UNIVERSITÄT PADERBORN
*Die Universität der Informationsgesellschaft*

Institut für Informatik
Fachgebiet Softwaretechnik
Warburger Straße 100
33098 Paderborn

# Kompositionale Softwareverifikation mechatronischer Systeme

Schriftliche Arbeit
zur Erlangung des Grades
“Doktor der Naturwissenschaften”

vorgelegt von

**Daniela Schilling**
Rochusweg 10a
33102 Paderborn

Paderborn, Februar 2006

# Zusammenfassung

Die Ansprüche an die Leistungsfähigkeit und Sicherheit technischer Systeme, wie Autos oder Flugzeuge, steigt stetig. Um den wachsenden Anforderungen gerecht zu werden ist es notwendig, dass maschinenbauliche, elektrotechnische und Softwarekomponenten zusammenarbeiten. Ein System, dass aus solchen Komponenten zusammengesetzt ist, wird als mechatronisches System bezeichnet.

Ein solches System kann über Sensoren Informationen über seine Umwelt sammeln. Es hat aber auch die Möglichkeit, mit anderen mechatronischen Systemen zu kommunizieren oder zu kooperieren. Die Software, die für eine solche Interaktion erforderlich ist, ist sicherheitskritisch, d.h. ein Fehlverhalten der Software kann einen großen finanziellen Schaden verursachen und im schlimmsten Fall auch Menschenleben kosten. Da die Software gleichzeitig aber auch sehr komplex ist und zumeist einen unendlichen Zustandsraum hat, reicht Testen alleine nicht aus, um die Korrektheit der Software nachzuweisen. Automatische Ansätze zur formalen Verifikation wie Model Checking können nur Systeme mit einem endlichen Zustandsraum verifizieren. Theorembeweiser, die einen Korrektheitsnachweis auch für solche Systeme führen können, benötigen die Interaktion mit einem Benutzer, der mit formalen Methoden vertraut ist.

Deshalb wird in dieser Arbeit ein kompositionaler Ansatz vorgestellt, der die Software, die zur Interaktion zwischen mehreren mechatronischen Systemen notwendig ist, automatisch formal verifizieren kann. Der Ansatz baut auf den existierenden Ansatz der MECHATRONIC UML auf. Er nutzt dabei die Tatsache aus, dass ein Systemzustand in der MECHATRONIC UML, charakterisiert durch die mechatronischen Systeme und deren laufende Interaktion, als Graph dargestellt werden kann. Zustandsübergänge wie beispielsweise das Starten oder Beenden einer Interaktion können dann als Graphtransformationsregel beschrieben werden. Zudem können die hier betrachteten strukturellen Sicherheitseigenschaften lokal nachgewiesen werden. Für eine Menge von Graphtransformationsregeln und eine Menge von Sicherheitseigenschaften wird dann gezeigt, dass die Regeln niemals einen korrekten Graphen, also einen Zustand, der alle Sicherheitseigenschaften erfüllt, in einen inkorrekten Graphen überführen können. Die Erreichbarkeit der Graphen wird dabei jedoch nicht berücksichtigt. Somit wird bei der Verifikation nachgewiesen, dass die Sicherheitseigenschaften induktive Invarianten des Systems darstellen.

# Danke

Bevor ich meine Forschungsergebnisse der letzten Jahre präsentiere, möchte ich mich bei all den Menschen bedanken, die mir auf dem Weg zur Promotion geholfen, mich ermutigt oder mir freundschaftlich zur Seite gestanden haben.

Als erstes gilt mein Dank meinem Doktorvater Prof. Wilhelm Schäfer. Bei Wilhelm möchte ich mich dafür bedanken, dass er mich 2002 zur Promotion ermutigte, mir die Chance gegeben hat mich in seiner Arbeitsgruppe mit spannenden Themen der Informatik zu beschäftigen und mir viele wertvolle Tipps gegeben hat. An dieser Stelle möchte ich mich auch bei Prof. Gregor Engels und Prof. Jürgen Gausemeier dafür bedanken, dass sie meine Promotion betreut haben.

Bei meinem Kollegen Holger Giese möchte ich mich für viele fruchtbare Diskussionen, Anregungen und einen riesigen Berg „spannender Bettlektüre“bedanken.

Wer mich kennt, der weiß, dass meine Computer aus Prinzip nicht das tun, was sie tun sollen und Fehler produzieren, die nie zuvor jemand gesehen hat. Deswegen gilt mein Dank Jürgen Maniera, der nie die Geduld verloren hat und am Ende jedes noch so seltsame technische Rätsel lösen konnte.

Bürokratische Probleme sind während meiner Promotionszeit in Paderborn dank Jutta Haupt nie aufgetreten. Mein Dank gilt Jutta aber vor allem für die freundschaftliche Unterstützung und die Dienstag-Mittagessen.

„Wer solche Kollegen hat braucht keine Feinde mehr“(Sprichwort in der Arbeitsgruppe). Jungs, so schlimm wart ihr gar nicht. Im Gegenteil, ich möchte mich an dieser Stelle bei allen meinen Kollegen - Björn Axenath, Sven Burmester, Matthias Gehrke, Holger Giese, Stefan Henkler, Martin Hirsch, Florian Klein, Ekkart Kindler, Jörg Niere, Ahmet Mehic, Matthias Meyer, Vladimir Rubin, Matthias Tichy, Robert Wagner, Jörg Wadsack und Lothar Wendehals - für die gute Zusammenarbeit, gemeinsame Kinoabende und eine schöne Promotionszeit bedanken. Lothar möchte ich darüber hinaus für drei freundschaftliche Jahre im gemeinsamen Büro danken.

Matthias Gehrke, Martin Hirsch, Florian Klein, Lothar Wendehals, Matthias Meyer, Matthias Tichy, Robert Wagner und Reiko Heckel möchte ich dafür danken, dass sie ein oder mehrere Kapitel meiner Arbeit, zum Teil auch mehrfach, Korrektur gelesen und mir gute Tipps zur Verbesserung gegeben haben.

Bei Basil Becker und Victor Neumann möchte ich mich dafür bedanken, dass sie im Rahmen ihrer Studienarbeiten und während ihrer Tätigkeiten als studentische Hilfskräfte meine Ideen implementiert und in die Fujaba Real-Time Tool Suite integriert haben.

Bei Huberta Vahle und Riccarda Vollmers sowie ihren Familien und Teams möchte ich mich für die gute Freundschaft bedanken und dafür, dass ich auf dem Rücken ihrer Pferde nicht nur Glück und Erholung sondern auch so manch eine Problemlösung finden konnte. Meiner besten Freundin Silke Kaspari möchte ich für die vergnüglichen gemeinsamen Stunden und die vielen aufmunternden Emails bedanken. Außerdem gilt Silke mein Dank, da sie viele Stunden dafür aufgebracht hat, um die Rechtschreibung und Zeichensetzung in dieser Arbeit zu korrigieren.

Bei meinen Eltern möchte ich mich dafür bedanken, dass sie immer für mich da waren, mich unterstützt haben (auch dann, wenn ihnen meine Entscheidungen mal nicht gefallen haben) und sie sich mit mir und für mich über Erfolge gefreut haben.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

„The bug is always in the case you did not test“ Murphy’s law.

## 1.1 Motivation

Technische Systeme spielen eine immer größere Rolle im täglichen Leben. Beispiele für solche Systeme reichen von der Kaffeemaschine bis hin zu Autos und Flugzeugen oder den an der Universität Paderborn entwickelten autonom fahrenden Shuttles (siehe Abschnitt 1.2). Dabei steigen die Ansprüche an die Leistungsfähigkeit, aber auch an die Sicherheit dieser Systeme, stetig. Zu den Ansprüchen an die Leistungsfähigkeit gehört, dass die Systeme zunehmend autonom agieren können müssen. In Autos wird dies durch Fahrerassistenzsysteme realisiert und im Flugzeug durch Autopiloten.

Um den wachsenden Ansprüchen gerecht zu werden, ist es erforderlich, dass maschinenbauliche, elektrotechnische und Softwarekomponenten zusammenarbeiten. Ein System, das aus solchen Komponenten zusammengesetzt ist, wird als mechatronisches System bezeichnet.

Über Sensoren erhält ein mechatronisches System Informationen über seine Umwelt. Bei Autos oder Shuttles liefern die Sensoren beispielsweise Informationen über die Beschaffenheit der Strecke. Weitere Sensoren werden dazu eingesetzt, andere Verkehrsteilnehmer oder Hindernisse zu erkennen. Die durch die Sensoren erfassten Daten werden dann an Softwarekomponenten weitergeleitet. Die Softwarekomponenten bestimmen anhand dieser Daten, welche Aktionen durchzuführen sind und steuern die entsprechenden Aktoren, die die Aktionen wie Bremsen oder Beschleunigen durchführen.

Neben dem Informationsgewinn über Sensoren, können verschiedene mechatronische Systeme aber auch durch den Austausch von Daten über ein Netzwerk miteinander interagieren.

Die Verarbeitung dieser Daten durch die Software des mechatronischen Systems darf jedoch nicht beliebig lange dauern, sondern muss innerhalb fest vorgegebener Fristen (engl. deadline) erfolgen. Deshalb wird die Software auch als Echtzeitsystem bezeichnet.

Aufgrund ihrer Einbettung in ein maschinenbauliches Produkt und ihrer kontrollierenden Aufgabe sowie der harten Echtzeitrestriktionen, stellt die Software eines mechatronischen Systems ein sicherheitskritisches System dar. Eine Fehlfunktion der Software, z.B. Überschreiten

von Fristen oder fehlerhafte Berechnungen, resultiert im schlimmsten Fall in einem Unfall, der zu großen finanziellen Schäden führen oder Menschenleben kosten kann.

Die Komplexität der Software erfordert einen Entwicklungsansatz, der es erlaubt, ein abstraktes Modell der Software kompositional zu entwerfen, d.h. statt das System als Ganzes zu modellieren, muss es möglich sein das System in Teile zu untergliedern, jedes Teil einzeln zu modellieren und das Modell des Gesamtsystems dann aus diesen Teilen zusammen zufügen. Aufgrund des sicherheitskritischen Charakters muss es zudem möglich sein, für das Modell bestimmte Sicherheitseigenschaften nachzuweisen. Dabei stellt eine Sicherheitseigenschaft eine Eigenschaft dar, die in jedem erreichbaren Systemzustand erfüllt ist.

Der Nachweis, dass das Modell der Software eines mechatronischen Systems eine Menge von Sicherheitseigenschaften erfüllt, gestaltet sich in der hier betrachteten Domäne als sehr schwierig. Zwar stellen die Modelle eine Abstraktion der spezifizierten Systeme dar, jedoch sind auch sie im Allgemeinen zu komplex, um vollständig getestet zu werden. Vollautomatische Verifikationstechniken wie Model Checking machen Aussagen über alle erreichbaren Zustände eines Modells. Allerdings skalieren solche Verfahren nicht für Systeme mit beliebig großem bzw. unendlich großem Zustandsraum. Die Software eines mechatronischen Systems kann jedoch einen sehr großen oder unendlich großen Zustandsraum besitzen. Verfahren wie Theorembeweise können zwar auch zur Verifikation von solchen Systemen eingesetzt werden, sind jedoch nicht vollautomatisch und erfordern Eingaben von Benutzern mit Erfahrungen im Bereich von formalen Methoden.

## 1.2 Anwendungsbeispiel

Das in dieser Arbeit verwendete Anwendungsbeispiel stammt aus dem Sonderforschungsbereich (SFB) 614 - „Selbstoptimierende Systeme des Maschinenbaus“[1]. Innerhalb dieses SFBs wird exemplarisch die Software für Shuttles entwickelt, wobei jedes Shuttle ein mechatronisches System darstellt.

Die Shuttles sind Teil eines Transportsystem, dass die Vorteile von individuellen und öffentlichen Verkehrsmitteln verbindet. So sollen die Shuttles nicht nach fest vorgegebenen Fahrplänen fahren, sondern Personen und Güter auf Anforderung vom Start- zum Zielort bringen. Andererseits sollen die Shuttles aber auch einen geringeren Energiebedarf haben als zum Beispiel Autos, sodass sie umweltfreundlicher als der Individualverkehr sind.

Um dieses Ziel zu erreichen, fahren die Shuttles autonom und können selbständig und dezentral Entscheidungen treffen. Die Shuttles fahren auf Schienen. Ein Satellitensystem unterstützt die Shuttles bei der Positionsberechnung. Sowohl die Kommunikation zwischen zwei Shuttles, als auch die Kommunikation zwischen einem Shuttle und den Bahnhöfen, auf denen die Shuttles angefordert werden können, erfolgt über ein Funknetz. In Abbildung 2.14 ist ein solches System graphisch dargestellt.

Der Antrieb der Shuttles erfolgt, ähnlich wie bei der Magnetschwebebahn, über einen elektromagnetischen Linearmotor. Die dazu notwendigen Statorwellen werden von zwischen den Schienen eingelassenen Statoren erzeugt.

[1] www.sfb614.de

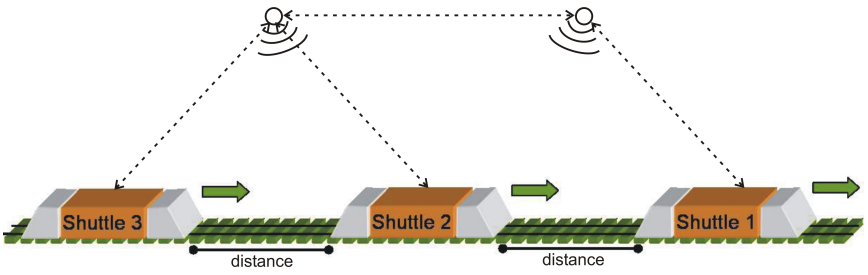



Abbildung 1.1: Zwei hintereinander herfahrende Shuttles

Um den Energieverbrauch der Shuttles zu minimieren, können sie Konvois bilden. Diese Konvois sind kontaktfrei, sodass das Bilden der Konvois während der Fahrt stattfinden kann. Zur Bildung von Konvois wird der Abstand zwischen zwei Shuttles minimiert. Je kleiner der Abstand ist, desto mehr Energie kann von den Shuttles eingespart werden, da der Luftwiderstand verringert wird. Die Abstandshaltung erfolgt zum einen über einen Abstandssensor, der den Abstand zum vorher fahrenden Shuttle misst und zum anderen dadurch, dass die Shuttles über das Funknetz Daten austauschen. Diese Kommunikation kann jedoch nicht beliebig erfolgen, sondern unterliegt einem fest vorgeschriebenen Protokoll (siehe Abschnitt 2.2.1).

