnittstellentypen den gleichen Namen besitzen. Die Interface-Parts müssen dabei im gleichen hierarchischen Komponententyp eingebettet sein. Da Informationen von benötigten zu angebotenen Schnittstellen fließen [Obj07b], ist die Quelle eines Kompositionskonnektortyps das Interface-Part einer benötigten Schnittstelle, während das Ziel das Interface-Part der angebotenen Schnittstelle ist. Abb. 5.6 zeigt das Schema für Kompositionskonnektortypen.

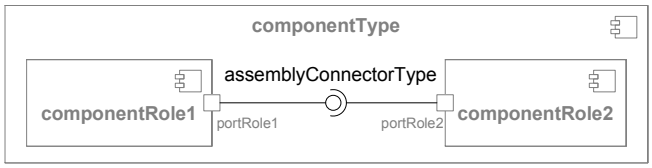


Abbildung 5.6: Kompositionskonnektortyp

Der *Delegationskonnektortyp* verbindet ein eingebettetes Port-Part mit einem Porttyp des übergeordneten Komponententypen, falls die entsprechenden Schnittstellen den gleichen Namen haben und vom gleichen Typ sind. Abb. 5.7 zeigt das Schema. Delegiert der Delegationskonnektor Aufrufe einer angebotenen Schnittstelle, so ist der Port der übergeordneten Komponente die Quelle, da Informationen zu dessen angebotenen Schnittstelle fließen und an ein eingebettetes Port-Part mit angebotener Schnittstelle weiter delegiert werden. Repräsentiert der Delegationskonnektor dagegen die Weiterleitung von Aufrufen einer benötigten Schnittstelle, ist das eingebettete Port-Part die Quelle, da die Aufrufe zur benötigten Schnittstelle der umgebenden Komponente delegiert werden.

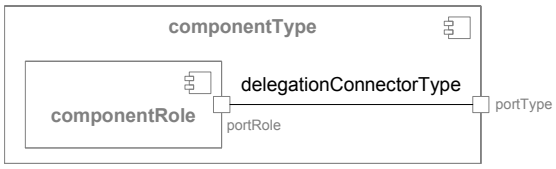
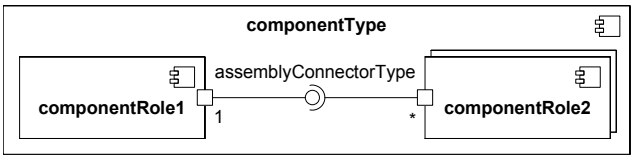


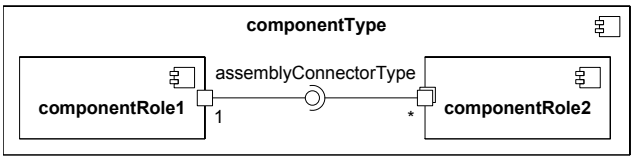
Abbildung 5.7: Delegationskonnektortyp

Konnektortypen besitzen eine Quell- und eine Zielmultiplizität. Diese Multiplizitäten beschreiben das mengenmäßige Verhältnis, in dem Instanzen der verbundenen Elemente zueinander stehen. So gibt die Zielmultiplizität gibt an, wie viel Instanzen auf der Zielseite mit einer Instanz auf der Quellseite verbunden werden können, und vice versa. Die Quell- und Zielmultiplizität von Konnektortypen beträgt als Vorgabe 1. Sofern eines der

verbundenen Elemente die Multiplizität $*$ besitzt (Multipart oder Port-Part eines Multiports), kann die korrespondierende Multiplizität des verbindenden Konnektortyps (d. h. Quell- oder Zielmultiplizität, je nachdem ob das Element mit Multiplizität $*$ Quelle oder Ziel ist) ebenfalls auf $*$ gesetzt werden. Abb. 5.8 stellt dies exemplarisch für die Quellmultiplizität $*$ eines Kompositionskonnektortyp vor. Die Quellelemente mit Multiplizität $*$ können auch in Kombination auftreten, d. h. ein Multipart mit Port-Part eines Multiports. Die Zielmultiplizitäten sowie die Multiplizitäten für Delegationskonnektortypen werden analog definiert.



(a) Multipart



(b) Multiport

Abbildung 5.8: Beispiele für Möglichkeiten der Quellmultiplizität $*$ eines Kompositionskonnektortyps

Auch für die oberste Systemebene wird im neuen Ansatz ein Typgraph spezifiziert, um eine Typdefinition für ein gesamtes System in MECHATRONIC UML zu ermöglichen und somit auch Graphtransformationen auf Systemebene zu typisieren. Die Systemebene stellt in der Komponententyphierarchie die oberste Hierarchieebene dar, in der die eigentliche Verteilung der verschiedenen Komponenteninstanzen vorgenommen wird. Sie wird als spezieller Komponententyp ohne Ports und mit dem Stereotyp ≪system≫ modelliert, vgl. Abb. 5.9. Diese Information wird beim Mapping von Komponentenstorydiagrammen auf Transformationsdiagramme benötigt, um die oberste Systemebene von tiefer liegenden Hierarchieebenen zu unterscheiden.

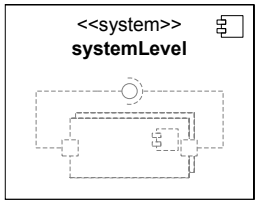


Abbildung 5.9: Systemebene

RailCab-Beispiel Abb. 5.10 zeigt den Komponententyp RailCab als variablen, hierarchischen Komponententyp. Zwecks Vollständigkeit werden die Namen aller eingebetteten Port-Parts angezeigt, welche die Namen der korrespondierenden Porttypen (vgl. Abb. 5.2) mit einem klein geschriebenen Anfangsbuchstaben übernehmen. Anstelle der Komponenteninstanzen des alten RailCab-Komponententypen aus Unterabschnitt 2.3.1 treten entsprechend benannte Komponenten-Parts. Die einzelnen, durchnummerierten Komponenteninstanzen :PosCalc werden durch das Multipart posCalc:PosCalc ersetzt, welches für eine beliebige Anzahl von :PosCalc-Komponenteninstanzen zur Laufzeit steht. Dadurch wird die maximale Größe eines Konvois nicht mehr durch die Anzahl eingebetteter :PosCalc-Instanzen bestimmt.

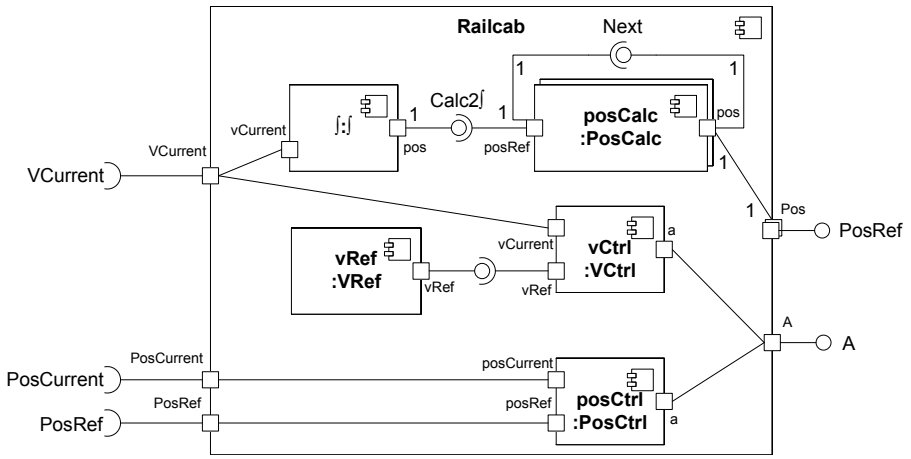


Abbildung 5.10: Hierarchischer, variabler Komponententyp RailCab

Die Konnektorinstanzen werden durch korrespondierende Konnektortypen ersetzt. Verbinden die Konnektortypen Elemente mit einer Multiplizität von 1, so wird sowohl für die Quell- als auch für die Zielmultiplizität der Konnektortypen implizit 1 angenommen, da sie zur Laufzeit lediglich eine Quell-Portinstanz mit einer Ziel-Portinstanz verbinden können. In diesem Fall werden die Multiplizitäten nicht dargestellt. Für Konnektortypen, welche Elemente mit einer Multiplizität von $*$ verbinden, muss hingegen die Quell- und Zielmultiplizität angegeben werden, um die Verbindungsmuster zu konkretisieren. In RailCab besitzt jeder dieser Konnektortypen eine Quell- und Zielmultiplizität von 1. Dies bedeutet für Instanzen des Konnektortyps Calc2$\int$, dass sie genau eine Komponenteninstanz des Parts posCalc mit der Komponenteninstanz von $\int$ verbinden. Instanzen des Kompositionskonnektortyps Next verbinden jeweils zwei Komponenteninstanzen des Parts :PosCalc untereinander, und jede dieser Instanzen wird mit exakt einer Portinstanz des Multiports Pos verbunden.

Vergleicht man den an dieser Stelle vorgestellten variablen Komponententypen RailCab mit dem nach dem alten Prinzip definierten Komponententypen RailCab in Abb. 2.5 aus Unterabschnitt 2.3.1, so ist die Steigerung der Wiederverwendbarkeit von Komponententypen zu erkennen. Anstatt einen Komponententypen mit fest vorgegebenen Anzahlen von :PosCalc- und Portinstanzen für eine bestimmte Konvoizusammenstellung

festzulegen, wird ein Komponententyp für beliebige Anzahlen von :PosCalc- und :Pos-Instanzen definiert, welche alle Konvoikonstellationen abdecken. Gleichzeitig reduziert sich die Komplexität der Komponententypmodellierung. So muss man für einen Konvoi aus mehreren nachfolgenden RailCabs nicht entsprechend viele eingebettete Komponenteninstanzen und Porttypen für RailCab spezifizieren, sondern definiert ein Komponenten-Part und einen Porttyp mit entsprechenden Multiplizitäten.

Abb. 5.11 stellt die oberste Systemebene und somit das Konvoisystem dar. Die Systemebene ist als hierarchischer Komponententyp ConvoySystem mit dem Stereotyp ≪system≫ modelliert. ConvoySystem ist ein Beispiel dafür, dass unterschiedliche Parts desselben Typs verschiedene Rollen einnehmen können. Das System besteht aus den beiden Rollen convoyLeader und convoyFollowers, welche beide über den Komponententyp RailCab klassifiziert sind. Im gesamten System gibt es ein Vorkommen der Rolle convoyLeader und eine beliebige Anzahl an Instanzen der Rolle convoyFollowers. Der convoyLeader bekommt die aktuelle Geschwindigkeit vCurrent als Eingabe und beantwortet über den Kompositionskonnektortyp PosRef Anfragen aller convoyFollowers nach den Referenzpositionen. Wie in Abschnitt 2.4 bereits motiviert, beträgt sowohl Quell- als auch Zielmultiplizität von PosRef 1. Somit wird jede der beliebig vielen Instanzen des Port-Parts pos mit einer Instanz des Port-Parts posRef verbunden, von der es pro convoyFollowers-Komponenteninstanz genau eine gibt. Die Port- und Interface-Parts, welche in der jeweiligen Rolle nicht benötigt sind, werden in der Abbildung zwecks einer besseren Übersicht nicht dargestellt.

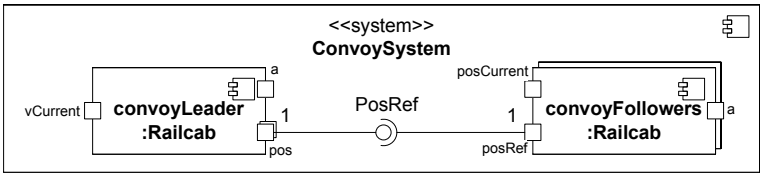


Abbildung 5.11: Systemebene ConvoySystem

5.1.3 Komponenteninstanzstrukturen

Nachdem in den vorangegangenen Unterabschnitten die Erweiterungen auf Typebene eingeführt wurden, stellt dieser Unterabschnitt die Auswirkungen dieser Erweiterungen auf konkrete Komponenteninstanzstrukturen bzw. Konfigurationen vor. Anstatt direkt einen Typ zu instanziieren, werden i. A. Instanzen von Parts erstellt, und somit wird die Zugehörigkeit von Instanzen zu ihrem Typ indirekt hergestellt. Ein Komponentenmodell lässt sich also in Typ-, Part- und Instanzelemente einteilen. Konnektortypen werden bei dieser Strukturierung zu den Partelementen gezählt, da sie wie Parts in hierarchische Komponententypen eingebettet werden.

Die Korrespondenz von Instanzen zu ihren Parts wird, wie in Abschnitt 2.4 bereits beschrieben, durch ein „/“ zwischen dem Instanz- und dem Rollennamen angedeutet. Optional kann zusätzlich die Typzugehörigkeit annotiert werden, welche bereits implizit durch die Rollenzugehörigkeit gegeben ist. Abb. 5.12 zeigt das Schema der Namensgebung

anhand einer Komponenteninstanz. Diese Namensgebung wird für Port- und Schnittstelleninstanzen in benötigten Fällen übernommen, vgl. Abb. 5.13.

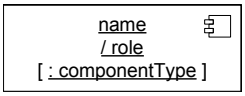


Abbildung 5.12: Namensgebung von Komponenteninstanzen

Konnektorinstanzen stellen ihre Typzugehörigkeit mit dem Typnamen dar, welcher in der bekannten Schreibweise durch einen Doppelpunkt vom Instanznamen getrennt wird. Dies wird ebenfalls in Abb. 5.13 am laufenden Beispiel dargestellt.

Eine Konfiguration, wie in Abb. 5.13 teilweise gezeigt, ist ein Graph, welcher anstatt Knoten und Kanten Komponenten-, Port- und Konnektorinstanzen enthält. Dieser Graph sei *Komponenteninstanzgraph* genannt. Ein solcher Komponenteninstanzgraph lässt sich, ähnlich einem Graphen für objektorientierte Instanzstrukturen, auf eine Typdefinition abbilden. Ein Komponenteninstanzgraph ist *gültig*, wenn für ihn folgende Eigenschaften gelten:

1. Ein Komponenteninstanzgraph ist *wohlgeformt*, wenn
   - jede Konnektorinstanz zwei Portinstanzen verbindet, und
   - jede Portinstanz an einer Komponenteninstanz angebracht ist.
2. Ein Komponenteninstanzgraph ist *typkonform*, wenn
   - jede Instanz entsprechend der Komponententyphierarchie eindeutig einem Typ oder einem Part zugeordnet ist, und
   - von jedem Typ oder Part so viel Instanzen existieren und über Konnektorinstanzen verbunden sind, wie es die Multiplizitäten erlauben.

Bei der Zuordnung von Instanzen zu Typen oder Parts dürfen letztere lediglich aus dem übergeordneten Komponententyp der Instanz gewählt werden, in den die zuzuordnende Instanz eingebettet ist. Somit gibt die Hierarchie der Komponententypen vor, wie die jeweiligen Komponenteninstanzen geschachtelt werden dürfen. Obwohl die meisten dieser Eigenschaften selbstverständlich erscheinen, muss bei der Konzeptionierung der Komponentenstorydiagramme darauf geachtet werden, dass bei deren Ausführung diese Eigenschaften erhalten bleiben.

RailCab-Beispiel Abb. 5.13 zeigt eine Konfiguration für einen Konvoi mit zwei nachfolgenden RailCabs, die Möglichkeit weiterer Komponenten-, Port- und Konnektorinstanzen sind angedeutet. Die umgebende Instanz der Systemebene ConvoySystem ist nicht abgebildet.

Offensichtlich ist diese Konfiguration ein gültiger Komponenteninstanzgraph. Jede Konnektorinstanz verbindet zwei Portinstanzen, und alle Portinstanzen sind an einer

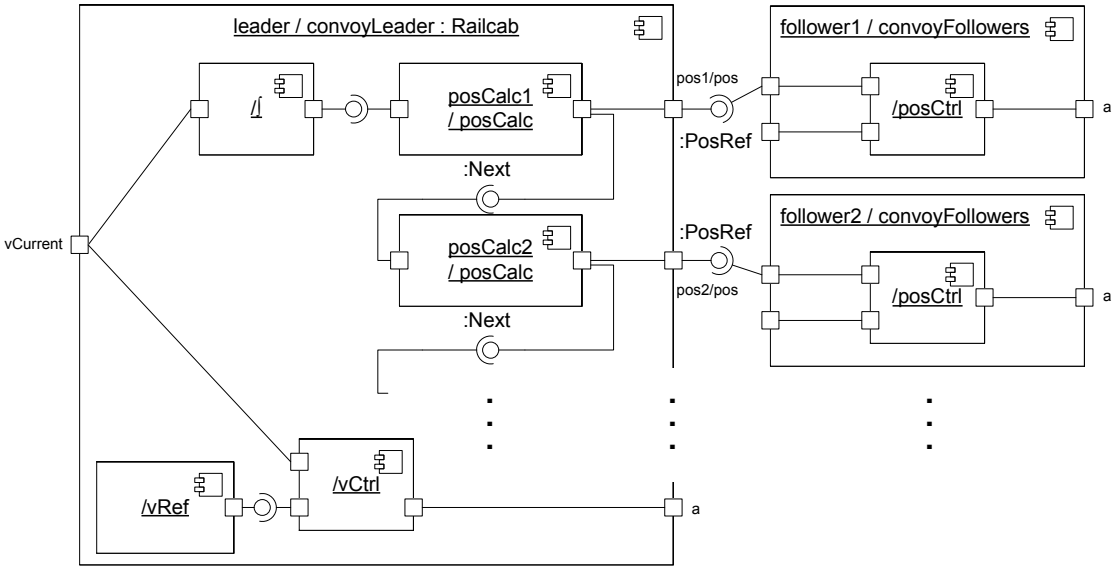


Abbildung 5.13: Konfiguration als Komponenteninstanzgraph

Komponenteninstanz angebracht. Somit ist der Graph wohlgeformt. Des Weiteren ist jede in :RailCab eingebettete Instanz einem Part oder einem Typ zugeordnet[1], welcher eingebettet ist in den Komponententyp RailCab. Konnektorinstanzen sind direkt einem Typ zugeordnet (in der Abb. ist dies lediglich für :Next und :PosRef erkennbar), alle anderen Instanzen sind über ihre jeweiligen Parts typisiert. Falls von einem Part mehrere Instanzen existieren (/posCalc, /pos und /convoyFollowers), wurden für die Instanzen zwecks einer eindeutigen Identifikation nummerierte Namen vergeben. Da die entsprechenden Klassifizierer Multi-Parts oder Multiports sind und des Weiteren die Konnektormultiplizitäten eingehalten werden, ist der Graph ebenso konform bzgl. der Multiplizitäten, und somit insgesamt gültig.

Wie am Beispiel zu erkennen ist, gibt es dank der beschriebenen Erweiterungen im Gegensatz zum Komponententyp RailCab aus Unterabschnitt 2.3.1 keine Elemente mehr auf Typebene, welche die maximale Größe eines Konvois bestimmen. Für Porttypen fällt dank der Zuordnung von Multiplizitäten die Beschränkung weg, dass exakt eine Instanz von ihnen gebildet werden kann. Mit Komponenten-Parts werden typisierte Rollen zur Verfügung gestellt, welche eine beliebige Anzahl eingebetteter Komponenteninstanzen zur Laufzeit repräsentieren. Konnektortypen setzen über Quell- und Zielmultiplizitäten diese verschiedenen Elemente zueinander in Relation.

[1] Die Namen einiger Port-Parts oder -typen sowie einiger Konnektortypen wurden aus Gründen einer besseren Übersicht ausgelassen.

5.2 Erweiterung des Komponentenmetamodells

Nachdem im letzten Abschnitt die Erweiterungen der Komponententypstrukturen informal vorgestellt wurden, werden sie in diesem Abschnitt durch ihre Repräsentation im Metamodell von MECHATRONIC UML formalisiert. Dabei werden einige technische Details zwecks einer besseren Übersicht ausgelassen.

Wie im letzen Unterabschnitt erwähnt, gibt es durch die Erweiterung der Komponententypmodellierung mit MECHATRONIC UML eine Einteilung in Typen, Parts und Instanzen. Daher lässt sich das zu Grunde liegende Metamodell ebenso strukturieren, was mit den UML Paketen Types, Parts und Instances vorgenommen wird (Abb. 5.14). Das Paket Parts enthält neben den Parts auch Konnektortypen und somit alle Elemente, die sich in einen hierarchischen Komponententyp einbetten lassen. Offensichtlich greift das Types-Paket auf das Parts-Paket zu und vice versa, da Typen aus Parts aufgebaut sind und Parts einen Typ repräsentieren. Das Instances-Paket greift ebenso auf das Parts-Paket zu, da Instanzen über Parts und Konnektortypen klassifiziert sind. In einem Fall greifen auch Instanzen direkt auf die Typen zu (vgl. Unterabschnitt 5.2.2), daher gibt es des Weiteren eine direkte Abhängigkeit von Instances nach Types.

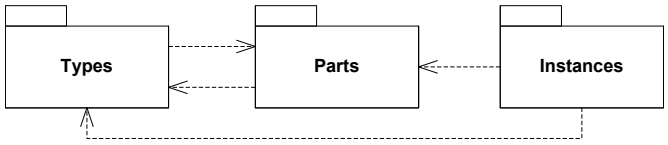


Abbildung 5.14: Erweitertes Komponentenmetamodell – Strukturierung

5.2.1 Typ- und Partklassen

Abbildung 5.15 stellt die Klassen für die Repräsentation der Typen vor und führt die Klassen ein, mit denen sich die eingebetteten Komponenteninstanzstrukturen in hierarchischen Komponententypen modellieren lassen. Klassen und Assoziationen, welche bereits im Metamodell vorhanden waren, werden grau dargestellt, um die Differenz zum ursprünglichen Metamodell hervorzuheben.

Die Typklassen (Modell Types in Abb. 5.14) befinden sich auf der linken Seite in Abb. 5.15. Komponententypen werden durch die Klasse Component, Porttypen durch Port und Schnittstellentypen durch Interface repräsentiert. Eine Komponente kann beliebig viele Ports besitzen, und diese wiederum beliebig viele Schnittstellen. Die Unterscheidung zwischen angebotenen und benötigten Schnittstellen wird mit Hilfe der Spezialisierung durch die Klassen ProvidedInterface bzw. RequiredInterface vorgenommen. Eine Schnittstelle besitzt eine unidirektionale Assoziation zur Klasse UMLClass, welche die Interface-Klasse in der internen Datenstruktur der korrespondierenden Komponente definiert (vgl. Unterabschnitt 5.1.1). UMLClass wird als unveränderliche Referenz aus dem Paket FujabaCore importiert, welches das Metamodell des FUJABA-Kerns repräsentiert (siehe Architekturbeschreibung dieser Arbeit in Unterabschnitt 7.1.2).

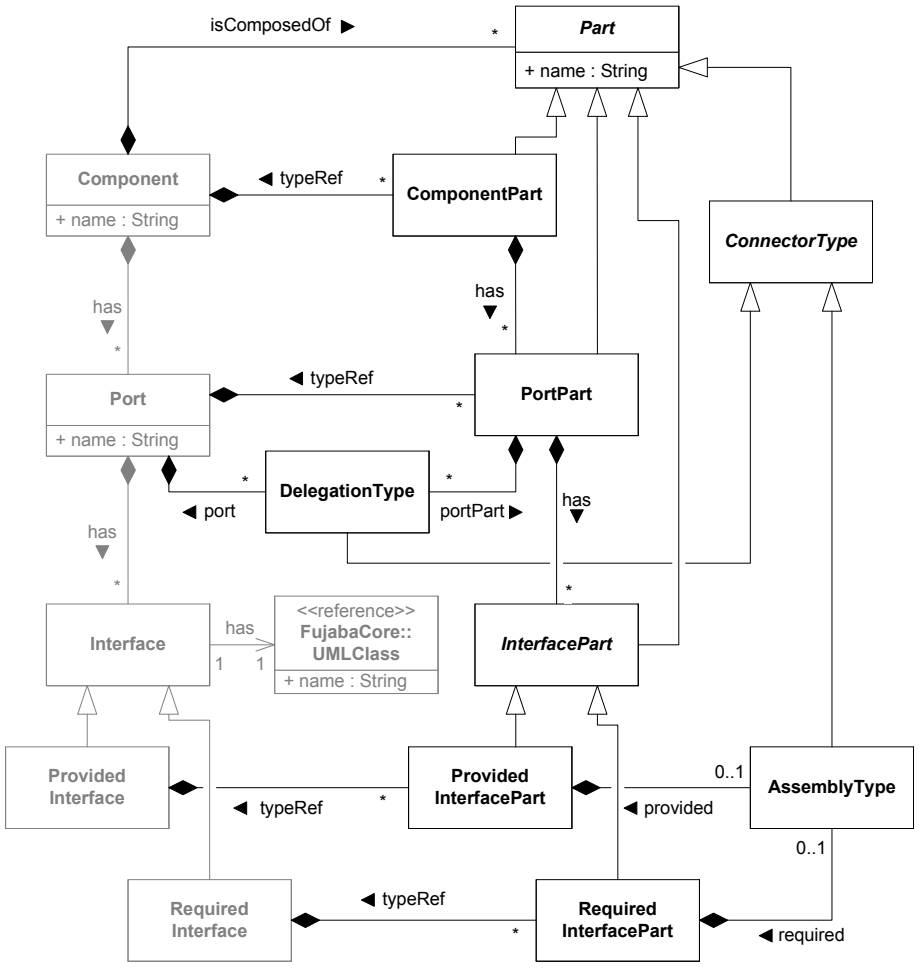


Abbildung 5.15: Erweitertes Komponentenmetamodell – Typ- und Partklassen

Die abstrakte Klasse Part beschreibt alle Elemente, welche in hierarchische Komponententypen eingebettet werden können, dargestellt auf der rechten Seite von Abb. 5.15 (Modell Parts in Abb. 5.14). Die Einbettung in einen Komponententyp wird über die Assoziation isComposedOf zwischen Component und Part ausgedrückt. Die Assoziationen has zwischen Component, Port und Interface haben ebenso eine Entsprechung zwischen den Partklassen ComponentPart, PortPart und InterfacePart, so dass ein Komponenten-Part beliebig viele Port-Parts und diese wiederum beliebig viele Interface-Parts besitzen können. Des Weiteren besitzen die Partklassen eine $*$:1-Assoziation typeRef auf ihre jeweilige Typklasse (z. B. zwischen ComponentPart und Component) zur Typisierung der jeweiligen Parts bzw. Rollen.

Konnektortypen werden durch die Subklassen von ConnectorType realisiert. Auch ConnectorType erbt von Part, obwohl Konnektortypen keine Parts im eigentlichen Sinne darstellen, da sie keine Rollen für einen Typ ausdrücken (siehe Unterabschnitt 5.1.2). Durch die Ableitung von Part wird allerdings ausgedrückt, dass ein Konnektortyp ebenso wie Parts in Komponententypen eingebettet werden kann. Die Differenzierung der Konnektortypen in Kompositions- und Delegationskonnektortypen wird auf Metamodellebene mit den Klassen AssemblyType bzw. DelegationType durchgeführt. Kompositionskonnektortypen verbinden ein ProvidedInterfacePart und ein RequiredInterfacePart, d. h. sie verbinden Interface-Parts einer angebotenen und einer benötigten Schnittstelle. Delegationskonnektortypen verbinden ein Objekt der Klasse PortPart und ein Objekt der Klasse Port, was bedeutet, dass sie ein eingebettetes Port-Part mit dem Port des umgebenden Komponententypen verbinden.

Abb. 5.16 zeigt, wie das Komponentenmetamodell um Multiplizitäten erweitert wird. Für die Definition von Multiplizitäten wird die bereits vorhandene Klasse UMLCardinality benutzt. UMLCardinality besitzt die Integer-Attribute lowerBound zur Definition der unteren und upperBound für die Definition der oberen Multiplizitätsgrenze. Die Zuordnung der Multiplizitäten zu Komponenten-Parts und Porttypen erfolgt über unidirektionale 1:1-Assoziationen multiplicity von ComponentPart und Port zu UMLCardinality. Die Quell- und Zielmultiplizitäten von Konnektortypen werden über unidirektionale Assoziationen sourceMultiplicity bzw. targetMultiplicity von ConnectorType zu UMLCardinality zugeordnet. Instanzen von UMLCardinality werden vom FUJABA-Framework über ein Flyweight-Entwurfsmuster [GHJV95] erstellt, so dass nicht für jedes Typ- bzw. Partobjekt ein eigenes Objekt zur Beschreibung der gleichen Multiplizität notwendig ist.

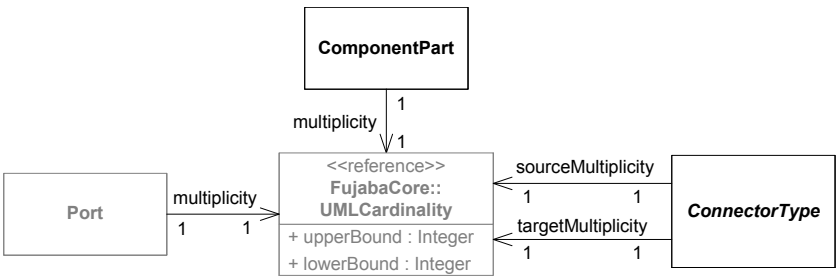


Abbildung 5.16: Erweitertes Komponentenmetamodell – Multiplizitäten

5.2.2 Part- und Instanzklassen

Das Klassendiagramm in Abb. 5.17 visualisiert die Verbindung der neuen Part- und Konnektortypklassen zu den bereits vorhandenen Instanzklassen. Partklassen, siehe Paket Parts in Abb. 5.14, befinden sich auf der linken, Instanzklassen (Paket Instances in Abb. 5.14) auf der rechten Seite der Abbildung. Bereits vorhandene Klassen und Assoziationen werden wie im vorherigen Unterabschnitt grau dargestellt.

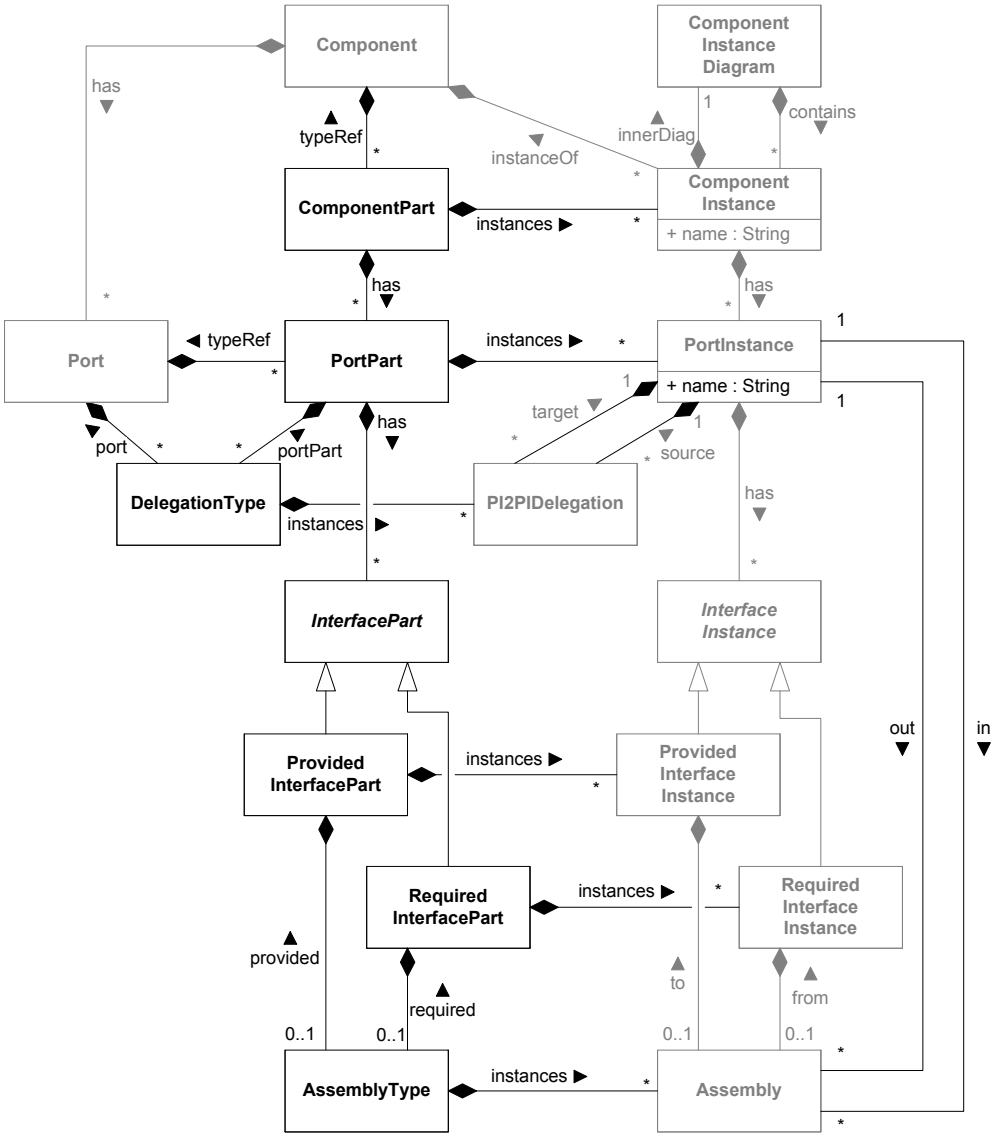


Abbildung 5.17: Erweitertes Komponentenmetamodell – Part- und Instanzklassen

Zur Modellierung der Instanzebene werden die Klassen ComponentInstance, PortInstance, InterfaceInstance sowie deren Subklassen aus dem ursprünglichen Metamodell

verwendet. Wie bei den Typ- und die Partklassen findet sich auch auf der Instanzebene die Struktur wieder, dass Komponenteninstanzen beliebig viele Portinstanzen und diese wiederum beliebig viele Schnittstelleninstanzen besitzen können.

Alle Instanzklassen sind über die Assoziationen instances zu den korrespondierenden Partklassen typisiert, eine direkte Assoziation zu den Typklassen ist nicht mehr notwendig. Eine Ausnahme diesbezüglich stellt die Assoziation instanceOf zwischen ComponentInstance und Component dar, welche die Instanziierung der Systemebene ermöglichen soll. Da die Systemebene die oberste aller Hierarchieebenen darstellt, ist sie nicht als Part in einen übergeordneten Komponententyp eingebettet und muss direkt über ihren Komponententyp instanziiert werden. In diesem Fall besteht also ein direkter Zugriff des Instanzpakets auf das Typpaket, vgl. Abb. 5.14.

Ein Objekt der Klasse ComponentInstance kann über ein ComponentInstanceDiagram weitere ComponentInstance-Objekte enthalten, wodurch die kompositionalen Beziehungen zwischen verschiedenen Komponenteninstanzen modelliert werden.

Im alten Ansatz reichte zur Identifikation von Portinstanzen der Name des zu Grunde liegenden Porttypen aus, da für jeden Porttyp lediglich eine Portinstanz pro Komponenteninstanz existieren konnte. Die Einführung von Multiports ändert dies. Zwecks einer besseren Unterscheidung von Portinstanzen, Port-Parts und Porttypen wird daher die Klasse PortInstance um das Attribut name ergänzt.

Die Klasse PI2PIDelegation repräsentiert eine Delegationskonnektorinstanz und verbindet eine Quell- und eine Ziel-Portinstanz. Die entsprechenden Assoziationen source und target wurden in Kompositionen umgewandelt, um der Definition der Wohlgeformtheit eines Komponenteninstanzgraphen in Unterabschnitt 5.1.3 gerecht zu werden. Je nachdem, ob der korrespondierende Delegationskonnektortyp für die Delegierung von Aufrufen einer angebotenen oder einer benötigten Schnittstelle steht, ist die entsprechende Portinstanz Quelle oder Ziel (siehe Unterabschnitt 5.1.2).

Ein Assembly stellt eine Kompositionskonnektorinstanz dar und verbindet nach wie vor zwei Schnittstelleninstanzen unterschiedlicher Art. Die Quelle eines Kompositionskonnektors ist immer die Instanz einer benötigten Schnittstelle (Assoziation from), das Ziel immer die Instanz einer angebotenen Schnittstelle (Assoziation to), siehe auch Unterabschnitt 5.1.2. Zusätzlich wurden die redundanten Assoziationen in und out zwischen PortInstance und Assembly hinzugefügt, da diese Information aus technischen Gründen bei der Abbildung von Komponentenstorydiagrammen auf Transformationsdiagramme benötigt wird.

6 Instanziierung und Rekonfiguration von Komponenteninstanzstrukturen

Im letzten Kapitel wurde die Integration der Konzepte der UML Kompositionsstrukturen in das Komponentenmetamodell von MECHATRONIC UML vorgestellt. Die resultierenden variablen Komponententypstrukturen erlauben nun beliebig große Konfigurationsmengen, was die Beschreibung von Transformationen zwischen den einzelnen Konfigurationen anstatt der Auflistung jeder einzelnen Konfiguration ermöglicht.

Ausgehend von diesen Möglichkeiten stellt sich die Frage, wie einzelne eingebettete Komponenteninstanzstrukturen und komplette Systemkonfigurationen initial erstellt und anschließend rekonfiguriert werden können. Dieses Kapitel beantwortet diese Frage mit der Vorstellung von Komponentenstorydiagrammen, einer neuen Graphtransformationssprache basierend auf Storydiagrammen.