Die Abstandssensoren haben allerdings nur eine begrenzte Reichweite und sind, besonders in Kurven, zur Abstandshaltung alleine nicht ausreichend. Damit ein Shuttle frühzeitig erfährt, welche anderen Shuttles sich in seiner Nähe befinden, ist das System in überlappende kritische Abschnitte, ControlledArea, unterteilt. Jeder dieser kritischen Abschnitte wird von einer Abschnittskontrolle, der BaseStation, überwacht. Bevor ein Shuttle in einen kritischen Abschnitt einfahren darf, muss es sich bei der entsprechenden Abschnittskontrolle anmelden. Nach der Anmeldung muss das Shuttle in regelmäßigen Abständen seine Position an die Abschnittskontrolle senden. Diese wiederum sendet die Daten an alle anderen bei ihr gemeldeten Shuttles. Fällt ein Shuttle aus und sendet seine Positionsdaten nicht, so warnt die Abschnittskontrolle die anderen Shuttles. Die Abschnittskontrolle hat also die Aufgabe, den Shuttles mitzuteilen, welche anderen Shuttles in der Nähe sind und vor Gefahren durch defekte Shuttles zu warnen. Die Abschnittskontrolle kann die einzelnen Shuttles jedoch nicht koordinieren; dies erfolgt ausschließlich durch die Shuttles.

## 1.3 Ziel und Lösungsansatz

Die Informationsverarbeitung eines mechatronischen Systems kann als Operator-Controller-Modul (OCM) aufgefasst werden [OHG04, HOG04, Ge05]. Ein solches Modul ist in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator unterteilt. Während der Controller direkten Zugriff auf die Aktoren des Systems hat, wird der reflektorische Operator dazu verwendet, um den Controller zu steuern und die Interaktion mit anderen OCMs zu koordinieren. Die Aufgabe des kognitiven Operator besteht darin Wissen über die Umwelt und das OCM selber zu sammeln und dazu zu nutzen, um das Verhalten des OCM besser an die gegebenen Anforderungen anzupassen.

Da die Software des reflektorischen Operators für die Steuerung des Controllers sowie die Interaktion des OCMs mit anderen OCMs verantwortlich ist, ist sie sicherheitskritisch. Deshalb besteht das Ziel dieser Arbeit in der Entwicklung eines kompositionalen Modellierungs- und Verifikationsansatzes für die Software des reflektorischen Operators und hier besonders für die Koordination zwischen zwei OCMs. Der Ansatz soll eine automatische und kompositionale Verifikation der Software, modelliert durch Graphtransformationssysteme, auch dann ermöglicht, wenn diese einen unendlichen Zustandsraum besitzen und ihr Initialzustand zum Zeitpunkt der Verifikation nicht bekannt ist.

Die Grundlage für die Softwareentwicklung stellt dabei die MECHATRONIC UML (siehe [BGT05, GTB+03, GBSO04, Bur05]) dar. Sie ist eine Anpassung der UML für die Modellierung und Verifikation der Software mechatronischer Systeme. Dabei bietet sie vor allem Möglichkeiten, um die Interaktion zwischen verschiedenen mechatronischen Systemen zu spezifizieren und zu verifizieren.

Jedes OCM wird in der MECHATRONIC UML als Komponente aufgefasst. Eine solche Komponente kann wiederum aus internen Komponenten bestehen. Da die Interaktion zwischen zwei Komponenten im Allgemeinen sicherheitskritisch ist und strengen Restriktionen unterliegt, muss sie fest vorgegebenen Protokollen folgen. In der MECHATRONIC UML werden diese Protokolle in Koordinationsmustern festgehalten. Damit zwei Komponenten miteinander interagieren können, müssen sie das entsprechende Protokoll ausführen, d.h. jede der an der Interaktion beteiligten Komponenten muss das Protokoll komponentenintern umsetzen. Die Modellierung mittels MECHATRONIC UML ist kompositional, d.h. jedes Koordinationsmuster kann unabhängig von anderen Koordinationsmustern entwickelt werden. Ebenso können die Komponenten unabhängig voneinander entwickelt werden.

Betrachtet man das Anwendungsbeispiel aus Abschnitt 1.2, so stellt jedes Shuttle und jede BaseStation ein OCM dar, das in der MECHATRONIC UML als Komponente vom Typ Shuttle bzw. vom Typ BaseStation modelliert wird. Fahren zwei Shuttles dicht hintereinander her, so müssen sie, um nicht zu kollidieren, ein Protokoll zur Abstandshaltung ausführen.

Die kompositionale Modellierung der MECHATRONIC UML lässt auch eine kompositionale Verifikation zu. Für jedes der Koordinationsmuster kann mittels Model Checking unabhängig von anderen Koordinationsmustern und Komponenten automatisch nachgewiesen werden, dass es eine Menge von Sicherheitseigenschaften erfüllt. Auch die Verifikation der Komponenten erfolgt durch Model Checking und kann unabhängig von anderen Komponenten und den Koordinationsmustern erfolgen. Bei der Verifikation einer Komponente wird überprüft, ob sich die Komponente so verhält, wie sie es durch die Umsetzung der Protokolle aus den Koordinationsmustern versprochen hat.

Das gesamte Modell eines Systems wird dann aus Instanzen der Komponenten und Koordinationsmuster zusammengesetzt. Wurden die Komponenten und Koordinationsmuster erfolgreich verifiziert, das gesamte Modell syntaktisch korrekt aus Instanzen der Komponenten und Koordinationsmuster zusammengesetzt und erfolgt die Interaktion der Komponenten ausschließlich über die Koordinationsmuster, so gilt, dass auch das gesamte Modell die vorgegebenen Sicherheitseigenschaften erfüllt. Eine zusätzliche Verifikation, um nachzuweisen, dass die Sicherheitseigenschaften auch vom gesamten Modell erfüllt werden, ist nicht notwendig.

Der von Burmester und anderen vorgestellte Ansatz der MECHATRONIC UML garantiert somit, dass die Interaktion mehrerer Komponenten sicher ist, wenn diese die notwendigen Koordinationsmuster miteinander ausführen. Da mechatronische Systeme sehr dynamisch sind, ist bei der Instanzierung einer Komponente jedoch nicht bekannt, mit welchen anderen Komponenteninstanzen sie zur Laufzeit interagieren muss.

Im Anwendungsbeispiel stellt jedes Shuttle eine Instanz der Komponente Shuttle dar. Die Shuttles bewegen sich auf den Schienen und begegnen dabei anderen Shuttles, mit denen sie interagieren müssen, z.B. um eine Abstandshaltung durchzuführen. Da sich die Shuttles im Schienennetz frei bewegen können, ist bei der Instanzierung einer Shuttle-Komponenten, also der Inbetriebnahme eines Shuttles, nicht bekannt, welchen anderen Shuttles dieses Shuttle jemals begegnen wird.

Außerdem gilt: Existiert zwischen zwei oder mehr Komponenteninstanzen eine Instanz eines Koordinationsmusters, so bedeutet dies, dass die Komponenten Daten austauschen. Diese Daten müssen von den Komponenteninstanzen verarbeitet und evtl. gespeichert werden. Jede der Komponenteninstanzen stellt ein mechatronisches System dar. In solchen Systemen stehen nur begrenzte Rechen- und Speicherkapazitäten zur Verfügung. Das bedeutet, selbst wenn alle anderen Komponenteninstanzen eines Systems bei der Instanzierung einer Komponente bekannt sind, so ist es nicht möglich, zwischen jedem Komponentenpaar alle möglichen Muster zu instanzieren, d.h. es ist nicht möglich, dass eine Komponenteninstanz mit allen anderen Komponenteninstanzen kommuniziert.

Eine Beschreibung, wann eine bestimmte Musterinstanz benötigt bzw. wann sie nicht mehr benötigt wird und wie eine Instanz erzeugt oder gelöscht werden muss, ist mittels der MECHATRONIC UML nicht möglich.

Aus diesem Grund wird ein Ansatz benötigt, der es erlaubt, die Instanzierung und das Löschen von Koordinationsmustern zu modellieren. Um garantieren zu können, dass eine Instanz eines Koordinationsmusters immer vorhanden ist wenn sie benötigt wird, muss der Ansatz zudem eine formale Verifikation ermöglichen. Da mechatronische Systeme in der Regel zu komplex sind, um als Ganzes modelliert zu werden, muss der Ansatz zusätzlich eine kompositionale Modellierung und Verifikation erlauben.

Ein solcher Ansatz wird in dieser Arbeit vorgestellt und in den existierenden Modellierungs- und Verifikationsprozess der MECHATRONIC UML integriert.

Der in dieser Arbeit vorgestellte Ansatz basiert auf der folgenden Idee: Ein Systemzustand wird durch seine Koordinationsmuster- und Komponenteninstanzen sowie durch die existierenden Verbindungen zwischen den Instanzen charakterisiert. Eine solche Instanzsituation kann auch als Graph aufgefasst werden. In einem derartigen Graphen entspricht jede Koordinationsmuster- und Komponenteninstanz einem Knoten und jede Verbindung zwischen Instanzen einer Kante. Ein Übergang von einem Systemzustand in einen anderen beschreibt dann das Erzeugen oder das Löschen von Instanzen oder deren Verbindungen. Fasst man einen Systemzustand als Graphen auf, so werden bei einem Zustandsübergang Knoten und Kanten erzeugt bzw. gelöscht. Wann ein Konten oder eine Kante erzeugt oder gelöscht werden muss und wie ein neu erzeugter Knoten oder eine neu erzeugte Kante in einen Graphen eingefügt wird, wird mittels Graphtransformationsregeln definiert. Damit entspricht ein Zustandsübergang einer Graphtransformation.

In dieser Arbeit werden zur Beschreibung von Graphtransformationen Story Patterns [FNTZ98, Zün01] verwendet. Story Patterns wurden speziell für die objektorientierte Modellierung entwickelt und stellen eine Erweiterung der UML-Aktivitätendiagramme dar. Der Vorteil bei einer Verwendung einer UML-nahen Modellierungssprache liegt darin, dass der Einarbeitungsaufwand in diese Sprache gering ist und somit eine größere Akzeptanz des Ansatzes zu erwarten ist. Da Story Patterns eine spezielle Form von Graphtransformationsregeln darstellen, besitzen sie eine formal definierte Semantik, sodass sie mittels formaler Verifikationstechniken überprüft werden können.

Fasst man einen Systemzustand als Graphen auf, so bietet dies einen weiteren Vorteil: Strukturelle Sicherheitseigenschaften, die vom System erfüllt werden müssen, können ebenfalls als Graphen aufgefasst werden. In dieser Arbeit werden Graphmuster eingeführt, die eine spezielle Form von Graphen darstellen, um solche strukturellen Eigenschaften zu spezifizieren. Von besonderem Interesse werden dabei die so genannten verbotenen Graphmuster sein, da diese dazu genutzt werden können, um kritische Situationen und Unfälle zu modellieren, die nie eintreten dürfen. Auch solche Graphmuster können als Story Patterns beschrieben werden.

Im Anwendungsbeispiel kann ein solches verbotenes Graphmuster dazu verwendet werden, um zu beschreiben, dass zwei Shuttles sehr dicht hintereinander herfahren, ohne jedoch ein Koordinationsmuster zur Abstandshaltung miteinander auszuführen. Dies entspricht einer kritischen Situation, da das vordere Shuttle bei seinen Aktionen keine Rücksicht auf das hintere Shuttle nimmt und ihm beispielsweise nicht signalisiert, dass es bremsen möchte. In diesem Fall droht eine Kollision der beiden Shuttles.

Für eine Menge von verbotenen Graphmustern soll durch formale Verifikation gezeigt werden, dass die Anwendung der Graphtransformationsregeln niemals einen Graphen erzeugen können, der ein solches verbotenes Graphmuster enthält. Das bedeutet, es soll geprüft werden, dass niemals eine der durch die verbotenen Graphmuster beschriebenen kritischen Situationen oder Unfälle eintritt.

Zur Verifikation von Graphtransformationssystemen existieren zwar einige Ansätze (siehe Kapitel 6), jedoch sind diese zumeist auf Systeme beschränkt, in denen nur endliche viele Zustände erreichbar sind. Die Regeln zur Instanzierung und zum Löschen von Koordinationsmuster- und Komponenteninstanzen sowie deren Verbindungen, können aber unter Umständen unendlich viele Zustände erzeugen. Darüber hinaus benötigen die meisten dieser Ansätze den Anfangszustand des Systems. Da die Verifikation zum frühst möglichen Zeitpunkt im Entwicklungsprozess erfolgen soll, der Anfangszustand im Allgemeinen jedoch erst recht spät bekannt ist, würde die Verwendung dieser Ansätze den Entwicklungsprozess verzögern. Deshalb sind die existierenden Ansätze im Allgemeinen in der Domäne der mechatronischen Systeme nicht einsetzbar.

Aus diesem Grund wird in dieser Arbeit ein Verfahren eingeführt, dass die Verifikation von Graphtransformationssystemen auch dann erlaubt, wenn das System unendlich viele erreichbare Zustände besitzt. Statt wie die existierenden Verfahren für jeden Zustand zu prüfen, ob dieser korrekt ist, d.h. zu prüfen ob er keines der verbotenen Graphmuster enthält, prüft der hier vorgestellte Ansatz, ob ein inkorrekter Zustand das Resultat einer Regelanwendung auf einen korrekten Graphen sein kann. Wie in der Arbeit von Heckel und Wagner (siehe [HW95] und Abschnitt 6.1) wird dazu die rechte Seite der betrachteten Graphtransformationsregel mit einem verbotenen

Graphmuster verknüpft, daraus resultiert das so genannte Ergebnisgraphmuster. Auf dieses Ergebnisgraphmuster wird dann die entsprechende Regel rückwärts angewendet, woraus das Startgraphmuster resultiert. Enthält dieses Startgraphmuster kein verbotenes Graphmuster, so wurde ein Beispiel gefunden, das zeigt, dass die betrachtete Regel einen korrekten Zustand in einen inkorrekten überführen kann. Dabei wird allerdings nicht betrachtet, ob der inkorrekte Zustand ausgehend vom Startzustand des Systems erreichbar ist. Damit prüft der Ansatz für jedes verbotene Graphmuster, ob sein „Nicht-Auftreten“ eine induktive Invariante des Graphtransformationssystems ist. Im Gegensatz zum Ansatz von Heckel und Wagner wird jedoch die betrachtete Regel nicht modifiziert. Zudem wird gezeigt, wie die Regelmenge in voneinander unabhängige Teilmengen unterteilt werden kann. Diese Teilmengen können dann unabhängig voneinander verifiziert werden.

Wird ein Beispiel gefunden, in dem eine Regel einen korrekten Graphen in einen inkorrekten überführt, so werden diese beiden Graphen zusammen mit der angewendeten Regel als Gegenbeispiel zurückgeliefert. Die Generierung eines solchen Gegenbeispiels sowie seiner Darstellung in der gleichen Notation, mit der auch das Modell entwickelt wurde, ermöglicht eine einfache Fehlerdiagnose.