Abschnitt 6.1 führt zunächst die Syntax und Semantik von Komponentenstorydiagrammen ein. Bzgl. der operationalen Semantik der Komponentenstorydiagramme wird versucht, die Konzepte der konventionellen Storydiagramme wie z. B. die Vermeidung von dangling edges und die Ersetzung von über :1-Assoziationen typisierte Links zu adaptieren. Diese Konzepte werden in den Abschnitten 6.2 und 6.3 vorgestellt. Abschnitt 6.4 formalisiert die Syntax der Komponentenstorydiagramme über das zu Grunde liegende Metamodell. Komponentenstorydiagramme werden übersetzt in Transformationsdiagramme auf dem Komponentenmetamodell von MECHATRONIC UML. Der letzte Abschnitt dieses Kapitels stellt dieses Mapping vor.

6.1 Komponentenstorydiagramme

Komponentenstorydiagramme [THHO08] basieren auf konventionellen Story- und Transformationsdiagrammen (siehe Unterabschnitt 2.1.1 bzw. 2.1.2), daher finden sich viele Gemeinsamkeiten in den Grundideen sowie in der graphischen Syntax. Wie Storydiagramme werden auch Komponentenstorydiagramme in einzelne Aktivitäten unterteilt, welche als Kontrollstrukturen dienen können oder reinen programmiersprachlichen Code bzw. *Komponentenstorypatterns* enthalten. In letzteren findet die Modellierung der Graphtransformationen basierend auf der konkreten Syntax der Komponentendiagramme von MECHATRONIC UML statt.

Komponentenstorydiagramme sind einem hierarchischen, variablen Komponententypen zugeordnet. Komponentenstorypatterns werden basierend auf der eingebetteten Struktur des Komponententypen modelliert, d. h. die Elemente des Komponentenstorypatterns werden über die eingebetteten Parts und Konnektortypen klassifiziert. Komponentenstorydiagramme lassen sich in *Konstruktoren* und *Rekonfigurationsregeln* un-

terteilen. Ein Konstruktor erstellt die initiale eingebettete Struktur einer neuen Komponenteninstanz, während eine Rekonfigurationsregel die eingebettete Struktur einer bereits existierenden Komponenteninstanz verändert. Konstruktoren und Rekonfigurationsregeln können aus Komponentenstorydiagrammen von hierarchisch höher liegenden Komponententypen aufgerufen werden, somit spannt die Komponententyphierarchie auch indirekt eine Hierarchie über ihre Transformationen auf.

Im Folgenden führt Unterabschnitt 6.1.1 zunächst die neuen Syntax-Elemente für Storypatterns auf Komponenteninstanzstrukturen ein, anschließend werden in Unterabschnitt 6.1.2 die von den konventionellen Story- und Transformationsdiagrammen bekannten Kontrollstrukturen der Komponentenstorydiagramme vorgestellt. Unterabschnitt 6.1.3 verdeutlicht die hierarchische Beziehung zwischen den einzelnen Transformationen.

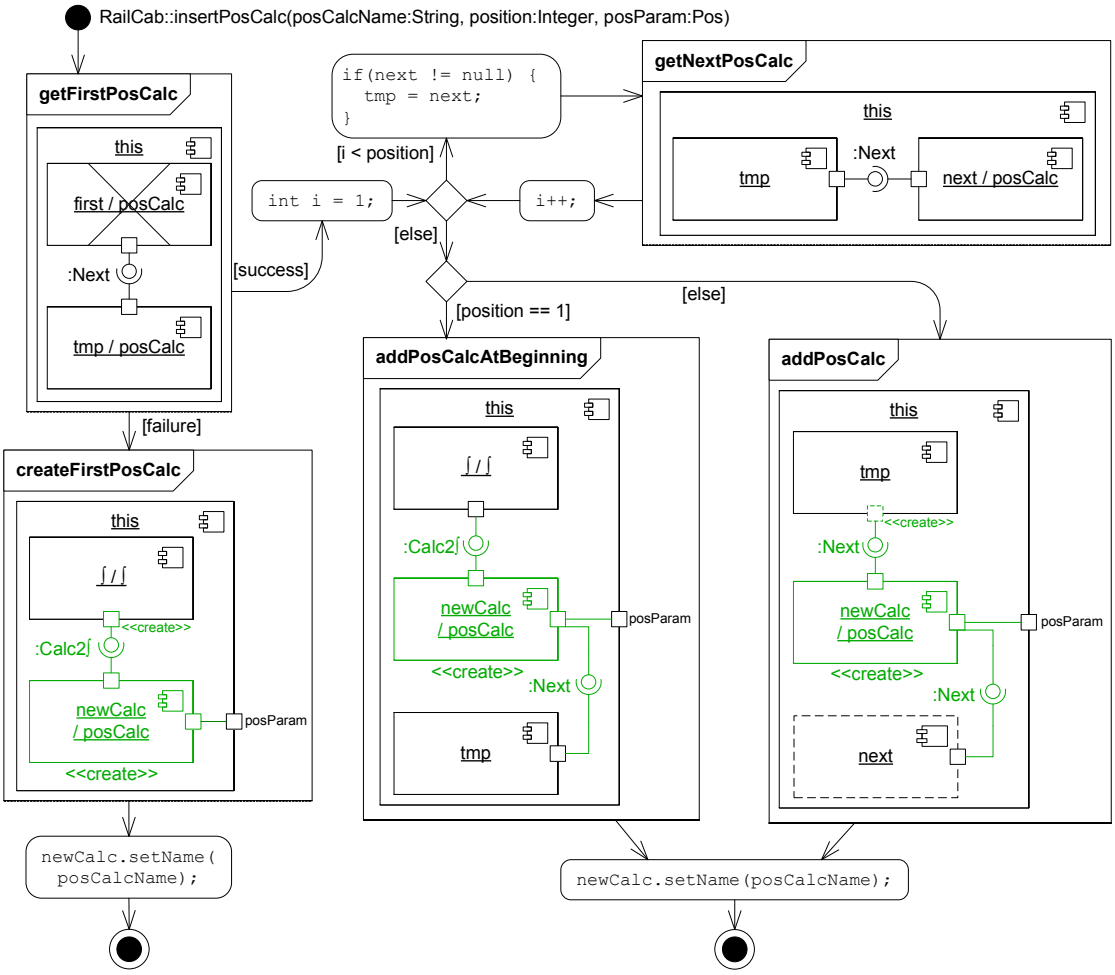


Abbildung 6.1: Rekonfigurationsregel insertPosCalc [THHO08]

Das in Abb. 6.1 gezeigte Komponentenstorydiagramm insertPosCalc für den Kompo-

nententyp RailCab wird in den folgenden Unterabschnitten als laufendes Beispiel benutzt, um die einzelnen vorgestellten Konstrukte im Zusammenhang zu zeigen. Für die Komponenteninstanz eines führenden RailCabs erstellt insertPosCalc eine neue Komponenteninstanz :PosCalc und fügt sie an einer parametrisierten Position innerhalb der Liste der :PosCalc-Instanzen ein. Dazu wird zunächst die erste Komponenteninstanz in einer :PosCalc-Liste ermittelt, und anschließend die Liste in einer Schleife durchlaufen bis entweder die Position oder das Ende erreicht ist. An der entsprechenden Stelle wird dann die neue Komponenteninstanz erstellt und der Typdefinition entsprechend über Konnektorinstanzen verbunden.

6.1.1 Komponentenstorypatterns

Innerhalb von Komponentenstorypatterns findet die Modellierung einzelner Graphtransformationsregeln statt. Wie konventionelle Storypatterns enthalten sie sowohl LHS als auch RHS im gleichen Diagramm. Um eine übersichtliche, leicht verständliche sowie einfach zu erlernende Graphtransformationssprache zu entwickeln, wird die konkrete Syntax der Komponentendiagramme von MECHATRONIC UML mit der Notation der konventionellen Storydiagramme kombiniert.

Für Komponentenstorypatterns wird das Konzept der typisierten Objektvariablen und Links durch *Variablen* und *Konnektorlinks* übernommen. Variablen repräsentieren Komponenten- oder Portinstanzen und Konnektorlinks Konnektorinstanzen und somit verbindende Objekte im Wirtsgraphen. Somit ist auch ein Konnektorlink eine Variable im eigentlichen Sinne. Da er aber wie ein herkömmlicher Link Elemente verbindet, wird er nach diesem benannt. Die Variablen und Konnektorlinks sind dabei klassifiziert über hierarchische Komponententypstrukturen, wie sie im letzten Kapitel definiert worden sind. Wie bei den konventionellen Storypatterns, ist die Abbildung der Variablen und Konnektorlinks auf die korrespondierenden Instanzen im Wirtsgraph isomorph, d. h. dass zwei verschiedene Variablen oder Konnektorlinks nicht auf die gleiche Instanz im Wirtsgraph abgebildet werden.

Variablen und Konnektorlinks Ein Komponentenstorypattern besitzt einen Namen, die *this-Variable* und optional ein *Constraint*, siehe Abb. 6.2. Ein Constraint ist wie in konventionellen Storypatterns eine in der Zielprogrammiersprache formulierte boolesche Bedingung. Wenn ein solches Constraint für ein Komponentenstorypattern definiert ist, wird nur dann versucht das Storypattern anzuwenden, falls die boolesche Bedingung positiv ausgewertet wird.

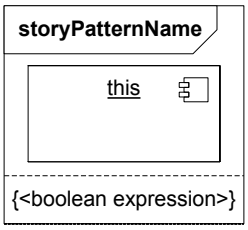


Abbildung 6.2: Komponentenstorypattern mit this-Variable

Die this-Variable ist eine *Komponentenvariable*. Komponentenvariablen sind Repräsentanten für eine Komponenteninstanz im Wirtsgraph. Die this-Variable stellt eine Selbst-

referenz auf die aktuelle Komponenteninstanz dar, auf dessen eingebetteter Struktur das Storypattern operiert. Sie ist über den Komponententyp des Storypatterns klassifiziert, so beziehen sich z. B. alle this-Variablen in insertPosCalc in Abb. 6.1 auf den Komponententyp RailCab. Abb. 6.3 visualisiert exemplarisch die Typisierung zwischen Elementen des Komponentenstorypatterns createFirstPosCalc und dem hierarchischen Komponententypen RailCab.

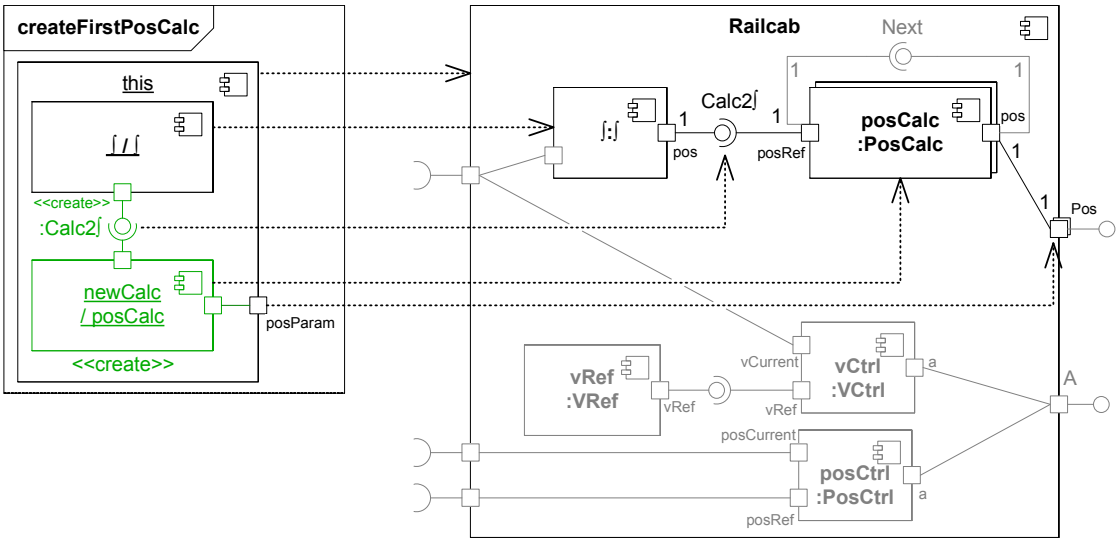
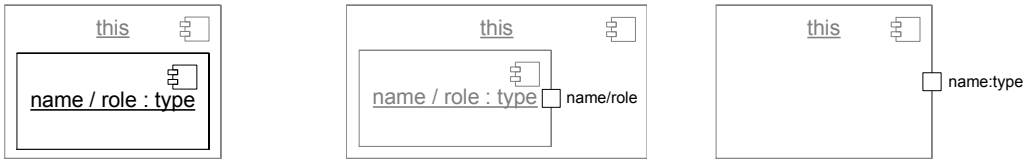


Abbildung 6.3: Typisierung von Variablen und Konnektorlinks

Innerhalb der this-Variable findet die eigentliche Regelmodellierung mit weiteren eingebetteten Variablen und Konnektorlinks statt. Die this-Variable stellt dabei den typisierenden Kontext zwischen diesen eingebetteten Elementen und der internen Struktur des Komponententyps her. Innerhalb der this-Variable verwendete Variablen sind über ein Part und Konnektorlinks über einen Konnektortyp klassifiziert. Diese Parts und Konnektortypen müssen in dem Komponententyp eingebettet sein, dem das Storydiagramm zugeordnet ist und welcher somit die this-Variable klassifiziert. In Storypattern createFirstPosCalc aus Abb. 6.3 z. B. werden die Komponentenvariablen $\int$ und newCalc verwendet, welche über die Parts $\int$:$\int$ bzw. posCalc:PosCalc typisiert sind. Diese wiederum sind eingebettet in den Komponententyp RailCab, für den das umgebende Storydiagramm insertPosCalc definiert ist und welcher somit die this-Variable im Storypattern createFirstPosCalc (sowie die this-Variablen in den übrigen Storypatterns) klassifiziert.

Komponentenvariablen werden, wie Komponenteninstanzen, mit unterstrichener Beschriftung dargestellt, siehe Abb. 6.4(a). Die Typinformation :type von Variablen ist optional und dient lediglich zur Verdeutlichung, denn das Part stellt die Beziehung zwischen Variable und Typ her.

Neben Komponentenvariablen gibt es *Portvariablen*. Analog zur Definition von Komponentenvariablen sind Portvariablen Repräsentanten von Portinstanzen im Wirtsgraph. Man unterscheidet zwischen einer innerhalb der this-Variable verwendeten Portvariable



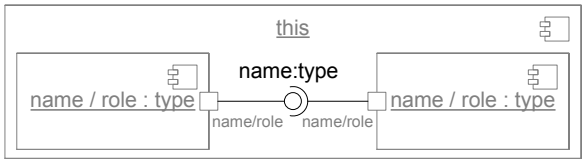
(a) Eingebettete Komponentenvariable (b) Eingebettete Portvariable (c) Externe Portvariable

Abbildung 6.4: Ungebundene Variablen

(Abb. 6.4(b)), und einer externen, direkt mit der this-Variable verbundenen Portvariable (Abb. 6.4(c)). Während erstere wie eingebettete Komponentenvariablen über ein Part role typisiert ist, ist letztere wie die this-Variable über einen Typ type klassifiziert, da in den Unterabschnitten 5.1.1 und 5.1.2 definiert wurde, dass Komponententypen Porttypen besitzen. Diese Struktur wird wie bei den eingebetteten Elementen durch die Typisierung der Variablen widergespiegelt. Die this-Variablen sind über einen Komponententyp klassifiziert, also sind auch direkt mit ihr verbundene Portvariablen über Porttypen klassifiziert. Im Storypattern createFirstPosCalc z. B. wird die Portvariable posParam eingesetzt, welche über den Porttyp Pos klassifiziert ist. Da posParam in Form eines Parameters vorliegt, ist dessen Typinformation an der Signatur des Storydiagramms zu erkennen. Die eingebetteten Portvariablen in createFirstPosCalc sind durch die Port-Parts typisiert, welche durch die entsprechenden Konnektortypen in RailCab verbunden werden. Die Namen dieser Portvariablen und die durch Pfeile angedeuteten Typisierungen wurden in Abb. 6.3 zwecks einer besseren Übersicht ausgelassen, da die Zuordnungen der übrigen Variablen und Konnektorlinks zu den Komponenten-Parts bzw. zu den Konnektortypen für eine eindeutige Typinformation aller Elemente ausreichen.

Konnektorlinks werden entsprechend ihrer Natur in *Kompositionskonnektorlinks* und *Delegationskonnektorlinks* unterschieden. Erstere repräsentieren eine Kompositionskonnektorinstanz, letztere eine Delegationskonnektorinstanz im Wirtsgraph. Ein Kompositionskonnektorlink verbindet zwei eingebettete Portvariablen (vgl. Abb. 6.5(a)), während ein Delegationskonnektorlink eine eingebettete und eine an der this-Variable angebrachten Portvariable verbindet (siehe Abb. 6.5(b)). Konnektorlinks sind über Konnektortypen type klassifiziert, die wie die typisierenden Parts von Variablen ebenfalls im Komponententyp der übergeordneten this-Variable eingebettet sein müssen. Im Storypattern createFirstPosCalc z. B. wird der Konnektorlink :Calc2$\int$ verwendet.

Die Klassifizierung der this-Variable und der mit ihr direkt verbundenen Portvariablen über Typen erlaubt es, Komponentenstorydiagramme für möglichst viele Anwendungsfälle zu definieren. Würde man die this-Variable über ein Part typisieren, würde auch das Komponentenstorydiagramm einem Komponenten-Part und nicht einem Komponententyp zugeordnet werden. Somit müsste für jede benutzte Rolle eines Komponententyps eine eigene Transformation definiert werden, obwohl deren modelliertes Verhalten der gleiche sein kann. Dieses Verhalten sollte also typabhängig sein und nicht spezifischen Rollen zugewiesen werden. Bei einer objektorientierten Klasse wird z. B. ebenfalls mit



(a) Kompositionskonnektorlink (b) Delegationskonnektorlink

Abbildung 6.5: Ungebundene Konnektorlinks

Methoden ein Verhalten für einen Typ spezifiziert, anstatt Verhalten lediglich für eine Assoziationsrolle festzulegen.

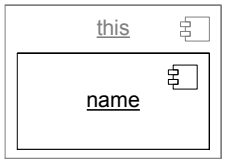
Die vorgestellte Typisierung der Variablen und Konnektorlinks erfüllt die in Kapitel 4 geforderte statische Typisierung auf Ebene der Transformationssprache. Es ist nicht möglich, eine Variable oder einen Link ohne Typinformationen anzulegen. Die Zuordnung des übergeordneten Storydiagramms zu einem Komponententypen stellt dabei den Kontext zu den möglichen Typisierungen der Variablen her.

Ungebundene und gebundene Elemente Analog zur Differenzierung in ungebundene und gebundene Objektvariablen der konventionellen Storypatterns, können auch Variablen entweder *ungebunden* oder *gebunden* sein. Da Konnektorlinks anders als die Links der klassischen Storypatterns verbindende Objekte im Wirtsgraph darstellen, kann diese Unterscheidung ebenso für sie gemacht werden. Ungebundene Elemente werden im Wirtsgraph anhand ihres Kontextes und ihrer Typinformationen gebunden. Ein einmal gebundenes Element kann anschließend in nachfolgenden Komponentenstorypatterns im gleichen Storydiagramm wiederverwendet werden.

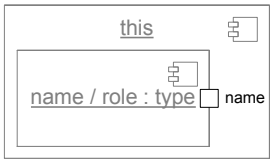
Wie bei konventionellen Storypatterns muss grundsätzlich ein gebundenes Element existieren, von dem ausgehend die ungebundenen Elemente gebunden werden. Dies ist in Komponentenstorypatterns die this-Variable, welche zu Beginn der Ausführung des Storydiagramms per se gebunden ist und eine Referenz auf die aktuelle Komponenteninstanz darstellt.

Das Storypattern getFirstPosCalc in Abb. 6.1 bindet z. B. innerhalb der aktuellen Komponenteninstanz this die eingebettete Komponenteninstanz tmp/posCalc. Ist dies erfolgreich, wird in bestimmten Situationen das Storypattern getNextPosCalc ausgeführt, in dem tmp nun als gebundene Variable vorliegt. Gebundene Elemente werden in Storypatterns ohne Typinformationen dargestellt, da sie bereits an einer anderen Stelle anhand ihres Klassifizierers gebunden wurden. Abb. 6.6 zeigt gebundene Variablen und Konnektorlinks. Liegen Elemente in Form eines Parameters vor, so sind diese ebenfalls bereits gebunden. Der Parameter posParam:Pos ist ein Beispiel hierfür, er wird in diversen Storypatterns als bereits gebundene Portvariable benutzt. Auf diese Weise erfolgt der Zugriff auf den Inhalt eines Parameters.

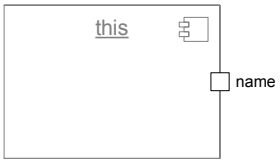
Modifizierer Mit den bisher eingeführten Variablen und Konnektorlinks lassen sich lediglich unveränderliche Anwendungsstellen definieren. Um konkrete Transformationen zu



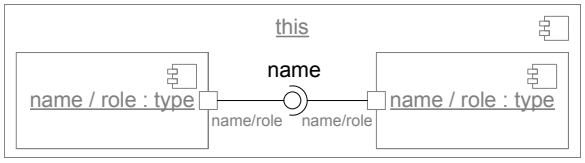
(a) Eingebettete Komponentenvariable



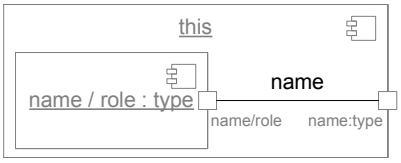
(b) Eingebettete Portvariable



(c) Externe Portvariable



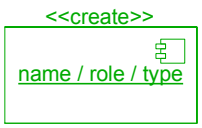
(d) Kompositionskonnektorlink



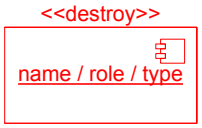
(e) Delegationskonnektorlink

Abbildung 6.6: Gebundene Elemente

beschreiben, können eingebetteten Variablen und Konnektorlinks die aus den konventionellen Storypatterns bekannten Modifizierer ≪create≫ (Abb. 6.7(a)) und ≪destroy≫ (Abb. 6.7(b)) zugeordnet werden. Variablen oder Links übernehmen dabei die entsprechende Farbe des Modifizierers. Elemente mit ≪create≫-Modifizierer werden bei einer erfolgreichen Regelanwendung instanziiert, Elemente mit ≪destroy≫-Modifizierer zerstört. Im Storypattern createFirstPosCalc z. B. wird eine Komponenteninstanz newCalc/posCalc erstellt, durch eine Delegationskonnektorinstanz mit posParam und über :Calc2$\int$ mit $\int$/$\int$ verbunden.



(a) ≪create≫-Modifizierer



(b) ≪destroy≫-Modifizierer

Abbildung 6.7: Modifizierer

Analog zur Definition aus Unterabschnitt 2.1.1 definieren die Modifizierer für die einzelnen Elemente eines Komponentenstorypatterns dessen LHS und RHS, vgl. auch Definition A.5 in [Zün01]. Gebundene Elemente können nicht mit dem ≪create≫-Modifizierer versehen werden, da sie bereits eine Instanz repräsentieren und es Mehrdeutigkeiten bzgl. der vorhandenen Instanz bei einer erneuten Instanziierung geben würde [FNT98].

Instanzen der this-Variable und von mit ihr verbundenen Portvariablen lassen sich nicht erstellen oder zerstören. Dies wird mit Storydiagrammen von Komponententypen bewerkstelligt, welche eine Hierarchieebene höher liegen. Ein Komponentenstorydiagramm operiert also lediglich auf der Hierarchieebene der eingebetteten Komponenteninstanzstruktur des korrespondierenden Komponententypen, um nicht die Kapselung einer Komponente aufzubrechen. Um mehrere Hierarchieebenen abzudecken, werden Transforma-

tionen von Komponententypen aufgerufen, welche eine Ebene tiefer liegen. Dies wird in Unterabschnitt 6.1.3 näher beleuchtet.

Anwendungsbedingungen Ebenfalls von den konventionellen Storydiagrammen werden negative und optionale Anwendungsbedingungen übernommen. Ein negatives Element darf für eine erfolgreiche Regelausführung nicht im Wirtsgraph vorhanden sein und wird durchgestrichen dargestellt, siehe Abb. 6.8(a)). Ein optionales Element kann für eine erfolgreiche Bindung der LHS im Wirtsgraph vorhanden sein, muss aber nicht zwangsweise existieren. Optionale Elemente werden mit gestrichelten Linien visualisiert, vgl. Abb. 6.8(b) und Abb. 6.8(c).

(a) Negative Anwendungsbedingung (b) Optionale Variablen (c) Optionale Konnektorlinks

Abbildung 6.8: Anwendungsbedingungen

Im Beispiel wird im Storypattern getFirstPosCalc die negative Komponentenvariable first/posCalc verwendet, um die erste Komponenteninstanz einer :PosCalc-Liste zu ermitteln. Dazu wird die Komponenteninstanz tmp/posCalc gesucht, welche nicht mit einer vorherigen Komponenteninstanz über den Konnektor :Next verbunden ist. Die Anordnung der angebotenen und benötigten Schnittstellen gibt dabei die Konnektorrichtung vor.

Optionale Elemente sind im Storypattern addPosCalc zu finden, welches das Hinzufügen neuer Komponenteninstanzen auch am Ende einer :PosCalc-Liste erlaubt. Das Storypattern getNextPosCalc übernimmt vorher die Aufgabe, die Komponenteninstanzen tmp und next zu binden. Soll newCalc am Ende der :PosCalc-Liste hinzugefügt werden, kann in diesem Storypattern next nicht gebunden werden, da tmp das letzte Element der Liste ist. In diesem Fall können auch die Konnektorinstanz :Next sowie die entsprechenden Portinstanzen – insbesondere die Portinstanz von tmp – nicht existieren. Um diesen Fall nicht in einem gesonderten Storypattern zu behandeln, werden in addPosCalc next und die Portinstanz von tmp als optional markiert. Somit wird newCalc nur dann mit next verbunden, wenn next existiert. Falls next nicht existiert, existiert auch die Portinstanz von tmp nicht und müsste in diesem Fall neu kreiert werden. Diese wird dank der optionalen Anwendungsbedingung nur dann instanziiert, wenn sie nicht bereits existiert.

Fischer et al. [FNT98] definieren für konventionelle Storypatterns die möglichen Kombinationen negativer und optionaler Anwendungsbedingungen mit gebundenen Objektvariablen und Modifizierern. So ist eine negative Objektvariable per Definition ungebunden, daher kann diese mit einem ≪create≫-Modifizierer versehen werden. In diesem Fall wird ein Objekt des spezifizierten Typs instanziiert, falls es nicht im Wirtsgraph existiert. Gebundene negative Objektvariablen sind nicht definiert. Ungebundene dagegen können nicht zerstört werden, da sie im Widerspruch zu ihrer Definition existieren müssten. Optionale Objektvariablen lassen sich mit beiden Modifizierern kombinieren:

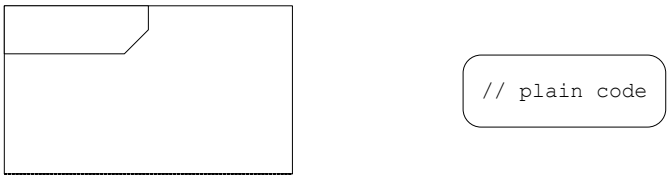
Ein optionales, zu erstellendes Objekt wird instanziiert, falls es nicht existiert (siehe Storypattern addPosCalc); ein optionales, zu löschendes Objekt wird gelöscht, falls es existiert. Letztendlich können optionale Objekte als gebunden markiert und gleichzeitig mit ≪create≫-Modifizierer versehen werden. Diese Semantiken werden für Variablen und Konnektorlinks übernommen.

Mengenvariablen werden in dieser Arbeit nicht eingeführt, da sie von der technischen Seite her kompliziert zu realisieren sind und dabei die Ausdrucksfähigkeit von Komponentenstorydiagrammen kaum erhöhen. Des Weiteren unterliegen negative und optionale Anwendungsbedingungen, insbesondere so wie sie in Abb. 6.1 formuliert wurden, im Kontext dieser Arbeit starken Einschränkungen. Diese Themen werden in Abschnitt 6.5.5 genauer erläutert.

6.1.2 Kontrollstrukturen

(Komponenten-)Storydiagramme sind adaptierte Aktivitätsdiagramme, somit finden die wichtigsten Kontrollstrukturen aus Aktivitätsdiagrammen ebenfalls Anwendung in Komponentenstorydiagrammen. Im Prinzip werden die Kontrollstrukturen der konventionellen Storydiagramme und deren Semantik [FNTZ98, Zün01, FNT98] übernommen. Dieser Unterabschnitt stellt sie in einer kurzen Übersicht vor. Ein Komponentenstorydiagramm besteht aus diversen *Aktivitäten*, welche untereinander mit *Transitionen* verbunden sind[1].

Aktivitäten Aktivitäten können ein Komponentenstorypattern (Abb. 6.9(a), siehe Unterabschnitt 6.1.1) oder reinen Code der Zielprogrammiersprache enthalten, welcher direkt an der entsprechenden Stelle in den aus dem Komponentenstorydiagramm generierten Code kopiert wird. Letztere Aktivität wird *Statement-Aktivität* genannt und ist in Abb. 6.9(b) dargestellt. Mit Statement-Aktivitäten können Berechnungen durchgeführt werden, die mit reinen Graphtransformationen gar nicht oder nur sehr umständlich modelliert werden können. Im Komponentenstorydiagramm insertPosCalc in Abb. 6.1 werden z. B. Statement-Aktivitäten mit Java-Code für die Initialisierung und Inkrementierung eines Zählers sowie für das Kopieren der Variable next nach tmp benutzt.

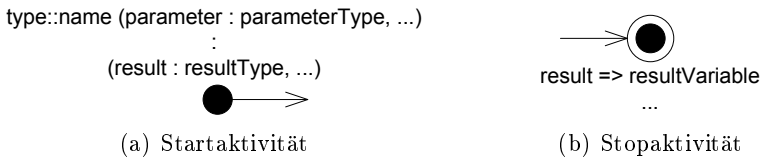


(a) Komponentenstorypattern (b) Statement-Aktivität

Abbildung 6.9: Aktivitäten

[1] Wie in Unterabschnitt 2.1.1 bereits erwähnt, basieren Storydiagramme auf UML 1 Aktivitätsdiagrammen. Somit werden an dieser Stelle die Begrifflichkeiten und Semantiken der Literatur über Storydiagramme sowie der UML 1 [BRJ99] benutzt.

Der Einstiegspunkt eines Komponentenstorydiagramms ist die *Startaktivität (Startzustand)*, welche keine eingehende Transition besitzt (siehe Abb. 6.10(a)). Die Startaktivität visualisiert über name den Namen des Komponentenstorydiagramms und über type die Information über den Komponententyp, dem das Diagramm zugeordnet ist. Ein Komponentenstorydiagramm kann mit typisierten Parametern und Rückgabedeklarationen versehen werden, dies wird in Unterabschnitt 6.1.3 genauer vorgestellt. Die Informationen über die Parameter und Rückgabedeklarationen werden ebenfalls vom Startzustand dargestellt. Zusammen bilden diese Informationen die *Signatur* eines Komponentenstorydiagramms. Das Storydiagramm insertPosCalc z. B. ist dem Komponententyp RailCab zugeordnet, besitzt drei Parameter und keine Rückgabedeklaration.

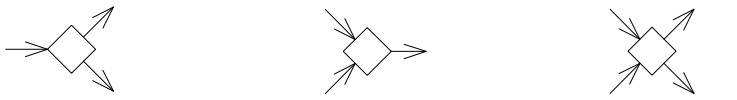


(a) Startaktivität (b) Stopaktivität

Abbildung 6.10: Start- und Endaktivitäten

Eine *Stopaktivität (Endzustand)* (vgl. Abb. 6.10(b)) beendet die Ausführung eines Komponentenstorydiagramms, besitzt somit keine ausgehende Transition und kann Werte zurückgeben. Im Schema wird der Wert resultVariable an die Rückgabedeklaration result gebunden, was in Unterabschnitt 6.1.3 näher erläutert wird.

Zur Auswertung von alternativen Transitionen gibt es *NOP-Aktivitäten (Pseudo-Aktivitäten)*, welche in Abb. 6.11 dargestellt werden. Eine Verzweigung trennt den Kontrollfluss in zwei Pfade auf, während eine Vereinigung zwei unterschiedliche Pfade zusammenführt. Die Codegenerierung von Fujaba erlaubt dabei nicht (im Gegensatz zu [Zün01]) mehr als zwei aus-/eingehende Pfade. Aufeinander folgende Verzweigungen und Vereinigungen lassen sich zusammenziehen, wie in Abb. 6.11(c) zu sehen ist. In einfachen Fällen kann die Modellierung von NOP-Aktivitäten auch eingespart werden, indem direkt einer vorausgehenden/folgenden Aktivität zwei aus-/eingehende Transitionen zugeordnet werden.



(a) Verzweigung (b) Vereinigung (c) Kombination

Abbildung 6.11: NOP-Aktivitäten

Eine Verzweigung findet sich beispielsweise im Kontrollfluss von insertPosCalc vor den Komponentenstorypatterns addPosCalcAtBeginning und addPosCalc, eine Vereinigung direkt danach. Bei dieser Vereinigung wurde die explizite NOP-Aktivität eingespart. Eine Kombination von Verzweigung und Vereinigung ist die oberste NOP-Aktivität in in-

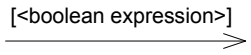
sertPosCalc. Die ein- und ausgehenden Transitionen dieser NOP-Aktivität werten eine Schleifenbedingung aus, dementsprechend komplex ist der resultierende Kontrollfluss.

Abbildung 6.12: Transition

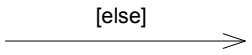
Transitionen und Bedingungen Aktivitäten werden untereinander mit Transitionen verbunden, welche durch gerichtete Kanten dargestellt werden. Abb. 6.12 zeigt eine Transition ohne Bedingungen. Der Nachfolger einer Aktivität muss eindeutig bestimmbar sein, d. h. eine Aktivität darf nicht zwei ausgehende Transitionen besitzen, sofern eine der Transitionen keine Bedingung besitzt.

Transitionen können mit *Bedingungen (Guards)* versehen werden. Eine bedingte Transition kann nur schalten, sofern die entsprechende Bedingung positiv ausgewertet wird. Da der Nachfolger einer Aktivität eindeutig bestimmbar sein muss, müssen sich die Bedingungen mehrerer aus einer Aktivität ausgehender Transitionen gegenseitig ausschließen. Wie bereits erwähnt, lässt die Codegenerierung von FUJABA lediglich zwei aus einer Aktivität ausgehende Transitionen zu.

Abb. 6.13(a) zeigt eine Transition mit einer beliebigen booleschen Bedingung. Diese Bedingung wird direkt als Ausdruck in einer if-Anweisung der Zielprogrammiersprache ausgewertet, daher muss die Bedingung ein in der Zielprogrammiersprache formulierter boolescher Ausdruck sein. Diese boolesche Bedingung kann komfortabel mit der *alternativen Transition* (Abb. 6.13(b)) mit der Bedingung else kombiniert werden, um einen gegenseitigen Ausschluss zweier Guards zu erreichen. Diese Transition schaltet, falls die boolesche Bedingung negativ ausgewertet wird. Dieses Konstrukt findet sich z. B. im Kontrollfluss von insertPosCalc vor den Komponentenstorypatterns addPosCalcAtBeginning und addPosCalc.



(a) Transition mit boolescher Bedingung



(b) Alternative Transition

Abbildung 6.13: Transitionen mit Bedingungen

Da Storypatterns in den Kontrollfluss eines Storydiagramms eingebettet werden, sind Konstrukte für die Reaktion auf eine erfolgreiche bzw. erfolglose Regelanwendung wünschenswert. Storydiagramme bieten diese Konstrukte in Form von success- und failure-Bedingungen für Transitionen, welche ein Storypattern verlassen. Eine success-Transition schaltet nur dann, wenn das Storypattern erfolgreich angewendet werden konnte, während die failure-Transition im erfolglosen Fall schaltet. Abb. 6.14 zeigt diese beiden speziellen Transitionen. Im laufenden Beispiel werden diese Transitionsbedingungen nach dem Storypattern getFirstPosCalc benutzt, um zu ermitteln ob bereits eine /posCalc-Komponenteninstanz existiert.

(a) Transition für erfolgreiche Regelanwendung (b) Transition für erfolglose Regelanwendung

Abbildung 6.14: Transitionen mit Bedingungen für Komponentenstorypatterns

forEach-Aktivität Ein Spezialfall einer Aktivität mit Storypattern ist die *forEach-Aktivität*. Sie wird durch einen zweiten angedeuteten Rahmen dargestellt, vgl. Abb. 6.15. Während konventionelle Storypatterns bei einem einmaligen Aufruf nichtdeterministisch eine Anwendungsstelle der LHS suchen, werden Storypatterns einer forEach-Aktivität für jede mögliche Anwendungsstelle im Wirtsgraph ausgeführt. Für diese Aktivität gibt es zwei spezielle Transitionsbedingungen. Die Transition mit der Bedingung each time schaltet jedes Mal, nachdem vom Storypattern der forEach-Aktivität eine neue Anwendungsstelle der LHS gefunden wurde. Die Transition mit end-Bedingung schaltet, nachdem alle Anwendungsstellen abgearbeitet sind.