In dieser Arbeit wird das Verfahren zunächst formalisiert. Dies geschieht jedoch nicht auf Ebene der Story Patterns, sondern allgemeiner für Graphtransformationssysteme, wie sie zumeist in der Literatur verwendet werden. Die dabei gewonnenen Erkenntnisse werden auf Story Pattern übertragen und prototypisch in der Fujaba Real-Time Tool Suite[2] umgesetzt. Das Verfahren soll dann anhand eines kleinen Ausschnitts des im vorangegangenen Abschnitts vorgestellten Anwendungsbeispiels evaluiert werden.

Zusammen mit dem bereits existierenden Ansatz der MECHATRONIC UML und dessen Umsetzung in der Fujaba Real-Time Tool Suite bietet der Ansatz dann die Möglichkeit, die Software eines mechatronischen Systems kompositional zu modellieren und automatisch zu verifizieren.

Damit ist es dann möglich, die Leistungsfähigkeit technischer Systeme zu erhöhen, indem Softwarekomponenten in Systeme, bestehend aus maschinenbaulichen und elektrotechnischen Komponenten, integriert werden. Trotz der daraus resultierenden Komplexität der Software kann der vorgestellte Ansatz ihre Korrektheit im Bezug auf bestimmte Sicherheitseigenschaften gewährleisten.

## 1.4 Aufbau der Arbeit

Die Arbeit ist in die folgenden Kapitel unterteilt:

Kapitel 2 stellt die Architektur des Operator-Controller-Moduls sowie die existierenden Modellierungs- und Verifikationskonzepte der MECHATRONIC UML dar. Neben dem Modellierungs- und Verifikationsansatz wird in diesem Kapitel auch beschrieben, wie die relevanten Komponenten und Koordinationsmuster eines mechatronischen Systems sowie die Regeln zu deren Instanzierung und Löschen identifiziert werden können. Am Ende des Kapitels wird der Prozess vorgestellt, der die Identifikation der relevanten

[2] www.fujaba.de

Komponenten und Koordinationsmuster, deren Modellierung und Verifikation beinhaltet. Dieser Prozess beschreibt auch, wie die Verifikation der Regeln zum Instanzieren und Löschen von Komponenten und Koordinationsmustern sowie deren Verbindungen in den Gesamtprozess integriert wird.

Kapitel 3 beschreibt einen Ansatz zum Nachweis von induktiven Invarianten in Graphtransformationssystemen. Dieser Ansatz wird bei der Entwicklung sicherheitskritischer Software für mechatronische Systeme dazu verwendet, um die Regeln zum Instanzieren und Löschen von Komponenten und Koordinationsmustern zu verifizieren.

Kapitel 4 beschreibt informal die Abbildung der in Kapitel 2 eingeführten Story Patterns auf die in Kapitel 3 beschriebenen Graphtransformationsregeln und verbotenen Graphmuster. Diese Abbildung ermöglicht die Verwendung des Verifikationsansatzes aus Kapitel 3 für die Verifikation von Story Patterns.

Kapitel 5 beschreibt die prototypische Umsetzung des gesamten Modellierungs- und Verifikationsansatzes. Am Ende des Kapitels erfolgt eine Evaluierung anhand des Anwendungsbeispiels.

Kapitel 6 fasst verwandte Arbeiten zur Verifikation von Graphtransformationssystemen zusammen.

Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und liefert einen Überblick über mögliche Erweiterungen des Ansatzes.

# Kapitel 2

# Grundlagen

In diesem Kapitel werden die existierenden Ansätze vorgestellt, die für die vorliegende Arbeit als Grundlage dienen. Darüber hinaus wird gezeigt, wie der in dieser Arbeit vorgestellte Ansatz in die existierenden Ansätze integriert werden kann.

In [OHG04, HOG04, Ge05] wurde die Architektur des Operator-Controller-Moduls vorgestellt. Diese bietet die Möglichkeit die Informationsverarbeitung eines mechatronischen Systems zu Strukturieren.

Die Software eines mechatronischen Systems ist im Allgemeinen sehr komplex und sicherheitskritisch. Dies gilt insbesondere für den Teil der Software, der für die Steuerung der Hardware sowie die Interaktion mit anderen mechatronischen Systemen verantworlich ist. Um diese Komplexität handhaben zu können und eine effiziente Analyse zu ermöglichen, wird eine modellbasierte Softwareentwicklung verwendet.

Bei der modellbasierten Softwareentwicklung wird sowohl die Architektur des Softwaresystems als auch sein Verhalten in einem Modell festgehalten. Dabei wird von implementierungsspezifischen Details abstrahiert. Das resultierende Modell soll mittels formaler Methoden verifiziert werden können, damit Modellierungsfehler möglichst früh im Entwicklungsprozess aufgedeckt werden.

Die Unified Modeling Language (UML, siehe [UML05]) stellt die Standardmodellierungssprache für die modellbasierte Softwareentwicklung dar. Sie enthält verschiedene Notationen für die Modellierung von Softwarearchitekturen und Verhalten.

Um ein mechatronisches System spezifizieren zu können, muss auch echtzeitfähige Software spezifiziert werden können. Dies wird durch die UML, wie sie in [UML05] definiert ist, jedoch nicht bzw. nur unzureichend unterstützt. Deshalb ist es notwendig, einige Notationen der UML auszuwählen und diese zu verfeinern oder zu erweitern. Eine solche Anpassung der UML an die Aufgaben bei der Modellierung von mechatronischen Systemen erfolgte im Rahmen des SFB 614. Die dabei entstandene Anpassung wird als MECHATRONIC UML [BGT05, BTG04, GBSO04, GTB+03, Bur05] bezeichnet und soll in diesem Kapitel eingeführt werden.

In diesem Kapitel wird zunächst die Modellierung der Architektur vorgestellt 2.1. Nachdem die Architektur der Software modelliert wurde, erfolgt in Abschnitt 2.2 die Modellierung des Koordinationsverhaltens. Der vorgestellte Modellierungsansatz ist kompositional, sodass das System nicht als ganzes sondern in Form kleinerer Teilsysteme (Komponenten und Koordinati-

onsmuster) modelliert wird. Die Komposition des Gesamtsystems aus diesen Teilsystemen ist Inhalt von Abschnitt 2.3. Da die hier betrachtete Software sicherheitskritisch, auf der anderen Seite aber auch zu komplex ist, um vollständig getestet werden zu können, wird in Abschnitt 2.4 ein Ansatz zur formalen, kompositionalen Verifikation vorgestellt. Der gesamte Modellierungs- und Verifikationsprozess wird in Abschnitt 2.5 erläutert.

## 2.1 Architektur

Die Strukturierung der Informationsverarbeitung erfolgt mittels der Architektur des Operator-Controller-Moduls (siehe Abschnitt 2.1.1). Diese teilt ein mechatronisches System in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator ein.

Von besonderem Interesse ist dabei die Software des reflektorischen Operators, da diese für die Steuerung des Controllers, der die Aktoren des Systems steuert, sowie die Interaktion mit anderen mechatronischen Systemen verantwortlich ist.

Zur Modellierung der Softwarearchitektur des reflektorischen Operators werden Komponenten- und Klassendiagramme verwendet.

### 2.1.1 Die Operator-Controller-Modul Architektur

Die Informationsverarbeitung, d.h. die Aufnahme von Daten zum Beispiel über Sensoren sowie deren Verarbeitung, eines mechatronischen Systems ist komplex. In [OHG04, HOG04, Ge05] stellen Oberschelp, Hestermeyer und Giese deshalb die Architektur des Operator-Controller-Modul (OCM) zur strukturierten und modularen Entwicklung der Informationsverarbeitung eines mechatronischen Systems vor.

In diesem Ansatz wird ein mechatronisches System als Operator-Controller-Modul aufgefasst, das in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator unterteilt werden kann. Die Struktur eines solchen Operator-Controller-Moduls ist in Abbildung 2.1 gegeben.

Die unterste Ebene des Moduls bildet der Controller. Dieser kann die mechanischen Teile des Gesamtsystems durch Zugriff auf die Aktoren direkt beeinflussen. Seine Aufgabe besteht darin Signale aufzunehmen, zu verarbeiten und weiter zu geben, deshalb wird er auch als motorischer Kreis bezeichnet. Dabei unterliegt die Verarbeitung der Signale harten Echtzeitbedingungen, das bedeutet, dass die Verarbeitung innerhalb einer fest vorgegebenen Zeit erfolgen muss. Ist die Verarbeitung innerhalb dieser Zeit nicht abgeschlossen, so kann das zu einer sicherheitskritischen Situation führen. Ein Controller kann aus mehreren Reglern bestehen, zwischen denen umgeschaltet werden kann. Die Informationsverarbeitung des Controllers ist quasi-kontinuierlich, d.h. die Sensoren nehmen kontinuierlich Daten auf und leiten diese zur Verarbeitung weiter.

Über der Controller-Ebene liegt die Ebene des reflektorischen Operators. Zu den Aufgaben des reflektorischen Operators gehört die Überwachung des Controllers. Der reflektorische Operator kann keinen direkten Einfluss auf die Aktorik des Systems nehmen. Er kann jedoch die Konfiguration des Controllers verändern und dadurch die Umschaltung der Regler bewirken. Auch die sicherheitskritische Koordination mit anderen Operator-Controller-Modulen erfolgt über den

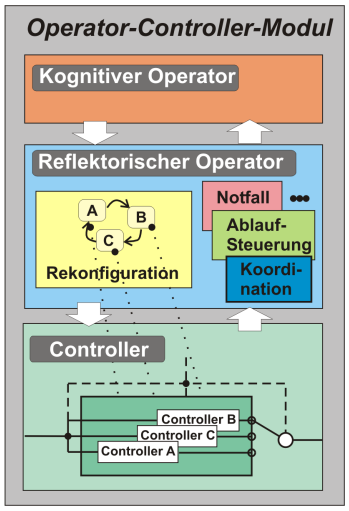



Abbildung 2.1: Die Architektur des Operator-Controller-Moduls. Quelle: [Ge05]

reflektorischen Operator. Die Überwachungs-, Steuerungs- und Koordinationsfunktionen des reflektorischen Operators sind diskret und ereignisbasiert.

Auf der obersten Ebene der OCM-Architektur befindet sich der kognitive Operator. Dieser Operator verwendet Lernverfahren, modellbasierte Optimierungsverfahren und wissensbasierte Systeme, um Wissen über sich und seine Umwelt zu sammeln und zur Verbesserung des eigenen Verhaltens auszunutzen.

Der Name Operator-Controller-Modul beschreibt die Zweiteilung eines mechatronischen Systems in den Teil, der auf die Aktorik des Systems direkt zugreifen kann, sowie den Teil der nur indirekten Zugriff auf die Mechanik besitzt. Diese beiden Teile werden als Operator und als Controller bezeichnet.

In dieser Arbeit werden Konzepte zur Modellierung und Verifikation des reflektorischen Operators vorgestellt, wobei die sicherheitskritische Koordination im Fokus der Arbeit liegt.

### 2.1.2 Komponentendiagramme

Die Software eines Operator-Controller-Moduls wird modular beschrieben. Dazu werden Teile des Systems als Komponenten aufgefasst, die miteinander interagieren können. Ein Operator-Controller-Modul stellt selber eine Komponente dar. Eine schematische Darstellung einer Komponente, die das Operator-Controller-Modul aus Abbildung 2.1 repräsentiert ist in Abbildung 2.2 gegeben. Im Folgenden wird die detaillierte Darstellung von Controller, reflektorischem Operator und kognitiven Operator weggelassen und nur die Koordination des OCM mit anderen OCMs modelliert.

Nach Szyperski [Szy02] ist „eine Softwarekomponente eine Kompositionseinheit mit vertraglich festgelegten Schnittstellen und expliziten Kontextabhängigkeiten. Eine Softwarekomponente kann unabhängig verteilt und durch Dritte mit anderen Komponenten verbunden werden.“

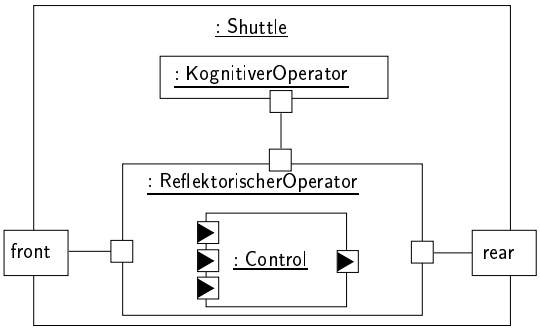



Abbildung 2.2: OCM als Komponente. Quelle: [Ge05]

Das bedeutet, eine Komponente ist eine Einheit, deren Implementierung nach außen nicht sichtbar ist. Für eine Komponente wird festgelegt, welche Nachrichten sie für andere Komponenten bereitstellt bzw. welche Nachrichten sie von anderen Komponenten erwartet.

In einem UML-Komponentendiagramm [UML05] werden die Komponententypen festgelegt. Das nach außen sichtbare Verhalten einer Komponente, d.h. die Interaktion mit anderen Komponenten, wird durch benötigte Schnittstellen (engl. required interfaces) und bereitgestellte Schnittstellen (engl. provided interfaces) beschrieben. Dabei entsprechen die benötigten Schnittstellen den „Kontextabhängigkeiten“von Szyperski und die bereitgestellten Schnittstellen den „vertraglich festgelegten Schnittstellen“. Eine benötigte Schnittstelle beschreibt, welche Nachrichten die Komponente von anderen Komponenten erwartet. Die bereitgestellte Schnittstelle beschreibt, welche Nachrichten die Komponente anderen Komponenten zur Verfügung stellt. Mehrere Schnittstellen einer Komponente können in einem Port zusammengefasst werden.

Im Beispiel stellt jedes Shuttle eine Komponente dar. Um sich gegenseitig koordinieren zu können, müssen die Shuttles sowohl Nachrichten versenden als auch empfangen können. Das Senden und Empfangen von Nachrichten erfolgt über benötigte und bereitgestellte Schnittstellen. Abbildung 2.3 zeigt die Komponente, die den Typ Shuttle definiert. Die Komponente besitzt zwei benötigte Schnittstellen, dargestellt durch einen Halbkreis, und zwei bereitgestellte Schnittstellen, dargestellt durch einen Kreis. Jeweils eine benötigte und eine bereitgestellte Schnittstelle werden in einem Port zusammengefasst. Die Ports werden durch Quadrate an den Rändern der Komponente repräsentiert.