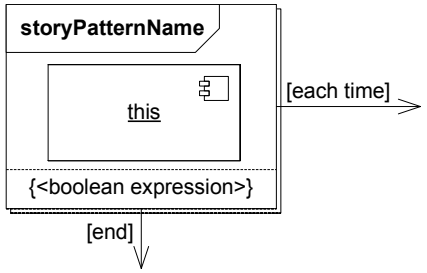
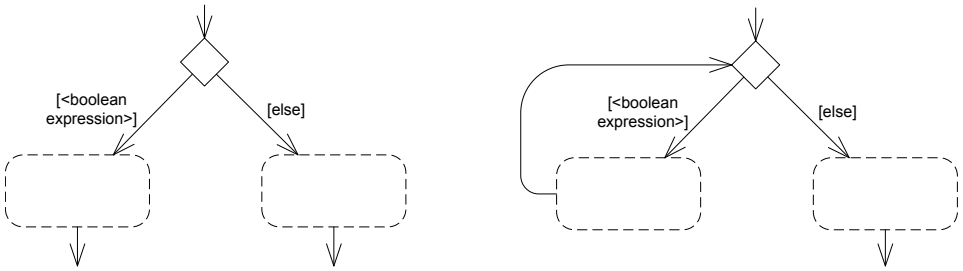


Abbildung 6.15: forEach-Aktivität

Verzweigungen und Iterationen Die beschriebenen Kontrollstrukturen lassen sich zu komplexeren, aus den Programmiersprachen bekannten Konstrukten kombinieren. Hierbei gelten die Einschränkungen bzgl. der Wohlgeformtheit von Aktivitätsdiagrammen [Zün01, FNT98]. Abb. 6.16 zeigt z. B. die Schemata einer Verzweigung und einer Schleife, weitere Variationen sind denkbar. Mit solchen Kombinationen lassen sich die von Bradbury et al. [BCDW04] geforderten Operationskonstrukte für dynamische Architekturtransformationen (vgl. Abschnitt 3.2) in Komponentenstorydiagrammen modellieren.



(a) Verzweigung (b) Schleife

Abbildung 6.16: Programmiersprachliche Kontrollstrukturen [Zün01]

6.1.3 Hierarchische Organisation

Um die Kapselung von Komponenten zu respektieren, operieren Komponentenstorypatterns lediglich direkt auf der eingebetteten Struktur ihres Komponententypen. Der direkte Zugriff auf höhere oder tiefere Hierarchieebenen oder auf andere Komponententypen derselben Hierarchieebene ist nicht erlaubt. Um ganze Systeme über mehrere Hierarchieebenen zu instanziieren und zu rekonfigurieren, werden *Aufrufe (Calls)* von Komponentenstorydiagrammen eingesetzt. Der Komponententyp des aufgerufenen Storydiagramms muss dabei der selbe sein wie der des aufrufenden Storydiagramms, oder einer, der genau eine Hierarchieebene tiefer liegt. Aufrufe bieten neben der Möglichkeit zur Überbrückung der Hierarchien den Vorteil, dass von der Wiederverwendbarkeit mehrerer kleinerer Transformationen profitiert werden kann, anstatt für jede Situation eine große Transformation für ein gesamtes System anzulegen.

Abb. 6.17 zeigt das Komponentenstorydiagramm insertFollower für die Systemebene ConvoySystem, welches im Folgenden mehrfach referenziert wird. Der Komponententyp ConvoySystem ist in Abb. 5.11 in Unterabschnitt 5.1.2 definiert. In insertFollower wird eine neue RailCab-Komponenteninstanz der Rolle convoyFollowers erstellt, welche mit der Komponenteninstanz leader des führenden RailCabs verbunden wird. Um leader anschließend intern zu rekonfigurieren und eine neue Instanz von PosCalc an der entsprechenden Position einzubetten, wird das bisherige laufende Beispiel insertPosCalc aus Abb. 6.1 aufgerufen.

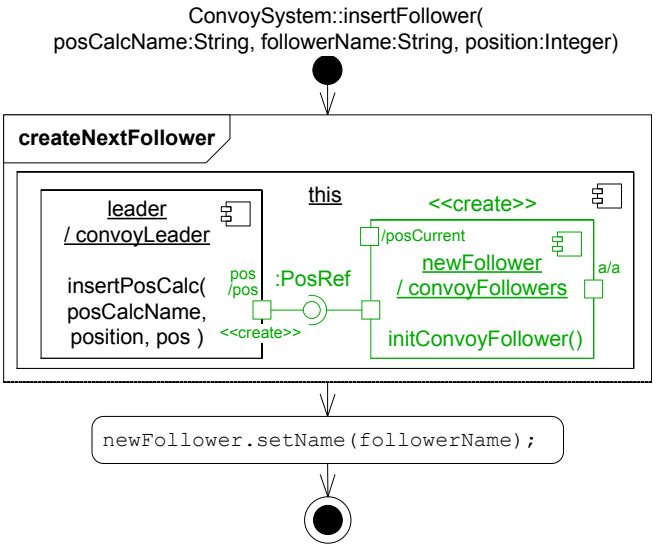
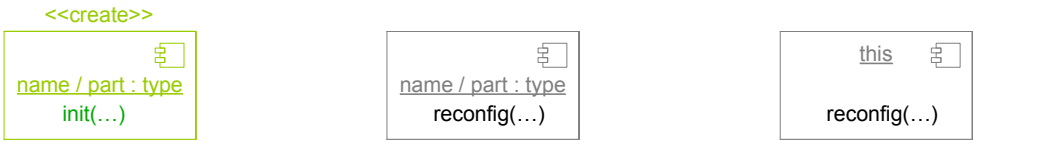


Abbildung 6.17: Rekonfigurationsregel insertFollower

Aufruf von Komponentenstorydiagrammen Wenn man in einem Komponentenstorydiagramm eine Komponente instanziiert, welche über ein Part eines hierarchischen Komponententypen klassifiziert ist, dann möchte man i. A. direkt die eingebettete Struktur

dieser neuen Komponenteninstanz initialisieren. Für diesen Zweck eignen sich *Konstruktoraufrufe (Constructor Calls)*. Wie bereits erwähnt, erstellen Konstruktoren die initiale eingebettete Struktur einer neuen Komponenteninstanz. Konstruktoraufrufe lassen sich für Komponentenvariablen mit einem ≪create≫-Modifizierer spezifizieren, wie in Abb. 6.18(a) zu sehen ist. Im Beispiel wird in insertFollower der Konstruktor initConvoyFollower nach der Instanziierung von newFollower aufgerufen, welcher dessen benötigte Komponenteninstanzstruktur für die Auswertung der vom führenden RailCab übermittelten Referenzpositionen initialisiert. Dieser Konstruktor ist technisch umgesetzt in Abschnitt 7.2.2 in Abb. 7.8 als Teil des Evaluierungsbeispiels dargestellt.



(a) Konstruktoraufruf (b) Rekonfigurationsaufruf (c) Rekonfigurationsaufruf auf this-Variable

Abbildung 6.18: Konstruktor- und Rekonfigurationsaufrufe

Rekonfigurationsaufrufe (Reconfiguration Calls) bieten prinzipiell die gleiche Funktionalität wie Konstruktoraufrufe, rufen allerdings eine Rekonfigurationsregel auf. Eine Rekonfigurationsregel transformiert die interne Struktur einer bereits vorhandenen Komponenteninstanz. Somit werden Rekonfigurationsaufrufe lediglich für Komponentenvariablen ohne Modifizierer spezifiziert, vgl. Abb. 6.18(b). Beispielsweise wird in insertFollower auf der Komponenteninstanz leader die Rekonfigurationsregel insertPosCalc (Abb. 6.1) aufgerufen, um in die eingebettete Komponenteninstanzstruktur von leader eine neue :PosCalc-Komponenteninstanz an der entsprechenden Position einzufügen. Dies geschieht, nachdem die Strukturanpassung auf der Ebene von ConvoySystem abgeschlossen ist.

Rekonfigurationsaufrufe auf der this-Variable sind ebenfalls erlaubt (siehe Abb. 6.18(c)), da diese nicht modifiziert wird und lediglich eine Rekonfigurationsregel auf derselben Komponenteninstanz aufgerufen wird. Daher wird die aktuelle Hierarchieebene nicht verlassen. Rekonfigurationsaufrufe auf der this-Variable ermöglichen die Dekomposition komplexerer Rekonfigurationsregeln und erhöhen somit ihre Wiederverwendbarkeit. Zusätzlich sind mit diesem Konstrukt auch rekursive Aufrufe möglich.

Aufgerufen werden können lediglich Komponentenstorydiagramme, die dem Komponententyp type bzw. dem Komponententyp der this-Variable zugeordnet sind. Mittels verketteter Aufrufe hinweg über die Komponententyphierarchie eines kompletten Systems lassen sich schließlich ganze Konfigurationen initialisieren und rekonfigurieren.

Abb. 6.19 zeigt ein UML Sequenzdiagramm [Obj07b] zur Verdeutlichung der Interaktionen zwischen insertFollower und den aufgerufenen Transformationen. Die Auswirkung von Statement-Aktivitäten zum Setzen des Namens von Komponenteninstanzen werden in dem Diagramm nicht dargestellt. Am Sequenzdiagramm lässt sich erkennen, dass sich durch die Komponententyphierarchie auch eine zeitliche Ausführungsreihenfolge der einzelnen Transformationen ergibt. So wird zunächst die eingebettete Struktur der

Komponenteninstanz :ConvoySystem transformiert. In der abgebildeten Situation ist dies die Instanziierung von convoyFollowers:RailCab und der anliegenden Port- und Konnektorinstanzen. Letzteres wird im Sequenzdiagramm nicht abgebildet. Anschließend werden die Komponentenstorydiagramme zur Transformation der nächsttieferen Hierarchieebene aufgerufen, in diesem Fall initConvoyFollower und insertPosCalc. Die Komponententyphierarchie im Sequenzdiagramm von links nach rechts abgebildet: Die Systemebene ConvoySystem ist zusammengesetzt aus Parts des Komponententyps RailCab (Ebene 1), welche u. a. Parts der Typen PosCtrl und PosCalc (Ebene 2) enthalten.

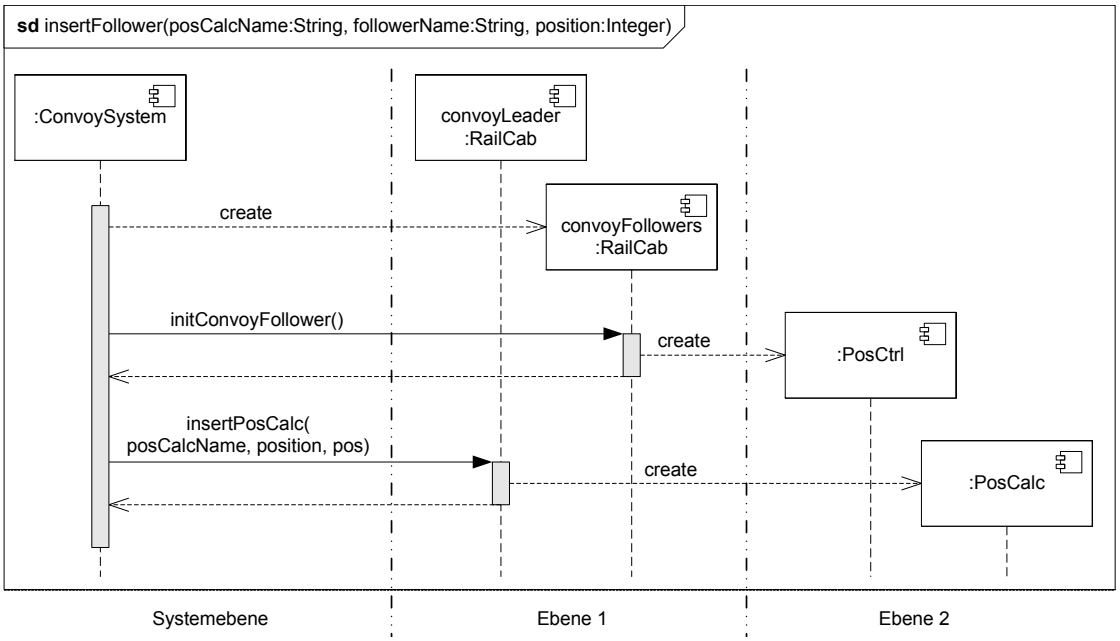
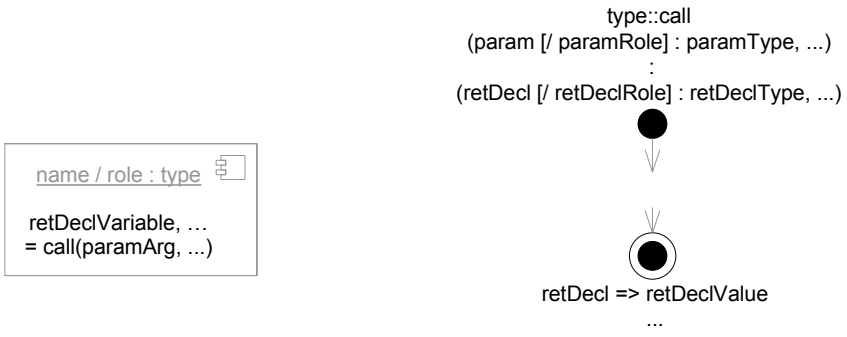


Abbildung 6.19: Sequenzdiagramm zu insertFollower

Parameter und Argumente Für Komponentenstorydiagramme können Parameter definiert werden. Name und Typ von Parametern werden in der Signatur des Komponentenstorydiagramms angezeigt, welche an dessen Startknoten annotiert ist. Dies ist in Abb. 6.20(b) visualisiert.

Primitive Parameter repräsentieren Parameter, welche entweder über die Klasse String oder über die Wrapper-Klassen der primitiven Datentypen der zu Grunde liegenden Programmiersprache klassifiziert sind, z. B. Boolean oder Integer. Der Zugriff auf den Parameter erfolgt über seinen Namen. In insertFollower sind beispielsweise alle Parameter primitiv. Der Parameter followerName wird zum Setzen des Namens von newFollower in der Statement-Aktivität verwendet, während die anderen Parameter beim Aufruf weiterer Komponentenstorydiagramme benutzt werden.

Komplexe Parameter sind typisiert über die Struktur des Komponententyps, für den das Komponentenstorydiagramm definiert ist. Ein komplexer Parameter kann zum einen



(a) Aufruf Komponentenstorydiagramm

(b) Aufgerufenes Komponentenstorydiagramm

Abbildung 6.20: Zusammenhang Parameter/Argumente und Rückgabedeklarationen /-werte

Portvariablen der this-Variable repräsentieren, in diesem Fall ist der Parameter also über einen Porttyp klassifiziert. Zum anderen kann er eine eingebettete Variable darstellen, in diesem Fall klassifiziert ein Part oder ein Konnektortyp den komplexen Parameter. Komplexe Parameter, welche Elemente anderer Komponententypen repräsentieren, sind – ebenfalls um die Kapselung der Komponenten zu respektieren – nicht erlaubt.

Der Zugriff auf einen komplexen Parameter erfolgt, indem eine typ-/partgleiche Variable oder ein typgleicher Konnektorlink des korrespondierenden Komponentenstorydiagramms an ihn gebunden wird. Dazu wird die Variable oder der Link als gebunden markiert und mit dem Namen des Parameters versehen. Beim Zugriff auf die Variable bzw. den Link wird somit auf den Parameter zugegriffen. In insertPosCalc z. B. wird der komplexe Parameter posParam:Pos eingesetzt, um eine neue Komponenteninstanz newCalc mit der :Pos-Portinstanz zu verbinden, welche in einer aufrufenden Transformation erstellt wurde. Da die Variable posParam gebunden ist, wird sie in den jeweiligen Storypatterns ohne Typinformation dargestellt.

Parametern müssen beim Aufruf des korrespondierenden Komponentenstorydiagramms konkrete Werte zugewiesen werden. Ein solcher Parameterwert nennt sich *Argument*. Für einen primitiven Parameter wird als Argument z. B. abhängig vom Typ eine konkrete Zeichenfolge oder eine konkrete Zahl als Wert spezifiziert. Das Argument eines komplexen Parameters ist eine Variable oder ein Konnektorlink im aufrufenden Komponentenstorydiagramm.

Abb. 6.20 stellt schematisch den Zusammenhang zwischen komplexen Parametern und Argumenten her. In Abb. 6.20(b) wird die Signatur eines Komponentenstorydiagramms gezeigt, welche den Parameter param spezifiziert. Dieser Parameter ist über einen Typ paramType oder ein Part paramRole des Typs paramType klassifiziert. Beim Aufruf des Storydiagramms wird diesem Parameter ein Argument zugewiesen. Dies ist eine Variable oder ein Konnektorlink paramArg des aufrufenden Storydiagramms, welche(r) ebenfalls über paramType oder paramRole typisiert ist.

Auch bei der Parametrisierung von Transformationsaufrufen muss die Kapselung von

Komponenten berücksichtigt werden. Somit sind lediglich komplexe Parameter bzw. Argumente sinnvoll, auf welche sowohl die aufgerufene als auch die aufrufende Transformation Zugriff haben.

Jedes Komponentenstorydiagramm, welches ein hierarchisch tiefer liegendes Storydiagramm über eine Komponentenvariable aufruft, hat auch Zugriff auf die Portvariablen dieser Komponentenvariable. Diese werden im aufgerufenen Storydiagramm – eine Hierarchieebene tiefer – durch Portvariablen der this-Variable repräsentiert, auf welche ebenfalls zugegriffen kann. Somit eignen sich deren Porttypen als Klassifizierer für einen Parameter. Das Argument ist in diesem Fall also eine Portvariable der Komponentenvariable, welche den Aufruf spezifiziert. In insertFollower beispielsweise wird für leader die Portinstanz pos erstellt, über dessen Konnektor leader mit newFollower verbunden wird. Diese Portinstanz wird im darauffolgenden Aufruf von insertPosCalc als Argument übergeben. Diese Rekonfigurationsregel benötigt diese Information, um eine neue Komponenteninstanz :PosCalc über eine Delegationskonnektorinstanz mit pos zu verbinden. Hier zeigt sich, dass ein Port ein Interaktionspunkt einer Komponente mit seiner Umwelt ist. Andere Parameterdeklarationen für hierarchieübergreifende Transformationsaufrufe sind nicht möglich.

Bei Rekonfigurationsaufrufen auf der this-Variable können dagegen auch Parameter über eingebettete Parts oder Konnektortypen klassifiziert werden. Sowohl aufrufende als auch aufgerufene Transformation haben Zugriff auf die eingebetteten Variablen und Konnektorlinks der this-Variable, da diese in beiden Transformationen dieselbe Komponenteninstanz darstellt.

Rückgabedeklarationen und -werte Komponentenstorydiagramme können Werte zurückgeben. Mittels *Rückgabedeklarationen* lassen sich Rückgabetypen eines Komponentenstorydiagramms spezifizieren. Anders als in Funktionen und Methoden der Programmiersprachen sind hier wie bei den Transformationsdiagrammen auch mehrere Rückgabedeklarationen mit Namen möglich. Dadurch fällt der Zwang weg, für mehrere Rückgabewerte eine Menge zu bilden, was in Storydiagrammen eher umständlich geschehen müsste. Rückgabedeklarationen werden wie Parameter in der Signatur des Komponentenstorydiagramms angezeigt, wie in Abb. 6.20(b) zu sehen ist.

Rückgabedeklarationen werden über Komponententypstrukturen klassifiziert. Wie komplexe Parameter lassen sie sich also durch eingebettete Parts und Konnektortypen sowie durch Porttypen der Komponente des Storydiagramms typisieren. Der Zugriff auf Rückgabedeklarationen erfolgt ähnlich wie bei den komplexen Parametern, allerdings im aufrufenden Storydiagramm: Eine typ-/partgleiche Variable oder ein typgleicher Konnektorlink wird über den Namen an die Rückgabedeklaration gebunden.

Rückgabewerte sind das Pendant zu Argumenten, also konkrete Variablen oder Konnektorlinks, welche über denselben Typ oder dasselbe Part wie die korrespondierende Rückgabedeklaration klassifiziert sind. Ein Rückgabewert wird über einen Endknoten an eine Rückgabedeklaration gebunden.

In Abb. 6.20 wird neben den Parametern/Argumenten auch der Zusammenhang zwischen Rückgabedeklarationen und -werten dargestellt. Im Transformationsaufruf in Abb.

6.20(a) z. B. wird einer Variable retDeclVariable des umgebenden Storypatterns der Rückgabewert der aufgerufenen Transformation call zugewiesen. Die Variable muss dabei über retDeclRole oder retDeclType klassifiziert sein. In Abb. 6.20(b) dagegen wird eine innerhalb von call benutzte Variable retDeclValue als Rückgabewert an die Rückgabedeklaration retDecl gebunden und zurückgegeben, wenn die Stopaktivität erreicht wird.

Für die Typisierung von Rückgabedeklarationen gelten die gleichen Einschränkungen wie für die komplexen Parameter. Eine Rückgabedeklaration lässt sich lediglich im aufrufenden Storydiagramm verwenden, wenn dessen Komponententyp auch Zugriff auf den Klassifizierer der Rückgabedeklaration hat. Bei hierarchieübergreifenden Transformationsaufrufen können dies, wie bei den komplexen Parametern, wieder nur Porttypen sein. Das aufgerufene Storydiagramm besitzt in diesem Fall eine Rückgabedeklaration, welche über einen Porttyp des Komponententypen des Storydiagramms klassifiziert ist und somit eine Portvariable der this-Variable repräsentiert. Im aufrufenden Storydiagramm entspricht diese externe Portvariable einer eingebetteten Portvariable der Komponentenvariable, welche den Aufruf spezifiziert. Bei Rekonfigurationsaufrufen auf der this-Variable dagegen können wieder eingebettete Strukturelemente als Klassifizierer verwendet werden. Eine Rückgabedeklaration, welche typisiert ist über ein eingebettetes Part oder einen eingebetteten Konnektortyp, lässt sich in diesem Fall im aufrufenden Storydiagramm verwenden, da dieses dem gleichen Komponententyp zugeordnet ist wie das aufgerufene Storydiagramm.

Ein konkretes Beispiel für die Rückgabe einer Portinstanz bei einem hierarchieübergreifenden Transformationsaufruf findet sich im Evaluierungsbeispiel in den Rekonfigurationsregeln removeFollower (Abb. 7.12) und removePosCalc (Abb. 7.11) im Abschnitt 7.2.3. Dort ermittelt removePosCalc die Portinstanz :Pos, welche mit der :PosCalc-Komponenteninstanz an der zu löschenden Position verbunden ist bzw. war. In removeFollower wird diese Portinstanz nach dem entsprechenden Aufruf per Rückgabewert gebunden, so dass anschließend die mit ihr verbundene /convoyFollowers:RailCab-Instanz gelöscht wird. Obwohl die aufrufende Transformation keinen Zugriff auf die interne Struktur der /convoyLeader-Komponenteninstanz hat, kann sie über den zurückgegebenen Wert von removePosCalc die anhand ihrer Delegationskonnektorverbindung ermittelte Portinstanz binden.

Die in diesem Unterabschnitt beschriebenen Konzepte für den hierarchieübergreifenden Aufruf von Komponentenstorydiagrammen erlauben es, hierarchische Graphtransformationen im Sinne von Abschnitt 3.1 zu spezifizieren. Dabei wird eine strikte Kapselung der Komponentenhierarchien eingehalten. Die einzigen Elemente, auf die sowohl aufrufende als auch aufgerufene Transformation Zugriff haben sind die Ports, welche die Interaktionspunkte einer Komponente mit ihrer Umwelt darstellen. Die Auflösung der hierarchischen Beziehungen geschieht durch die Aufrufe der einzelnen Graphtransformationen untereinander sowie die Abbildung der Komponentenstorydiagramme auf Transformationsdiagramme, welche auf flachen Objektstrukturen spezifiziert sind. Grenzüberschreitende Kanten, welche bei hierarchischen Graphtransformationen problematisch sein können, werden durch den generellen Komponentenaufbau vermieden, bei dem die Umwelt mit eingebetteten Elementen einer Komponente lediglich über Ports und Delegationskonnektoren kommunizieren darf.

6.2 Berücksichtigung von kompositionalen Beziehungen

Im letzten Abschnitt wurden die Basiselemente für die Modellierung von Komponentenstorydiagrammen beschrieben. Dabei wurde auch die Typisierung der einzelnen Variablen und Konnektorlinks der Storypatterns betrachtet. Bei der Modellierung von Komponentenstorypatterns und deren Abbildung auf Transformationsdiagramme sind zusätzlich die kompositionalen Beziehungen zwischen den einzelnen Elementen der resultierenden Komponenteninstanzstrukturen zu berücksichtigen. So kann z. B. keine Konnektorinstanz ohne die beiden Portinstanzen existieren, welche sie verbindet. Eine dieser Eigenschaft widersprechende Konnektorinstanz würde einer Kante ohne Quelle oder Ziel (dangling edge) in einem Graphen gleichkommen (vgl. Abschnitt 2.1). Ebenso enthält ein gültiger Komponenteninstanzgraph keine Portinstanzen, welche nicht an einer Komponenteninstanz angebracht sind. Zusammen werden diese Eigenschaften als Wohlgeformtheit eines Komponenteninstanzgraphen bezeichnet, wie in Unterabschnitt 5.1.3 definiert.

Um diese kompositionalen Eigenschaften zu wahren und somit strukturell wohlgeformte Komponenteninstanzstrukturen zu erreichen, werden verschiedene Techniken eingesetzt. Um eine intuitive Modellierung zu gewährleisten, orientieren sich diese Techniken an den Semantiken der konventionellen Storypatterns. In diesen werden Links eines zu zerstörenden Objekts ebenfalls gelöscht, um nach der Regelausführung wieder einen gültigen Graph zu erhalten [Zün01, FNT98]. Des Weiteren werden im Storypattern modellierte Links zu zu instanziierenden Objekten ebenso erstellt, auch wenn sie keinen expliziten ≪create≫-Modifizierer besitzen. Dadurch ergibt sich insgesamt eine praktikable Modellierung von Storypatterns, welche sich am SPO-Ansatz orientiert und somit dangling edges vermeidet, vgl. auch Unterabschnitt 2.1.1. Diese Semantik wird für den komplexeren Kontext der Komponentenstorypatterns adaptiert, indem das Verhalten auf Komponenten-/Portinstanzen und Port-/Konnektorinstanzen übertragen wird.

Auf Ebene des Metamodells für Komponenteninstanzstrukturen werden bereits Kompositionen zwischen den entsprechenden Klassen eingesetzt, um die o. g. kompositionalen Beziehungen zu beschreiben und teilweise zu erhalten. Dadurch wird erreicht, dass die Zerstörung einer Komponenteninstanz die Zerstörung aller anliegenden Portinstanzen impliziert, und dass die Zerstörung einer Portinstanz die Zerstörung aller mit ihr verbundenen Konnektorinstanzen verursacht. Ein Beispiel hierfür wird in Abb. 6.21 dargestellt. Somit kann ein wohlgeformter Komponenteninstanzgraph nicht durch die Zerstörung einzelner Elemente in einen missgebildeten Graphen transformiert werden. Daher ist für Komponentenstorypatterns bzgl. der Zerstörung von Instanzen bereits eine Semantik gegeben, welche die der konventionellen Storypatterns simuliert.

Auf der Modellierungsebene für Komponentenstorypatterns wird durch die Benutzung der konkreten Syntax der Komponentendiagramme sichergestellt, dass Portvariablen an einer Komponentenvariablen angebracht werden und Konnektorlinks zwei Portvariablen verbinden. Dies impliziert, dass auch die entsprechenden Instanzen nicht autonom erstellt werden können. Dabei dürfen die einzelnen, miteinander verbundenen Elemente nicht unterschiedliche Modifizierer besitzen, da es ansonsten Widersprüchlichkeiten gibt. Eine Komponenteninstanz darf z. B. nicht zerstört werden, während eine anliegende Portinstanz erstellt wird. Wenn diese Eigenschaften erfüllt sind, dann kann aus einem

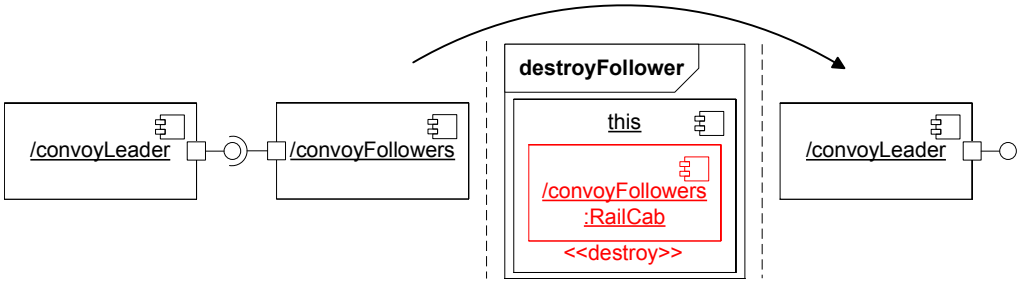


Abbildung 6.21: Implizite Zerstörung kompositional abhängiger Instanzen

wohlgeformten Komponenteninstanzgraph durch die Instanziierung einzelner Elemente nicht ein missgebildeter Graph entstehen.

Um die Semantik konventioneller Storypatterns bzgl. der Erstellung von Instanzen vollständig zu simulieren und somit die Modellierung praktikabel zu gestalten, werden Modifizierer von Komponenten- auf Portvariablen und von Portvariablen auf Konnektorlinks übernommen. Abb. 6.22 zeigt noch einmal das Storypattern createFirstPosCalc aus dem Komponentenstorydiagramm insertFollower in Abb. 6.1 aus dem letzten Abschnitt. Man beachte, dass lediglich ≪create≫-Modifizierer für newCalc und für die Portvariable von ∫ spezifiziert sind. Alle im Storypattern modellierten, von newCalc kompositional abhängigen Elemente wie die beiden anliegenden Portvariablen sowie die beiden Konnektorlinks müssten ebenfalls mit einem ≪create≫-Modifizierer versehen werden. Ein Binden der entsprechenden Instanzen macht keinen Sinn, da diese nicht existieren können. Um nicht jedem einzelnen dieser Elemente den gleichen Modifizierer zuordnen zu müssen, werden die Modifizierer auf sie übertragen. Der Portvariable von ∫ dagegen muss explizit ein Modifizierer zugeordnet werden, da sich der kompositionale Einfluss von newCalc lediglich bis zur Kompositionskonnektorinstanz auswirkt. Im Gegensatz zu den anderen angesprochenen Elementen kann deren Portinstanz nämlich bereits vorhanden sein. Mit dieser Vorgehensweise wird für Komponentenstorypatterns eine Semantik erreicht, die für Variablen mit ≪create≫-Modifizierer – analog zu in konventionellen Storypatterns modellierten Links, welche inzident zu Objektknoten mit ≪create≫-Modifizierer sind – die Instanzen für alle kompositional abhängigen Portvariablen und Konnektorlinks automatisch mit erstellt.

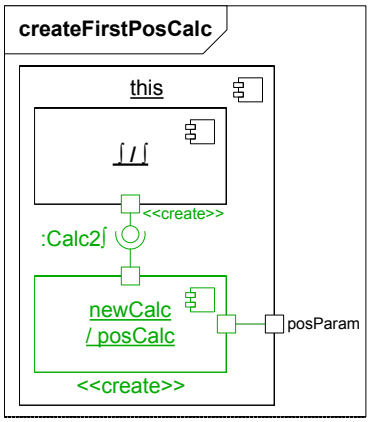


Abbildung 6.22: Modifizierer und kompositional abhängige Elemente

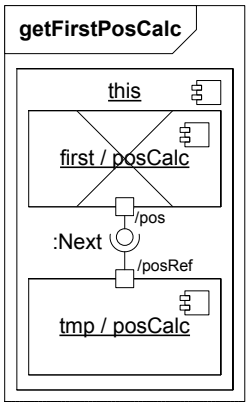


Abbildung 6.23: Anwendungsbedingungen und kompositional abhängige Elemente

Auch Anwendungsbedingungen für eine Variable haben Auswirkungen auf einen Bereich, welcher aufgespannt wird durch die kompositionalen Abhängigkeiten anderer Variablen oder Konnektorlinks von dieser Variable. Abb. 6.23 stellt noch einmal separat das Storypattern getFirstPosCalc dar, in dem eine negative Anwendungsbedingung für die Komponentenvariable first/posCalc verwendet wird. Diese Anwendungsbedingung muss, ebenso wie ein Modifizierer, auf die Portvariable /pos und von dieser auf den Konnektorlink :Next übertragen werden. Deckt die Anwendungsbedingung nicht diesen Bereich ab, dann würde nach einer Komponenteninstanzstruktur gesucht, welche aus tmp/posCalc, /posRef, :Next und /pos besteht und first nicht enthält. Diese Komponenteninstanzstruktur kann offensichtlich nicht existieren, da sie keinen wohlgeformten Komponenteninstanzgraph darstellt und, wie oben gezeigt, nicht erstellt werden kann. Die gewünschte Semantik des Storypatterns getFirstPosCalc ist also, dass tmp und deren Portvariable /posRef nur dann erfolgreich gebunden werden können, wenn diese nicht über einen Kompositionskonnektorlink :Next und eine Portvariable /pos mit einer Komponentenvariable first/posCalc verbunden sind.

Für eine optionale Anwendungsbedingung gilt analog, dass sie sich auf einen bestimmten Bereich auswirkt und somit auf kompositional abhängige Variablen übertragen werden muss. Falls first optional wäre und diese optionale Anwendungsbedingung nicht auf abhängige Variablen übertragen werden würde, wäre das Storypattern erfolgreich, wenn die modellierte Struktur komplett oder ohne first gebunden werden kann. Da :Next und /pos nicht ohne first existieren können, ist die Regel also nur dann erfolgreich, falls die Struktur komplett gebunden kann. Somit wäre in diesem Fall die Optionalität überflüssig. Das gewünschte Verhalten wird erreicht, wenn der Bereich first, /pos und :Next zusammenhängend als optional betrachtet wird.

6.3 Berücksichtigung von Multiplizitäten

In konventionellen Storypatterns werden bei der Erstellung von Links zu einem existierenden Objekt am Objekt eventuell bereits vorhandene Links derselben Assoziation ersetzt, falls die Assoziation eine oberen Multiplizitätsgrenze von 1 auf der Seite der Objektklasse besitzt. Dies verhindert resultierende Objektstrukturen, welche nicht der korrespondierenden Klassendefinition entsprechen [Zün01], siehe auch Unterabschnitt 2.1.1. Wie in Kapitel 5 vorgestellt, werden bei der neuen variablen Komponententypdefinition von MECHATRONIC UML ebenfalls Multiplizitäten eingesetzt. Diese ermöglichen es, beliebig viele Instanzen eines Parts oder Typs erstellen zu können. Bisher wurde allerdings

nicht berücksichtigt, wie Multiplizitäten mit einer oberen Grenze von z. B. 1 eingehalten werden sollen. Auch bzgl. dieser Herausforderung wird die Semantik der konventionellen Storypatterns als Vorbild genommen.

In dieser Arbeit werden für Multiplizitäten lediglich die oberen Grenzen 1 und $*$ ausgewertet. In konventionellen Storypatterns werden ebenfalls ausschließlich vorhandene Links ersetzt, deren Assoziationsmultiplizitäten eine obere Grenze von 1 besitzen. Untere Grenzen sowie konkrete Werte zwischen 1 und $*$ können zwar angegeben werden, werden allerdings nicht berücksichtigt. Genauer gesagt werden konkrete Werte zwischen 1 und $*$ in konventionellen Storydiagrammen auf $*$ abgebildet. Das beschriebene Verhalten wird in dieser Arbeit übernommen.

Bei herkömmlichen Storydiagrammen sorgt die Codegenerierung auf Metamodellebene für die Ersetzung von vorhandenen Links, welche über :1-Assoziationen typisiert sind. In Komponentenstorydiagrammen besitzen die Modellelemente Multiplizität, so dass von der Codegenerierung nicht profitiert werden kann. Eine Überprüfung während der Modellierung von Komponentenstorypatterns, dass nicht mehr Variablen/Konnektorlinks angelegt werden als die korrespondierenden Multiplizitäten erlauben, wäre unzureichend. Diese Überprüfung würde nämlich die multiple Erstellung von Instanzen eines :1-Parts oder -Typs lediglich für die einmalige Ausführung dieses einen Storypatterns verhindern. Auf andere Weise erstellte und somit bereits im Wirtsgraph vorhandene Instanzen würden nicht berücksichtigt werden. Daher wird für Komponentenstorydiagramme ein Ausführungsverhalten spezifiziert, welches wie in konventionellen Storydiagrammen bereits vorhandene Instanzen eines :1-Parts/-Typs ersetzt.