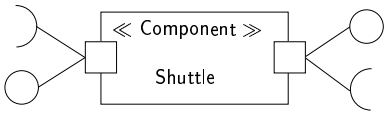



Abbildung 2.3: Definition des Komponententyps Shuttle

Nachdem die Komponententypen festgelegt wurden, können konkrete Komponenteninstanzen betrachtet werden. Auch Komponenteninstanzen werden in einem UML-Komponentendiagramm dargestellt. Wird die bereitgestellte Schnittstelle einer Komponente mit der benötigten Schnittstelle einer zweiten Komponente verbunden, so bietet die MECHATRONIC UML die Möglichkeit, dies abkürzend durch einen Pfeil darzustellen. Dieser Pfeil verläuft

von der Komponente, die die Schnittstelle bereitstellt, zu der Komponente mit der benötigten Schnittstelle. Tauschen zwei Komponenten in beide Richtungen Nachrichten aus, d.h. beide haben jeweils eine benötigte und eine bereitgestellte Schnittstelle, so wird abkürzend ein Doppelpfeil verwendet. Eine solche Verbindung von zwei Komponenten wird als Konnektor bezeichnet. Abbildung 2.4 zeigt zwei Instanzen der Shuttle-Komponente, die miteinander über einen Konnektor kommunizieren.

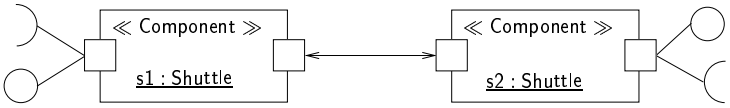



Abbildung 2.4: Zwei Instanzen der Shuttle-Komponente

Ein Komponententyp beschreibt, welche Schnittstellen eine Komponente bereitstellt bzw. benötigt. Das Verhalten dieser Schnittstellen wird intern durch die in Abschnitt 2.2.1 eingeführten Real-Time Statecharts spezifiziert. Das interne Verhalten der Komponente wird durch Instanzen interner Komponenten oder durch Real-Time Statecharts beschrieben.

Weitere interne Strukturen einer Komponente wie beispielsweise Datenstrukturen, können durch UML-Klassendiagramme definiert werden.

### 2.1.3 Klassendiagramme

UML-Klassendiagramme [UML05] werden dazu verwendet, die interne Architektur einer Komponente zu spezifizieren, ihre Datenstrukturen festzulegen und um das interne Verhalten der Komponenten realisieren zu können.

Abbildung 2.5 zeigt das Klassendiagramm, das die interne Architektur der Shuttle-Komponente festlegt. Es definiert die Ontologie des Shuttles, d.h. in ihm wird die Sicht des Shuttles auf seine Umgebung definiert.

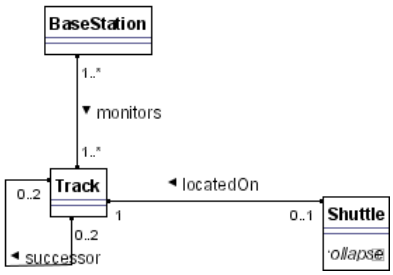



Abbildung 2.5: Klassendiagramm, das die Ontologie der Shuttle-Komponente beschreibt

Im Beispiel besteht das Klassendiagramm, das die Ontologie beschreibt, aus den Klassen Shuttle, Track und BaseStation sowie den Assoziationen locatedOn, successor und monitors. Die locatedOn-Assoziation wird dazu benutzt, um zu beschreiben, auf welchem Track sich ein

Shuttle befindet. Die Kardinalitäten an der Assoziation besagen, dass sich jedes Shuttle auf genau einem Track befindet. Die Tracks sind so kurz, dass sich höchstens ein Shuttle darauf befinden kann. Dies wird durch die Kardinalität $0 \ldots 1$ ausgedrückt. Das Schienennetz wird durch die successor-Assoziationen modelliert. Für Weichen gibt es keine eigene Klasse. Eine Weiche wird dadurch modelliert, dass ein Track zwei Vorgänger oder zwei Nachfolger hat. Die Aufgabe der BaseStation besteht darin, die Tracks und die auf ihnen fahrenden Shuttles zu überwachen. Jede BaseStation kann beliebig viele Tracks überwachen und überwacht mindestens einen, dargestellt durch die Kardinalität $1 \cdots *$ an der monitors-Assoziation. Umgekehrt kann ein Track von beliebig vielen BaseStations überwacht werden und wird von mindestens einer überwacht, d.h. die von den BaseStations überwachten Bereiche können sich überlappen.

In diesem Abschnitt wurden Komponentendiagramme eingeführt, um die Architektur eines Softwaresystems zu modellieren. Für jede der Komponenten kann eine Menge von bereitgestellten und benötigten Schnittstellen angegeben werden, die festlegen, welches Verhalten eine Komponente nach außen zeigt bzw. welches Verhalten sie von anderen Komponenten erwartet. Intern wird eine Komponente mittels eines Klassendiagramms strukturiert, das auch die Datenstrukturen der Komponente festlegt. Im folgenden Abschnitt soll betrachtet werden, wie das Verhalten der Schnittstellen sowie das interne Verhalten einer Komponente spezifiziert werden kann.

## 2.2 Verhalten

Nachdem die Architektur der Software durch Komponenten- und Klassendiagramme spezifiziert wurde, kann das interne Verhalten der Komponenten definiert werden.

Ein Teil dieses Verhaltens ist die Kommunikation zwischen zwei Komponenten. Die Schnittstellen der Komponenten stellen dar, welche Nachrichten eine Komponente bereitstellt bzw. welche Nachrichten sie von anderen Komponenten benötigt. Existiert zwischen zwei Komponenteninstanzen ein Konnektor, d.h. die benötigten und bereitgestellten Schnittstellen der Komponenten wurden miteinander verbunden, so muss garantiert werden können, dass entweder Nachrichten, die von der einen Komponente versandt werden, von der anderen Komponente auch empfangen werden oder beim Verlust oder zu späten Eintreffen einer Nachricht kein Unfall und keine kritische Situation eintreten kann. Eine Kommunikation, die diese Eigenschaft erfüllt wird als sicher bezeichnet. Um eine solche sicher Kommunikation zu spezifizieren werden Kommunikationsprotokolle verwendet.

### 2.2.1 Kommunikation

Bei der Modellierung einer sicheren Kommunikation zwischen Komponenten muss zum einen berücksichtigt werden, dass das Senden und Empfangen von Nachrichten zwischen Komponenten Zeit benötigt. Zudem muss berücksichtigt werden, dass Nachrichten verloren gehen können, zum Beispiel durch ein unzuverlässiges Übertragungsmedium. Die Echtzeit-Koordinationsmuster der MECHATRONIC UML unterstützen die Modellierung einer sicheren Kommunikation.

Ein Echtzeit-Koordinationsmuster [GTB+03] (oder kurz Koordinationsmuster) besteht aus

- einer Menge von Rollen, die die Kommunikationspartner darstellen,
- einer Menge von Rolleninvarianten,
- Konnektoren, die die Rollen verbinden und
- einem Musterconstraint.

Eine Rolle beschreibt die externe Kommunikation eines Kommunikationspartners. Sie gibt an, welche Nachrichten versendet werden, welche Nachrichten erwartet werden, in welcher Reihenfolge die Nachrichten versandt oder empfangen werden, wie viel Zeit mindestens vergehen muss oder maximal vergehen darf, wenn eine Nachricht verschickt oder empfangen werden soll. Zur Spezifikation der Kommunikation werden Zustandsautomaten verwendet. Da die in der UML eingeführten Protokoll-Zustandsautomaten [UML05] kaum die Modellierung von Echtzeitverhalten unterstützen, werden für die Spezifikation der Rollen Real-Time Statecharts verwendet (siehe unten).

Die Rollen und ihr durch Real-Time Statecharts spezifiziertes Verhalten sind eine abstrakte Beschreibung eines Kommunikationspartners bzw. seines extern sichtbaren Kommunikationsverhaltens. Soll eine Komponente in einer Kommunikation eine bestimmte Rolle übernehmen, so muss sie die entsprechende Rolle verfeinern. Das bedeutet, die Komponente muss das Kommunikationsverhalten der Rolle realisieren. Nach [Gie03, GTB+03] darf dabei jedoch kein zusätzliches extern sichtbares Verhalten hinzugefügt werden. Zudem muss Verhalten, das durch die Rolle garantiert wird, auch von der Komponente gezeigt werden. Außerdem muss die Komponente die Rolleninvarianten der von ihr realisierten Rollen erfüllen.

Eine Rolleninvariante ist einer bestimmten Rolle zugeordnet. Sie beschreibt eine Eigenschaft, die von dem entsprechenden Kommunikationspartner erfüllt werden muss. Durch formale Verifikation kann gezeigt werden, dass eine Komponente, die eine Rolle bei der Kommunikation übernommen hat, die entsprechende Rolleninvariante einhält (siehe Abschnitt 2.4.2). Beschrieben werden Rolleninvarianten zum Beispiel durch TCTL-Formeln [ACD90].

Die Konnektoren beschreiben die Verbindung zwischen den Rollen. Sie werden wie die Rollen durch ein Real-Time Statechart beschrieben. Im Konnektor werden die Qualitätseigenschaften (engl. quality of service) der Verbindung zwischen den Rollen spezifiziert. Die Beschreibung enthält zum Beispiel, wie lange das Verschicken einer Nachricht benötigt oder wie zuverlässig die Verbindung ist.

Das Musterconstraint beschreibt eine Eigenschaft, die von allen Kommunikationspartnern und den Konnektoren eingehalten werden muss. Wie die Rolleninvarianten werden auch die Musterconstraints als TCTL-Formeln spezifiziert.

Zur Modellierung einer Komponente werden die Rollen ausgewählt, deren Kommunikationsverhalten die Komponente umsetzen soll. Eine Komponente kann dadurch an der Kommunikation verschiedener Koordinationsmuster teilnehmen. Die Modellierung des Komponentenverhaltens ist Inhalt von Abschnitt 2.2.2.

Die Trennung von Kommunikations- und Komponentenverhalten ermöglicht einerseits eine Wiederverwendung der Koordinationsmuster, andererseits kann auf diese Weise eine formale

Verifikation durchgeführt werden. In Abschnitt 2.3.1 wird gezeigt, wie ein komplexes Softwaresystem aus Komponenten- und Koordinationsmusterinstanzen zusammengesetzt werden kann.

Im Shuttle-System müssen sich beispielsweise zwei Shuttle-Instanzen gegenseitig koordinieren, um hintereinander fahren zu können, ohne zu kollidieren [BGH+05]. Für diese Koordination wird das DistanceCoordination-Koordinationsmuster aus Abbildung 2.6 verwendet. Dieses Koordinationsmuster besteht aus den beiden Rollen frontRole und rearRole und einem bidirektionalem Konnektor. Die Rollen werden durch die beiden Quadrate im Diagramm dargestellt, das Koordinationsmuster als gestricheltes Oval.

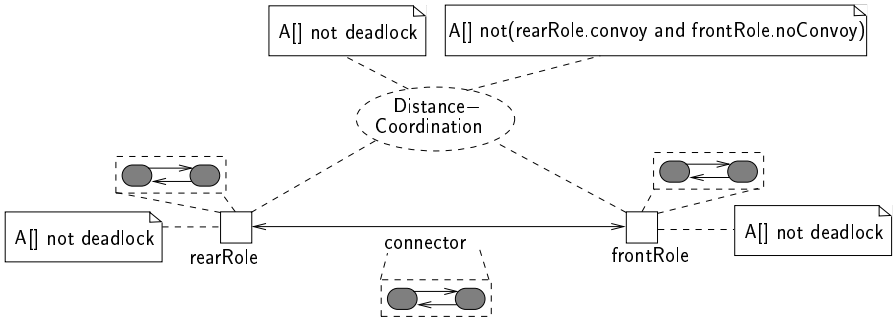



Abbildung 2.6: Das DistanceCoordination-Koordinationsmuster

An das Koordinationsmuster sind zwei Musterconstraints angefügt. Das Constraint A[] not deadlock besagt, dass die Kommunikation der Rollen über den Konnektor immer verklemmungsfrei (engl. deadlock free) sein muss. Das Constraint A[] not (rearRole.convoy and frontRole.noConvoy) verlangt, dass das vordere Shuttle nicht im Zustand noConvoy fährt, wenn das hintere Shuttle im Zustand convoy fährt. Andernfalls würde das vordere Shuttle bei seinen Aktionen nicht berücksichtigen, dass das hintere Shuttle mit sehr geringem Abstand hinter ihm herfährt.

Ebenso ist an jede der beiden Rollen eine Rolleninvariante angefügt. Im Beispiel wird von den Rollen nur verlangt, dass sie verklemmungsfrei arbeiten.

Abbildung 2.7 zeigt die Umsetzung des Koordinationsmusters durch zwei Instanzen der Shuttle-Komponente. Wobei der Port front des Shuttles s1 die frontRole realisiert und der Port rear von Shuttle s2 die rearRole.

Damit zwei Shuttle-Komponenten kollisionsfrei hintereinander herfahren können, muss dass Verhalten der Rollen so spezifiziert werden, dass die Shuttles Konvois bilden und wieder auflösen können. Zudem müssen die Protokolle, die das Rollenverhalten beschreiben, so modelliert werden, dass sie eventuelle Nachrichtenverluste abfangen können, ohne dass die beteiligten Shuttles kollidieren. Bisher wurde jedoch nur die Struktur des Koordinationsmusters betrachtet, das Verhalten wurde noch nicht spezifiziert. Die Spezifikation des Rollenverhaltens, aber auch des Verhaltens des Konnektors, erfolgt über die im folgenden Abschnitt vorgestellten Real-Time Statecharts.

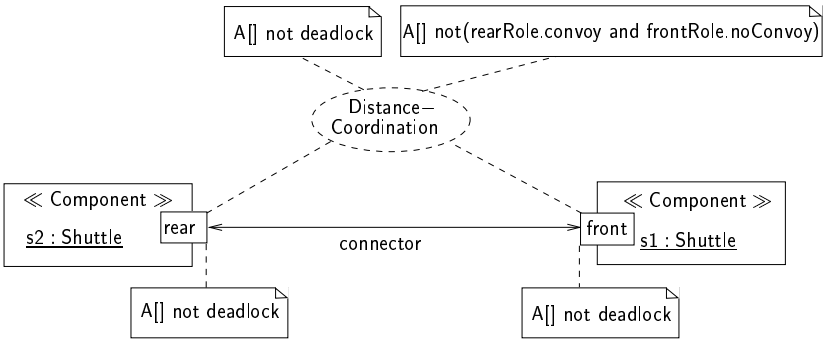



Abbildung 2.7: Instanz des DistanceCoordination-Musters

### Real-Time Statecharts

Zur Spezifikation von Protokollen und Verhalten werden in der UML-Zustandsautomaten [UML05] verwendet. Protokolle werden durch Protokoll-Zustandsautomaten (engl. protocol statemachines) modelliert und Verhalten durch Verhaltens-Zustandsautomaten (engl. behavioral statemachines).