Bei Storydiagrammen auf Objektebene besitzen Assoziationen Multiplizität. Diese entsprechen in Komponentenstorydiagrammen den Konnektortypen, welche wie Assoziationen Quell- und Zielmultiplizitäten besitzen. Vorhandene Konnektorinstanzen müssen basierend auf diesen Multiplizitäten in bestimmten Fällen – ähnlich Objektbeziehungen – ersetzt werden. Zusätzlich besitzen Porttypen sowie Komponenten-Parts Multiplizität, so dass deren Instanzen ebenfalls bei einer oberen Grenze von 1 ersetzt werden müssen.

Zunächst wird die Methodik für die Ersetzung einer vorhandenen Komponenteninstanz vorgestellt, dessen Part eine Multiplizität mit einer oberen Grenze von 1 besitzt. Als Beispiel für ein solches Komponenten-Part wird convoyLeader:RailCab des Komponententyps ConvoySystem betrachtet, vgl. Abb. 5.11 in Unterabschnitt 5.1.2. Abb. 6.24 zeigt das Komponentenstorypattern createLeader, welches eine neue Komponenteninstanz newLeader/convoyLeader mit den entsprechenden Portinstanzen für die eingebettete Struktur erstellt. Dieses Storypattern wird z. B. im Systemkonstruktor des Evaluierungsbeispiels verwendet (Abb. 7.6 in Abschnitt 7.2.2). In dem Fall, dass bereits eine Komponenteninstanz /convoyLeader innerhalb derselben übergeordneten Komponenteninstanz :ConvoySystem existiert, wird zunächst /convoyLeader zerstört, und im Anschluss newLeader instanziiert. Besitzt die zu löschende /convoyLeader-Instanz Portinstanzen und diese wiederum Konnektorinstanzen, so werden diese bei einer Löschung dank der Verwendung von Kompositionen zwischen den korrespondierenden Metamodellklassen ebenfalls zerstört (siehe letzer Abschnitt).

Die beschriebene Methodik funktioniert nur dann, wenn bei der Modellierung eines Komponentenstorypatterns nicht mehr Komponentenvariablen angelegt werden, als die

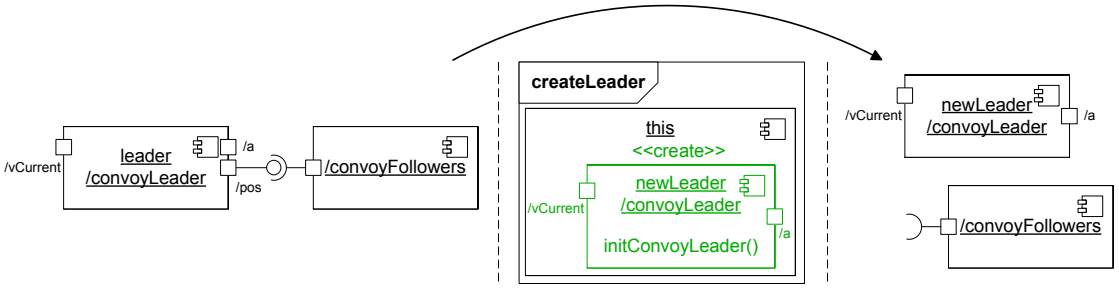


Abbildung 6.24: Ersetzung von Komponenteninstanzen aufgrund oberer Multiplizitätsgrenzen

Multiplizität des entsprechenden Komponenten-Parts erlaubt. Dies ist eigentlich ein Modellierungsfehler, muss allerdings trotzdem ausgeschlossen werden. Falls z. B. in createLeader eine zweite Komponentenvariable /convoyLeader ohne Modifizierer angelegt wird, wird eine evtl. bereits vorhandene Komponenteninstanz /convoyLeader gebunden und bleibt nach Ausführung des Storypatterns erhalten. Somit kann aufgrund der isomorphen Abbildung auf den Wirtsgraphen diese existierende Instanz nicht gelöscht werden, während die zu kreierende Instanz zusätzlich erstellt wird. Bei mehreren Komponentenvariablen /convoyLeader mit ≪create≫-Modifizierer können zwar bereits existierende Komponenteninstanzen zerstört werden, im Anschluss werden allerdings genauso viele instanziiert. Wenn diese Regel bei der Modellierung von Komponentenstorypatterns eingehalten wird, kann also garantiert werden, dass bereits existierende Komponenteninstanzen eines Parts mit einer oberen Multiplizitätsgrenze von 1 durch neu zu kreierende Komponenteninstanzen desselben Parts ersetzt werden.

Bei der Erstellung von Portinstanzen wird ein analoges Konzept verwendet. Hier muss anstatt des Parts der Porttyp auf die obere Multiplizitätsgrenze untersucht werden, und es muss sichergestellt werden, dass lediglich vorhandene Portinstanzen des gleichen Port-Parts an der gleichen Komponenteninstanz ersetzt werden. Im Fall einer Komponentenvariable /convoyLeader ohne Modifizierer würde also die Erstellung einer neuen Portinstanz /a die vorherige Zerstörung einer bereits an /convoyLeader angebrachten Portinstanz /a bewirken. Offensichtlich ist ebenfalls auf die Einhaltung der maximal erlaubten Portvariablen zu achten.

Bei der Ersetzung von Konnektorinstanzen muss zwischen Quell- und Zielportinstanzen und der Quell- bzw. Zielmultiplizität des korrespondierenden Konnektortyps unterschieden werden. Abb. 6.25 zeigt eine vereinfachte Variante des Storypatterns addPosCalc aus dem Komponentenstorydiagramm insertPosCalc in Abb. 6.1 aus Abschnitt 6.1. Dieses Storypattern fügt eine neue Komponenteninstanz newCalc/posCalc zwischen zwei vorhandene /posCalc-Instanzen ein. Dazu werden neben newCalc an den Portinstanzen der existierenden /posCalc-Instanzen neue Kompositionskonnektorinstanzen :Next erzeugt. Die Quell- bzw. Zielmultiplizitäten des Konnektortyps geben das mengenmäßige Verhältnis an, in dem Quell- und Zielelemente miteinander verbunden werden dürfen (vgl. Unterabschnitt 5.1.2). Dies bedeutet für Instanzen des Konnektortyps Next mit Quell-

und Zielmultiplizität von 1:1, dass sie exakt eine Quellportinstanz mit einer Zielportinstanz verbinden dürfen. Somit müssen also evtl. vorhandene Instanzen :Next an den zu erhaltenden Portinstanzen von prev oder next in der linken Seite von Abb. 6.25 ersetzt werden, wenn die Regel addPosCalc auf diese Instanzstruktur angewendet wird.

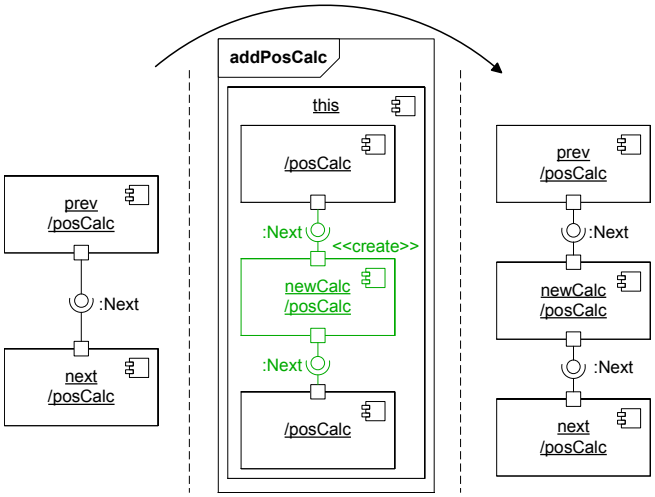


Abbildung 6.25: Ersetzung von Kompositionskonnektorinstanzen aufgrund oberer Multiplizitätsgrenzen

Auch für Kompositionskonnektoren dürfen nicht mehr Links verbunden werden als durch die Multiplizitäten des Konnektortyps erlaubt. In addPosCalc darf z. B. kein weiterer Link :Next zwischen den beiden zu erhaltenden /posCalc-Instanzen modelliert werden. Dieser Konnektorlink würde nach dem Einfügen von newCalc erhalten bleiben. Deshalb werden im Storydiagramm insertPosCalc in Abb. 6.1 aus Abschnitt 6.1 die Komponenteninstanzen tmp und next in einem eigens dafür vorgesehenen Storypattern gebunden. Das Einfügen der neuen Komponenteninstanz newCalc in die /posCalc-Liste geschieht in separaten Storypatterns, um das Ersetzen der :Next-Instanzen zu ermöglichen.

Bei Delegationskonnektorinstanzen wird analog verfahren. Soll eine Konnektorinstanz zwischen zwei zu erhaltenden Portinstanzen[2] kreiert werden, so werden sowohl Quell- als auch Zielmultiplizität berücksichtigt.

Durch die beschriebene Ersetzung von Instanzen kann für eine obere Multiplizitätsgrenze von 1 garantiert werden, dass ein hinsichtlich der Multiplizität typkonformer Komponenteninstanzgraph (vgl. Definition in Unterabschnitt 5.1.3) durch die Anwendung eines Komponentenstorypatterns diese Konformität nicht verliert. Dabei gilt als Voraussetzung, dass das Storypattern nicht mehr Variablen oder Konnektorlinks enthält, wie es durch die obere Multiplizitätsgrenze erlaubt ist. Andere konkrete obere Grenzen als 1 können nicht garantiert werden.

[2]Die externe Portinstanz der aktuellen, umgebenden Komponenteninstanz kann nicht modifiziert werden und bleibt somit bei der Ausführung eines Komponentenstorypattern immer erhalten.

6.4 Metamodell für Komponentenstorydiagramme

Nachdem in Abschnitt 6.1 die Syntax von Komponentenstorydiagrammen informal vorgestellt wurde, wird diese im folgenden mit dem Metamodell der einzelnen syntaktischen Konstrukte formalisiert. Abb. 6.26 zeigt die Zuordnung zu Komponententypen und die Verwandtschaft mit konventionellen Storydiagrammen als Einstiegspunkt.

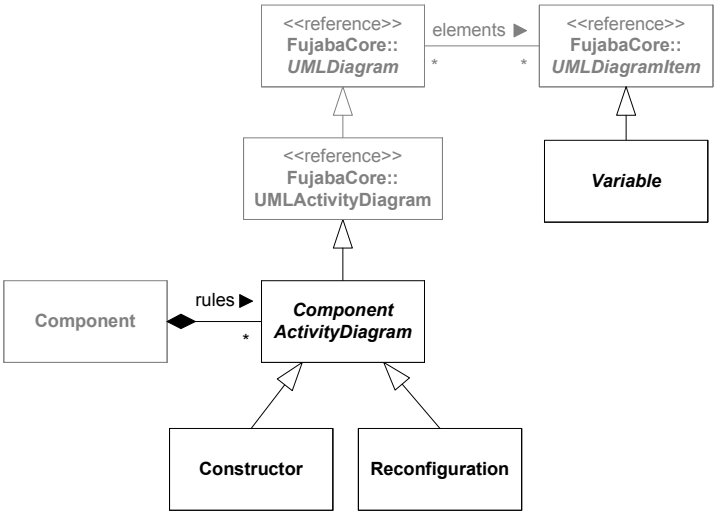


Abbildung 6.26: Metamodell Komponentenstorydiagramme – Zuordnung zu Komponententypen und Verwandtschaft mit konventionellen Storydiagrammen

Die Klasse UMLActivityDiagram aus dem FUJABA-Kern repräsentiert ein Aktivitätsdiagramm und insbesondere auch ein herkömmliches Storydiagramm. Über die Assoziation elements zu UMLDiagramItem werden Elemente wie die verschiedenen Arten von Aktivitäten und Kontrollstrukturen in Storydiagramme eingebettet. Diese Klassen werden zur Modellierung der in Unterabschnitt 6.1.2 vorgestellten Kontrollstrukturen von Komponentenstorydiagrammen wiederverwendet und werden an dieser Stelle aus Platzgründen nicht vorgestellt. Als Referenz sei auf das Metamodell von Aktivitätsdiagrammen in Abschnitt 4.4 von [FNT98] verwiesen.

Eine neue, von UMLDiagramItem erbende Klasse ist dagegen Variable, die abstrakte Oberklasse aller in Komponentenstorypatterns verwendeten Variablen und Konnektorlinks. Diese wird nachfolgend näher erklärt. Die neue Klasse ComponentActivityDiagram stellt ein Komponentenstorydiagramm dar und erweitert UMLActivityDiagram. Somit können Komponentenstorydiagramme die Kontrollstrukturen der konventionellen Storydiagramme über die Assoziation elements wiederverwenden. Die Unterscheidung in Konstruktoren und Rekonfigurationsregeln wird über die erbenden Klassen Constructor und Reconfiguration vorgenommen. Die Zuordnung zu Komponententypen erfolgt über die neue Komposition rules.

Abb. 6.27 stellt das Metamodell für Variablen und Konnektorlinks vor. Die Klasse Variable besitzt neben dem Namensattribut die Attribute modifier für die Zuordnung eines

Modifizierers und type für die Zuordnung einer negativen oder optionalen Anwendungsbedingung. Modifizierer und Anwendungsbedingungen werden dabei über Konstanten spezifiziert. Eine durch ComponentVariable repräsentierte this-Variable kann eingebettete Komponentenvariablen über die Komposition contains besitzen. Ein Komponentenvariable besitzt außerdem beliebig viele Portvariablen, was durch die Komposition has zu PortVariable dargestellt wird. Portvariablen wiederum besitzen eine beliebige Anzahl ein- und ausgehender Konnektorlinks, was die Kompositionen in bzw. out zur abstrakten Klasse ConnectorLink ausdrücken. Konnektorlinks werden über die Subklassen DelegationLink und AssemblyLink in Delegations- bzw. Kompositionskonnektorlinks unterschieden. Mit einer Instanz von AssemblyLink werden zusätzlich Schnittstellenvariablen in Form von Instanzen der von InterfaceVariable ableitenden Klassen verbunden. Diese stellen bei der Modellierung von Komponentenstorypatterns keine eigenständigen Modellelemente dar und werden lediglich für das Mapping auf Transformationsdiagramme benötigt. Daher werden diese über die Kompositionen in bzw. out beim Löschen einer Kompositionskonnektorinstanz automatisch gelöscht.

Konnektorlinks werden über die Kompositionen type den korrespondierenden Konnektortypen und Variablen über die Kompositionen part den korrespondierenden Parts zugeordnet und somit typisiert. Zusätzlich besitzt PortVariable eine Assoziation type zu Port, welche verwendet wird, falls die repräsentierte Portvariable zur this-Variable gehört. In diesem Fall ist, wie in Unterabschnitt 6.1.1 definiert, die Portvariable über einen Porttyp klassifiziert. Die Information des Komponententypen der this-Variable kann dagegen über die Komposition rules zwischen Component und dem übergeordneten ComponentActivityDiagram ermittelt werden (vgl. Abb. 6.26), somit ist keine zusätzliche Assoziation type zwischen ComponentVariable und Component notwendig.

Abb. 6.28 zeigt den für Transformationsaufrufe relevanten Ausschnitt des Metamodells und visualisiert somit den Zusammenhang zwischen Storydiagrammen von Komponententypen verschiedener Hierarchieebenen. Komponentenstorydiagramme besitzen Parameter und Rückgabedeklarationen, was über die geordneten Assoziationen params/ returnDeclarations zu den Klassen Param bzw. ReturnDeclaration ausgedrückt wird. Diese erweitern die Klasse ComponentTransformationDeclaration, welche ein Namensattribut sowie eine unidirektionale Assoziation type zu UMLType aus dem Fujaba-Kern besitzt. Die Klasse UMLType stellt dabei eine Schnittstelle auf einen in einem Fujaba-Projekt definierten Typ dar, welche für das Mapping auf Transformationsdiagramme benötigt wird. Die Unterteilung von Parametern in primitive und komplexe Parameter wird über die von Param erbenden Klassen PrimitiveParam und ComplexParam vorgenommen. Primitive Parameter werden direkt über UMLType klassifiziert. Komplexe Parameter und die Rückgabedeklarationen ReturnDeclaration dagegen werden über die Klasse Part oder Port typisiert, je nachdem ob sie eine Portvariable der this-Variable oder ein eingebettetes Element referenzieren. Bei komplexen Parametern und Rückgabedeklarationen wird UMLType mit zusätzlichen Typinformationen für das Mapping gefüllt.

Ein Komponentenstorydiagramm kann über eine Instanz der Klasse ComponentActivityDiagramCall aufgerufen werden. Ein solcher Aufruf wird für eine ComponentVariable über einen Link der Assoziation uses spezifiziert. Komplexe und primitive Argumente werden über die jeweiligen Assoziationen declaration mit den entsprechenden Parameter-

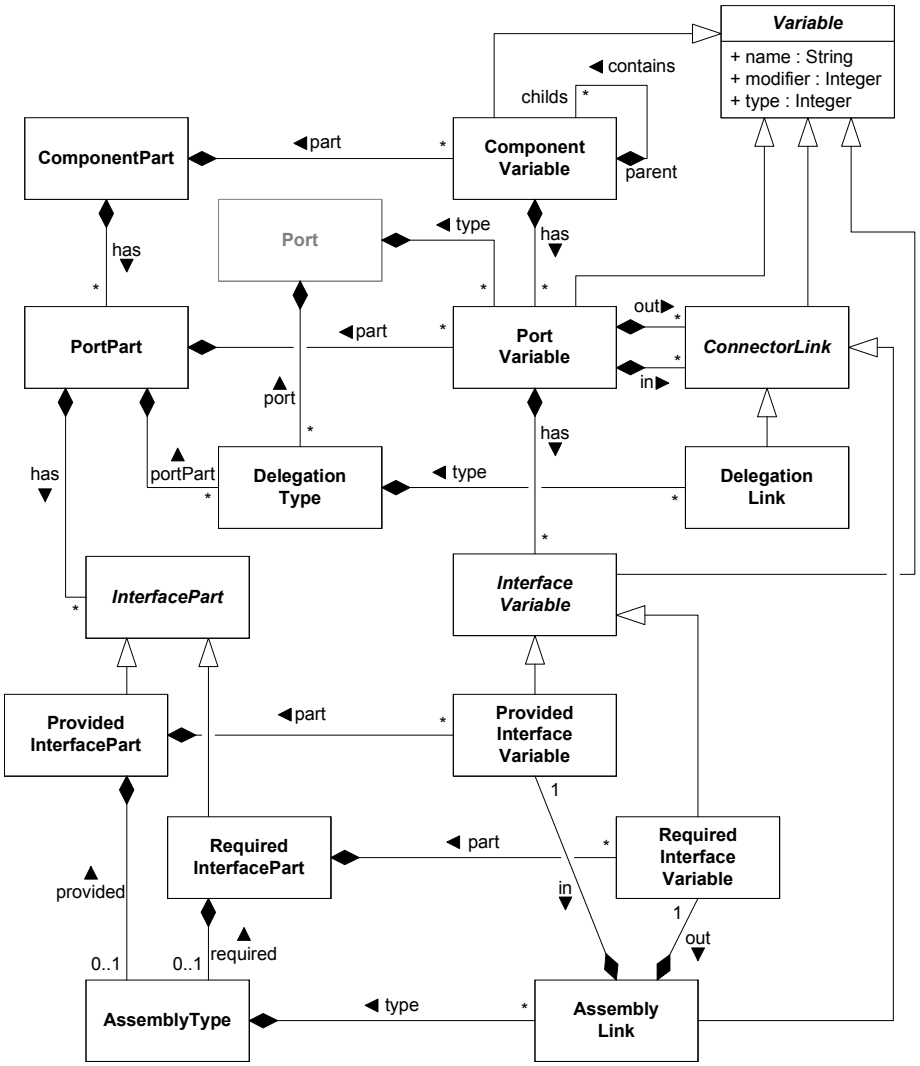


Abbildung 6.27: Metamodell Komponentenstorydiagramme – Variablen und Konnektorlinks

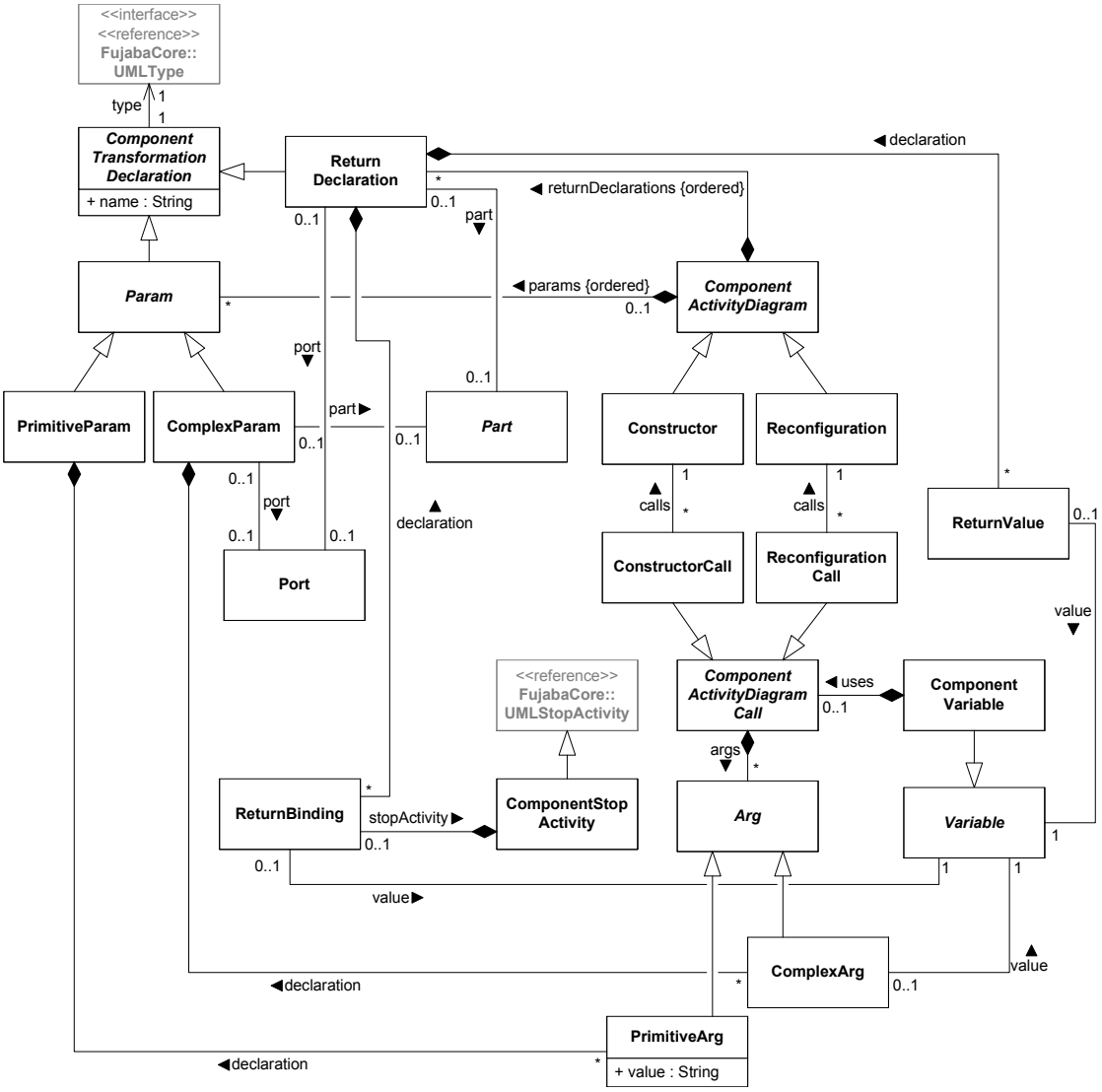


Abbildung 6.28: Metamodell Komponentenstorydiagramme – Aufruf von Komponentenstorydiagrammen

deklarationen verbunden. Ein primitives Argument speichert seinen Wert direkt in dem Attribut value. Ein komplexes Argument benutzt hierfür einen Link value auf eine Variable innerhalb des aufrufenden Storydiagramms, welche dem Typ des korrespondierenden Parameters entsprechen muss.

Der Wert einer Rückgabedeklaration im aufgerufenen Komponentenstorydiagramm wird über eine Instanz von ReturnBinding gebunden. Über stopActivity wird ein solches Objekt einer Instanz von ComponentStopActivity zugeordnet, einer Subklasse von UMLStopActivity. Diese Klasse repräsentiert Stop-Aktivitäten konventioneller Storydiagramme. Das eigentliche Binden geschieht über einen Link value auf eine Instanz von Variable. Die Benutzung eines Rückgabewerts im aufrufenden Storydiagramm dagegen geschieht über Objekte der Klasse ReturnValue. Ein solcher Rückgabewert wird über den Link value einer typgleichen Variable zugewiesen. Die Zuordnung von Bindungen und Rückgabewerten zu einer Rückgabedeklaration erfolgt über die jeweiligen Assoziationen declaration.

6.5 Abbildung auf Transformationsdiagramme

Um auf einem Komponententypmodell definierte Komponentenstorydiagramme auszuführen, werden diese abgebildet auf Transformationsdiagramme auf dem Komponentenmetamodell, um die vorhandene Implementierung zur Ausführung von Transformationsdiagrammen wiederzuverwenden. Das Mapping definiert dabei gleichzeitig die Semantik der Komponentenstorydiagramme.

Abb. 6.29 skizziert eine Übersicht über die verschiedenen Abstraktionsebenen, auf denen in dieser Arbeit operiert wird. Auf der linken Seite sind Ausschnitte von Modellen in der konkreten Syntax der Komponentendiagramme von MECHATRONIC UML abgebildet. Die Modelle der rechten Seite zeigt die entsprechende Repräsentation in der abstrakten Syntax, welche auf dem Komponentenmetamodell basiert. Das Komponentenstorydiagramm der linken Seite ist auf einem Komponententypmodell und somit auf der konkreten Syntax der Komponentendiagramme definiert. Das Transformationsdiagramm ist auf dem Komponentenmetamodell und somit auf der abstrakten Syntax der Komponentendiagramme definiert, ist allerdings formuliert in der konkreten Syntax von Transformationsdiagrammen. Diese Unterscheidung ist für das Mapping relevant. In den Modellen der linken Seite sind einige Teile ausgegraut, deren Repräsentation in der abstrakten Syntax aus Platzgründen nicht abgebildet ist.

Das obere Drittel zeigt das Komponenteninstanzmodell oder die Konfiguration G, auf das die Transformationsregel p im mittleren Drittel angewandt wird und somit das Modell bzw. die Konfiguration H im unteren Drittel generiert. Analog zu klassischen Graphtransformationen (vgl. Abschnitt 2.1), sei diese Anwendung die direkte Ableitung $G \overset{p,m}{\Longrightarrow} H$ eines Komponenteninstanz- bzw. eines Objektgraphen G genannt, wobei p aus Komponentenstorydiagrammen bzw. aus Transformationsdiagrammen besteht[3] und m

[3]Die Transformation p umfasst mehrere Komponentenstory-/Transformationsdiagramme, wenn sie mehrere Ebenen der Komponentenhierarchie beeinflusst und somit weitere Komponentenstory-/Transformationsdiagramme aufgerufen werden.

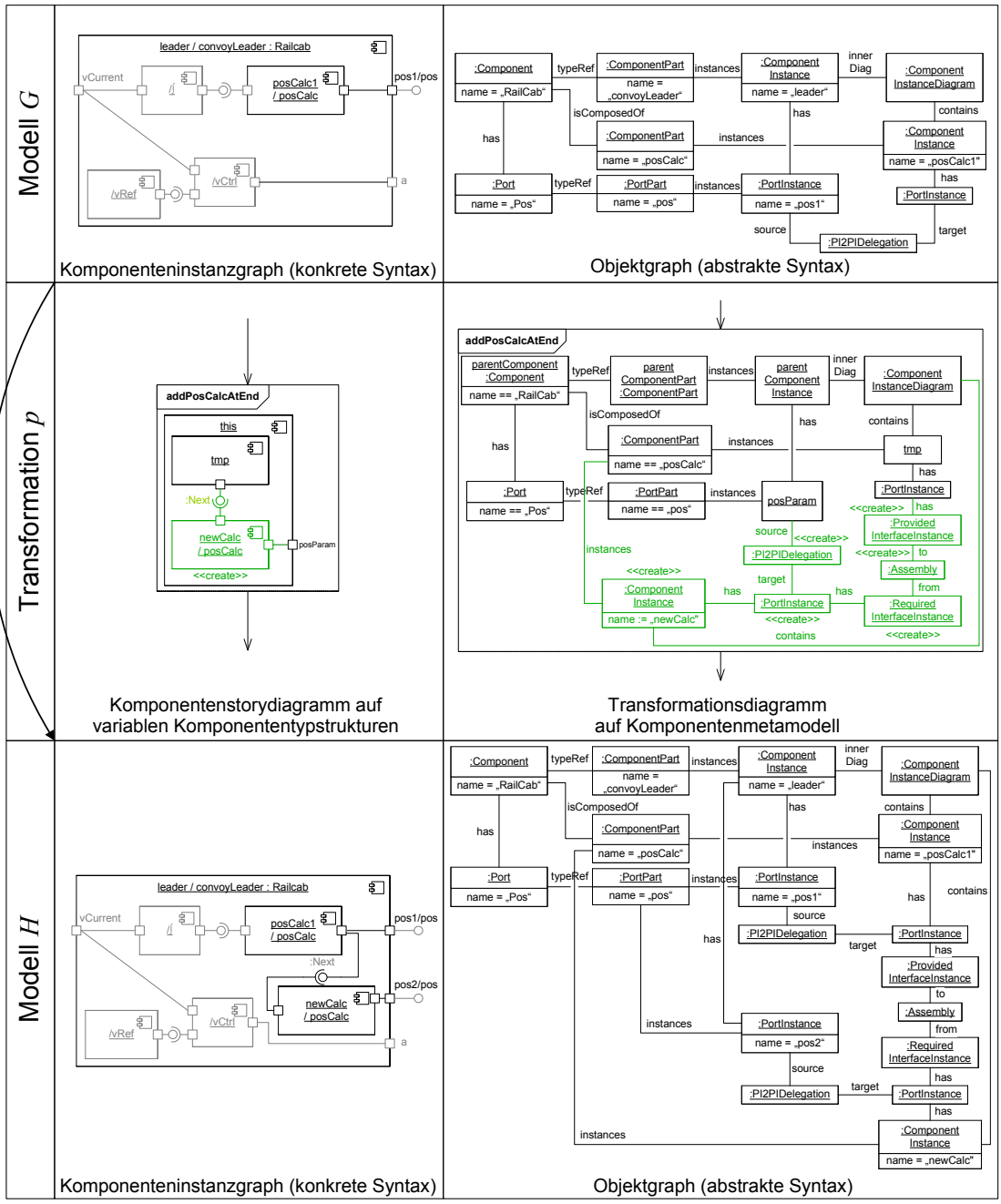


Abbildung 6.29: Übersicht der verschiedenen Abstraktionsebenen [THHO08]

die Abbildungsfunktion der Storypatterns auf den jeweiligen Wirtsgraphen ist. Während Komponenteninstanzgraphen über ihre Komponententypdefinition bzw. ein Komponententypmodell klassifiziert sind, werden die zu Grunde liegenden Objektgraphen über das Komponentenmetamodell von MECHATRONIC UML typisiert. Analog sind Komponentenstorydiagramme auf Komponententypen und ihre übersetzten Transformationsdiagramme auf den korrespondierenden Metamodellklassen definiert.

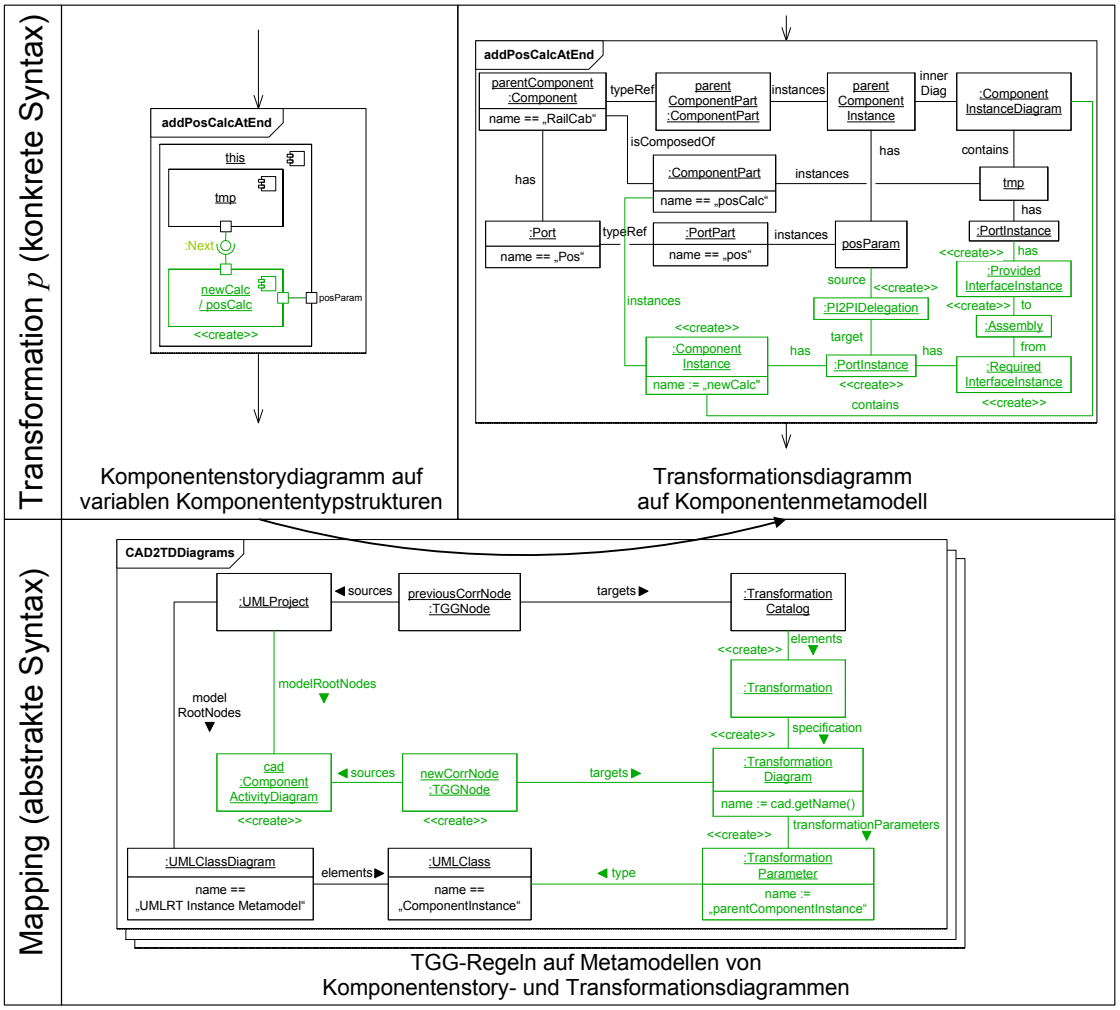


Abbildung 6.30: Übersicht der Abstraktionsebenen des Mappings

Der konkrete Vorgang des Mappings der Komponentenstorydiagramme auf Transformationsdiagramme wird durch Abb. 6.30 visualisiert. In der oberen Hälfte werden wieder die Ausschnitte des Komponentenstorydiagramms und des übersetzten Transformationsdiagramms aus Abb. 6.29 gezeigt, welche wie bereits erwähnt in ihrer jeweiligen konkreten Syntax formuliert sind. Das Mapping ist in der unteren Hälfte in Form einer TGG-Regel abgebildet. Diese TGG-Regel steht exemplarisch für mehrere TGG-Regeln, welche zu-

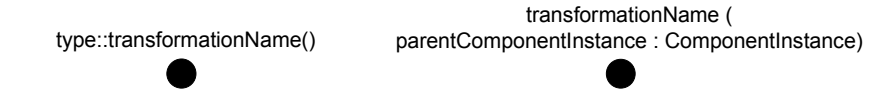
sammen das Mapping ausmachen, was durch den dreifachen Rahmen angedeutet wird. Die TGG-Regeln werden auf den jeweiligen Metamodellen von Komponentenstory- und Transformationsdiagrammen somit auf deren abstrakten Syntax definiert. Formuliert sind sie wiederum in ihrer eigenen konkreten Syntax.

Für jedes Komponentenstorydiagramm wird ein Transformationsdiagramm mit gleichem Namen generiert. Darauf basierend werden die Inhalte des Komponentenstorydiagramms in entsprechende Konstrukte des korrespondierenden Transformationsdiagramms übersetzt. Das Mapping für diese Übersetzung wird im Folgenden vorgestellt.

6.5.1 Kontrollstrukturen

Dieser Unterabschnitt stellt die zumeist trivialen Mappings der einzelnen Kontrollstrukturen von Komponentenstorydiagrammen auf Kontrollstrukturen der Transformationsdiagramme vor.

Abb. 6.31 zeigt das Mapping einer Startaktivität eines Komponentenstorydiagramms ohne Parameter und ohne Rückgabedeklarationen auf eine Startaktivität eines Transformationsdiagramms. Jedem Transformationsdiagramm wird ein Parameter parentComponentInstance:ComponentInstance zugeordnet. Dieser Parameter ist eine Referenz auf das Objekt der aktuellen Komponenteninstanz, dessen eingebettete Struktur transformiert werden soll. In den Komponentenstorypatterns wird diese Komponenteninstanz durch die this-Variable repräsentiert. Die Abbildung von Parametern und Rückgabedeklarationen wird in Unterabschnitt 6.5.4 vorgestellt.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.31: Mapping von Startaktivitäten

Abb. visualisiert 6.32 das triviale Mapping einer Stopaktivität ohne die Bindung von Rückgabewerten, deren Mapping in Unterabschnitt 6.5.4 betrachtet wird. Die übrigen, ebenfalls trivialen Mappings von NOP-Aktivitäten, Statement-Aktivitäten, Storypatterns, forEach-Storypatterns und der verbindenden Transitionen mit eventuell vorhandenen Guards sind in den Abbildungen 6.33 bis 6.37 dargestellt.

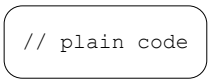
(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.32: Mapping von Stopaktivitäten

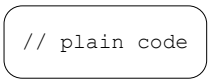
(a) Komponentenstorydiagramm

(b) Transformationsdiagramm

Abbildung 6.33: Mapping von NOP-Aktivitäten

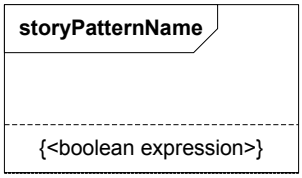


(a) Komponentenstorydiagramm

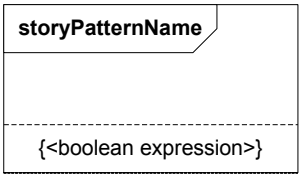


(b) Transformationsdiagramm

Abbildung 6.34: Mapping von Statement-Aktivitäten

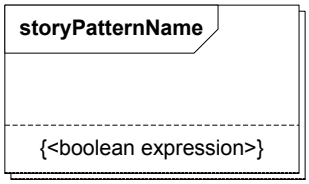


(a) Komponentenstorydiagramm

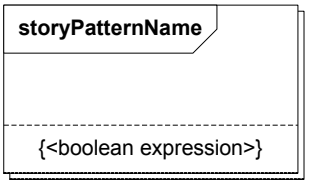


(b) Transformationsdiagramm

Abbildung 6.35: Mapping von Storypatterns

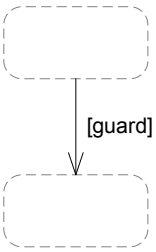


(a) Komponentenstorydiagramm

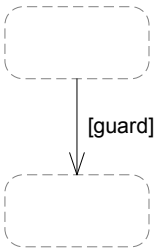


(b) Transformationsdiagramm

Abbildung 6.36: Mapping von forEach-Aktivitäten



(a) Komponentenstorydiagramm



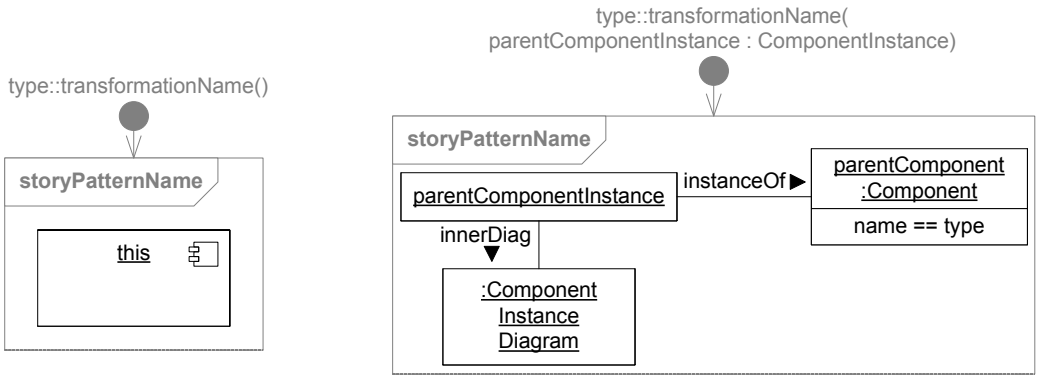
(b) Transformationsdiagramm

Abbildung 6.37: Mapping von Transitionen

6.5.2 Variablen und Konnektorlinks

In diesem Unterabschnitt werden die Mappings der einzelnen Elemente von Komponentenstorypatterns auf die Elemente von Storypatterns der Transformationsdiagramme erläutert. Die Storypatterns der Transformationsdiagramme sind dabei auf dem Komponentenmetamodell definiert, siehe Unterabschnitt 5.2.

this-Variable der Systemebene Abb. 6.38 zeigt das Mapping für eine this-Variable, welche eine Instanz der Systemebene repräsentiert. Der klassifizierende Komponententyp type ist also mit dem Stereotyp ≪system≫ ausgestattet. Diese Information wird beim Mapping ausgewertet, um zwischen Systemkomponententypen an der Spitze und Komponententypen innerhalb der Hierarchie zu unterscheiden. Systemkomponententypen werden nicht als Parts in andere Komponententypen eingebettet und werden daher nicht über ein Part, sondern direkt instanziiert. Die als gebunden markierte Objektvariable parentComponentInstance wird als Einstiegspunkt verwendet. Dieses greift auf den gleichnamigen Parameter zu, welcher die aktuelle Komponenteninstanz des Systems darstellt und in jedem Storypattern des Transformationsdiagramms verfügbar ist. Die Objektvariable parentComponentInstance entspricht somit der this-Variable im Komponentenstorypattern. Das Objekt parentComponent:Component, welches den Komponententyp type repräsentiert, wird anhand seines Namens über den Link instanceOf gebunden. Dieser Link repräsentiert die direkte Instanziierung des Systemkomponententypen. Des Weiteren ist ein Objekt :ComponentInstanceDiagram zu binden, welches die Einbettung weiterer Komponenteninstanzen in die aktuelle Komponenteninstanz ausdrückt.

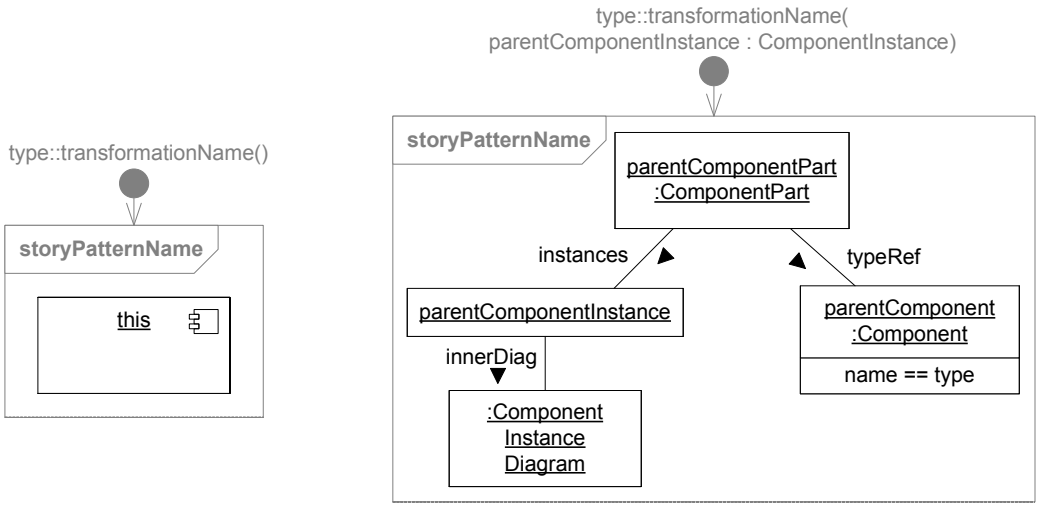


(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.38: Mapping einer this-Variable für die Systemebene

this-Variable eines Komponententypen innerhalb Hierarchie Abb. 6.39 zeigt das Mapping der this-Variable zur Repräsentation von Instanzen eines Komponententypen type, welcher sich innerhalb der Komponentenhierarchie befindet. Mindestens ein Part von type ist also in einen anderen Komponententypen eingebettet; die aktuelle Komponentenin-

stanz wurde über dieses Part instanziiert. Diese Typisierungsinformation drückt sich auf Seite des Transformationsdiagramms über die Objektvariablen parentComponentPart und parentComponent aus. Die Variable parentComponent wird wieder über den Namen des Komponententypen type gebunden, während der Name des typisierenden Parts irrelevant ist.

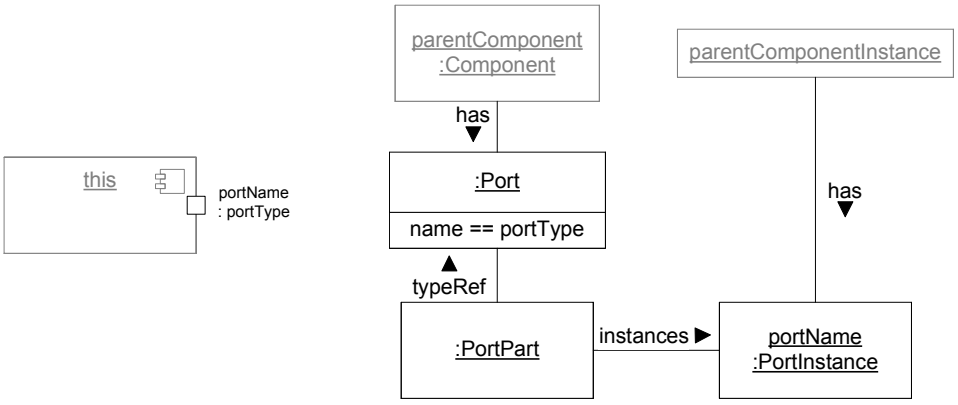


(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.39: Mapping der this-Variable eines Komponententypen innerhalb der Hierarchie

Portvariablen der this-Variable Das Mapping einer externen, an der this-Variable angebrachten Portvariable wird durch Abb. 6.40 visualisiert. Die Portvariable wird auf die Objektvariable portName:PortInstance abgebildet. Die Übernahme des Namens der Portvariable als Objektname ermöglicht den Zugriff auf gleich benannte Parameter. Die Portinstanz wurde, eine Hierarchieebene höher, über ein Port-Part kreiert, was durch die Verbindung von portName mit der Objektvariable :PortPart ausgedrückt wird. Die Portvariable ist durch den Porttyp portType klassifiziert. Dies wird auf Ebene des Transformationsdiagramms durch die Objektvariable :Port mit der entsprechenden Attributbedingung repräsentiert.

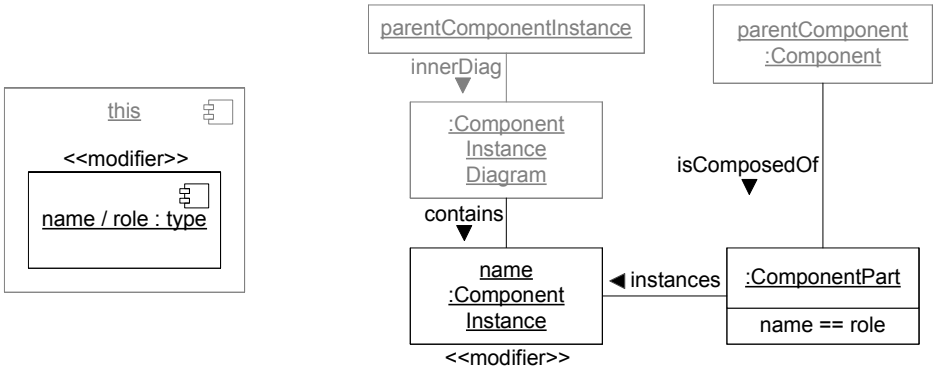
Eingebettete Komponentenvariablen Abb. 6.41 stellt das Mapping für eingebettete Komponentenvariablen vor. Diese Komponentenvariablen sind typisiert über ein eingebettetes Part role, welches auf Objektebene durch :ComponentPart repräsentiert wird. Der übergeordnete Komponententyp kann diverse Komponenten-Parts enthalten kann, daher wird :ComponentPart über den Namen qualifiziert. Da von einem Multipart mehrere Komponenteninstanzen und somit auch Komponentenvariablen existieren können, wird für eine Komponentenvariable im Storypattern des Transformationsdiagramms nur dann



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.40: Mapping einer an der this-Variable angebrachten Portvariable

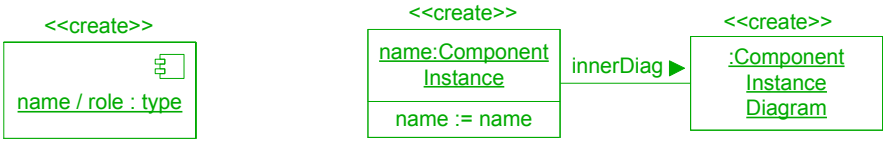
ein neue Objektvariable :ComponentPart generiert, falls diese nicht bereits existiert. Für jede Kombination aus Komponentenvariable und Komponenten-Part wird also genau eine Objektvariable :ComponentPart angelegt. Die Information über den Komponententyp, dessen Rolle das Komponenten-Part spielt, ist bereits über das Komponenten-Part selber gegeben und muss nicht in Form einer Objektvariable im Storypattern modelliert werden. Die eigentliche Komponentenvariable wird auf name:ComponentInstance abgebildet. Die Einbettung in die übergeordnete Komponenteninstanz wird über die Objektbeziehung contains zwischen :ComponentInstanceDiagram und name ausgedrückt. Ein eventuell vorhandener Modifizierer ≪modifier≫ wird von der Komponentenvariable übernommen. Falls die Komponentenvariable als gebunden markiert ist, so wird auch die korrespondierende Objektvariable als gebunden markiert, was in der Abbildung nicht dargestellt ist.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.41: Mapping einer eingebetteten Komponentenvariable

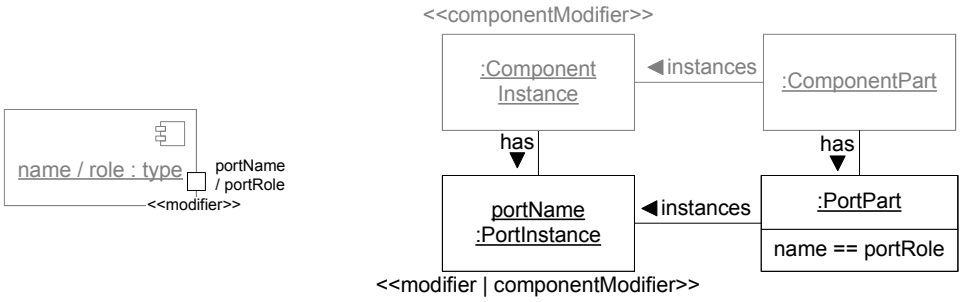
Falls eine Komponenteninstanz erstellt werden soll, so wird zum Einen ihr Namensattribut besetzt, zum Anderen muss ihr Komponenteninstanzdiagramm zur Einbettung weiterer Komponenteninstanzen kreiert werden. Das zusätzliche Mapping hierfür findet sich, ergänzend zu Abb. 6.41, in Abb. 6.42. Das Objekt :ComponentInstanceDiagram ist erforderlich, da es in tieferen Hierarchieebenen gebunden wird, wie in Abb. 6.39 zu sehen ist.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.42: Mapping einer Komponentenvariable mit ≪create≫-Modifizierer

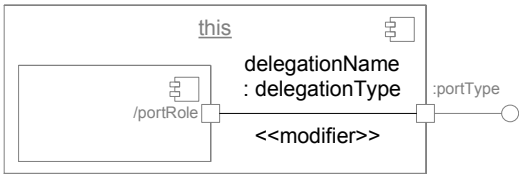
Eingebettete Portvariablen Abb. 6.43 zeigt das Mapping für eine eingebettete Portvariable. Diese wird auf eine Objektvariable portName:PortInstance abgebildet. Die Information über das Port-Part wird auf die Variable :PortPart abgebildet, welche analog zu den Objektvariablen für Komponenten-Parts für jede Kombination von Portvariable und Port-Part nur einmal generiert wird. Die Zugehörigkeit einer Portinstanz zu einer Komponenteninstanz wird über die Objektbeziehung has zwischen :ComponentInstance und portName ausgedrückt. Ist die Portvariable gebunden, so wird ebenso die Objektvariable portName als gebunden markiert. Des Weiteren übernimmt die Objektvariable einen möglichen Modifizierer ≪modifier≫ der Portvariable, verknüpft diesen allerdings zusätzlich mit dem evtl. vorhandenen Modifizierer ≪componentModifier≫ von :ComponentInstance. Dadurch wird die in Abschnitt 6.2 vorgestellte Semantik bzgl. der Übernahme der Modifizierer aufgrund kompositionaler Abhängigkeiten technisch umgesetzt. Im Falle eines ≪create≫-Modifizierers wird analog zu Komponenteninstanzen das Attribut name der Portinstanz mit dem Wert portName belegt.



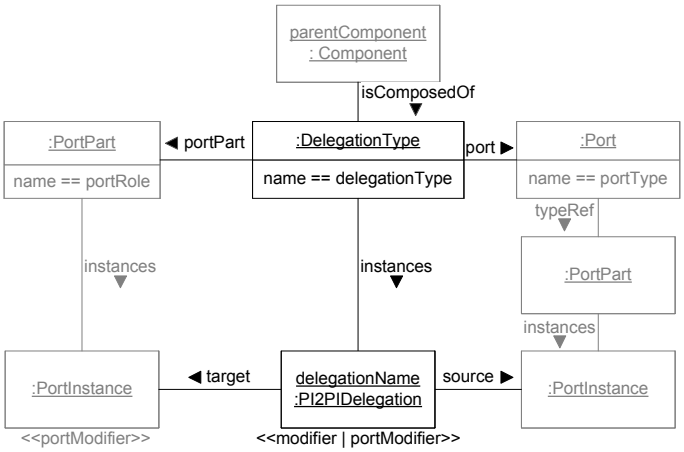
(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.43: Mapping einer eingebetteten Portvariable

Delegationskonnektorlinks Das Mapping für Delegationskonnektorlinks ist in Abb. 6.44 visualisiert. In der abgebildeten Situation delegiert der Konnektor Aufrufe einer angebotenen Schnittstelle, also fließen Informationen von der externen Portinstanz :portType zur eingebetteten Portinstanz /portRole. Die diese Portinstanzen repräsentierenden Objekte sind dementsprechend über die Links source bzw. target mit dem Objekt delegationName:PI2PIDelegation verbunden, welches die Delegationskonnektorinstanz repräsentiert. Delegiert der Konnektor Aufrufe einer benötigten Schnittstelle, sind die Rollen dieser Objekte vertauscht. Der Delegationskonnektortyp wird durch das Objekt :DelegationType dargestellt, welches mit den Objekten für ein Port-Part und einen Porttyp verbunden ist. Im speziellen Fall des Delegationskonnektorlinks muss lediglich der evtl. vorhandene Modifizierer ≪portModifier≫ der Objektvariable :PortInstance übernommen werden, welche die eingebettete Portinstanz /portRole repräsentiert. Die externe Portinstanz :portType kann nicht durch das gleiche Komponentenstorydiagramm modifiziert werden. Man beachte, dass der Modifizierer ≪portModifier≫ auf Objektvariablenebene übernommen wird. Das bedeutet, dass dieser auch den möglichen Modifizierer ≪componentModifier≫ aus Abb. 6.43 beinhaltet. Dieser wurde beim Mapping von /portRole von der Objektvariable übernommen, welche die Komponenteninstanz darstellt.



(a) Komponentenstorydiagramm

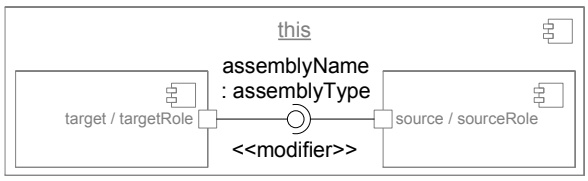


(b) Transformationsdiagramm

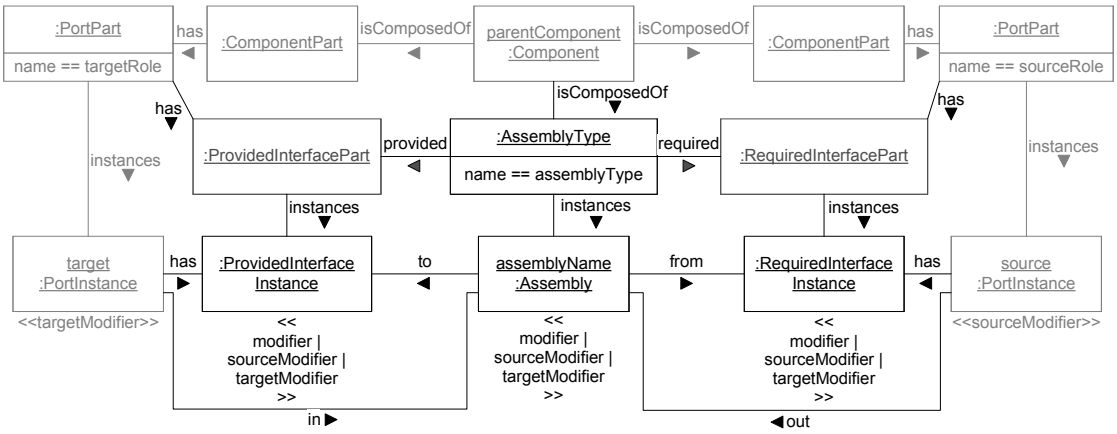
Abbildung 6.44: Mapping eines Delegationskonnektorlinks

Kompositionskonnektorlinks Abb. 6.45 zeigt das Mapping für Kompositionskonnektorlinks. Im Gegensatz zu den bisher vorgestellten Strukturelementen besteht eine Kom-

positionskonnektorinstanz aus multiplen Objekten. Sie verbindet Instanzen einer angebotenen und einer benötigten Schnittstelle, somit werden sie durch die Objekte :ProvidedInterfaceInstance, assemblyName:Assembly und :RequiredInterfaceInstance dargestellt. Deren Objektvariablen entsprechen zusammenhängend einem Kompositionskonnektorlink im Komponentenstorypattern. Die Typisierungsinformationen der einzelnen Elemente der Kompositionskonnektorinstanz wird durch die Objekte :ProvidedInterfacePart, :AssemblyType und :RequiredInterfacePart ausgedrückt. Bei diesem Konnektorlink-Mapping müssen die möglichen Modifizierer beider Portvariablen zusammenhängend auf die drei verschiedenen Objekte übernommen werden, da sowohl Quell- als auch Zielportinstanzen sowie ihre jeweiligen Komponenteninstanzen modifiziert werden können.



(a) Komponentenstorydiagramm

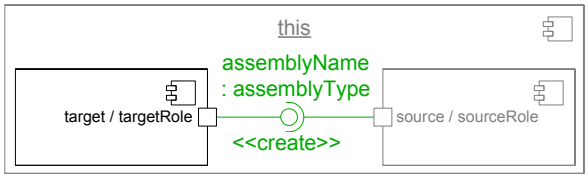


(b) Transformationsdiagramm

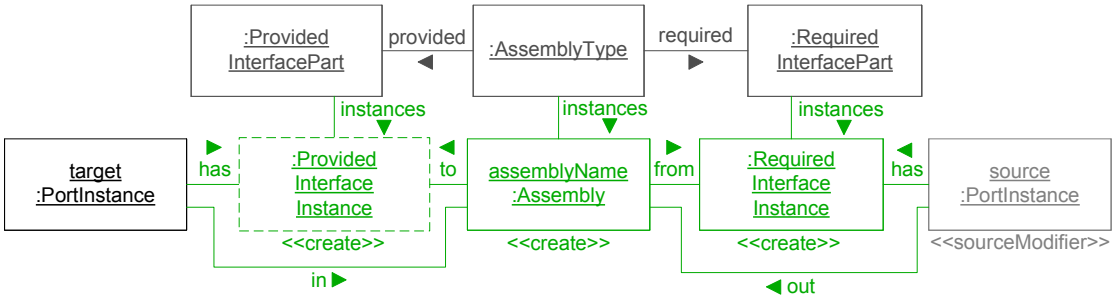
Abbildung 6.45: Mapping eines Kompositionskonnektorlinks

Würde man eine Portinstanz source/sourceRole bei einer Struktur wie in Abb. 6.46(a) zerstören, ohne dass die Zerstörung der Kompositionskonnektorinstanz explizit modelliert ist, wird die Instanz der angebotenen Schnittstelle auf der gegenüberliegenden Seite nicht gelöscht. Das kommt daher, dass die Kompositionen im Metamodell für Komponenteninstanzstrukturen (vgl. Unterabschnitt 5.2.2) von Provided- bzw. RequiredInterfaceInstance zu Assembly definiert sind. Eine unverbundene Schnittstelleninstanz verletzt nicht die Wohlgeformtheit eines Komponenteninstanzgraphen. Es kann allerdings passieren, dass diese Situation mehrfach auftritt und somit diverse unverbundene Schnittstelleninstanzen des gleichen Parts existieren, welche über eine Kompositionskonnektorinstanz wieder neu verbunden werden könnten. Um diese Neuverbindung zu erreichen, wird das spezielle

Mapping aus Abb. 6.46 verwendet. Im Fall, dass die Portinstanz target nicht neu instanziiert wird, kann es sein dass sie eine unverbundene Schnittstelleninstanz besitzt. Um diese neu zu verbinden, wird die korrespondierende Objektvariable :ProvidedInterfaceInstance mit einer optionalen Anwendungsbedingung versehen. Somit wird nur dann eine neue Schnittstelleninstanz des gleichen Parts erstellt, wenn nicht bereits eine vorhanden ist. Ansonsten wird die existierende Schnittstelleninstanz neu verbunden. Das Mapping wird analog für Instanzen benötigter Schnittstellen verwendet, und ergänzt das Mapping aus Abb. 6.45.



(a) Komponentenstorydiagramm



(b) Transformationsdiagramm

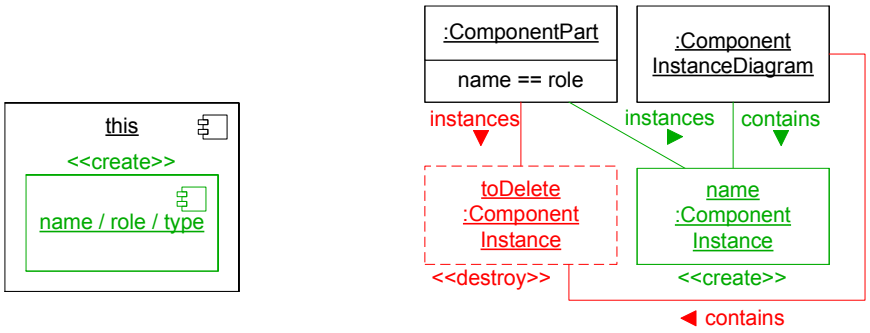
Abbildung 6.46: Mapping eines Kompositionskonnektorlinks mit ≪create≫-Modifizierer an einer Portvariable ohne Modifizierer

6.5.3 Ersetzung von Instanzen aufgrund oberer Multiplizitätsgrenzen

Die operationale Semantik von Komponentenstorypatterns bzgl. der Konformität der resultierenden Komponenteninstanzstrukturen zu den auf Typebene definierten Multiplizitäten, wie sie in Abschnitt 6.3 informal vorgestellt wurde, wird ebenfalls beim Mapping auf Transformationsdiagramme festgelegt.

Ersetzung Komponenten- und Portinstanzen Abb. 6.47 zeigt das Mapping für die Erstellung einer neuen Komponenteninstanz, dessen Part eine obere Multiplizitätsgrenze von 1 besitzt. Dieses Mapping ergänzt das allgemeine Mapping für Komponentenvariablen aus Abb. 6.41 aus Unterabschnitt 6.5.2. Um zur Laufzeit des Storypatterns im Falle einer Komponenteninstanziierung bereits existierende Komponenteninstanzen desselben Parts zu ersetzen, wird für jede Komponentenvariable mit ≪create≫-Modifizierer auf Ebene des Transformationsdiagramms eine zweite Objekt-

variable toDelete:ComponentInstance angelegt. Diese Objektvariable ist optional, besitzt einen ≪destroy≫-Modifizierer und hat ebenso Links zu :ComponentPart und :ComponentInstanceDiagram wie die Objektvariable name:ComponentInstance. In dem Fall, dass bereits eine Komponenteninstanz /role innerhalb derselben übergeordneten Komponenteninstanz existiert, wird also zunächst /role zerstört, und im Anschluss name/role instanziiert. Besitzt die zu löschende /role-Instanz Portinstanzen und diese wiederum Konnektorinstanzen, so werden diese bei einer Löschung durch die Verwendung von Kompositionen zwischen den korrespondierenden Metamodellklassen ebenfalls zerstört.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

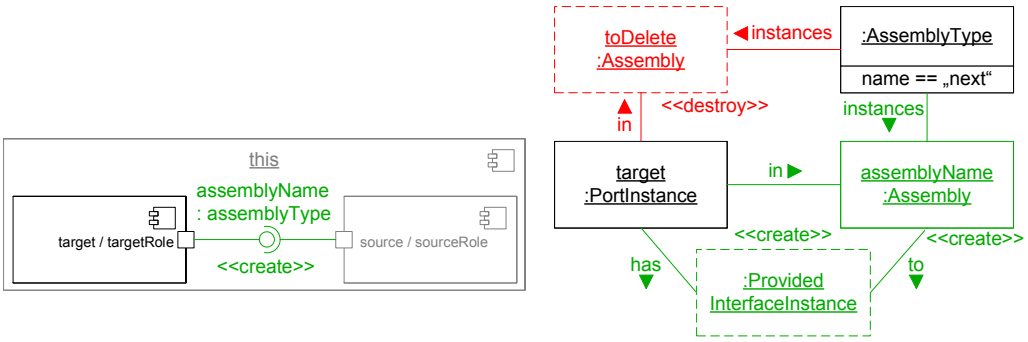
Abbildung 6.47: Mapping für Komponentenvariable mit ≪create≫-Modifizierer und oberer Part-Multiplizitätsgrenze von 1

Das Mapping für Portvariablen mit ≪create≫-Modifizierer, dessen Porttyp[4] eine obere Multiplizitätsgrenze von 1 besitzt, wird ähnlich definiert. Im Transformationsdiagramm wird eine zusätzliche optionale :PortInstance-Variable mit ≪destroy≫-Modifizierer generiert, welche Links zu den Objektvariablen hat, die das klassifizierende Port-Part und die zugeordnete Komponenteninstanz repräsentieren.

Ersetzung von Konnektorinstanzen Für zu erstellende Konnektorinstanzen müssen bei der Berücksichtigung der Multiplizitäten des Typs die Quelle und das Ziel unterschieden werden. Die Quell- und Zielmultiplizitäten des Konnektortyps beschreiben das mengenmäßige Verhältnis, in dem Quell- und Zielportinstanzen miteinander verbunden werden dürfen. Abb. 6.48 zeigt das Mapping für Kompositionskonnektorlinks mit ≪create≫-Modifizierer, deren Konnektortyp eine Quellmultiplizität mit einer oberen Grenze von 1 besitzt. Das Mapping ergänzt das allgemeine Mapping für Kompositionskonnektorlinks aus Abb. 6.45 aus dem letzten Unterabschnitt. Die Zielportvariable target und ihre Komponenteninstanz besitzen keinen Modifizierer, da ansonsten die entsprechende Portinstanz erstellt oder zerstört werden würde und somit keine evtl. vorhandene Konnektorinstanz ersetzt werden müsste. Da der Konnektortyp eine Quellmultiplizität mit einer oberen Grenze von 1 besitzt, muss eine an target vorhandene Instanz :assemblyType, welche target mit einer weiteren Quellportinstanz verbindet, zerstört werden. Dies wird

[4]Die Porttypen eingebetteter Portvariablen werden durch das klassifizierende Port-Part vorgegeben.

auf Ebene des Transformationsdiagramms durch die optionale Objektvariable toDelete:Assembly mit ≪destroy≫-Modifizierer spezifiziert. Um sicherzustellen, dass lediglich Kompositionskonnektorinstanzen vom selben Typ und an der gleichen Zielportinstanz zerstört werden, besitzt toDelete Links zu den Variablen :AssemblyType und target. Die zugehörigen Schnittstelleninstanzen werden nicht gelöscht, sondern, wie im letzten Unterabschnitt vorgestellt, bei Bedarf neu verbunden. In der abgebildeten Situation wird bei einer Löschung von toDelete direkt im Anschluss die durch :ProvidedInterfaceInstance repräsentierte Schnittstelleninstanz durch assemblyName neu verbunden. Für die umgekehrte Konnektorrichtung und einer oberen Grenze von 1 für die Zielmultiplizität wird das Mapping analog spezifiziert. Für einen 1:1-Konnektortyp werden zwei zu löschende :Assembly-Variablen generiert, welche mit den Variablen zur Repräsentation von Quell- und Zielportinstanz verbunden werden.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.48: Mapping für Kompositionskonnektorlinks mit ≪create≫-Modifizierer und oberer Quellmultiplizitätsgrenze von 1

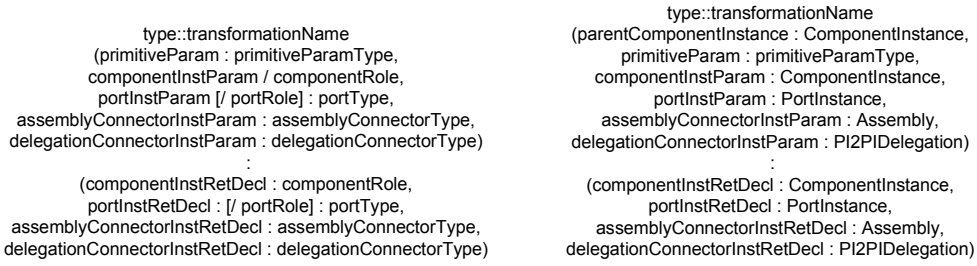
Das Mapping für die Ersetzung von vorhandenen Delegationskonnektorinstanzen eines Typs mit oberen Multiplizitätsgrenzen von 1 sei nach dem Vorbild des allgemeinen Mappings für Delegationskonnektorlinks (Abb. 6.44 im letzten Unterabschnitt) analog definiert.

6.5.4 Transformationsaufrufe

Bisher wurden lediglich die Mappings von Elementen betrachtet, welche sich auf den inneren Anwendungsbereich eines Storydiagramms oder eines Storypatterns beziehen. Im Folgenden werden die Mappings vorgestellt, welche die Kollaboration verschiedener Komponentenstorydiagramme untereinander ermöglichen. Auf Ebene der Transformationsdiagramme werden dafür die in Unterabschnitt 2.1.2 vorgestellten Elemente in Bezug auf Transformationsaufrufe verwendet.

Parameter und Rückgabedeklarationen Abb. 6.49 zeigt das Mapping für Parameter und Rückgabedeklarationen. Jeder komplexe Parameter eines Komponentenstorydia-

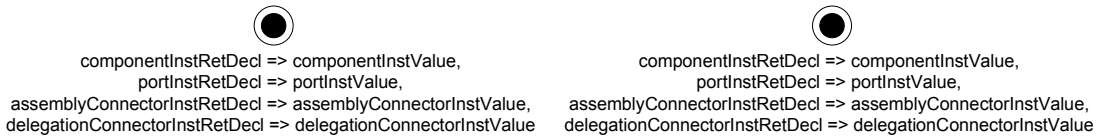
gramms wird in die entsprechende Repräsentation auf Objektebene übertragen. Ein Parameter componentInstParam beispielsweise, welcher eine Komponenteninstanz darstellt und über ein Komponenten-Part typisiert ist, wird auf den Transformationsdiagramm-Parameter componentInstParam:ComponentInstance abgebildet. Die Information über die Typisierung der Komponenteninstanz wird in der Signatur des Transformationsdiagramms nicht deklariert, ist aber nach wie vor durch den Gesamtkontext gegeben. Primitive Parameter werden direkt vom Transformationsdiagramm übernommen. Rückgabedeklarationen werden analog spezifiziert.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.49: Mapping für Parameter und Rückgabedeklarationen

Rückgabewerte Abb. 6.50 visualisiert das Mapping für Rückgabewerte. Um einen Rückgabewert einer Rückgabedeklaration zuzuordnen, wird eine part-/typgleiche Variable oder ein typgleicher Konnektorlink an die Deklaration gebunden. Dies wird z. B. durch componentInstRetDecl ⇒ componentInstValue ausgedrückt, wobei eine Komponentenvariable an eine Rückgabedeklaration des gleichen Klassifizierers gebunden wird. Dies wird auf die Objektrepräsentation der Variablen und Konnektorlinks im Transformationsdiagramm übertragen. Die gleichnamige Objektvariable componentInstValue:ComponentInstance wird also auf Seite des Transformationsdiagramms an die Rückgabedeklaration componentInstRetDecl:ComponentInstance gebunden.