Bei der Spezifikation der Software eines mechatronischen Systems spielt die Modellierung von Zeit eine besondere Rolle. UML-Zustandsautomaten bieten zur Modellierung von Zeitverhalten jedoch nur das after-Konstrukt an. Dieses Konstrukt ist für die Modellierung der hier betrachteten mechatronischen Systeme nicht ausreichend. Zudem besitzen die UML-Zustandsautomaten eine Nullzeitsemantik, d.h. die Transitionen der Zustandsautomaten können schalten, ohne dass dabei Zeit vergeht. In mechatronischen Systemen, in denen das Schalten einer Transition mit der Ausführung von Methoden auch auf physikalischer Ebene verbunden sein kann wie beispielsweise dem Versenden oder empfangen einer Nachricht, ist eine solche Semantik nicht realisierbar. Deshalb werden in der MECHATRONIC UML Real-Time Statecharts [GB03, BG03, BGS05b, Bur05] eingesetzt, um die Kommunikationsprotokolle und das Komponentenverhalten zu spezifizieren.

Real-Time Statecharts stellen eine Kombination aus UML-Zustandsautomaten und Timed Automata [AD90, AD94] dar. Sie sollen an dieser Stelle nur in einem informalen Überblick vorgestellt werden. Eine ausführlichere Beschreibung ist in [Bur05, GB03] enthalten.

Real-Time Statecharts bieten verschiedene Möglichkeiten um Zeit zu spezifizieren. Eine Transition kann mit einem Intervall beschriftet werden, das angibt, wie lange das Schalten der Transition mindestens bzw. höchstens dauert. Zeitinvarianten geben an, wann ein Zustand spätestens wieder verlassen werden muss. Dazu können mehrere Uhren spezifiziert werden. Diese Uhren werden beim Schalten bestimmter Transitionen zurückgesetzt.

Eine Transition ist aktiviert, wenn ihre Zeitbedingung (engl. time guard) erfüllt ist, das auslösende Ereignis vorliegt und die Bedingung der Transition, bestehend aus einem boolschen Ausdruck über Variablen und Methoden, wahr ist. Ist die aktivierte Transition zwingend (engl. urgent), so muss sie sofort nach ihrer Aktivierung schalten. Nicht-zwingende Transitionen können das Schalten verzögern, so lange ihre Zeitbedingung erfüllt ist. Nicht-zwingende Transitionen werden durch gestrichelte Pfeile dargestellt.

Während die Ausführungssemantik der Real-Time Statecharts der von Timed Automata entspricht, enthalten sie auch einige Konstrukte von UML-Zustandsautomaten. Dazu gehören zum Beispiel sowohl flache als auch tiefe Historie (engl. shallow und deep history) und parallele und hierarchische Zustände.

Für einen Zustand können außerdem entry()-, do()- und exit()- Methoden definiert werden. Enthält ein Zustand eine do()-Methode, so wird diese Methode periodisch ausgeführt solange der Zustand aktiviert ist. Wird der Zustand aktiviert und sofort wieder verlassen, so muss seine do()-Methode jedoch mindestens einmal ausgeführt werden. Für jede der Methoden muss eine maximal zulässige Ausführungszeit (engl. worst case execution time) spezifiziert werden. Zusätzlich zu den entry()-, do()- und exit()-Methoden in den Zuständen können die Transitionen mit Methodenaufrufen belegt sein. In diesem Fall wird, sobald die Transition schaltet, zunächst die entsprechende Methode ausgeführt. Wie in den Zustandsautomaten der UML werden diese Methoden jedoch nur in den Real-Time Statecharts verwendet, die das Verhalten einer Komponente beschreiben. Statecharts mit Methodenaufrufen werden in Abschnitt 2.2.2 noch einmal betrachtet. In UML-Protokoll-Zustandsautomaten und Real-Time Statecharts, die Protokolle modellieren, werden keine Methoden aufgerufen.

Für Real-Time Statecharts kann Code generiert werden, der auf echtzeitfähigen Plattformen ausgeführt werden kann. Eine solche Codegenerierung ist möglich, da für die Real-Time Statecharts durch Abbildung auf Timed Automata eine formale Semantik definiert wurde und für jede Methode, die innerhalb eines Real-Time Statecharts aufgerufen wird, eine maximal zulässige Ausführungszeit angegeben werden muss.

In Abbildung 2.8 ist ein Beispiel für ein Real-Time Statechart gegeben, das das Protokoll der Rolle rearRole beschreibt. Standardmäßig ist die rearRole im Zustand inactive, d.h. es findet keine Kommunikation statt. Nichtdeterministisch wird aus diesem Zustand in den Zustand active gewechselt. Die entsprechende Transition ist nicht-zwingend (sie ist gestrichelt), deshalb kann ihr Schalten beliebig verzögert werden. raisedEvents an der Transition gibt an, dass die folgenden Nachrichten erzeugt und versendet werden. Mit empfängerName.nachricht wird spezifiziert, wer der Empfänger ist und welche Nachricht ihm gesandt werden soll. Durch frontRole.startCommunication wird der frontRole signalisiert, dass eine Kommunikation erforderlich ist. Die Beschriftung der Transition mit $\{t0\}$ sagt, dass beim Schalten der Transition neben dem Versenden der Nachricht auch die Uhr t0 auf den Wert 0 zurückgesetzt werden soll. Das Intervall $[0; 100]$ legt fest, dass das Schalten der Transition maximal 100 msek dauern darf.

Nachdem der Schaltvorgang beendet ist, ist rearRole im Zustand active. In diesem Zustand wartet die rearRole auf eine Bestätigung, dass die zuvor versandte Nachricht von der frontRole empfangen wurde. Mit der Invariante $t0 < 500$ wird spezifiziert, dass die rearRole maximal 500 msek im Zustand active auf eine Empfangsbestätigung warten darf. Trifft die Empfangsbestätigung in dieser Zeit nicht ein, muss die rearRole davon ausgehen, dass die Nachricht möglicherweise verloren gegangen ist und schaltet wieder in den Zustand inactive.

Trifft die Empfangsbestätigung innerhalb der erlaubten Zeit ein, so wechselt die rearRole in den Zustand noConvoy. Ausgehend von diesem Zustand kann die rearRole der frontRole nichtdeterministisch den Vorschlag machen, einen Konvoi zu bilden. Im Zustand wait wartet die rearRole dann darauf, dass die frontRole der Konvoibildung zustimmt (Nachricht startConvoy) oder die Konvoibildung ablehnt (Nachricht convoyProposalRejected).

Soll ein bestehender Konvoi aufgelöst werden, so schickt die rearRole die Nachricht breakConvoyProposal. Die frontRole kann daraufhin den Konvoi durch das Versenden der Nachricht breakConvoy auflösen oder die Anfrage der rearRole mit breakConvoyProposalRejected zurückweisen.

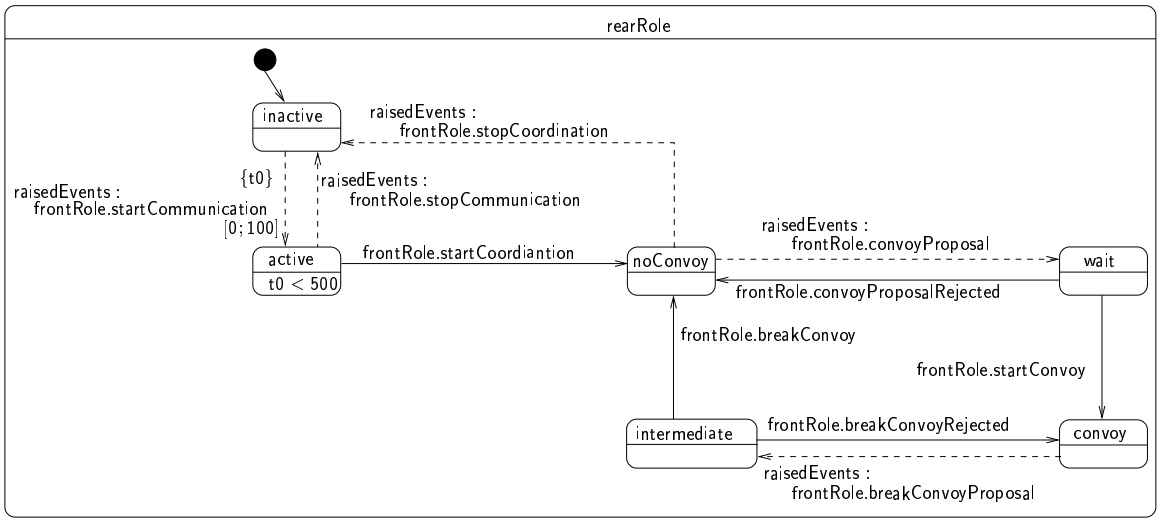



Abbildung 2.8: Real-Time Statechart, das das Verhalten der Rolle rearRole beschreibt

Wie bei der rearRole wird auch das Verhalten der frontRole und des Konnektors mittels Real-Time Statecharts spezifiziert.

### 2.2.2 Komponentenverhalten

Das Verhalten einer Komponente wird, wie das Verhalten einer Rolle, durch ein Real-Time Statechart beschrieben. Besitzt eine Komponente Schnittstellen und Ports, die durch eine Rolle beschrieben werden, so muss das Komponentenverhalten das Verhalten der entsprechenden Rollen realisieren.

Die Umsetzung des Rollenverhaltens darf jedoch nicht beliebig erfolgen [Gie03, GTB+03]. Zum einen darf die Verfeinerung kein Verhalten hinzufügen, das nach außen sichtbar ist und das nicht in der realisierten Rolle enthalten ist. Das bedeutet, für jede Transition im Real-Time Statechart des Ports, die ein Ereignis an einen anderen Port sendet oder von einem anderen Port empfängt, existiert eine Transition im Real-Time Statechart der Rolle, die das gleiche Ereignis sendet oder empfängt. Zum anderen darf bei der Umsetzung kein Verhalten entfernt werden. Das heißt, Verhalten, das die Rolle enthält, muss auch in der Komponente enthalten sein. Oder anders ausgedrückt, für jede Transition im Real-Time Statechart der Rolle muss es eine Transition im Real-Time Statechart des Ports geben, dass das gleiche Ereignis empfängt oder versendet. Die Verfeinerung stellt somit eine Simulation (siehe [CGP02] Kapitel 11) dar für die zusätzlich gelten muss, dass jedes Verhalten der Rolle auch im Port realisiert ist. Die Verfeinerung stellt im Allgemeinen jedoch keine Bi-Simulation dar. Daraus resultiert, dass bei der Realisierung einer Rolle durch eine Komponente hauptsächlich Nichtdeterminismus entfernt wird.

Realisiert eine Komponente das Verhalten mehrerer Rollen oder besitzt sie zusätzliches internes Verhalten, das nach außen nicht sichtbar ist, so muss eine Synchronisation durch ein internes Real-Time Statechart erfolgen.

Im Gegensatz zu den Real-Time Statecharts der Rollen, beschreibt das komponenteninterne Real-Time Statechart reaktives Verhalten, d.h. in den Zuständen dürfen entry()-, do()- und exit()-Methoden verwendet werden und die Transitionen können mit Methodenaufrufen belegt sein.

Schaltet eine Transition, die mit einem Methodenaufruf belegt ist, so wird der Aufruf als Seiteneffekt (engl. side effect) ausgeführt. Ein solcher Seiteneffekt kann das Versenden von Nachrichten an eine andere Komponente sein. Andererseits kann als Seiteneffekt eine Methode ausgeführt werden, die auf der internen Struktur der Komponente, beispielsweise auf ihrer Ontologie, arbeitet.

Ein Beispiel für eine solche Komponente ist die Shuttle-Komponente. Jedes Shuttle muss sowohl als vorderes als auch als hinteres Shuttle agieren können. Deshalb besitzt die Komponente Shuttle sowohl einen Port front als auch einen Port rear. Die beiden Ports verfeinern das Verhalten, das in den Rollen frontRole und rearRole des DistanceCoordination-Musters spezifiziert ist.

Abbildung 2.9 zeigt die drei parallelen Zustände der Shuttle-Komponente. Der oberste Zustand stellt die Umsetzung der frontRole dar, der untere die Umsetzung der rearRole. Der Zustand in der Mitte dient zur Synchronisation der beiden Ports, sein Verhalten ist nach außen nicht sichtbar.

Die Aufgabe des mittleren Zustands besteht darin, die beiden Ports der Komponente zu synchronisieren. Dazu wird eine synchrone Kommunikation verwendet, dargestellt durch Nachrichten, die entweder mit einem „?“oder einem „!“beginnen. Zunächst ist das Shuttle im Zustand noCoordination, d.h. es koordiniert sich mit keinem anderen Shuttle. Periodisch wird geprüft, ob die Methode createDC anwendbar ist. Dies ist genau dann der Fall, wenn sich ein anderes Shuttle in der Nähe befindet, mit dem eine Koordination erforderlich ist. Sobald die Methode anwendbar ist, schaltet die entsprechende Transition und stößt eine Koordination zwischen den beiden Shuttles an, falls sich der mittlere Zustand mit dem Zustand des rear-Ports synchronisieren kann. Die Synchronisation erfolgt über das Versenden von !startDC im mittleren Zustand und das Empfangen von ?startDC im rear-Port. Sind die beiden Shuttles soweit voneinander entfernt, dass eine Koordination nicht mehr notwendig ist, so ist die Methode deleteDC anwendbar und das Shuttle schaltet wieder in den Zustand noCoordination. Die beiden Methoden werden in Abschnitt 2.3.2 noch einmal genauer betrachtet.

Die in Abbildung 2.9 dargestellte Spezifikation des Shuttle-Verhaltens lässt nur die Koordination von jeweils zwei Shuttles zu, d.h. ein Shuttle kann niemals zeitgleich ein vorderes und ein hinteres Shuttle sein. Somit können die Konvois auch nur aus zwei Shuttles bestehen. Burmester verwendet das Konzept des DistanceCoordination-Musters in [Bur05], um auch Konvois beliebiger Länge zu bilden.

Das Kommunikationsverhalten der Komponente, das in ihren beiden Ports umgesetzt ist, entspricht dem Kommunikationsverhalten der beiden Rollen rearRole und frontRole. Allerdings wurde der Nichtdeterminismus der beiden Rollen durch die Kommunikation mit der internen Synchronisation aufgelöst. Beispielsweise kann die rearRole nichtdeterministisch aus dem Zustand inactive in den Zustand active wechseln. Im rear-Port ist ein Wechsel nur dann möglich,

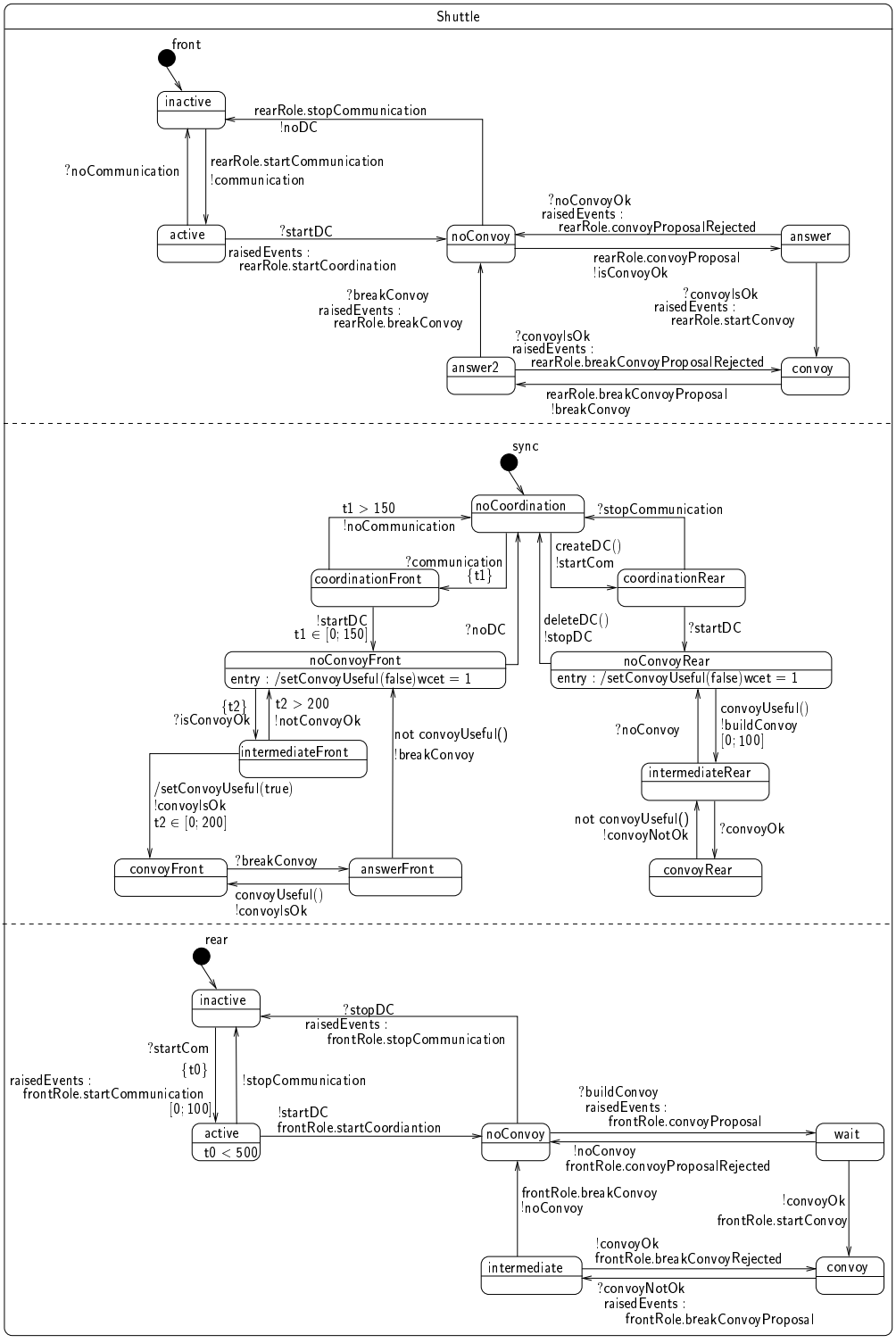



Abbildung 2.9: Verhalten der Shuttle-Komponente

wenn die Synchronisation die Nachricht startDC sendet. Da in der rearRole die Transition nicht zwingend war, konnte das Schalten der Transition beliebig verzögert werden. Erhält der rear-Port jedoch die Nachricht startDC so muss sie sofort schalten, da die Transition nun zwingend ist. In den Rollen ist genau spezifiziert, wann eine bestimmte Nachricht gesendet werden muss und wie lange dieses Senden maximal dauern darf. Die interne Synchronisation der Komponente muss diese Zeiten respektieren.

### Hybride Komponenten

Mit den bisher vorgestellten Verhaltensbeschreibungen ist es möglich, die Kommunikation zwischen mehreren Komponenten zu modellieren und beispielsweise die Ontologie zu verändern. Es ist jedoch noch nicht möglich, das bei der Kommunikation vereinbarte Verhalten auf physikalischer Ebene umzusetzen. Deshalb werden in [BGO04, BGT05, Bur05] hybride Komponenten eingeführt. Hybride Komponenten werden benötigt, um die Controller (unterste Ebene der OCM-Architektur, siehe Abbildung 2.1) zu steuern, die die entsprechenden Signale dann an die Aktoren des Systems weiterleiten und damit auch das bei der Koordination vereinbarte Verhalten auf physikalischer Ebene umzusetzen.

Ein Beispiel für eine hybride Komponente ist die Komponente Control in Abbildung 2.2. Die Ports der Komponente sind kontinuierlich, dies wird durch das ausgefüllte Dreieck innerhalb des Ports dargestellt.

Hybride Komponenten werden an dieser Stelle nur der Vollständigkeit halber genannt. Im weiteren Verlauf der Arbeit werden jedoch nur diskrete Komponenten, wie zuvor eingeführt, betrachtet. Die in Kapitel 3 und ff. vorgestellte Verifikationstechnik kann jedoch auch für hybride Komponenten verwendet werden.

### Story Diagramme

Das reaktive Verhalten der Komponenten resultiert aus den entry()-, do()- und exit()-Methoden der Zustände sowie den Methoden, die als Seiteneffekte beim Schalten einer Transition ausgeführt werden.

Solche Methoden werden in der UML durch UML-Aktivitätendiagramme beschrieben. In [Zün01, FNTZ98] wurden die UML-Aktivitätendiagramme zu Story Diagrammen erweitert. Sie stellen den Kontrollfluss einer Methode graphisch dar. Über Parameter können den Story Diagrammen Attributwerte und Objektreferenzen übergeben werden.

Die Basisstruktur von Story Diagrammen stellen die so genannten Aktivitäten dar. Aktivitäten können durch Transitionen verbunden werden, die die Ausführungsreihenfolge festlegen. Die Ausführung beginnt bei einer eindeutigen initialen Aktivität. Verlassen mehrere Transitionen eine Aktivität, so müssen diese mit sich gegenseitig ausschließenden Bedingungen belegt sein. Die Ausführung terminiert, wenn die Stopaktivität erreicht wurde. Die Aktivitäten können entweder durch Codefragmente oder durch Story Patterns beschrieben werden. Story Patterns stellen eine modellbasierte Notation dar, die auf Graphtransformationen basiert. Aus einem Story Pattern kann Code generiert werden. In dieser Arbeit werden Story Diagramme verwendet, die aus genau einer Aktivität bestehen und keine Attribute besitzen.

Ein UML-Objektdiagramm beschreibt eine bestimmt Instanzsituation, d.h. welche Objekte zu einem bestimmten Zeitpunkt existieren und durch welche Links sie verbunden sind. Mittels Story Patterns kann beschrieben werden, wann eine solche Instanzsituation verändert werden soll. In ihm wird definiert, welche Objekte und Links (im Folgenden kurz Elemente genannt) bei einer Änderung neu erzeugt oder gelöscht werden sollen. Zudem kann mittels Story Patterns der Wert von Attributen geändert werden. Elemente, die durch die Anwendung eines Story Patterns gelöscht werden, sind mit ≪ destroy ≫ gekennzeichnet und in rot dargestellt. Elemente, die durch die Anwendung eines Story Patterns erzeugt werden, sind durch ≪ create ≫ gekennzeichnet und in grün dargestellt. Elemente, die nicht gekennzeichnet sind, bleiben bei der Anwendung des Story Patterns unverändert erhalten.

Abbildung 2.10 zeigt ein einfaches Beispiel für ein Story Pattern. Dieses Story Pattern, moveSimple, beschreibt die Bewegung des Shuttles von einem Track (rt1) auf den nächsten (rt2). Für diese Vorwärtsbewegung des Shuttles wird die locatedOn-Kante zwischen dem Shuttle-Objekt und dem Track-Objekt rt1 entfernt (diese Kante ist mit ≪ destroy ≫ beschriftet) und eine neue locatedOn-Kante zwischen dem Shuttle und rt2 erzeugt (diese Kante ist mit ≪ create ≫ beschriftet).

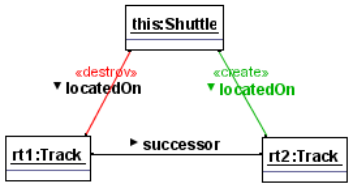



Abbildung 2.10: Story Pattern moveSimple, das die Bewegung eines Shuttles von einem Track auf den nächsten beschreibt

Ein Story Pattern kann auf eine Instanzsituation (beschrieben durch ein UML-Objektdiagramm) angewendet werden, wenn alle ungekennzeichneten und mit ≪ destroy ≫ gekennzeichneten Elemente des Story Patterns auf Elemente des Objektdiagramms abgebildet werden können. Bei der Anwendung werden dann alle mit ≪ destroy ≫ gekennzeichneten Elemente gelöscht und die mit ≪ create ≫ beschrifteten Elemente erzeugt und zum Objektdiagramm hinzugefügt.

Abbildung 2.11 zeigt ein Beispiel für ein Objektdiagramm. Das Diagramm stellt die Ontologie eines Shuttles dar. Sie besteht aus fünf Track-Objekten, die im Kreis angeordnet sind sowie zwei Shuttle-Objekten, this und as2, wobei sich this auf Track at1 befindet und as2 auf Track at4. Das mit this beschriftete Shuttle entspricht dem Shuttle, dessen Ontologie das Objektdiagramm darstellt. Das Story Pattern aus Abbildung 2.10 kann auf dieses Objektdiagramm angewendet werden. Dazu wird das Objekt this : Shuttle aus dem Story Pattern mit dem Objekt this : Shuttle aus dem Objektdiagramm gleichgesetzt. Die Track-Objekte rt1 und rt2 werden auf die Track-Objekte at1 und at2 abgebildet und die locatedOn-Kante zwischen this und rt1 wird auf die locatedOn-Kante zwischen this und at1 abgebildet. Die Abbildung des Story Patterns auf das Objektdiagramm ist in Abbildung 2.11 grün hinterlegt. Abbildung 2.12 zeigt das Objektdiagramm nachdem das Story Pattern angewendet wurde.

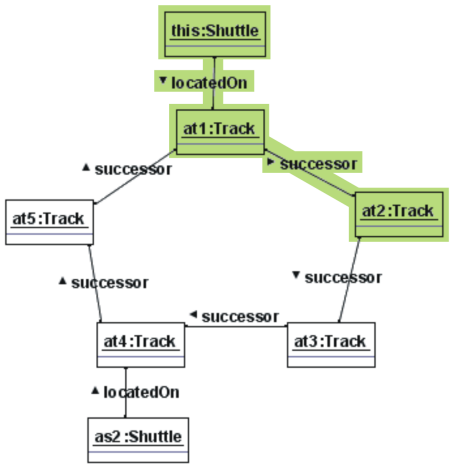



Abbildung 2.11: Beispiel für ein Objektdiagramm. Die Abbildung des Story Patterns aus Abbildung 2.10 auf dieses Objektdiagramm ist grün hinterlegt

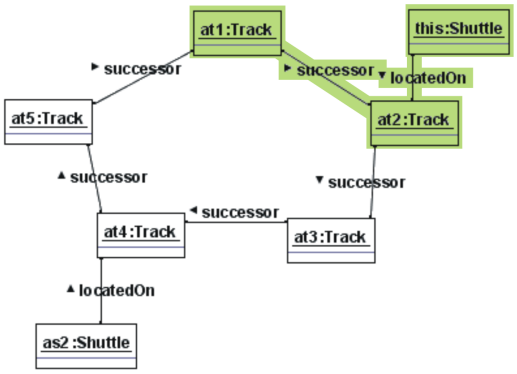



Abbildung 2.12: Objektdiagramm aus Abbildung 2.11 nachdem das Story Pattern moveSimple darauf angewendet wurde

Mit den bisher vorgestellten Konstrukten lässt sich definieren, wann ein Story Pattern angewendet werden kann. Mittels negativer Anwendungsbedingungen kann die Anwendung eines Story Patterns eingeschränkt werden.

Im Shuttle-Beispiel sind die Tracks so kurz, dass nur jeweils ein Shuttle darauf passt. Befinden sich zwei Shuttles auf einem Track, so kollidieren sie. Fahren zwei Shuttles auf aufeinander folgenden Tracks, so bedeutet dies, dass die Shuttles so dicht hintereinander herfahren, dass das hintere Shuttle nicht genügend Zeit zum Reagieren hat, wenn das vordere Shuttle plötzlich bremst. Um einen genügend großen Sicherheitsabstand zu garantieren, der eine solche Kollision vermeidet, wird deshalb verlangt, dass sich zwischen zwei Shuttles ein freier Track befindet. Es muss deshalb eine Bedingung aufgestellt werden, die verhindert, dass die moveSimple-Regel erneut auf das Shuttle this aus Abbildung 2.12 angewendet wird. Andernfalls würde das Shuttle this auf den Track at3 fahren. Da sich Shuttle as2 auf dem Track at4 befindet, wäre zwischen den beiden Shuttles jedoch kein Sicherheitsabstand mehr.

Um eine negative Anwendungsbedingungen zu beschreiben, bieten Story Patterns negative Objekte und Links. Ein Story Pattern mit negativen Elementen darf nur dann auf ein Objektdiagramm angewendet werden, wenn alle positiven Elemente auf Elemente des Objektdiagramms abgebildet werden können, die negativen jedoch nicht. Sobald eines der negativen Elemente auf das Objektdiagramm abgebildet werden kann, ist die Anwendung des entsprechenden Story Patterns nicht mehr erlaubt.

In Abbildung 2.13 ist ein Beispiel für ein Story Pattern mit negativer Anwendungsbedingung gegeben. Bei diesem Story Pattern handelt es sich um eine Erweiterung des Story Patterns aus Abbildung 2.10. Die Erweiterung fordert, dass das Shuttle nur dann auf den nächsten Track weitergesetzt werden darf, wenn sich weder auf dem nächsten noch auf dem übernächsten Track ein anderes Shuttle befindet (dargestellt durch die durchgestrichenen Shuttle-Objekte). Dieses Story Pattern kann nun nicht mehr auf das Shuttle this aus dem Objektdiagramm in Abbildung 2.12 angewendet werden, da das negative Shuttle-Objekt rs2 aus dem Story Pattern auf das Shuttle-Objekt as2 aus dem Objektdiagramm abgebildet werden kann.