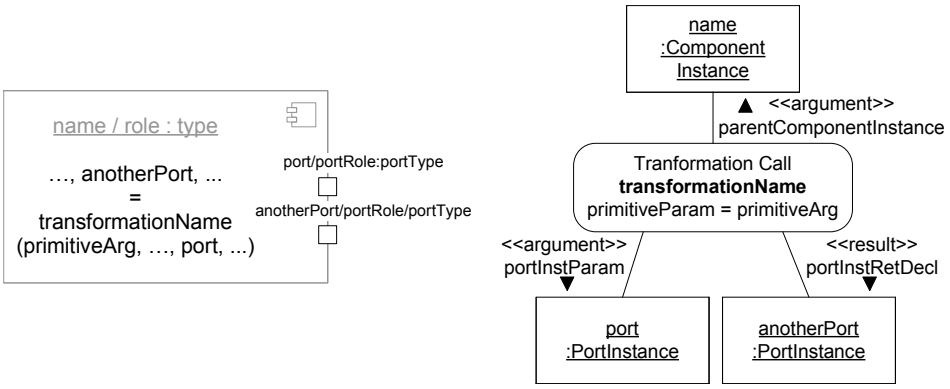


(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.50: Mapping für Rückgabewerte

Konstruktor- und Rekonfigurationsaufrufe Abb. 6.51 zeigt das Mapping für den Aufruf eines Komponentenstorydiagramms. Für Konstruktor- und Rekonfigurationsaufrufe

wird dabei das gleiche Prinzip verwendet. Ein Aufruf eines Komponentenstorydiagramms wird auf einen Transformation Call abgebildet. Der Transformation Call bekommt als Ziel das Transformationsdiagramm transformationName, welches dem aufzurufenden Komponentenstorydiagramm entspricht. Die Referenz auf die aktuelle Komponenteninstanz, deren eingebettete Struktur transformiert werden soll, wird dem aufzurufenden Transformationsdiagramm in Form der Objektvariable name:ComponentInstance als Argument für den Parameter parentComponentInstance übergeben. Wie in Unterabschnitt 6.5.2 erläutert, repräsentiert dieser Parameter die this-Variable von Komponentenstorypatterns. Für die Bindung des Arguments an den Parameter wird der Link mit Namen des Parameters und dem ≪argument≫-Stereotyp benutzt. In der dargestellten Situation wird des Weiteren die Portvariable port als Argument für den Parameter portInstParam verwendet, dessen Zuweisung zum Parameter ebenfalls auf einen ≪argument≫-Link abgebildet wird. Die anderen Arten komplexer Parameter werden analog behandelt. Der Inhalt des primitiven Arguments primitiveArg wird direkt auf die Ebene des Transformationsdiagramms übertragen. Der Rückgabewert des Aufrufs im Komponentenstorydiagramm wird an die Portvariable anotherPort gebunden, was im Transformationsdiagramm durch den Link mit dem Stereotyp ≪result≫ ausgedrückt wird. Dieser Link bezieht sich auf die Rückgabedeklaration portInstRetDecl, was im Komponentenstorydiagramm durch die Position in der Liste der Rückgabewerte ausgedrückt wird.



(a) Komponentenstorydiagramm (b) Transformationsdiagramm

Abbildung 6.51: Mapping für den Aufruf eines Komponentenstorydiagramms

6.5.5 Anwendungsbedingungen

Wie in Abschnitt 6.2 erläutert, müssen negative und optionale Anwendungsbedingungen für Variablen von Komponentenstorypatterns ebenso wie Modifizierer auf kompositional abhängige Elemente übertragen werden. Somit beziehen sich diese Anwendungsbedingungen auf einen Bereich mit mehreren Elementen, welche in der abstrakten Syntax durch multiple, untereinander verbundene Objektvariablen repräsentiert werden. Hierbei entsteht die Problematik, dass Anwendungsbedingungen für solche adjazente Objektva-

riablen nicht durch Storypatterns definiert sind [Zün01]. Daher können Anwendungsbedingungen nicht einfach, wie Modifizierer (vgl. Unterabschnitt 6.5.2), auf kompositional abhängige Elemente propagiert werden.

Um die gewünschte, in den Abschnitten 6.1.1 und 6.2 beschriebene Semantik zu gewährleisten, muss bei der Übersetzung eines Komponentenstorydiagramms mit Anwendungsbedingungen in ein Transformationsdiagramm der Kontrollfluss aufgespalten werden. Da die für das Mapping verwendeten TGG-Regeln einen sehr feingranularen Anwendungsbereich besitzen und bei einer solchen Kontrollflussaufspaltung ein Komponentenstorydiagramm in seiner Gesamtheit betrachtet werden muss, ist die Aufspaltung in einem gesonderten Schritt vor dem Mapping durchzuführen. Eine Möglichkeit für dessen Umsetzung wären z. B. konventionelle Storydiagramme.

Da dieser Schritt eine gesonderte Implementierung benötigt, konnte er in der zur Verfügung stehenden Bearbeitungszeit dieser Diplomarbeit nicht technisch realisiert werden. Dennoch wurden Konzepte erarbeitet, welche die benötigte Methodik beschreiben und welche in zukünftigen Arbeiten umgesetzt sowie evaluiert werden können. Diese Konzepte werden im Folgenden vorgestellt.

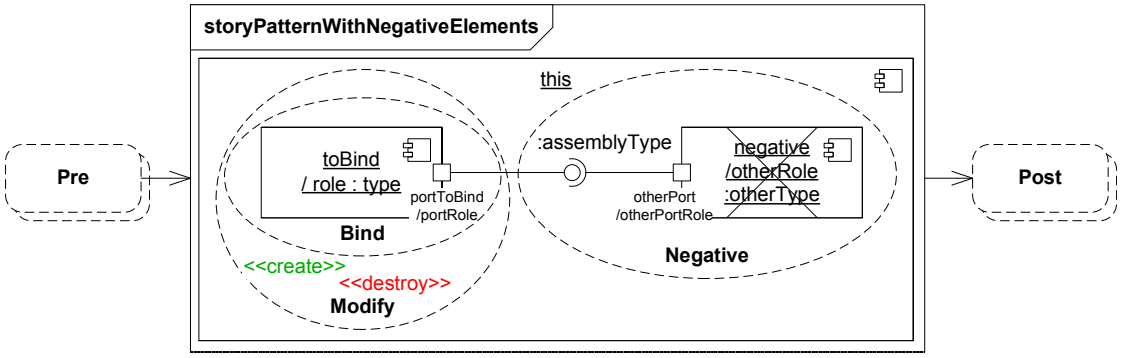
Dabei ist zu erwähnen, dass bestimmte Formulierungen von Bedingungen, welche keine Bereiche aufspannen, durchaus funktionieren. Als Beispiele seien negative Bedingungen auf einem Delegationskonnektorlink oder das simple Binden einer Komponenten- ohne Portinstanz genannt.

Negative Elemente Um komplette Bereiche negativer Elemente von der Anwendung eines Komponentenstorypatterns auszuschließen, wird eine Aufteilung in drei Storypatterns vorgenommen, welche Teilaufgaben des ursprünglichen Storypatterns übernehmen. Abb. 6.52 skizziert diese Aufteilung.

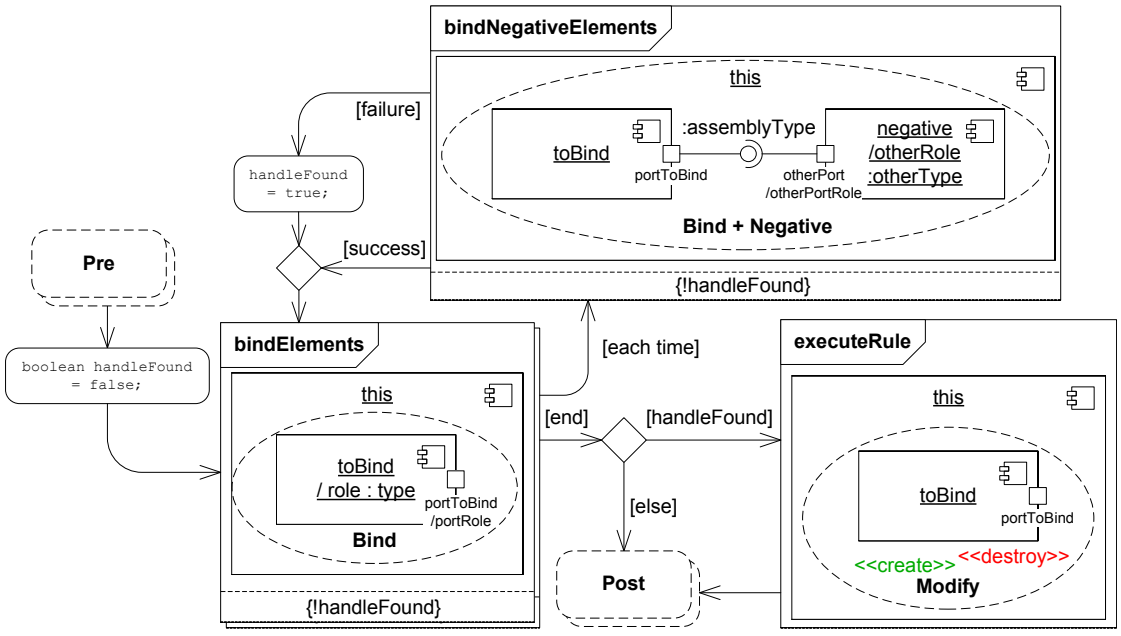
Abb. 6.52(a) zeigt das ursprüngliche Storypattern storyPatternWithNegativeElements. Dieses besitzt exemplarisch die Variablen toBind und portToBind ohne Modifizierer. Diese Variablen stehen für den Bereich oder die Menge Bind aller Variablen und Konnektorlinks, welche für eine erfolgreiche Regelanwendung gebunden werden müssen. Neben diesen Elementen enthält storyPatternWithNegativeElements die negative Komponentenvariable negative, welche über die Portvariable otherPort und den Kompositionskonnektorlink :assemblyType mit der Portvariable portToBind von toBind verbunden ist. Die Variablen und der Konnektorlink :assemblyType, otherPort und negative stehen dabei allgemein für den Bereich Negative aller Elemente, welche nicht im Wirtsgraph existieren dürfen. Neben diesen Elementen ist der Bereich Modify des Storypatterns angedeutet, welche den Bereich Bind sowie alle Modifizierer enthält. Der Bereich Bind stellt also die LHS des Storypatterns dar, während Modify die Operationen des Storypatterns kapselt. Somit enthält Modify zusätzlich die ≪destroy≫-Modifizierer der LHS sowie die zu instanziierenden Variablen und stellt bei der Ausführung die RHS her.

Das Storypattern ist in den Kontext eines Komponentenstorydiagramms eingebettet und besitzt einen Vorbereich Pre, welcher den logisch vorgelagerten Kontrollfluss umfasst. Genauer gesagt kann das storyPatternWithNegativeElements ein oder zwei eingehende Transitionen besitzen, denen weitere Aktivitäten vorgeschaltet sind. Analog wird

der Nachbereich Post definiert. Das Storypattern hat entweder eine oder zwei ausgehende Transitionen. Bei zwei ausgehenden Transitionen wird davon ausgegangen, dass deren Guards konventionellen booleschen Bedingungen entsprechen, Transitionen mit success-/failure-Guards dagegen werden nachfolgend gesondert behandelt.



(a) Vor Aufspaltung des Kontrollflusses



(b) Nach Aufspaltung des Kontrollflusses

Abbildung 6.52: Komponentenstorypattern mit negativen Elementen

Die Anwendung eines Storypatterns mit negativen Elementen lässt sich in drei Schritte einteilen. Zunächst wird die LHS ohne die negativen Elemente gebunden. Im erfolgreichen Fall wird die gefundene Struktur daraufhin untersucht, ob sie mit einem der modellierten negativen Elemente verbunden ist. Falls dem so ist, ist die gefundene Struktur keine gültige Anwendungsstelle bzgl. der Anwendungsbedingungen und es wird eine neue Anwendungsstelle der LHS gebunden und auf die Anwendungsbedingung überprüft. Enthält

die gefundene Struktur keines der negativen Elemente, dann kann das Storypattern ausgeführt werden.

Im Prinzip wird das ursprüngliche Storypattern in diese einzelnen Schritte aufgespalten, was in Abb. 6.52(b) skizziert ist. Dieses orientiert sich an dem Beispiel für die Semantik negativer Objektvariablen aus Unterabschnitt A.6.5 in [Zün01]. Der Pre-Bereich mündet in eine Statement-Aktivität, welche die boolesche Variable handleFound deklariert und mit false initialisiert. Falls diese Variable den Wert false besitzt und somit das entsprechende Constraint erfüllt ist, sucht das forEach-Storypattern bindElements eine Anwendungsstelle des Bereichs Bind. Im erfolgreichen Fall werden dessen Elemente als gebundene Variablen im Storypattern bindNegativeElements weiterverwendet.

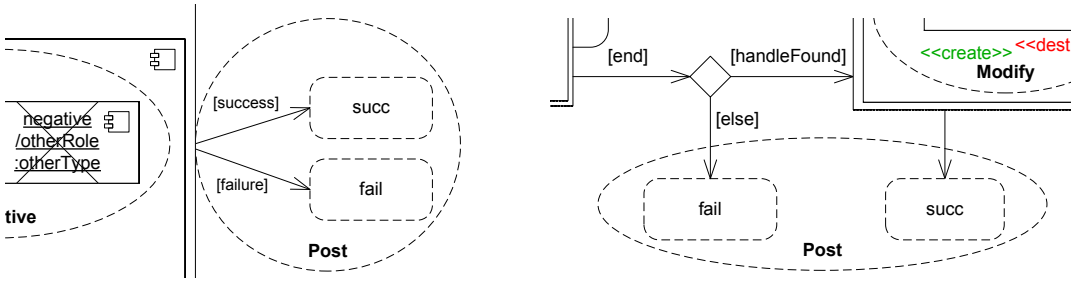
In diesem Storypattern wird eine Anwendungsstelle von Bind zusammen mit den negativem Bereich – in diesem Fall ohne Anwendungsbedingung – gesucht, falls handleFound den Wert false hat. Wird eine solche Struktur gefunden, so bilden die Elemente von Bind keine gültige Anwendungsstelle bzgl. der Anwendungsbedingung, und bindElements wird wieder ausgeführt. Da handleFound nicht modifiziert wurde, wird in dieser forEach-Aktivität die nächste Anwendungsstelle der LHS gesucht. Falls bindNegativeElements dagegen erfolglos war und somit Bind nicht zusammen mit dem negativen Bereich im Wirtsgraph gefunden werden konnte, wird handleFound auf true gesetzt. In diesem Fall wird aufgrund der Constraints in bindElements und bindNegativeElements die Anwendungsstelle nicht mehr verändert. Die forEach-Aktivität bindElements wird solange durchlaufen, bis alle möglichen Anwendungsstellen der LHS abgearbeitet sind, unabhängig vom Wert von handleFound.

Diese Variable wird von der nachfolgenden NOP-Aktivität ausgewertet. Konnte eine gültige Anwendungsstelle gefunden werden, so wird das Storypattern executeRule ausgeführt, welches den Bereich Modify des ursprünglichen Storypatterns enthält. In dieser Aktivität wird also die eigentliche Transformation vorgenommen, und anschließend der nachfolgende Bereich Post betreten. Falls keine Anwendungsstelle gefunden wurde, wird Post direkt betreten.

Besitzt eine forEach-Aktivität negative Elemente, so muss eine geringfügig andere Strategie verfolgt werden. In diesem Fall darf handleFound nicht ausgewertet werden, und executeRule ist nach jeder erfolgreichen Anwendung von bindNegativeElements auszuführen. Der Post-Bereich wird dann über die end-Transition von bindElements betreten.

Die beschriebene Vorgehensweise funktioniert nur dann, falls storyPatternWithNegativeElements lediglich eine ausgehende Transition oder zwei ausgehende Transitionen mit booleschen Bedingungen besitzt. In diesem Fall wird die nachfolgende Aktivität oder Verzweigung des Post-Bereichs unabhängig davon betreten, ob eine Anwendungsstelle gefunden werden konnte oder nicht. Zwei ausgehende Transitionen mit success-/failure-Guards werten dagegen den Erfolg eines Storypatterns bzgl. seiner Regelanwendung aus, die beschriebene Aufteilung gibt allerdings kein Feedback darüber. Somit wird für einen Post-Bereich mit success-/failure-Transitionen eine leicht modifizierte Übersetzung des Storypatterns benötigt, welche in Abb. 6.53 dargestellt wird.

Abb. 6.53(a) zeigt einen Ausschnitt des ursprünglichen Storypatterns storyPatternWithNegativeElements, welches nun zwei ausgehende success-/failure-Transitionen besitzt. Bei erfolgreicher Regelanwendung wird die Aktivität succ ausgeführt, ansonsten die Aktivi-



(a) Vor Aufspaltung des Kontrollflusses (b) Nach Aufspaltung des Kontrollflusses

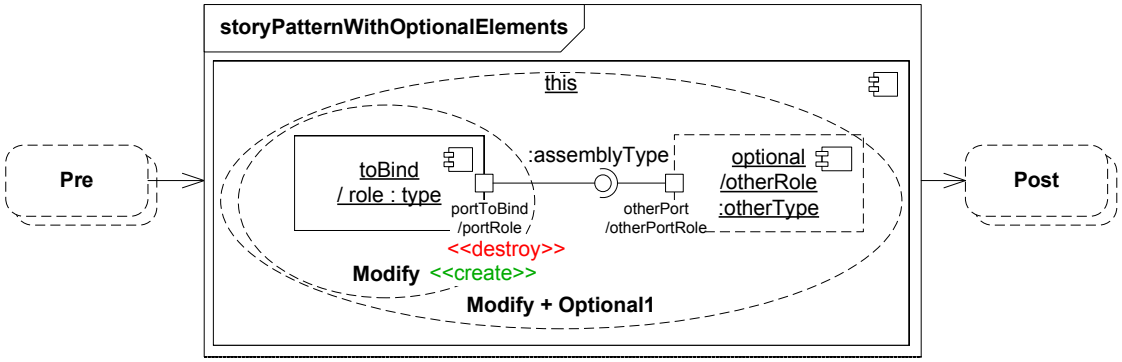
Abbildung 6.53: Komponentenstorypattern mit negativen Elementen und ausgehenden success-/failure-Transitionen (Ausschnitte)

tät fail. In Abb. 6.53(b) ist ein Ausschnitt der entsprechenden Übersetzung dargestellt. Im Wesentlichen besitzt diese die gleichen Aktivitäten und den gleichen Kontrollfluss wie zuvor beschrieben, unterscheidet sich aber in der Auswertung von handleFound durch die NOP-Aktivität. Konnte keine gültige Anwendungsstelle gefunden werden, wird direkt die Aktivität fail betreten. Konnte dagegen eine Anwendungsstelle gefunden werden, wird zunächst executeRule und anschließend die Aktivität succ ausgeführt.

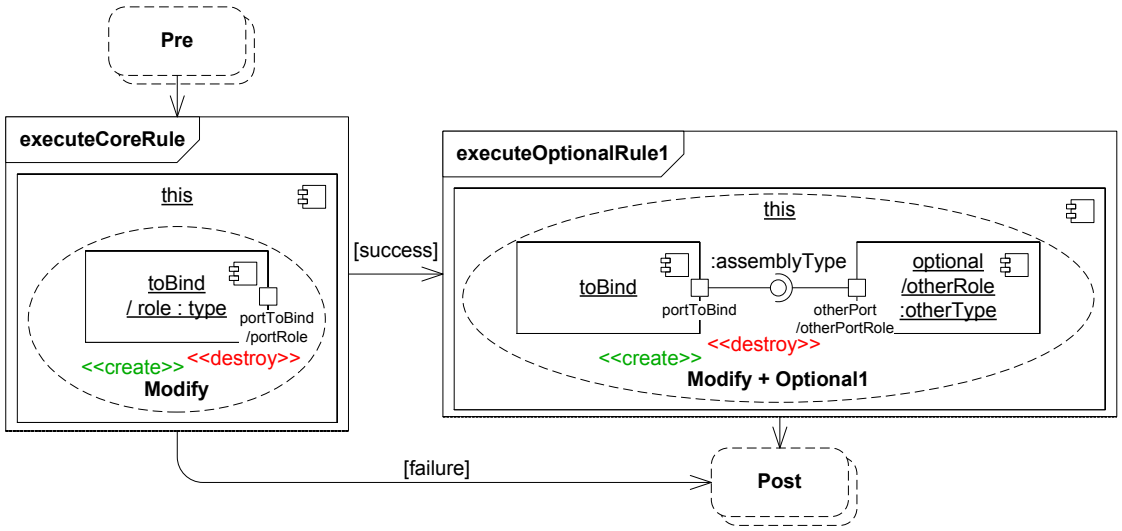
Optionale Elemente Abb. 6.54 skizziert die Übersetzung eines Komponentenstorypatterns mit optionalen Elementen. In Abb. 6.54(a) wird das ursprüngliche Storypattern storyPatternWithOptionalElements dargestellt. Es gibt einen Bereich Modify, welcher wie bei der Vorstellung der Umsetung für negative Bereiche definiert sei. Für jede optionale Variable gibt es einen Bereich Optional, welcher von der Anwendungsbedingung aufgespannt wird. Mit der Nummerierung für Optional wird angedeutet, dass es mehrere dieser Bereiche geben kann. Modifizierer aus Modify können sich auf jeden dieser Bereiche auswirken, somit ergibt sich jeweils ein Bereich Modify + Optional. Optionale Elemente, welche selber Modifizierer besitzen, müssen gesondert betrachtet werden. Die erfolgreiche Anwendung des Storypatterns ist unabhängig von den optionalen Bereichen.

In Abb. 6.54(b) wird die Übersetzung dieses Storypatterns gezeigt. Dem Pre-Bereich folgt das Storypattern executeCoreRule, in dem die LHS gebunden und alle Modifikationen aus Modify durchgeführt werden. Falls das Storypattern nicht erfolgreich war, wird direkt der Post-Bereich betreten. Falls erfolgreich eine Anwendungsstelle ermittelt werden konnte, folgt für jede optionale Variable – bzw. für den Bereich Optional, den diese aufspannt – ein Storypattern executeOptionalRule. Dieses Storypattern bindet die optionalen Elemente und transformiert diese im erfolgreichen Fall mit den Operationen aus Modify, welche Einfluss auf die Elemente von Optional haben. Unabhängig vom Erfolg des Storypatterns wird das Storypattern für den nächsten optionalen Bereich ausgeführt oder der Post-Bereich betreten.

Der Post-Bereich muss wie bei den negativen Elementen dahingehend differenziert werden, ob zwei success-/failure-Transitionen folgen oder nicht. Für optionale Variablen mit Modifizierern muss auf den Erfolg der Regelanwendung von executeOptionalRule reagiert



(a) Vor Aufspaltung des Kontrollflusses



(b) Nach Aufspaltung des Kontrollflusses

Abbildung 6.54: Komponentenstorypattern mit optionalen Elementen

werden. Besäße z. B. die Komponentenvariable optional einen ≪create≫-Modifizierer, so würde im erfolglosen Fall ein Storypattern folgen, welches optional instanziiert. Bei einem ≪destroy≫-Modifizierer würde im erfolgreichen Fall eine Regel zur Zerstörung von optional folgen.

Kombination von negativen und optionalen Elementen Möchte man optionale und negative Elemente im gleichen Komponentenstorypattern kombinieren, so müssen weitere Fälle betrachtet werden. Insbesondere lassen sich die hier beschriebenen Übersetzungen der Bedingungen nicht miteinander kombinieren, falls die Bindungsreihenfolge verschiedener Arten von Objekten mit Anwendungsbedingungen nach [Zün01] eingehalten werden soll. Nach Zündorf werden zunächst optionale Elemente gebunden, und danach die negativen Anwendungsbedingungen eines Storypatterns überprüft. Dies lässt sich mit den hier beschriebenen Übersetzungen nicht realisieren, da die Behandlung optionaler Bereiche zuerst die Operationen des Storypatterns für die LHS ohne optionale Elemente ausführt (Bereich Modify), und danach Schritt für Schritt die optionalen Bereiche abarbeitet. Im Konstrukt, welches aus einem Komponentenstorypattern mit negativen Elementen generiert wird, findet die eigentliche Regelausführung (Bereich Modify) erst am Ende statt.

Eine Alternative besteht darin, optionale Bereiche so zu übersetzen wie im Beispiel für die Semantik optionaler Objektvariablen aus Unterabschnitt A.6.7 in [Zün01]. In diesem Beispiel werden zwei Storypatterns für jede optionale Objektvariable generiert. Eines enthält die optionale Variable explizit in der LHS, während das andere sie explizit über eine negative Anwendungsbedingung aus der LHS ausschließt. Diese Variante würde allerdings unnötig viele Storypatterns pro optionaler Variable generieren.

Eine weitere Möglichkeit ist es, nicht die aus [Zün01] vorgeschlagene Reihenfolge zu übernehmen und zunächst die negativen Anwendungsbedingungen zu überprüfen. Vergleicht man die beiden in diesem Unterabschnitt beschriebenen Übersetzungen, fällt auf, dass diese leicht miteinander kombiniert werden können, indem im Anschluss der Überprüfung einer negativen Anwendungsbedingung optionale Bereiche gebunden werden können. Hierzu muss lediglich die letzte Regel executeRule der Überprüfung einer negativen Anwendungsbedingung (Abb. 6.52(b)) mit der ersten Regel executeCoreRule für das Binden der LHS eines Storypatterns mit optionalen Elementen (Abb. 6.54(b)) verschmolzen werden. Dies ist ohne weiteres möglich, da sie sich beide Regeln auf den Bereich Modify beziehen.

7 Evaluierung

Zur Evaluierung der vorgestellten Konzepte werden diese in die FUJABA Real-Time Tool Suite für Fujaba4Eclipse implementiert. Die wichtigsten Details dieser Implementierung werden im ersten Abschnitt vorgestellt. Das laufende Beispiel des RailCab-Konvois wird als Fallstudie zur Evaluierung herangezogen. Die Ergebnisse des Evaluierungsbeispiels werden im zweiten Abschnitt dieses Kapitels aufgelistet.

7.1 Technische Realisierung

Im folgenden Unterabschnitt werden zunächst die zentralen Use-Cases der neuen Konzepte vorgestellt. Der zweite Unterabschnitt skizziert in einer vereinfachten Übersicht, welche Plugins verwendet, modifiziert und neu implementiert wurden und welche Use-Cases von welchem Plugin realisiert werden.

7.1.1 Use-Cases

Abbildung 7.1 zeigt die relevanten Use-Cases der in dieser Diplomarbeit umgesetzten Implementierung. Das umgebende System stellt als vereinfachende Annahme die diversen Plugins der FUJABA Real-Time Tool Suite zusammenhängend dar. Eine genauere Analyse der zu Grunde liegenden und neu entwickelten Plugins wird im folgenden Unterabschnitt vorgenommen.

Der Akteur Benutzer stellt einen Benutzer der FUJABA Real-Time Suite mit Domänenwissen über die zu modellierenden Systeme dar. Mit einfachen und hierarchischen, variablen Komponententypen modelliert er die Architektur eines Systems. Basierend auf dieser Typdefinition werden Transformationen in Form von Komponentenstorydiagrammen spezifiziert. Mit Hilfe des vom Modellierer Mapping spezifizierten Mappings werden diese Komponentenstorydiagramme auf Transformationsdiagramme abgebildet und anschließend der Code für die konkrete Ausführung der Transformationsdiagramme generiert. Um das Mapping auf Transformationsdiagramme vorzunehmen, wird zunächst Code für das Mapping generiert und danach ausgeführt. Anschließend wird aus den resultierenden Transformationsdiagrammen Code generiert und letztendlich für die konkrete Ausführung der Transformationen benutzt. Der folgende Unterabschnitt stellt diesen Ablauf genauer vor.

Der Akteur Modellierer Mapping spezifiziert das Mapping der Komponentenstorydiagramme auf die Transformationsdiagramme. Dieser Akteur muss Kenntnis über die Metamodelle für Komponententypen, -instanzen und -storydiagramme von MECHATRONIC UML und für Transformationsdiagramme besitzen.

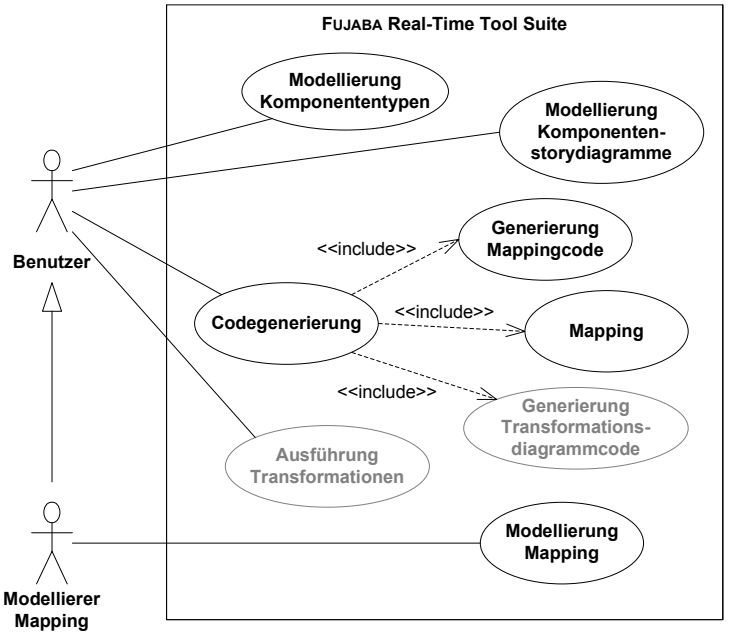


Abbildung 7.1: Use-Cases

Die Codegenerierung für Transformationsdiagramme sowie deren Ausführung sind Use-Cases, welche bereits durch vorhandene Plugins realisiert sind und nicht im Rahmen dieser Diplomarbeit implementiert wurden. Daher sind die entsprechenden Use-Cases ausgegraut.

7.1.2 Architektur

Abb. 7.2 zeigt eine stark vereinfachte Sicht auf die verwendeten, modifizierten und neu entwickelten Plugins. Die Implementierung der in dieser Arbeit vorgestellten Konzepte erfolgt dabei im Rahmen von Fujaba4Eclipse.

Das Plugin FujabaCore stellt den Kern von FUJABA dar, welches u. a. das Metamodell für konventionelle Storydiagramme, Funktionalitäten für die Codegenerierung sowie die Anbindung an Eclipse enthält. Dieses Plugin besteht im Detail wiederum aus weiteren dedizierten Plugins.

Das Plugin UMLRT stellt den Aufsatz für die FUJABA Real-Time Tool Suite dar und stellt die Sprache MECHATRONIC UML zur Verfügung. In diesem Plugin wurden zum einen die im Rahmen dieser Diplomarbeit nötig gewordenen Modifikationen für die variable Komponententypdefinition vorgenommen, zum anderen wurde der Editor für die Komponentenstorydiagramme in UMLRT implementiert. Auch UMLRT besteht intern aus weiteren Plugins, welche z. B. eine spezielle Codegenerierung für Echtzeitverhalten ermöglichen.

Das Plugin-Konglomerat Transformations repräsentiert vereinfacht gesehen die Funktionalitäten der Transformationsdiagramme, welche Elemente der Storydiagramme aus

FujabaCore erben. Dazu gehört einerseits die Codegenerierung für Transformationsdiagramme inklusive der Kompilierung des Codes und Speicherung in einem JAR-Archiv, andererseits die Ausführung dieses kompilierten Codes zur Manipulation von Objektstrukturen.

Das Plugin TGGEditor [Wag06] stellt den Editor für die TGG-Regeln bereit, mit denen das Mapping modelliert wird. Der Editor verwendet Syntaxelemente der Storydiagramme und übersetzt eine TGG-Regel in ein Tripel von Storydiagrammen, wodurch eine Abhängigkeit zu FujabaCore entsteht. Nachdem aus den Storydiagramm-Tripeln Java-Code generiert, kompiliert und in einem JAR-Archiv hinterlegt worden ist, ermöglicht das Plugin MoTE [Wag06] dessen Ausführung.

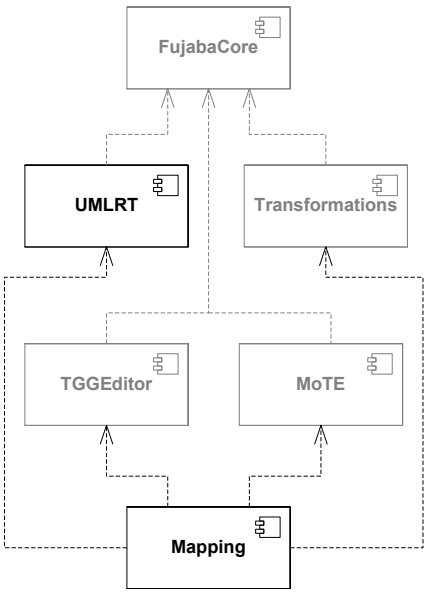


Abbildung 7.2: Plugins

Das Plugin Mapping wurde im Rahmen dieser Arbeit neu entwickelt, damit in UMLRT nicht zusätzliche Abhängigkeiten zu TGGEditor und MoTE entstehen. Mapping stellt eine Funktion zum direkten Export eines JAR-Archivs mit dem kompilierten Mappingcode sowie eine Importfunktion bereit, welche anhand dieses JAR-Archivs und aus Komponentenstorydiagrammen eines FUJABA-Modells Transformationsdiagramme generiert und in das Modell importiert.

Der komplette Use-Case Codegenerierung (siehe letzter Unterabschnitt) besteht aus mehreren Einzelschritten, bei denen unterschiedliche Plugins beteiligt sind. Abb. 7.3 zeigt zur Verdeutlichung den Ablauf dieses etwas komplexeren Use-Cases in Form eines UML 2 Aktivitätsdiagramms.

Zunächst wird der inkludierte Use-Case Generierung Mappingcode ausgeführt. Dabei werden die TGG-Regeln aus der Mapping-Modelldatei in ausführbaren Code kompiliert und in einem JAR-Archiv hinterlegt. Dieses wird für die Ausführung des Mappings benötigt. Mit einer Vorwärtstransformation durch die kompilierten TGG-Regeln im JAR-Archiv werden die Komponentenstorydiagramme aus der entsprechenden Modelldatei in Transformationsdiagramme übersetzt und diese in die Modelldatei importiert. Diese beiden Aktionen stellen Use-Cases des neuen Plugins Mapping dar. Vom bereits vorhandenen Plugin Transformations wird im Anschluss aus den erstellten Transformationsdiagrammen Code generiert und kompiliert, und die resultierenden ausführbaren Klassen in einem JAR-Archiv gespeichert. Dies vervollständigt den übergeordneten Use-Case bzw. die Aktivität Codegenerierung. Mit Hilfe des erstellten JAR-Archivs lassen sich nun Transformationen ausführen.

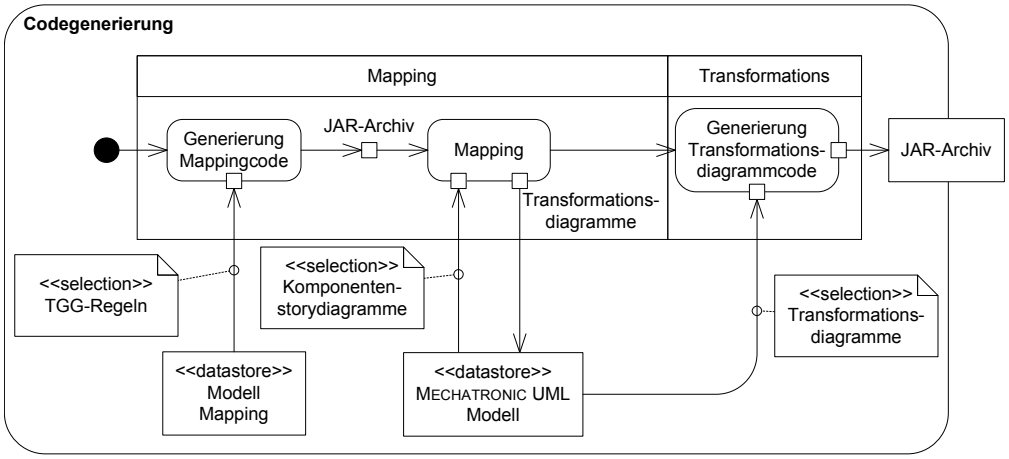


Abbildung 7.3: Ablauf Codegenerierung und beteiligte Plugins

7.2 Evaluierungsergebnisse

Zur Evaluierung der vorgestellten variablen Komponententypstrukturen aus Kapitel 5 und der darauf basierenden neuen Transformationssprache aus dem letzten Kapitel wurde das laufende Beispiel dieser Arbeit als Grundlage genommen und in Fujaba4Eclipse modelliert. Dabei soll in MECHATRONIC UML eine Architektur für ein RailCab-System modelliert werden, die es erlaubt, in einen bestehenden Konvoi an beliebiger Position neue RailCabs aufzunehmen und diese ebenfalls aus einem Konvoi zu entfernen. Dafür müssen die Komponenteninstanzen zur Positionsberechnung für die nachfolgenden RailCabs innerhalb der Komponenteninstanz des führenden RailCabs als Liste angeordnet werden. Zugriff auf diese Liste und auf die Komponenteninstanzen, welche die einzelnen RailCabs repräsentieren, wird über Operationen für diese Architektur in Form von Komponentenstorydiagrammen geschaffen. Da komplexere negative und optionale Anwendungsbedingungen aktuell nicht unterstützt werden, muss deren Verhalten durch zusätzliche Storypatterns simuliert werden.