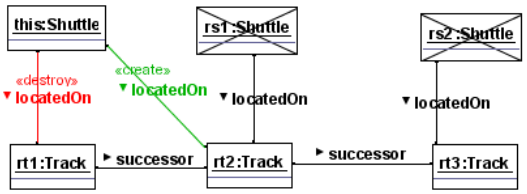



Abbildung 2.13: Story Pattern moveSimpleNAC, das das Story Pattern moveSimple um eine negative Anwendungsbedingung erweitert

Im Folgenden werden Story Patterns aber nicht nur dazu eingesetzt, um Methoden zu spezifizieren. Aufgrund ihrer Ausdrucksstärke und ihrer leicht verständlichen Darstellungsweise eignen sie sich auch dazu, kritische (Instanz-)Situationen und Unfälle zu beschreiben. Dies wird in Abschnitt 3.2.2 dargestellt.

## 2.3 Modellkomposition

Bisher wurde die Modellierung von Komponenten sowie deren Interaktion über Koordinationsmuster betrachtet. Sowohl Komponenten als auch Koordinationsmuster stellen einzelne Teile des gesamten Modells dar. In diesem Abschnitt soll nun betrachtet werden, wie aus den Teilmodellen das gesamte Modell zusammengesetzt wird.

### 2.3.1 Kommunikation und Komponenten

In Abschnitt 2.1.2 und 2.2.2 wurde die Modellierung von Komponenten und ihrem Verhalten betrachtet. Koordinationsmuster wurden in 2.2.1 eingeführt, um das Interaktionsverhalten der Komponenten zu beschreiben. Nun soll betrachtet werden, wie aus diesen Modellteilen ein gesamtes Modell gebildet wird.

Das Gesamtmodell besteht aus einer Menge von Komponenten- und Musterinstanzen [GTB+03, GST+03]. Die Interaktion von Komponenten erfolgt dabei immer über Instanzen der Koordinationsmuster. Das bedeutet, dass die Ports einer Komponente Implementierungen von Musterrollen darstellen. Soll in einem Komponentendiagramm dargestellt werden, dass der Port einer Komponente eine Musterrolle verfeinert, so wird in das Diagramm eine Komponenteninstanz sowie eine Instanz des Koordinationsmusters eingefügt. Das Quadrat am Rand der Komponente stellt dann sowohl ihren Port als auch die verfeinerte Rolle des Koordinationsmusters dar.

Eine solche Modellierung, bei der das Gesamtmodell aus einzelnen Teilen zusammengefügt wird, wird als kompositionale Modellierung bezeichnet. Die hier vorgestellte kompositionale Modellierung erlaubt zum einen die verschiedenen Komponenten- und Koordinationsmustertypen wieder zu verwenden. Zum anderen erlaubt sie aber auch eine formale Verifikation des Modells (siehe Abschnitt 2.4).

In Abbildung 2.14 ist exemplarisch ein Modell, bestehend aus zwei Shuttle-Komponenten und einer Komponente vom Typ BaseStation, dargestellt. Die Kommunikation der beiden Shuttles untereinander erfolgt über eine Instanz des DistanceCoordination-Koordinationsmusters. Die Kommunikation der beiden Shuttles mit der BaseStation erfolgt über die beiden Instanzen des Publication-Koordinationsmusters.

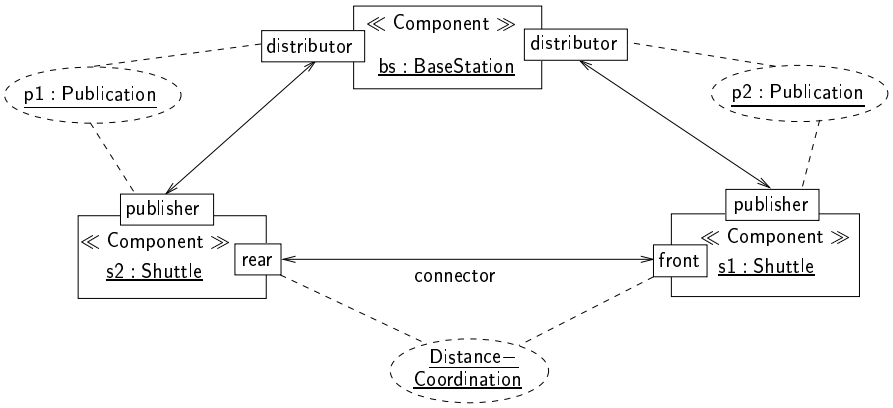



Abbildung 2.14: Modell bestehend aus je drei Komponenten und Koordinationsmustern

Wird das Modell auf diese Weise aus Komponenten und Koordinationsmustern zusammengefügt, so muss für jedes Paar von Komponenten, das sich zur Laufzeit eventuell koordinieren muss, das entsprechende Koordinationsmuster instanziert sein. Da die Ausführung eines Kommunikationsprotokolls bedeutet, dass Nachrichten ausgetauscht werden, kann eine Komponente nur mit einer begrenzten Anzahl anderer Komponenten kommunizieren. Hinzu kommt, dass mechatronische Systeme sehr dynamisch sind und bei der Instanzierung einer Komponente noch nicht bekannt ist, mit welchen anderen Komponenten sie sich zur Laufzeit koordinieren muss. Deshalb wird im nächsten Abschnitt das Instanzieren und das Löschen von Koordinationsmustern betrachtet.

### 2.3.2 Kulturen und Communities

Damit die Interaktion der Komponenten eines mechatronischen Systems sicher ist, muss sie durch Instanzen von Koordinationsmustern realisiert werden. Mechatronische Systeme sind jedoch dynamisch, d.h. bei der Instanzierung einer Komponente ist nicht bekannt, mit welchen andere Komponenten sie jemals zur Laufzeit interagieren muss. Außerdem stehen in einem mechatronischen System zur Ausführung der Software nur begrenzte Ressourcen (CPU, Speicher, etc. ) zur Verfügung. Dadurch kann eine Komponente nur mit einer begrenzten Anzahl anderer Komponenten kommunizieren. Für jedes Paar von Komponenten alle möglichen Koordinationsmuster zu instanzieren, ist somit nicht möglich. Stattdessen dürfen nur solche Koordinationsmuster instanziert werden, die zur Laufzeit wirklich benötigt werden.

Im Folgenden wird ein Ansatz vorgestellt, der es ermöglicht, festzustellen, welche Musterinstanzen zu einem bestimmten Zeitpunkt benötigt werden und die Instanzierung vornimmt bzw. nicht mehr benötigte Instanzen löscht.

Mechatronische Systeme sind im Allgemeinen sehr komplex, sodass ihre Modellierung aufwendig ist. Zudem führt die Komplexität dazu, dass eine automatische formale Verifikation erschwert oder verhindert wird. Deshalb werden an dieser Stelle soziale Metaphern eingeführt, die eine zusätzlich Strukturierung des Systems ermöglichen. Durch diese zusätzliche Strukturierung wird zum einen die Modellierung des Systems vereinfacht, zum anderen wird dadurch aber auch die Verifikation der Instanzierung und des Löschens von Musterinstanzen ermöglicht (siehe Abschnitt 2.4.3).

Die Strukturierung erfolgt über eine Hierarchie von Kulturen [GBK+03, KG04, KG05]. Jede dieser Kulturen garantiert die Einhaltung einer Menge von Systemeigenschaften.

Eine Kultur besteht aus

- einer Menge von Subkulturen,
- einer Menge von Rollen,
- einer Menge von Regeln:
  - Instanzierungsregeln,
  - Verhaltensregeln,
  - Absichtserklärungen,
- und Systemeigenschaften.

Die oben eingeführten Komponenten werden in diesem Kontext als Agenten aufgefasst. Eine Menge von Agenten, die die Regeln einer Kultur realisieren, werden in einer Community zusammengefasst. Eine solche Community wird dynamisch durch die Instanzierungsregeln der Kultur gebildet. Das physikalische und soziale Verhalten eines Agenten wird durch eine Menge von Verhaltensregeln definiert. Mit einer Absichtserklärung teilt der Agent mit, wie er sich in naher Zukunft verhalten wird.

Die Systemeigenschaften dürfen durch die Anwendung der Regeln nicht verletzt werden. Zu diesen Systemeigenschaften gehört zum Beispiel, dass die Agenten sich nicht anders verhalten dürfen als in einer Absichtserklärung versprochen. Andere Eigenschaften beschreiben, dass keine Unfälle und kritische Situationen eintreten dürfen. Solche Eigenschaften werden z.B. als

verbotene Story Patterns beschrieben. Das heißt, wenn das verbotene Story Pattern auf ein Objektdiagramm angewendet werden kann, ist ein Unfall oder eine kritische Situation eingetreten.

Nach Storey [Sto96] ist eine kritische Situation (engl. hazard) definiert als „eine Situation, in der Personen, die Umwelt oder Material möglicherweise oder tatsächlich gefährdet sind“. Demgegenüber ist ein Unfall (engl. accident) „ein nicht beabsichtigtes Ereignis oder eine Folge von nicht beabsichtigten Ereignissen, die sowohl den Tod oder Verletzungen von Personen sowie Umwelt- oder Materialschäden verursachen“.

Koordinationsmuster stellen eine eingeschränkte Form der Kulturen dar. Sie bestehen aus einer Menge von Rollen und Regeln, die in Form der Kommunikationsprotokolle festgelegt sind. Sie besitzen jedoch keine Subkulturen. Die Protokolle stellen dabei eine Absichtserklärung dar. Die in den Koordinationsmustern spezifizierten Rolleninvarianten und Musterconstraints entsprechen den in der Kultur definierten Systemeigenschaften. Wird ein Koordinationsmuster instanziert, so entspricht dies der Bildung einer Community, wobei den beteiligten Agenten die Rollen des Musters zugewiesen werden.

Die Idee besteht nun darin, mit Lösungen für einzelne kleine Probleme zu beginnen und dann diese Teillösungen zu Gesamtlösungen zusammenzufassen. Koordinationsmuster stellen solche Teillösungen dar. In einer solchen Teillösung ist eine Komponente ein Agent, dem bestimmte Rollen (die Musterrollen) zugewiesen werden. In den oben eingeführten Real-Time Statecharts ist bereits vorgesehen, dass ein Koordinationsmuster zur Laufzeit instanziert wird (modelliert durch die Zustände active und inactive). Somit ist festgelegt, dass die Komponente eine bestimmte Rolle übernehmen kann.

Ein Beispiel für eine Kulturhierarchie ist in Abbildung 2.15 dargestellt. Die oberste Kultur ist die Movement-Kultur, die die ControlledMovement-Kultur als Subkultur besitzt. Die ControlledMovement-Kultur hat zwei Subkulturen, das Publication-Koordinationsmuster und die CoordinatedMovement-Kultur. Diese hat wiederum das DistanceCoordination-Koordinationsmuster als Subkultur.

Um die Kulturen der Hierarchie modellieren zu können, muss zunächst die in Abschnitt 2.1.3 eingeführt Ontologie um Klassen und Assoziationen erweitert werden, die die konzeptionellen Elemente darstellen. Als erstes wird dazu die next Assoziation zwischen der Klasse Shuttle und der Klasse Track eingefügt. Diese Assoziation stellt eine Absichtserklärung dar. Mit ihrer Hilfe kann ein Shuttle den anderen Shuttles mitteilen, auf welche Tracks es als nächstes fahren wird.

Die Ontologie wird um ein Koordinationsmuster Publication erweitert. Mittels dieses Musters meldet sich ein Shuttle bei der BaseStation an, die den Track überwacht, auf dem sich das Shuttle befindet. Ist ein Shuttle bei einer BaseStation angemeldet, so sendet es in regelmäßigen Abständen seine Positionsdaten an die BaseStation. Diese sendet die Daten wiederum an alle bei ihr gemeldeten Shuttles. Auf diese Weise erfahren die Shuttles, welche anderen Shuttles sich in ihrer Nähe befinden. Erhält die BaseStation die Positionsdaten eines Shuttles nicht, so warnt sie die anderen Shuttles, dass dieses Shuttle möglicherweise defekt ist.

Durch das Publication-Koordinationsmuster erfährt ein Shuttle, welche anderen Shuttles sich in seiner Nähe befinden. Dies ermöglicht eine Koordination zwischen den Shuttles. Diese Koordination wird durch das DistanceCoordination-Muster spezifiziert. Es stellt die Umsetzung der Subkultur DistanceCoordination dar. Die Rollen dieser Subkultur werden als Assoziationen in die Ontologie eingefügt. Die erweiterte Ontologie ist in Abbildung 2.16 dargestellt.

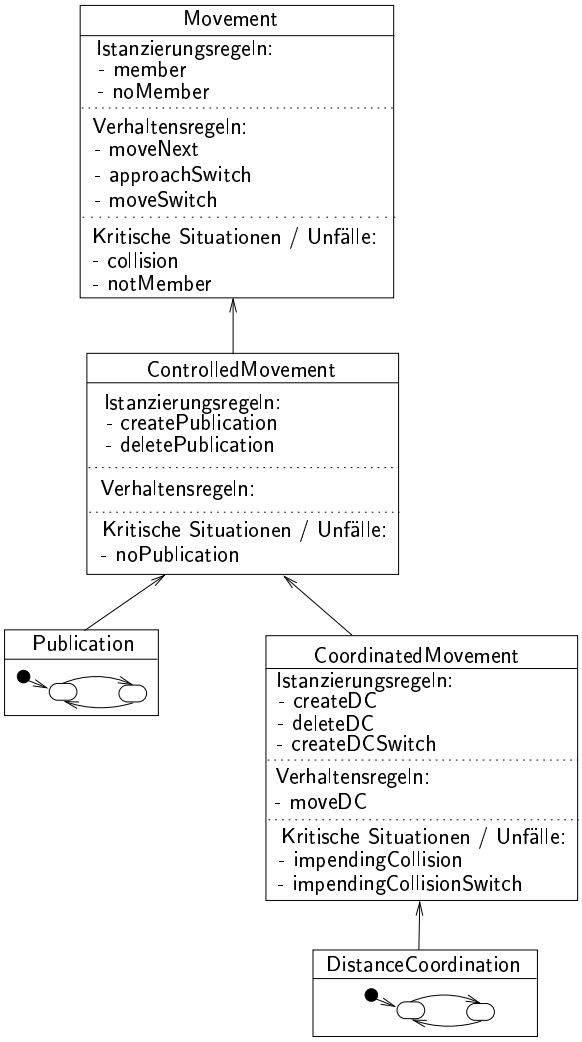



Abbildung 2.15: Beispiel für eine Kulturhierarchie

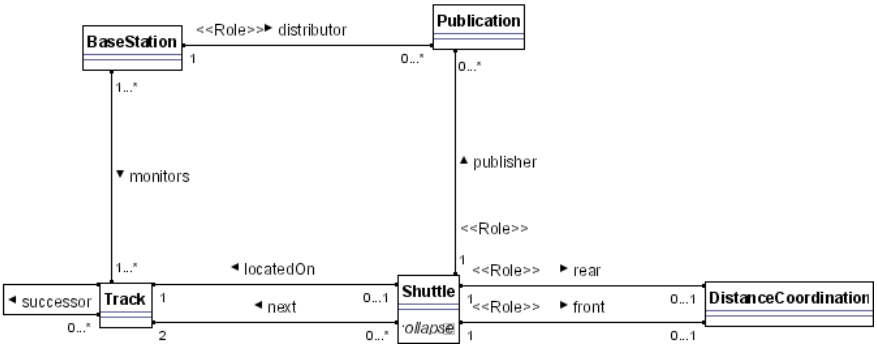



Abbildung 2.16: Ontologie erweitert um konzeptionelle Elemente

Ein Beispiel für eine Verhaltensregel der Movement-Kultur ist die Regel moveNext. Die Implementierung dieser Regel ist als Story Pattern in Abbildung 2.17 gegeben. Die Regel beschreibt, dass das Shuttle zwei next-Links besitzt. Ein next-Link zeigt auf den Track, auf den das Shuttle als nächstes fahren möchte, und der zweite auf den übernächsten Track. Befinden sich auf dem nächsten Track und dem übernächsten Track keine anderen Shuttles, so darf das Shuttle auf den nächsten Track weiterfahren. Dazu wird der locatedOn-Link zwischen dem Shuttle und dem Track rt1 gelöscht und ein neuer zum Track rt2 erzeugt. Außerdem wird der next-Link zu rt2:Track entfernt und ein neuer zum Track rt4 erzeugt. Durch die Anwendung dieser Regel wird das Shuttle auf den nächsten Track gesetzt und es wird gleichzeitig festgelegt, auf welchen Track das Shuttle als übernächstes fahren möchte.