7.2.1 Modellierung von Komponententypen

In diesem Unterabschnitt wird die technisch umgesetzte Modellierung der Komponententypen aus Kapitel 5 vorgestellt. Aus Platzgründen werden lediglich die hierarchischen Komponententypen gezeigt.

Abb. 7.4 zeigt den hierarchischen, variablen Komponententypen RailCab. Dieser ist aus Parts mit einer Multiplizität von 0..1 für die einfachen Komponententypen Integral[1], VRef, VCtrl und PosCtrl sowie dem Multipart für den einfachen Komponententypen PosCalc komponiert. Des Weiteren besitzt er Port- und Schnittstellentypen mit den gleichen

[1] Integral entspricht dem Komponententyp $\int$, dessen Sonderzeichen nicht von einer FUJABA-Modelldatei abgespeichert werden kann.

Namen wie in Abb. 5.10 in Kapitel 5 zu sehen, welche nicht im Diagramm visualisiert werden.

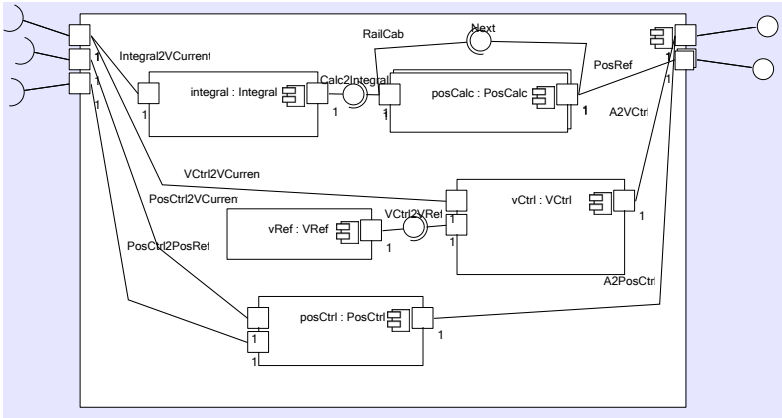


Abbildung 7.4: Hierarchischer Komponententyp Railcab

Die Systemebene ConvoySystem wird durch Abb. 7.5 dargestellt. Sie ist aus zwei Parts des Typs RailCab aufgebaut, welche die Rollen convoyLeader bzw. convoyFollowers einnehmen. Diese sind über den Kompositionskonnektortyp PosRef verbunden.

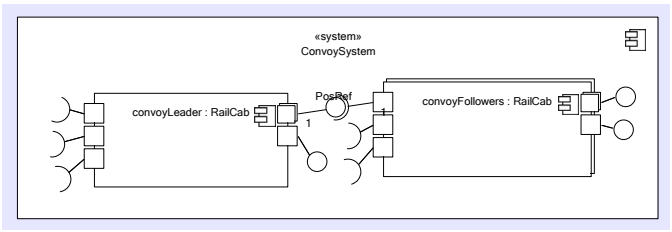


Abbildung 7.5: Systemkomponente ConvoySystem

Tabelle 7.1 listet alle Konstruktoren und Rekonfigurationsregeln dieser hierarchischen Komponententypen. Diese Transformationen werden in den beiden nachfolgenden Unterabschnitten im Detail vorgestellt.

Komponententyp	**Konstruktoren**	**Rekonfigurationsregeln**
ConvoySystem	initConvoySystem	insertFollower removeFollower
RailCab	initConvoyLeader initConvoyFollower	insertPosCalc removePosCalc

Tabelle 7.1: Komponentenstorydiagramme der hierarchischen Komponententypen

7.2.2 Modellierung von Konstruktoren

Abb. 7.6 zeigt den Konstruktor initConvoySystem für die Systemebene, welcher eine initiale Konfiguration eines kompletten Konvoisystems aus RailCabs instanziiert. Der Parameter followers des Typs Integer gibt dabei an, aus wievielen nachfolgenden :RailCab-Komponenteninstanzen diese Konfiguration bestehen soll.

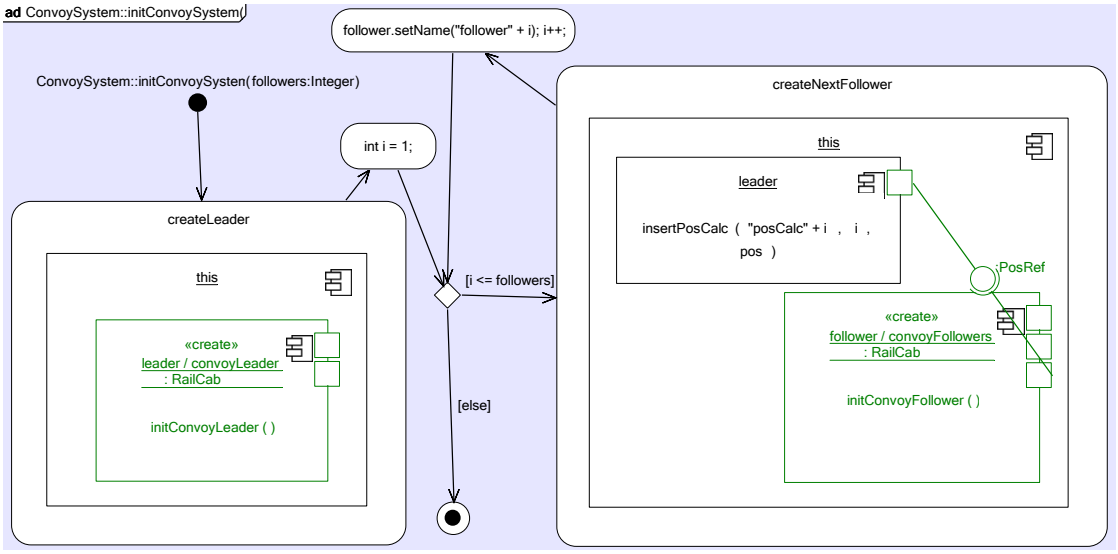


Abbildung 7.6: Konstruktor ConvoySystem::initConvoySystem

Zunächst wird im Storypattern createLeader eine Komponenteninstanz leader/convoyLeader:RailCab erstellt und der Konstruktor initConvoyLeader für diese Komponenteninstanz aufgerufen. Anschließend werden in einer Schleife sooft im Storypattern createNextFollower Komponenteninstanzen der Rolle convoyFollowers inklusive der benötigten Portinstanzen[2] kreiert und für diese der Konstruktor initConvoyFollower aufgerufen, wie es der Parameter followers vorgibt. Die neuen Komponenteninstanzen werden über Kompositionskonnektorinstanzen :PosRef mit der soeben erstellten und daher gebundenen Komponenteninstanz leader verbunden. Dazu wird in jeder Schleife zusätzlich eine neue Portinstanz pos/pos:Pos[3] für leader erstellt. Für jede Komponenteninstanz follower muss eine neue :PosCalc-Komponenteninstanz innerhalb von leader an der entsprechenden Position integriert werden. Dazu wird für leader die Rekonfigurationsregel insertPosCalc aufgerufen, welche den Namen für die neue :PosCalc-Komponenteninstanz, deren Position sowie die neue Portinstanz pos/pos:Pos als Argumente übergeben bekommt. Die Übergabe von pos ist notwendig, damit die neu zu erstellende :PosCalc-Instanz mit der soeben erstellten Portinstanz pos verbunden wird und nicht mit einer beliebigen Instanz der gleichen Rolle.

Abb. 7.7 zeigt den Konstruktor initConvoyLeader des Komponententypen RailCab. In

[2] Die Typinformation von Portvariablen wird nicht im Diagramm visualisiert.
[3] Auch die expliziten ≪create≫-Modifizierer von Portvariablen werden nicht angezeigt.

diesem Konstruktor wird die eingebettete Komponenteninstanzstruktur erstellt, welche für die führende Position im RailCab-Konvoi benötigt wird. Dazu werden die Komponenteninstanzen :Integral, :VCtrl und :VRef erstellt und untereinander sowie mit den Portinstanzen :VCurrent und :A verbunden, welche in der aufrufenden Transformation erstellt worden sind. Da die zu Grunde liegenden Porttypen keine Multiports sind und die Portinstanzen somit jeweils höchstens einmal an einer :RailCab-Instanz existieren können, müssen die Portinstanzen nicht als Parameter übergeben werden und können anhand ihrer Typinformation gebunden werden. Die genaue Funktion der einzelnen Komponenteninstanzen ist der Einleitung in Kapitel 1 zu entnehmen.

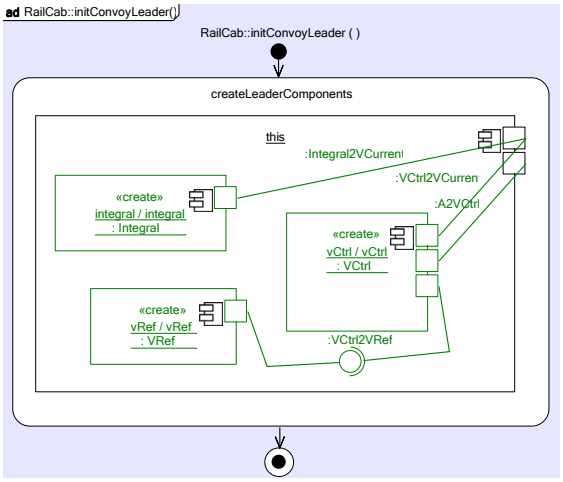


Abbildung 7.7: Konstruktor RailCab::initConvoyLeader

Der Konstruktor RailCab::initConvoyFollower in Abb. 7.8 erstellt die initiale Komponenteninstanzstruktur einer Komponenteninstanz :RailCab, welche für eine der folgenden Positionen im Konvoi benötigt wird. Dazu wird die Komponenteninstanz :PosCtrl erstellt und über Delegationskonnektorinstanzen mit den zuvor erstellten Portinstanzen verbunden.

7.2.3 Modellierung von Rekonfigurationsregeln

Abb. 7.9 zeigt die Rekonfigurationsregel insertFollower des Komponententypen RailCab. Mit dieser Transformation wird die Komponenteninstanz für ein neues RailCab im Konvoi erstellt und an der gewünschten Position in den Konvoi integriert. Zu diesem Zweck werden die Parameter posCalcName, followerName, und position verwendet. Im Storypattern createNewFollower wird die neue Komponenteninstanz newFollower/convoyFollowers:RailCab mit den entsprechenden Portinstanzen erstellt und über :PosRef mit einer neuen Portinstanz pos/pos:Pos der existierenden Komponenteninstanz leader/convoyLeader:RailCab des führenden RailCabs verbunden. Anschließend werden zum einen der bekannte Konstruktor initConvoyFollower auf newFollower sowie die Rekonfigurationsregel insertPosCalc auf leader aufgerufen. Letztere Transformation fügt dabei

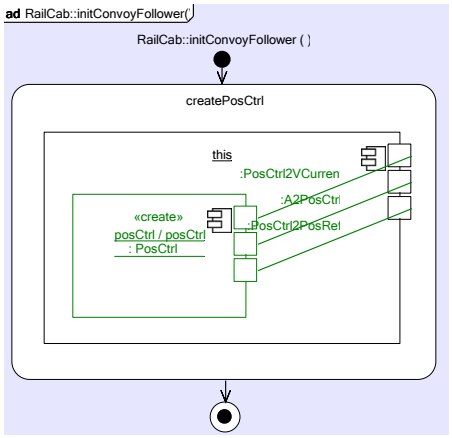


Abbildung 7.8: Konstruktor RailCab::initConvoyFollower

eine neue Komponenteninstanz in die :PosCalc-Liste ein und verbindet sie mit pos. Nach der Ausführung dieses Storypatterns wird eine Statement-Aktivität ausgeführt, welche direkt über eine Anweisung in der Zielsprache den Namen von leader mit followerName belegt. Dies ist notwendig, da Komponententypen nicht über Attribute verfügen und in Komponentenstorydiagrammen somit keine parametrisierten Attributzuweisungen formuliert werden können.

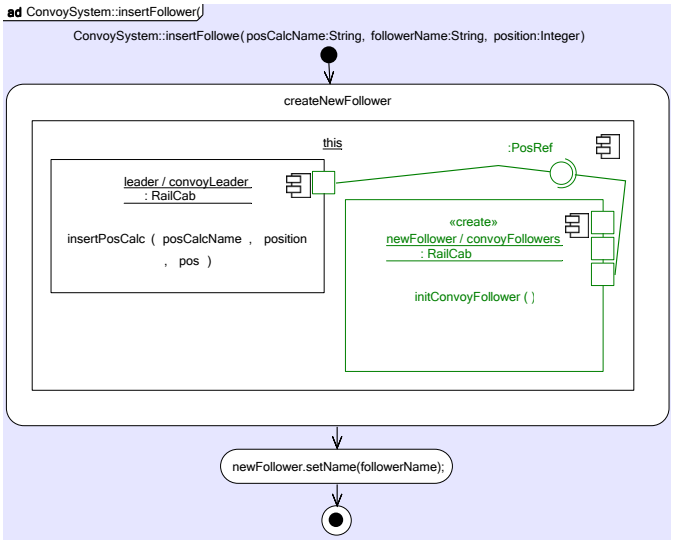


Abbildung 7.9: Rekonfigurationsregel ConvoySystem::insertFollower

Abb. 7.10 zeigt die komplexe Rekonfigurationsregel insertPosCalc des Komponententypen RailCab. Diese erlaubt die Kreierung und das anschließende Einfügen einer neuen Komponenteninstanz /posCalc an einer parametrisierten Position in die korrespondierende Liste. Der Parameter posParam:Pos beinhaltet die Portinstanz /pos:Pos, welche im

aufrufenden Komponentenstorydiagramm erstellt wurde und mit der die neue /posCalc-Instanz verbunden werden soll. Diese Rekonfigurationsregel modelliert im Prinzip das gleiche Verhalten wie die Transformation aus Abb. 6.1 in Abschnitt 6.1. Da allerdings keine komplexen negativen oder optionalen Anwendungsbedingungen von der technischen Umsetzung unterstützt werden, muss der Sonderfall des Hinzufügens einer Instanz am Ende der /posCalc-Liste, welcher von optionalen Anwendungsbedingungen abgedeckt wurde, in einem zusätzlichen Storypattern abgehandelt werden.

Im Storypattern getFirstPosCalc wird zunächst versucht, die erste /posCalc-Komponenteninstanz zu binden. Dies geschieht in diesem Fall nicht wie in Abb. 6.1 mit einer negativen Anwendungsbedingung, sondern über die /integral-Komponenteninstanz. Diese wurde bereits im Konstruktor RailCab::initConvoyLeader instanziiert und liefert immer die erste Position im Konvoi, nämlich die des führenden RailCabs. Kann über den Kompositionskonnektor :Calc2Integral nicht die erste /posCalc-Komponenteninstanz tmp gebunden werden und soll eine neue Komponenteninstanz newCalc/posCalc an der ersten Position erstellt werden, so wird das Storypattern addFirstPosCalc angewendet. Dort wird newCalc als erstes Element der /posCalc-Liste instanziiert und mit /integral und posParam verbunden.

Kann dagegen tmp gebunden werden, so sind zwei Fälle zu unterscheiden. Soll eine neue Instanz newCalc/posCalc an der ersten Position erstellt werden, so wird das Storypattern addPosCalcAtBeginning ausgeführt. Die Instanz newCalc wird in diesem Storypattern zwischen integral und tmp am Beginn der /posCalc-Liste eingefügt und mit posParam verbunden. Soll newCalc dagegen innerhalb oder am Ende der /posCalc-Liste eingefügt werden, so wird in einer Schleife solange im Storypattern getNextPosCalc der Nachfolger next von tmp ermittelt und in einer Statement-Aktivität tmp neu besetzt, bis die durch position vorgegebene Position erreicht ist oder getNextPosCalc nicht mehr erfolgreich angewendet werden kann. Wird die gewünschte Position innerhalb der Liste erreicht, wird das Komponentenstorypattern addPosCalcInbetween ausgeführt, in welchem newCalc instanziiert und zwischen tmp und next eingefügt sowie mit posParam verbunden wird. Ist dagegen getNextPosCalc nicht erfolgreich und soll ein neues Element am Ende der /posCalc-Liste erstellt werden, so wird das Storypattern addPosCalcAtEnd angewendet. Dort wird newCalc nur mit dem Vorgänger tmp und der Portinstanz posParam verbunden. In allen erfolgreichen Fällen wird anschließend der Name von newCalc mit posCalcName besetzt. Vorhandene Konnektorinstanzen werden, falls nötig, durch die neu erstellten Konnektorinstanzen ersetzt, wie in Abschnitt 6.3 beschrieben.

Die Rekonfigurationsregeln ConvoySystem::removeFollower (Abb. 7.12) und RailCab::removePosCalc (Abb. 7.11) sind die Pendants zu den entsprechenden Rekonfigurationsregeln insertFollower und insertPosCalc der jeweiligen Komponententypen. Beim Eintreten eines RailCabs in einen Konvoi wird in insertFollower zunächst auf der Systemebene eine neue Komponenteninstanz /convoyFollowers erstellt und mit einer neuen Portinstanz /pos von /convoyLeader verbunden. Die Position des neuen RailCabs wird erst anschließend durch insertPosCalc innerhalb von /convoyLeader über die Position innerhalb der /posCalc-Liste und über die Verbindung mit der /pos-Portinstanz fixiert. Beim Austritt eines RailCabs aus einem Konvoi dagegen muss anders herum vorgegangen werden. Zunächst muss durch removePosCalc innerhalb der /convoyLeader-Instanz die

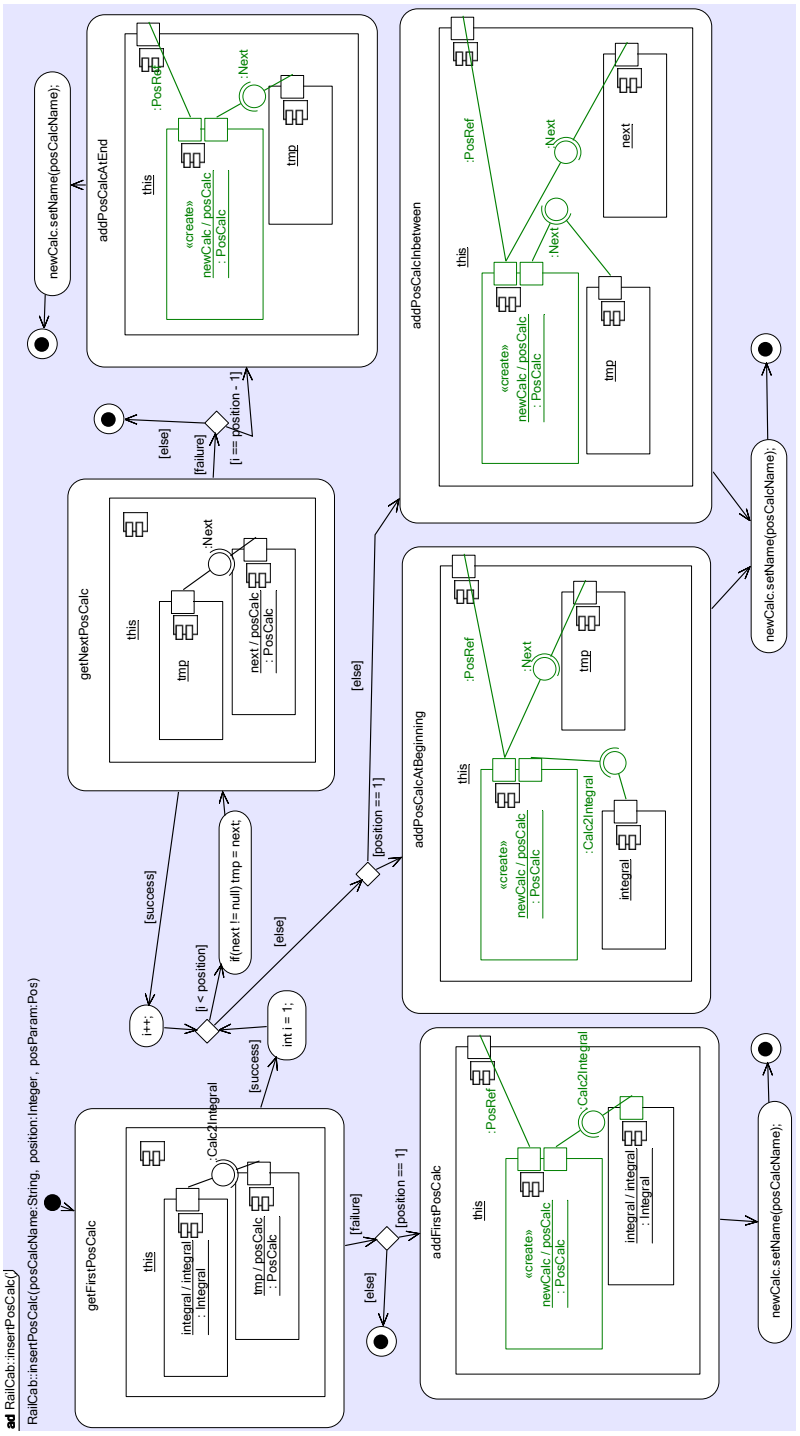


Abbildung 7.10: Rekonfigurationsregel RailCab::insertPosCalc

Komponenteninstanz der /posCalc-Liste an der gewünschten Position gelöscht werden. Anhand der Delegationskonnektorinstanz :PosRef dieser zu zerstörenden Komponenteninstanz kann die Portinstanz ermittelt werden, welche in der aufrufenden Transformation removeFollower – eine Hierarchieebene höher – gelöscht werden soll. In removeFollower lässt sich anhand dieser Portinstanz über die Kompositionskonnektorinstanz :PosRef die zu löschende Komponenteninstanz /convoyFollowers ermitteln. Die beschriebene Vorgehensweise wird über Rückgabedeklarationen modelliert.

Die Transformation removePosCalc wird durch Abb. 7.11 dargestellt. Dieses Komponentenstorydiagramm hat eine ähnliche Struktur wie insertPosCalc. Zunächst wird versucht, die erste /posCalc-Instanz tmp anhand /integral zu binden. Im Gegensatz zu insertPosCalc wird die darauffolgende Schleife aufgrund einer geringfügig anderen Abbruchbedingung mindestens einmal betreten[4]. Somit wird zunächst in getNextPosCalc versucht, den Nachfolger next von tmp in der /posCalc-Liste zu binden. Ist dieses nicht erfolgreich und soll die letzte /posCalc-Instanz gelöscht werden, so wird tmp im Storypattern destroyLastPosCalc zerstört. In der darauffolgenden Stop-Aktivität wird die Portinstanz pos, mit der tmp verbunden war, an die Rückgabedeklaration posPortAtPosition gebunden.

Ist getNextPosCalc dagegen erfolgreich, wird wiederholt tmp neu zugewiesen und next neu gebunden, bis next nicht mehr gebunden werden kann oder die Abbruchbedingung greift. Falls letzteres geschieht und falls die erste /posCalc-Instanz zerstört werden soll, wird das Storypattern destroyFirstPosCalc ausgeführt. Dort wird tmp zwischen integral und der in getNextPosCalc gebundenen Komponenteninstanz next zerstört und – um die Listenstruktur wieder herzustellen – eine neue Kompositionskonnektorinstanz :Next zwischen integral und next kreiert. Anschließend wird die Portinstanz pos zurückgeben. Im Fall dass eine /posCalc-Instanz innerhalb der Liste zerstört werden soll, wird destroyPosCalcInbetween angewendet. In diesem Storypattern wird der Vorgänger prev von tmp gebunden. Anschließend wird tmp zerstört und prev mit next verbunden. Analog zu den anderen Storypatterns wird im Anschluss die korrespondierende Portinstanz zurückgegeben.

Abb. 7.12 zeigt removeFollower. Im Storypattern destroyPosCalc wird removePosCalc auf der Komponenteninstanz leader/convoyLeader aufgerufen. Diese löscht die /posCalc-Instanz an der gewünschten Position und gibt die ursprünglich mit ihr verbundene Portinstanz zurück. Diese wird der Portvariable pos zugewiesen. Konnte destroyPosCalc erfolgreich angewendet werden, erfolgt im Storypattern destroyFollower die Zerstörung der Portinstanz pos von leader, der Kompositionskonnektorinstanz :PosRef und der Komponenteninstanz followerToDelete/convoyFollowers.

7.2.4 Generierte Transformationsdiagramme

Nachdem in den letzten Unterabschnitten alle Komponentenstorydiagramme ausführlich beschrieben wurden, soll im Folgenden ein kurzer Einblick in die generierten Transformationsdiagramme gegeben werden. Abb. 7.13 zeigt den Katalog mit den erstellten

[4] Es wird angenommen, dass position den Wert ≥ 1 hat.

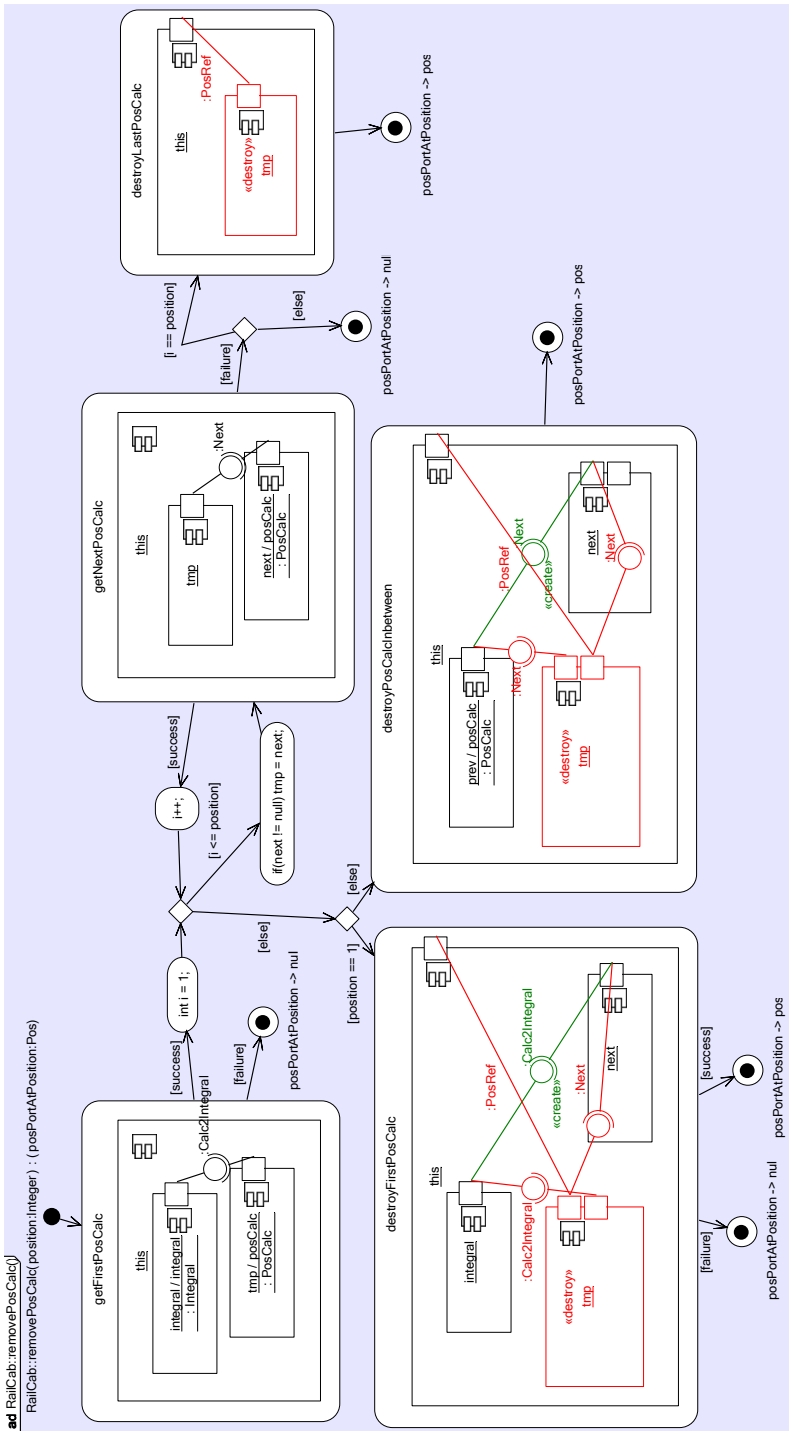


Abbildung 7.11: Rekonfigurationsregel RailCab::removePosCalc

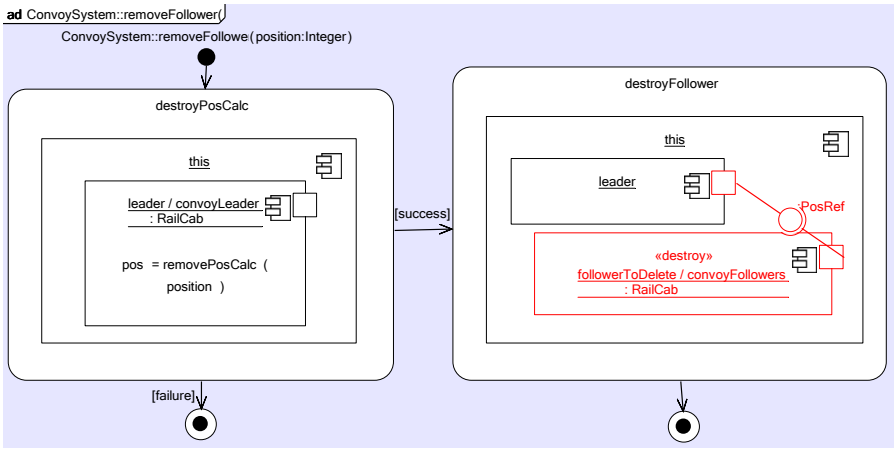


Abbildung 7.12: Rekonfigurationsregel ConvoySystem::removeFollower

Transformationsdiagrammen, in dem die Aufrufabhängigkeiten zwischen den einzelnen Transformationen gut zu erkennnen sind. Diese Abhängigkeiten werden von der zu Grunde liegenden Komponententyphierarchie beeinflusst, so sind die Transformationen in der oberen Hälfte der Abbildung für die Systemebene spezifiziert, die Transformationen der unteren Hälfte für den in der Hierarchie tiefer liegenden Komponententypen RailCab.

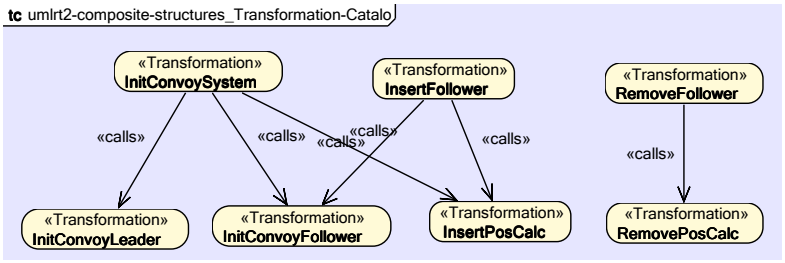


Abbildung 7.13: Generierter Transformationskatalog

Abb. 7.14 stellt exemplarisch das Transformationsdiagramm InsertFollower dar[5]. Obwohl das zu Grunde liegende Komponentenstorydiagramm ConvoySystem::insertFollower das am wenigsten umfangreiche ist, ergibt sich im korrespondierenden Transformationsdiagramm eine hohe Komplexität. Die Komplexität auf den verschiedenen Ebenen wird in Unterabschnitt 7.2.6 genauer untersucht.

7.2.5 Ausführung der Transformationsregeln

Um die Transformationen zu evaluieren, wurden die generierten Transformationsdiagramme auf konkreten Komponenteninstanzen ausgeführt. Zu diesem Zweck wird eine leere

[5] Das Layout von Transformationskatalog und Transformationsdiagramm musste manuell angepasst werden.

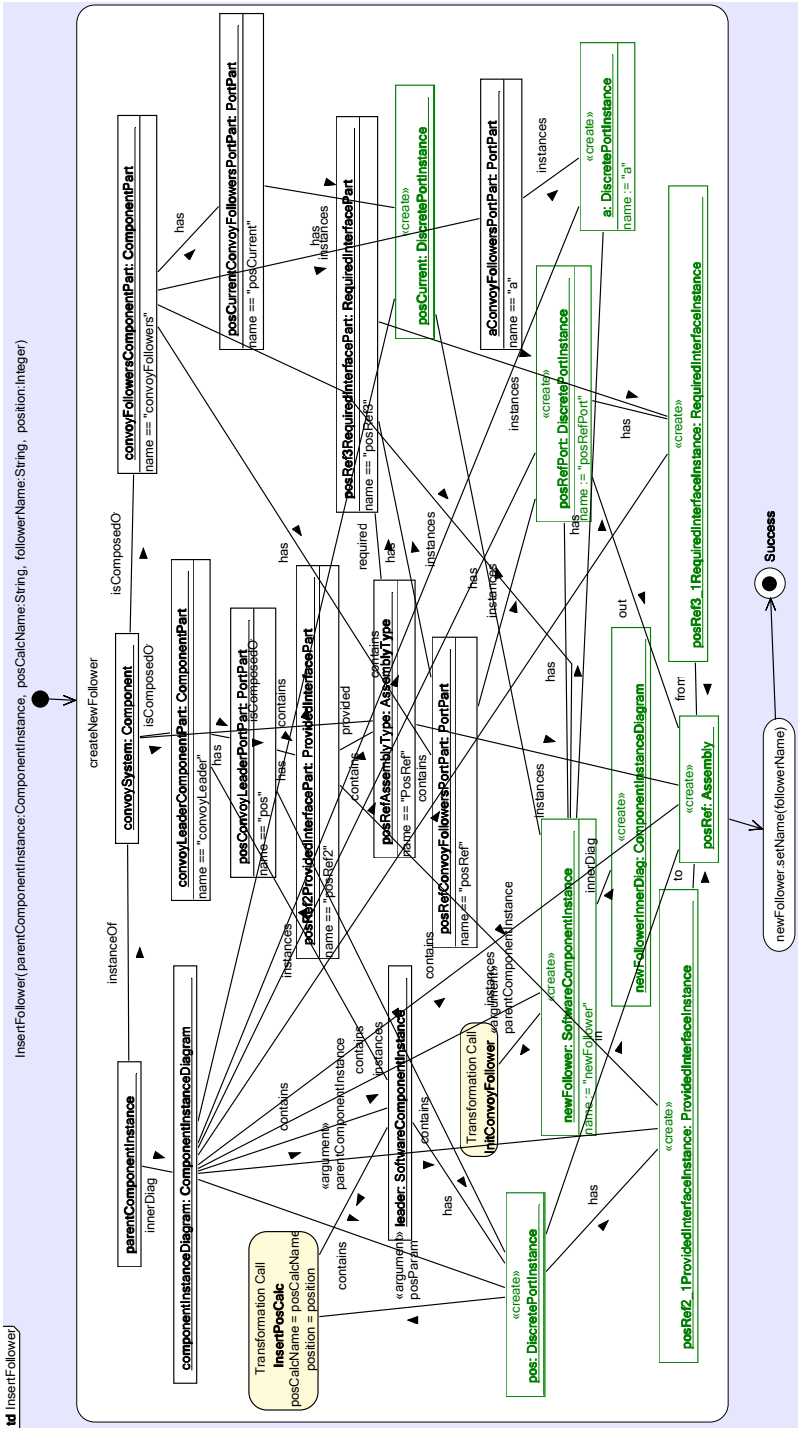


Abbildung 7.14: Generiertes Transformationsdiagramm InsertFollower

Komponenteninstanz :ConvoySystem erstellt, wie in Abb. 7.15 zu sehen ist. Diese Komponenteninstanz repräsentiert die Systemebene und stellt den Einstiegspunkt für den initialen Konstruktor initConvoySystem dar.

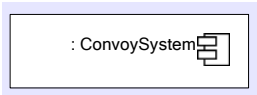


Abbildung 7.15: Systemebene

Ruft man initConvoySystem auf dieser Komponenteninstanz mit dem Argument 5 für den Parameter followers auf, so wird die initiale Konfiguration mit fünf Komponenteninstanzen /convoyFollowers aus Abb. 7.16[6] instanziiert. Alternativ lässt sich diese Konfiguration auch erreichen, indem man z. B. das System zunächst mit drei /convoyFollowers-Instanzen erzeugt und anschließend zwei mal insertFollower auf dem System aufruft.

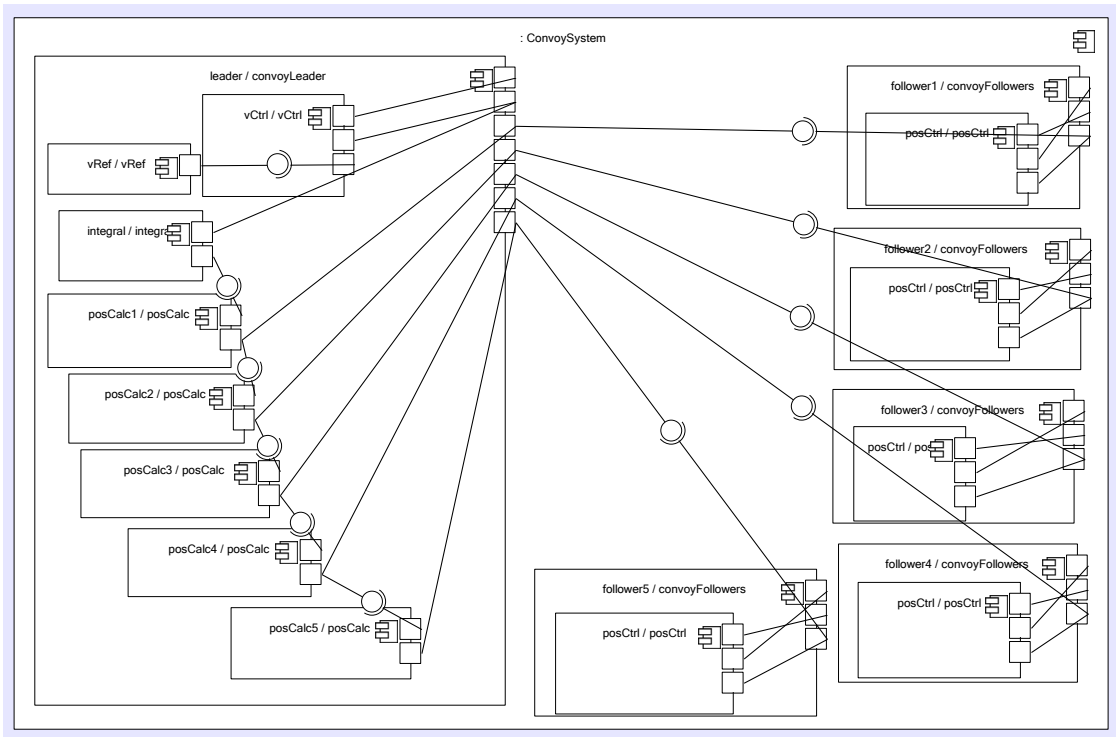


Abbildung 7.16: Komponenteninstanzdiagramm nach Konvoi-Initialisierung mit 5 folgenden RailCabs

Abb. 7.17 zeigt die Konfiguration, nachdem man mit der Transformation removeFollower die Instanzen der Konfiguration zerstört hat, welche relevant für das RailCab an

[6] Auch für Komponenteninstanzdiagramme muss bei dieser Form der Transformation eine manuelle Anpassung des Layouts erfolgen.

der dritten Position war. Fügt man im Anschluss mit insertFollower wieder entsprechende neue Instanzen für ein neues RailCab an der Position 3 hinzu, erhält man wieder eine Konfiguration, welche der aus Abb. 7.16 entspricht.

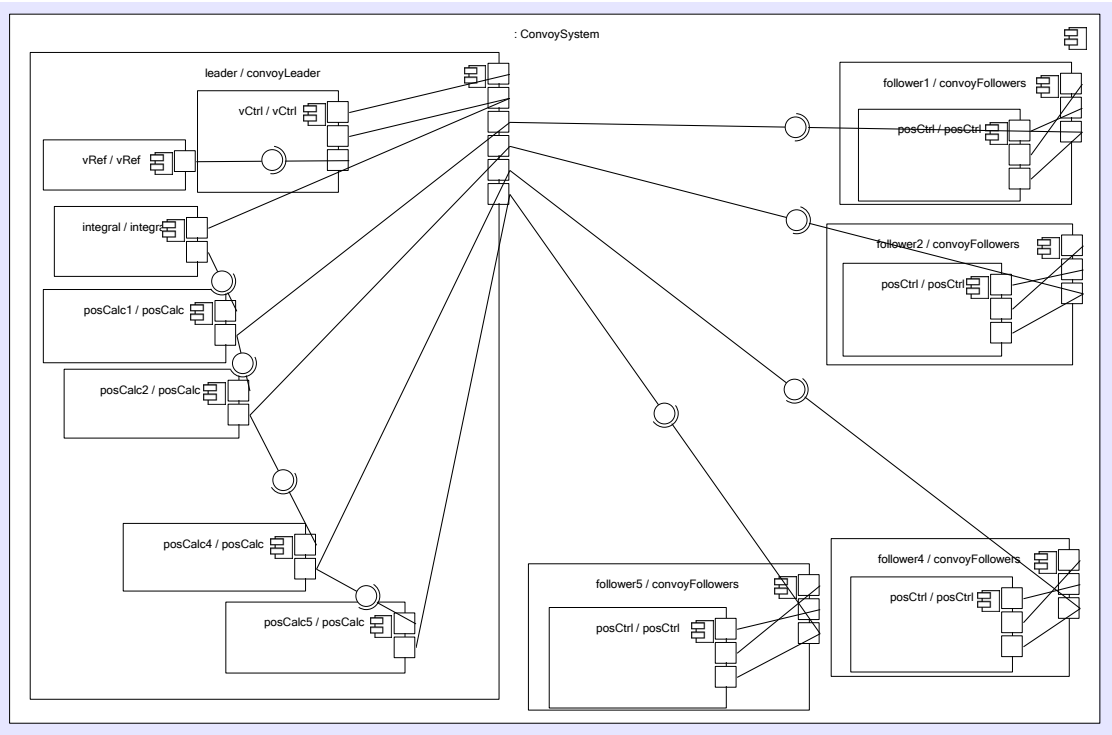


Abbildung 7.17: Komponenteninstanzdiagramm nach dem Entfernen des RailCabs an Position 3

Des Weiteren wurden Fälle wie das Hinzufügen und Entfernen von Instanzen für RailCabs am Anfang oder am Ende eines Konvois evaluiert. Die entsprechenden Screenshots werden an dieser Stelle aus Platzgründen nicht gezeigt.

7.2.6 Metriken

Wie aus Unterabschnitt 7.2.4 hervorgeht, sind die generierten Transformationsdiagramme im Vergleich zu den korrespondierenden Komponentenstorydiagrammen sehr komplex, was durch die große Anzahl der beim Mapping generierten Objektvariablen und Links verursacht wird. Aus dem in Abschnitt 6.5 vorgestellten Mapping lässt sich ableiten, wie viel Objektvariablen und Links pro Komponenten-/Portvariable und Konnektorlink erstellt werden. Hierbei spielt auch die Anzahl der verwendeten Parts und Konnektortypen eine Rolle, da beim Mapping innerhalb eines Storypatterns für mehrere Variablen/Konnektorlinks des gleichen Parts/Typs höchstens eine Objektvariable zur Repräsentation des Parts/Typs erstellt wird. Um an dieser Stelle eine schnell zu erfassende

Übersicht zu geben, wird diese Relation nicht formalisiert, sondern die experimentellen Ergebnisse des Evaluierungsbeispiels angegeben.

In Tabelle 7.2 werden diese Ergebnisse in Form von Metriken zusammengefasst. Dabei wird für jede Transformation aus Spalte 1 die Anzahl der im Komponentenstorydiagramm verwendeten Komponenten-/Portvariablen und Konnektorlinks in Spalte 2 notiert. Spalte 3 listet die Anzahlen der Objektvariablen und Links der jeweils übersetzten Transformationsdiagramme auf. Spalte 4 gibt letztendlich die Anzahlen der Codezeilen (*Lines of Code, LoC*) an, welche aus den Transformationsdiagrammen generiert wurden. In der letzten Zeile werden diese Daten zusammenaddiert.

Transformation	**Variablen und Konnektorlinks**	**Objektvariablen und Links**	**LoC**
ConvoySystem::initConvoySystem	12	89	760
ConvoySystem::insertFollower	8	69	605
ConvoySystem::removeFollower	9	71	618
RailCab::initConvoyFollower	11	89	695
RailCab::initConvoyLeader	15	137	968
RailCab::insertPosCalc	52	482	2806
RailCab::removePosCalc	42	389	2422
Gesamt	**149**	**1326**	**8874**

Tabelle 7.2: Gegenüberstellung der Variablen-/Link-Anzahl von Komponentenstory- und Transformationsdiagrammen und der generierten Lines of Code

Dabei ist zu beachten, dass zwischen den Variablen und Konnektorlinks auf Ebene der Komponentenstorydiagramme und den Objektvariablen und Links der Transformationsdiagramme eine direkte Relation durch das Mapping besteht. Die generierten LoC haben ebenfalls Bezug zur Anzahl der verwendeten Variablen, da jede Objektvariable deklariert werden muss. Allerdings hängen die LoC in weitaus größerem Maße von den modellierten Kontrollstrukturen wie Schleifen oder forEach-Aktivitäten ab. Ein weiterer Faktor, der bei der Codegenerierung die LoC erhöhen kann, ist das zu Grunde liegende Metamodell. So wird für das Binden eines Objekts über einen Link einer $:*$-Assoziation eine Schleife anstatt eines simplen Methodenaufrufs generiert. Die LoC sollen an dieser Stelle lediglich einen generellen Anhaltspunkt für den Vergleich zur Anzahl der Variablen geben.

Die Spalten der Übersicht mit den experimentellen Ergebnissen stellen Metriken für die Komplexität der jeweiligen Diagramme dar und zeigen somit, wie die Verlagerung der Graphtransformationen von der Objektebene auf die Komponenteninstanzebene die Komplexität der Modellierung reduzieren kann.

8 Zusammenfassung und Ausblick

Die bis dato in MECHATRONIC UML eingesetzte strukturelle Modellierung fixer, hierarchischer Komponententypen beschränkte die Größe möglicher Konfigurationsmengen. Die Spezifikation der darauf aufbauenden Strukturanpassung eines Systems in Form von hybriden Rekonfigurationscharts ist für größere Konfigurationsmengen unpraktikabel und auf Konfigurationen limitiert, welche zum Modellierungszeitpunkt bekannt sein müssen.

Die in dieser Diplomarbeit in Kapitel 5 vorgestellte Erweiterung hierarchischer Komponententypen um Charakteristika der strukturierten Klassen der UML Kompositionsstrukturen räumt die Größenbeschränkung der möglichen Konfigurationsmengen aus. Um dies zu erreichen, wurde ein Rollenkonzept für eingebettete Elemente eines hierarchischen Komponententypen in Form von Parts eingeführt. Über diese Parts können in einen variablen, hierarchischen Komponententypen verschiedene Rollen weiterer Komponententypen eingebettet werden. Durch die Zuordnung von Multiplizitäten zu Komponenten-Parts und Porttypen ist die Anzahl der korrespondierenden Instanzen pro übergeordneter Komponenteninstanz nicht mehr auf einen fixen Wert beschränkt. Dies steigert die Wiederverwendbarkeit und vereinfacht die Modellierung von Komponententypen, dessen Instanzen aus mehreren eingebettete Komponenteninstanzen desselben Typs komponiert werden oder mehrere gleichartige Portinstanzen besitzen sollen.

Die durch die neue Variabilität der Komponententypdefinition ermöglichte beliebige Größe der Konfigurationsmengen macht die Beschreibung von Differenzen zwischen den einzelnen Konfigurationen anstatt der Spezifikation jeder möglichen Konfiguration sinnvoll. Hierfür wurden in Kapitel 6 die in [THHO08] vorgestellten Komponentenstorydiagramme weiterentwickelt und konzeptionell sowie technisch umgesetzt. Diese neue, auf konventionellen Storydiagrammen basierende Transformationssprache für Komponenteninstanzstrukturen der MECHATRONIC UML ermöglicht die Modellierung und Ausführung zur Entwurfszeit von Transformationen zwischen Konfigurationen. Gleichzeitig müssen die möglichen Konfigurationen nicht a priori bekannt sein und können sich durch die Anwendung der Transformationen dynamisch entwickeln.

Konstruktoren erlauben die initiale Instanziierung der eingebetteten Kompositionsstruktur einer neu erstellten Komponenteninstanz. Rekonfigurationsregeln verändern die eingebettete Struktur von vorhandenen Komponenteninstanzen. Um die Kapselung der Komponenten zu respektieren, transformieren Komponentenstorydiagramme lediglich die eingebettete Struktur einer Komponenteninstanz. Daher werden Transformationen über mehrere Komponentenhierarchien über Aufrufe von Komponentenstorydiagrammen abgewickelt, wodurch komplette Konfiguration initialisiert und transformiert werden können. Durch die explizite Typisierung von Variablen und Konnektorlinks bei der Modellierung der Transformationen, durch die Berücksichtigung kompositionaler Beziehungen und der Wohlgeformtheit eines Komponenteninstanzgraphen sowie durch die Ersetzung

existierender Instanzen aufgrund oberer Multiplizitätsgrenzen von 1 wird weitestgehend sichergestellt, dass die von Komponentenstorydiagrammen generierten Konfigurationen typkonform sind. Die Verwendung der Kontrollstrukturen der konventionellen Storydiagramme ermöglicht auch komplexe Transformationen. Die Sprache vereint die konkrete Syntax der MECHATRONIC UML Komponentendiagramme mit der graphischen Notation der herkömmlichen Storydiagramme, was zu einer intuitiven Repräsentation bei einer geringen visuellen Komplexität führt. Gleichzeitig entsteht kein Bruch zwischen den Sprachen zur Spezifikation und Transformation von Komponenten. Wie das Evaluierungsbeispiel aus Kapitel 7 gezeigt hat, erlaubt die variable Typdefinition die Etablierung programmiersprachlicher Konstrukte wie Listen auf Komponententypebene, während mit Komponentenstorydiagrammen Zugriffsoperationen auf diese Konstrukte spezifiziert werden können. Der bisher verwendete Ansatz zur Komponententypdefinition und Strukturanpassung kann nach wie vor verwendet werden und stellt damit eine Alternative für wenige und zum Modellierungszeitpunkt bekannte Konfigurationen dar.

Mit der Kombination aus der beschriebenen variablen Komponententypdefinition und der neuen Transformationssprache wurden somit die Einschränkungen der Strukturmodellierung und -anpassung beseitigt. Die vorgestellten neuen Konzepte bieten viele Ansatzpunkte für weiterführende Ansätze. Im Folgenden werden daher einige mögliche Erweiterungen und Ideen für zukünftige Arbeiten vorgestellt.

Ausblick

In Unterabschnitt 6.5.5 wurde ein grundlegendes Konzept für das Mapping von Bereichen mit negativen und optionalen Anwendungsbedingungen auf die ausführbaren Transformationsdiagramme erarbeitet. In der Bearbeitungszeit dieser Arbeit konnte dieses Konzept nicht technisch umgesetzt werden, daher musste das Evaluierungsbeispiel mit zusätzlichen Storypatterns für bestimmte Sonderfälle der Nichtexistenz von Komponenteninstanzen modelliert werden. Somit bietet sich als erster offener Punkt eine Implementierung und Evaluierung dieses Konzepts an. Auch der Formalismus der Mengenobjekte der konventionellen Storydiagramme kann sich insbesondere für Variablen eines Multiparts oder -ports als sinnvolle Ergänzung erweisen. Auch für Mengenvariablen müssten beim Mapping Bereiche kompositional abhängiger Variablen oder Konnektorlinks gebildet und diese einzeln in entsprechenden Schleifen gebunden werden.

Die Strukturanpassung mit hybriden Rekonfigurationscharts kann mit syntaktischen Überprüfungen und ergänzendem Modelchecking auf *Konsistenz* überprüft werden [GBSO04, BGT05, Bur05]. Die Einführung von Graphtransformationen für die Strukturanpassung von Konfigurationen erfordert neue Ansätze zur Verifizierung der Korrektheit. Mit einem für Komponentenstorydiagramme angepassten Graphmodelchecker wie GROOVE können bestimmte Eigenschaften des generierbaren Zustands- bzw. *Konfigurationsraums* überprüft werden. Hierfür wird allerdings ein Komponenteninstanzgraph als Eingabe benötigt, und man ist evtl. mit unendlichen Konfigurationsräumen konfrontiert. Diese Probleme würde eine für Komponentenstorydiagramme adaptierte Version der Verifikationsansätze für konventionelle Storypatterns [Sch06] bzw. Transformations-

diagramme [Det08] lösen. Mit einem solchen Ansatz könnte man z. B. überprüfen, dass die generierten Konfigurationen konkrete untere oder obere Multiplizitätsgrenzen einhalten. Dazu würde man ein *unerwünschtes Komponenteninstanzmuster* modellieren, welches eine Komponentenvariablenstruktur mit einer außerhalb der auf Typebene vorgegebenen Multiplizitätsgrenzen liegenden Variablenanzahl enthält. Dieses Komponenteninstanzmuster lässt sich auf ein unerwünschtes Objektmuster für Transformationsdiagramme übertragen. Mit dem in [Det08] vorgestellten Verifikationsansatz für Transformationsdiagramme kann schließlich überprüft werden, ob durch ein Transformationsdiagramm eine dem Muster entsprechende Objektstruktur erstellt werden kann. Um dieses Vorgehen auch bei hierarchieübergreifenden Aufrufen zu ermöglichen, wird allerdings eine Erweiterung der Verifikation von Transformationsdiagrammen um die Unterstützung von Transformation Calls benötigt. Mit einer solchen Methode könnte die Sicherstellung der Typkonformität generierter Konfigurationen vervollständigt werden. Vgl. hierzu auch die Unterabschnitte 2.1.1 und 2.1.2.

Die in dieser Arbeit entwickelte Strukturanpassung wird zur Entwurfszeit ausgeführt. Eine Ausführung zur Laufzeit erfordert einen neuen Formalismus zur Initiierung und Selektion einer Transformation aufgrund bestimmter in einem System aufgetretener Ereignisse (siehe Abschnitt 3.2). Dies übernimmt momentan der Benutzer. Ein solcher Formalismus könnte z. B. an den der hybriden Rekonfigurationscharts angelehnt sein, mit dem sowohl ein Echtzeit- als auch ein hybrides Verhalten modelliert werden kann. Ein Komponentenstorydiagramm ließe sich als Seiteneffekt einer Transition aufrufen. Die aktuell von hybriden Rekonfigurationscharts unterstützte Strukturanpassung könnte dabei alternativ zu diesen Seiteneffekten unterstützt werden. Eine weitere Möglichkeit ist die Integration von Komponentenstorydiagrammen in *Storycharts* [KNNZ00], einer Variante von Statecharts, bei der die do()-Methode eines Zustands mit konventionellen Storypatterns beschrieben wird. Im Fall eines Echtzeitsystems müsste die maximale Anzahl der Komponenten- und Portinstanzen eines Systems beschränkt werden, um ein vorhersagbares Echtzeitverhalten zu ermöglichen [THHO08, TGS06]. Für ein Scheduling der Transformationen ist das Mapping so abzuändern, dass das zu Grunde liegende Echtzeitbetriebssystem bei der Transformationsausführung über entsprechende Aufrufe mit einbezogen wird [THHO08].

Der Sprachumfang von Komponentenstorydiagrammen sowie die Komponententypdefinition könnten ebenfalls erweitert werden. Vorbild hierfür könnten die Ausdrucksmöglichkeiten der konventionellen Storydiagramme und der zu Grunde liegenden Klassendiagramme sein. Zum einen könnten für Komponententypen Attribute spezifiziert werden. In Komponentenstorydiagrammen könnten dann Komponenteninstanzen Attributwerte zugewiesen werden oder mit Attributbedingungen die Auswahl möglicher Anwendungsstellen eingeschränkt werden. Die Zuweisung eines Attributs geschieht momentan in Form einer Namenszuweisung über den Umweg mit Statement-Aktivitäten. Zum anderen ist ein Vererbungskonzept für Komponententypen denkbar. Herkömmliche Storydiagramme berücksichtigen beim Binden einer LHS die Vererbungshierarchie, d. h. auch Objekte von Subklassen des Typs einer Objektvariable werden gebunden. Ein analoges Vererbungskonzept könnte für Komponententypen eingeführt werden, was z. B. bereits in der ADL LEDA [CPT99] praktiziert wird. Eine letzte Erweiterung ist die der geordneten

Konnektortypen. Klassendiagramme unterstützen geordnete Assoziationen, welche von Storypatterns für den simplen Zugriff auf bestimmte Objektpositionen in Listenstrukturen ausgenutzt werden können. Ein solcher Mechanismus würde weitere Vorteile im Hinblick auf Szenarien wie das Evaluierungsbeispiel bieten, in dem ebenfalls Listen zur Verwaltung von RailCabs benutzt wurden.

Durch den Umweg des Mappings auf Transformationsdiagramme und der anschließenden Codegenerierung wird die Benutzung von Statement-Aktivitäten, Guards oder Constraints erschwert, falls diese Code enthalten, der sich auf die modellierten Variablen und Konnektorlinks bezieht. In konventionellen Storydiagrammen wird direkt Code aus der modellierten Variablenstruktur generiert, so dass mit dem vorhandenen Wissen über das zu Grunde liegende Metamodell Code für Statement-Aktivitäten, Guards oder Constraints geschrieben werden kann, der sich auf diese Variablenstruktur bezieht. Bei Komponentenstorydiagrammen muss man dagegen im Vorfeld wissen, auf was für eine Objektvariablenstruktur in den Transformationsdiagrammen die Variablen und Links abgebildet werden, um einen korrekten Code zu schreiben, welcher beim Mapping direkt in den Code des Transformationsdiagramms kopiert wird. Eventuelle syntaktische Fehler fallen erst bei der Kompilierung des aus den Transformationsdiagrammen generierten Codes auf. Um dieses Problem zu umgehen, wäre ein Parser wie das in [Opp06] für FUJABA entwickelte Plugin hilfreich. Ein solcher Parser könnte abhängig von den im Komponentenstorydiagramm verwendeten Variablen und Konnektorlinks die Syntax des in Statement-Aktivitäten, Guards oder Constraints verwendeten Codes in Bezug auf das daraus generierte Transformationsdiagramm überprüfen.

Letztendlich ist der Einsatz von Komponentenstorydiagrammen im Umfeld der Gefahrenanalyse denkbar, welche wie die Komponentenstorydiagramme zur Entwurfszeit ausgeführt wird. Mit Komponentenstorydiagrammen können, wie in der Evaluierung in Kapitel 7 gezeigt, auf komfortable und schnelle Weise unterschiedliche komplexe Konfigurationen erzeugt werden. Diese könnten auf mögliche Gefahren untersucht und im Bedarfsfall mit vorgegeben Transformationen strukturell angepasst werden, um eine sicherere Konfiguration zu erhalten. Ein weiteres Szenario ist die Anwendung von *Fehlertoleranzmustern* [Tic06] auf eine gegebene Konfiguration. Ein solches Fehlertoleranzmuster macht ähnlich einem Entwurfsmuster [GHJV95] bestimmte Vorgaben bzgl. Struktur, Verteilungsbeschränkungen und Verhalten, um eine Fehlertoleranztechnik in ein System zu integrieren. Für die strukturelle Anpassung wäre eine erweiterte Version von Komponentenstorydiagrammen hilfreich. Es kann sein, dass die strukturellen Änderungen des Fehlertoleranzmusters eine Anpassung der Komponententypdefinition erfordert. In diesem Fall würde zunächst vor der Konfigurationstransformation durch das Fehlertoleranzmuster die zu Grunde liegende Komponententypdefinition so verändert, dass die Konfiguration nach der Transformation typkonform ist. Im Anschluss wird mit erweiterten Komponentenstorydiagrammen die Konfiguration entsprechend dem Fehlertoleranzmuster strukturell angepasst, indem die zu verändernde Anwendungsstelle der Transformation übergeben wird. Dies würde eine Erweiterung der Komponentenstorydiagramme um parametrisierte, generische Typen und Parts sowie eine entsprechende Anpassung des Mappings erfordern.

Literaturverzeichnis

[BCDW04] Bradbury, Jeremy S. ; Cordy, James R. ; Dingel, Juergen ; Wermelinger, Michel: A Survey of Self-Management in Dynamic Software Architecture Specifications. In: *WOSS '04: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems*. New York, NY, USA : ACM Press, 2004. – ISBN 1–58113–989–6, S. 28–33

[BG03] Burmester, Sven ; Giese, Holger: The Fujaba Real-Time Statechart Plug-In. In: Giese, Holger (Hrsg.) ; Zündorf, Albert (Hrsg.): *Proc. of the first International Fujaba Days 2003, Kassel, Germany* Bd. tr-ri-04-247, Universität Paderborn, Oktober 2003 (Technical Report), S. 1–8

[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

[BGO04] Burmester, Sven ; Giese, Holger ; Oberschelp, Oliver: Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In: Araujo, Helder (Hrsg.) ; Vieira, Alves (Hrsg.) ; Braz, Jose (Hrsg.) ; Encarnacao, Bruno (Hrsg.) ; Carvalho, Marina (Hrsg.): *Proc. of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, INSTICC Press, August 2004, S. 222–229

[BGS05] Burmester, Sven ; Giese, Holger ; Schäfer, Wilhelm: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: *Proc. of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany* Bd. 3748, Springer Verlag, November 2005 (Lecture Notes in Computer Science), S. 25–40

[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, August 2005 (Lecture Notes in Computer Science), S. 47–61

[BMR+96] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-oriented Software Architecture: A System of*

Patterns. New York, NY, USA : John Wiley & Sons, 1996. – ISBN 0–471–95869–7

[BRJ99] Booch, Grady ; Rumbaugh, James ; Jacobson, Ivar: *The Unified Modeling Language User Guide*. Redwood City, CA, USA : Addison-Wesley, 1999. – ISBN 0–201–57168–4

[Bur05] Burmester, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems*, Universität Paderborn, Diss., 2005

[Bus02] Busatto, Giorgio: *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*, Universität Paderborn, Diss., 2002

[CEER96] Cuny, Janice E. (Hrsg.) ; Ehrig, Hartmut (Hrsg.) ; Engels, Gregor (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Graph Gramars and Their Application to Computer Science, 5th International Workshop, Williamsburg, VA, USA, November 13-18, 1994, Selected Papers*. Bd. *1073*. Springer, 1996 (Lecture Notes in Computer Science). – ISBN 3–540–61228–9

[Cle96] Clements, Paul C.: A Survey of Architecture Description Languages. In: *IWSSD '96: Proceedings of the 8th International Workshop on Software Specification and Design*. Washington, DC, USA : IEEE Computer Society, 1996. – ISBN 0–8186–7361–3, S. 16

[CPT99] Canal, Carlos ; Pimentel, Ernesto ; Troya, José M.: Specification and Refinement of Dynamic Software Architectures. In: *WICSA1: Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Deventer, The Netherlands, The Netherlands : Kluwer, B.V., 1999. – ISBN 0–7923–8453–9, S. 107–126

[Det08] Detten, Markus von: *Verifikation von Transformationsspezifikationen*, Universität Paderborn, Institut für Informatik, Paderborn, Deutschland, Diplomarbeit, Januar 2008

[DHP02] Drewes, Frank ; Hoffmann, Berthold ; Plump, Detlef: Hierarchical Graph Transformation. In: *Journal of Computer and System Sciences* 64 (2002), Nr. 2, S. 249–283. `http://dx.doi.org/10.1006/jcss.2001.1790`. – DOI 10.1006/jcss.2001.1790. – ISSN 0022–0000

[EH00] Engels, Gregor ; Heckel, Reiko: Graph Transformation as a Conceptual and Formal Framework for System Modeling and Model Evolution. In: *ICALP '00: Proceedings of the 27th International Colloquium on Automata, Languages and Programming*. London, UK : Springer, 2000. – ISBN 3–540–67715–1, S. 127–150

[EPS73] Ehrig, Hartmut ; Pfender, Michael ; Schneider, Hans J.: Graph-Grammars: An Algebraic Approach. In: *14th Annual Symposium on Switching and Automata Theory*, 1973, S. 167–180

[FGK+04] Frank, Ursula ; Giese, Holger ; Klein, Florian ; Oberschelp, Oliver ; Schmidt, Andreas ; Schulz, Bernd ; Vöcking, Henner ; Witting, Katrin ; Gausemeier, Jürgen (Hrsg.): *Selbstoptimierende Systeme des Maschinenbaus - Definitionen und Konzepte*. 1. Auflage. Paderborn, Deutschland : Bonifatius GmbH, 2004 (HNI-Verlagsschriftenreihe Band 155)

[FNT98] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*, Universität Paderborn, Institut für Mathematik und Informatik, Paderborn, Deutschland, Diplomarbeit, Juli 1998

[FNTZ98] Fischer, Thorsten ; Niere, Jörg ; Torunski, Lars ; Zündorf, Albert: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Engels, Gregor (Hrsg.) ; Rozenberg, Grzegorz (Hrsg.): *Proc. of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany*, Springer, November 1998 (LNCS 1764), S. 296–309

[Gar95] Garlan, David: What is Style? In: *Proceedings of the Dagstuhl Workshop on Software Architecture*. Saarbrücken, Deutschland, Februar 1995

[GB00] Gosling, James ; Bollella, Greg: *The Real-Time Specification for Java*. Boston, MA, USA : Addison-Wesley, 2000. – ISBN 0201703238

[GB03] Giese, Holger ; Burmester, Sven: Real-Time Statechart Semantics / Lehrstuhl für Softwaretechnik, Universität Paderborn. Paderborn, Deutschland, Juni 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

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. Boston, MA : Addison-Wesley, 1995. – ISBN 0201633612

[GMK02] Georgiadis, Ioannis ; Magee, Jeff ; Kramer, Jeff: Self-Organising Software Architectures for Distributed Systems. In: *WOSS '02: Proceedings of the first workshop on Self-healing systems*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–609–9, S. 33–38

[Gru04] Grunske, Lars: *Strukturorientierte Optimierung der Qualitätseigenschaften von softwareintensiven technischen Systemen im Architekturentwurf*, Universität Potsdam, Diss., 2004

[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, September 2006 (Lecture Notes in Computer Science), S. 156–169

[GTB+03] Giese, Holger ; Tichy, Matthias ; Burmester, Sven ; Schäfer, Wilhelm ; Flake, Stephan: Towards the Compositional Verification of Real-Time UML Designs. In: *Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland*, ACM Press, September 2003, 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, September 2004 (Lecture Notes in Computer Science)

[Har87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. In: *Science of Computer Programming* 8 (1987), Nr. 3, S. 231–274. `http://dx.doi.org/10.1016/0167-6423(87)90035-9`. – DOI 10.1016/0167–6423(87)90035–9. – ISSN 0167–6423

[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. 1–8. – to appear

[Hoa78] Hoare, C. A. R.: Communicating Sequential Processes. In: *Communications of the ACM* 21 (1978), Nr. 8, S. 666–677. `http://dx.doi.org/10.1145/359576.359585`. – DOI 10.1145/359576.359585. – ISSN 0001–0782

[ICG+04] Ivers, James ; Clements, Paul ; Garlan, David ; Nord, Robert ; Schmerl, Bradley ; Silva, Jaime: Documenting Component and Connector Views With UML 2.0 / Carnegie Mellon, Software Engineering Institute. 2004 (CMU/SEI-2004-TR-008). – Forschungsbericht

[KNNZ00] Köhler, Hans J. ; Nickel, Ulrich A. ; Niere, Jörg ; Zündorf, Albert: Integrating UML Diagrams for Production Control Systems. In: *Proc. of the* 22^{nd} *International Conference on Software Engineering (ICSE), Limerick, Ireland*, ACM Press, 2000, S. 241–251

[KR06a] Kastenberg, Harmen ; Rensink, Arend: Model Checking Dynamic States in GROOVE. In: Valmari, A. (Hrsg.): *Model Checking Software (SPIN), Vienna, Austria* Bd. 3925. Berlin : Springer, 2006 (Lecture Notes in Computer Science). – ISBN 3–540–33102–6, S. 299–305

[Krä06b] Krämer, Helmer: *Laufzeitanpassungen von Softwarestrukturen für mechatronische Systeme*, Universität Paderborn, Institut für Informatik, Paderborn, Deutschland, Diplomarbeit, 2006

[Löw93] Löwe, Michael: Algebraic approach to single-pushout graph transformation. In: *Theoretical Computer Science* 109 (1993), Nr. 1-2, S. 181–224. http://dx.doi.org/10.1016/0304-3975(93)90068-5. – DOI 10.1016/0304–3975(93)90068–5. – ISSN 0304–3975

[MDEK95] Magee, Jeff ; Dulay, Naranker ; Eisenbach, Susan ; Kramer, Jeff: Specifying Distributed Software Architectures. In: *Fifth European Software Engineering Conference, ESEC '95 , Barcelona*, 1995

[MDK94] Magee, Jeff ; Dulay, Naranker ; Kramer, Jeff: Regis: A Constructive Development Environment for Distributed Programs. In: *Distributed Systems Engineering Journal* 1 (1994), Nr. 5. http://pubs.doc.ic.ac.uk/Regis/

[Mét96] Métayer, Daniel L.: Software architecture styles as graph grammars. In: *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*. New York, NY, USA : ACM Press, 1996. – ISBN 0–89791–797–9, S. 15–23

[Mey06] Meyer, Matthias: Pattern-based Reengineering of Software Systems. In: *Proceedings of the 13th Working Conference on Reverse Engineering (WCRE 2006), Benevento, Italy*, IEEE Computer Society, Oktober 2006, S. 305–306

[MT00] Medvidovic, Nenad ; Taylor, Richard N.: A Classification and Comparison Framework for Software Architecture Description Languages. In: *IEEE Transactions on Software Engineering* 26 (2000), Nr. 1, S. 70–93. http://dx.doi.org/10.1109/32.825767. – DOI 10.1109/32.825767. – ISSN 0098–5589

[Obj05] Object Management Group: *UML Profile for Schedulability, Performance, and Time Specification, Version 1.1*, Januar 2005. http://www.omg.org/cgi-bin/doc?formal/2005-01-02. – OMG document number: formal/05-01-02

[Obj07a] Object Management Group: *Normative SysML 1.0 Specification*, September 2007. http://www.omg.org/spec/SysML/1.0/PDF/. – OMG document number: formal/2007-09-01

[Obj07b] Object Management Group: *UML 2.1.2 Superstructure Specification*, November 2007. http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/. – OMG document number: formal/2007-11-02

[Opp06] Oppermann, Patrick: *Parsing and Analysing UML Text language in CASE-Tools*. Oktober 2006

[Roz97] Rozenberg, Grzegorz (Hrsg.): *Handbook of Graph Grammars and Computing by Graph Transformation : Foundations.* World Scientific Pub Co, 1997 http://www.amazon.com/exec/obidos/ASIN/9810228848/o/qid=966516781/sr=8-3/ref=aps_sr_b_1_3/102-9896583-3184124. – ISBN 9810228848. – Volume 1

[Sch95] Schürr, Andy: Specification of Graph Translators with Triple Graph Grammars. In: *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science.* London, UK : Springer, 1995. – ISBN 3–540–59071–4, S. 151–163

[Sch06] Schilling, Daniela: *Kompositionale Softwareverifikation mechatronischer Systeme*, Universität Paderborn, Diss., 2006

[Sel98] Selic, Bran: Using UML for Modeling Complex Real-Time Systems. In: *LCTES '98: Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems.* London, UK : Springer, 1998. – ISBN 3–540–65075–X, S. 250–260

[SG96] Shaw, Mary ; Garlan, David: *Software Architecture: Perspectives on an Emerging Discipline.* Upper Saddle River, NJ, USA : Prentice-Hall, Inc., 1996. – ISBN 0–13–182957–2

[SGW94] Selic, Bran ; Gullekson, Garth ; Ward, Paul T.: *Real-Time Object-Oriented Modeling.* John Wiley & Sons, 1994. – ISBN 0–471–59917–4

[Sto96] Storey, Neil: *Safety-Critical Computer Systems.* Boston, MA, USA : Addison-Wesley, 1996 http://www.amazon.com/exec/obidos/ASIN/0201427877/qid=1011279198/sr=8-2/ref=sr_8_3_2/002-6219168-0080849. – ISBN 0201427877

[Szy98] Szyperski, Clemens: *Component Software: Beyond Object-Oriented Programming.* New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 1998. – ISBN 0–201–17888–5

[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, 2000. – ISBN 3–540–67203–6, S. 179–193

[TGS06] Tichy, Matthias ; Giese, Holger ; Seibel, Andreas: Story Diagrams in Real-Time Software. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proc. of the 4^{th} International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, Universität Paderborn, September 2006 (Technical Report), S. 15–22

[THHO08] Tichy, Matthias ; Henkler, Stefan ; Holtmann, Jörg ; Oberthür, Simon: Towards a Transformation Language for Component Structures in Mechatronic Systems. In: *Post Proc. of the 4th Workshop on Object-oriented Modeling of Embedded Real-Time Systems (OMER 4), Paderborn, Germany*, 2008. – eingereicht

[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

[Wag06] Wagner, Robert: Developing Model Transformations with Fujaba. In: Giese, Holger (Hrsg.) ; Westfechtel, Bernhard (Hrsg.): *Proc. of the 4^{th} International Fujaba Days 2006, Bayreuth, Germany* Bd. tr-ri-06-275, Universität Paderborn, September 2006 (Technical Report), S. 79–82

[WF99] Wermelinger, Michel ; Fiadeiro, José Luiz: Algebraic Software Architecture Reconfiguration. In: *SIGSOFT Softw. Eng. Notes* 24 (1999), Nr. 6, S. 393–409. `http://dx.doi.org/10.1145/318774.319256`. – DOI 10.1145/318774.319256. – ISSN 0163–5948

[WLF01] Wermelinger, Michel ; Lopes, Antónia ; Fiadeiro, José Luiz: 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 Press, 2001. – ISBN 1–58113–390–1, S. 21–32

[Zün01] Zündorf, Albert: *Rigorous Object Oriented Software Development*. Universität Paderborn, 2001