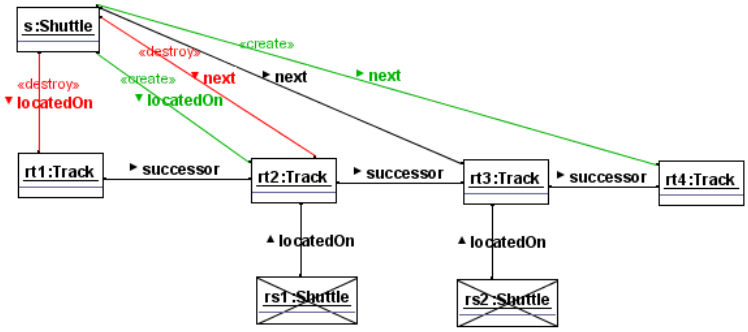



Abbildung 2.17: Beispiel für die Implementierung der Verhaltensregel moveNext

Eine Eigenschaft, die durch die Absichtserklärungen, Instanzierungs- und Verhaltensregeln eingehalten werden muss, ist, dass sich niemals zwei Shuttles auf dem gleichen Track befinden dürfen. Andernfalls sind die beiden Shuttles kollidiert. Diese Eigenschaft wird durch das verbotene Story Pattern collision in Abbildung in 2.18 modelliert. In dieser Arbeit wird der Begriff verbotene Story Patterns für solche Story Patterns verwendet, die kritische Situationen oder Unfälle beschreiben. Der Begriff Story Pattern wird nur für die Story Pattern verwendet, die Verhaltens- oder Instanzierungsregeln beschreiben.

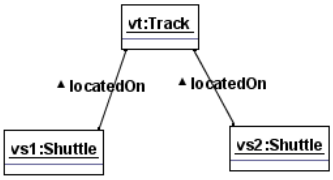



Abbildung 2.18: Verbotenes Story Pattern collision

In der DistanceCoordination-Kultur wird die eigentliche Abstandshaltung der Shuttles definiert. Sie stellt eine Subkultur der ControlledMovement-Kultur dar, die wiederum eine Subkultur der Movement-Kultur ist. Das DistanceCoordination-Muster implementiert diese Kultur.

Die Instanzierungsregeln der DistanceCoordination-Kultur stellen die beiden Regeln createDC und deleteDC dar. Dabei instanziert createDC das DistanceCoordination-Muster und deleteDC löscht eine Instanz dieses Musters. Die Instanzierung des Muster bedeutet in diesem

Fall, dass eine Funkverbindung (beschrieben durch den Konnektor des Koordinationsmusters) aufgebaut und die Kommunikation zwischen den beiden Komponenten gestartet wird. Ebenso bedeutet das Löschen des Koordinationsmusters, dass die Funkverbindung beendet wird. Beide Regeln werden in der Verhaltensbeschreibung der Shuttle-Komponente aufgerufen.

Die Implementierung der Regel createDC ist in Abbildung 2.19 gegeben. Sie wird von dem Shuttle ausgeführt, das mit this beschriftet ist. Die Regel beschreibt, dass das Koordinationsmuster erzeugt werden muss, wenn auf dem Track auf den das this:Shuttle als übernächstes fahren möchte, bereits ein anderes Shuttle ist. Das Koordinationsmuster darf nur dann erzeugt werden, wenn das mit this gekennzeichnete Shuttle noch nicht in einem Koordinationsmuster die rearRole ausführt und das mit rs2 beschriftete Shuttle noch nicht die frontRole ausführt. Dies verhindert zum einen, dass das Koordinationsmuster mehrfach zwischen den beiden Shuttles instanziert wird. Zum anderen wird dadurch verhindert, dass das Koordinationsmuster instanziert wird, obwohl sich zwischen den beiden Shuttles noch ein weiteres Shuttle befindet.

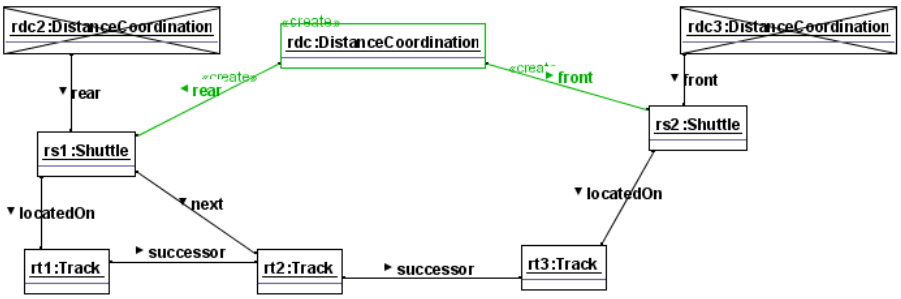



Abbildung 2.19: Instanzierungsregel createDC

Ein Beispiel für eine Verhaltensregel dieser Kultur ist die Regel moveDC, die eine koordinierte Fahrt des Shuttles beschreibt. Koordinieren sich zwei Shuttles über eine Instanz des DistanceCoordination-Musters, so darf das hintere Shuttle weiterfahren, wenn zwischen den beiden Shuttles ein Track frei ist. Dadurch ermöglicht das DistanceCoordination-Muster, dass zwei Shuttles dichter hintereinander herfahren können als ohne das Koordinationsmuster. Diese Regel hat eine höhere Priorität als die Regeln der Movement-Kultur. Das bedeutet, wenn die Regel moveDC angewendet werden kann und zeitgleich auch eine Regel der Movement-Kultur, z.B. die Regel moveNext, dann wird immer die Regel mit der höheren Priorität angewendet.

Generell gilt für die Vergabe der Prioritäten: Je weiter unten in der Kulturhierarchie eine Verhaltensregel eingeführt wird, desto höher ist ihre Priorität. Bei den Instanzierungsregeln verhält es sich genau umgekehrt, je weiter oben in der Hierarchie eine Instanzierungsregel steht, desto höher ist ihre Priorität und die Instanzierungsregeln haben eine höhere Priorität als die Verhaltensregeln. Die Instanzierungsregeln haben eine höhere Priorität als die Verhaltensregeln, da ein Agent erst bei der Instanzierung einer Community seine Rolle und die dazu gehörigen Verhaltensregeln zugewiesen bekommt. Die Instanzierungsregeln einer Kultur haben eine höhere Priorität als die Instanzierungsregeln ihrer Subkulturen, da der Agent erst dann eine Subkultur instanzieren kann, wenn er Teil einer Community der Kultur ist.

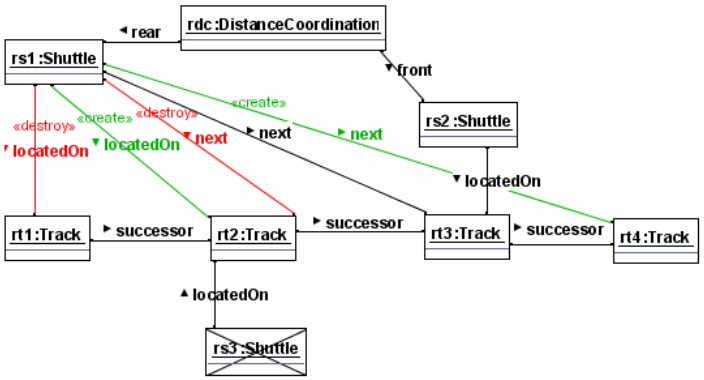



Abbildung 2.20: Verhaltensregel moveDC

Die ControlledMovement-Kultur muss zum einen die Eigenschaften der übergeordneten Movement-Kultur erfüllen. Zum anderen sind in der Kultur zusätzliche Eigenschaften spezifiziert, die die Regeln der Kultur erfüllen müssen. Eine dieser Eigenschaften beschreibt eine kritische Situation, die niemals eintreten darf. Diese kritische Situation besteht darin, dass sich zwei Shuttles auf benachbarten Tracks befinden, jedoch das DistanceCoordination-Muster nicht instanziert haben. In diesem Fall droht eine Kollision der Shuttles, denn obwohl die beiden Shuttles mit geringem Anstand hintereinander fahren, kommunizieren sie nicht. Das bedeutet, wenn das vordere Shuttle bremst, hat das hintere unter Umständen nicht mehr genug Zeit zum Reagieren und die beiden Shuttles würden kollidieren. Diese kritische Situation wird durch das verbotene Story Pattern impendingCollision in Abbildung 2.21 beschrieben.

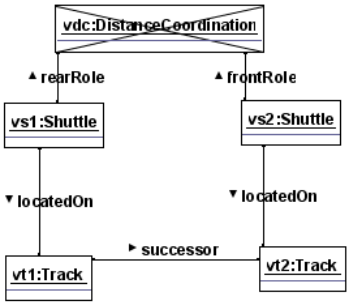



Abbildung 2.21: Verbotenes Story Pattern impendingCollision

In diesem Abschnitt wurde vorgestellt, wie durch die Verwendung sozialer Metaphern ein komplexes mechatronisches System strukturiert werden kann. Die Verwendung von hierarchischen Kulturen erlaubt es, Lösungen für einzelne Probleme zu entwickeln und diese dann zu einer Gesamtlösung zusammenzusetzen. Die in Abschnitt 2.2.1 eingeführten Koordinationsmuster stellen eine besondere Form dieser Kulturen dar. Mit Hilfe der Instanzierungsregeln der jeweiligen Kultur wird beschrieben, wann ein Koordinationsmuster instanziert wird bzw. wann eine Musterinstanz gelöscht wird.

Zusätzlich zu den Instanzierungsregeln wurden Verhaltensregeln eingeführt, die sowohl das physikalische als auch das soziale Verhalten der Agenten umsetzen. Wobei die Agenten den in Abschnitt 2.1.2 eingeführten Komponenten entsprechen.

Neben den Absichtserklärungen, Instanzierungs- und Verhaltensregeln enthalten die Kulturen auch eine Menge von Eigenschaften, die entweder in Form von TCTL-Formeln in den Koordinationsmustern angegeben oder in Form von verbotenen Story Patterns definiert werden. Diese Eigenschaften dürfen durch die Instanzierungs- und Verhaltensregeln sowie durch die Absichtserklärungen nicht verletzt werden. Im folgenden Abschnitt wird deshalb gezeigt, wie mittels kompositionaler formaler Verifikation nachgewiesen werden kann, dass die Eigenschaften eingehalten werden.

## 2.4 Verifikation

In Abschnitt 2.2.1 wurde gezeigt, wie die Echtzeitkommunikation in mechatronischen Systemen mittels Koordinationsmustern modelliert wird. Die in Abschnitt 2.1.2 eingeführten Komponenten verfeinern die Rollen der Koordinationsmuster, um die in den Mustern spezifizierten Kommunikationsprotokolle zu realisieren. In Abschnitt 2.3.2 wurden Kulturen eingeführt, um Systeme bestehend aus Komponenten und Koordinationsmustern zu strukturieren und die Systemkomposition aus einzelnen Teilsystemen zu ermöglichen.

Die Trennung der Kommunikation vom Komponentenverhalten sowie die Strukturierung des Gesamtsystems durch die Verwendung von Kulturen ermöglichen eine kompositionale Verifikation. Zur Verifikation werden dann zwei verschiedene Techniken eingesetzt. Zum einen werden die Real-Time Statecharts der Komponenten und Koordinationsmuster durch Model Checking verifiziert [GTB+03, GST+03]. Zum anderen werden die Story Patterns, die in den Kulturen definiert werden und die in den komponenteninternen Real-Time Statecharts aufgerufen werden, durch einen Induktionsbeweis verifiziert (siehe Kapitel 3).

### 2.4.1 Verifikation der Kommunikation

Ein Koordinationsmuster ist korrekt und damit die in ihm spezifizierte Kommunikation sicher, wenn das Muster die als temporal logische Formeln vorliegenden Musterconstraints erfüllt. Die Überprüfung, ob das Koordinationsmuster die Eigenschaften erfüllt, erfolgt durch Model Checking. Da die Semantik von Real-Time Statecharts durch Abbildung auf Timed Automata gegeben ist, ist keine zusätzlich Formalisierung des Modells notwendig.

Ein Koordinationsmuster kann durch einen Model Checker, wie zum Beispiel UPPAAL [LPY97] oder RAVEN [Ruf01] geprüft werden.

Bei der Verwendung der Koordinationsmuster wird davon ausgegangen, dass die Kommunikation zwischen zwei Kommunikationspartnern eine Punkt-zu-Punkt-Verbindung (engl. point-to-point) dargestellt. Dadurch können Seiteneffekte, die durch eine zusätzliche Kommunikation erzeugt werden können, ausgeschlossen werden. Bei der Verifikation ist es dann nicht notwendig, andere Koordinationsmuster oder Komponenten zu betrachten; alle notwendigen Informationen sind im Koordinationsmuster enthalten.

Der Model Checker prüft zum einen, ob das Muster das Musterconstraint einhält und zum anderen verklemmungsfrei ist.

Im Beispiel des DistanceCoordination-Musters muss der Model Checker prüfen, ob $M_{frontRole}||M_{Konnektor}||M_{rearRole} \models (A[]\ \text{not}\ (\text{rearRole.convoy and frontRole.noConvoy})) \wedge \neg\text{deadlock}$ gilt. Dabei bezeichnet $M_x$ den Timed Automata, auf den das entsprechende Real-Time Statechart abgebildet wurde und $||$ die Parallelausführung von Automaten.

### 2.4.2 Verifikation der Komponenten

Um nachzuweisen, dass eine Komponente korrekt ist, muss gezeigt werden, dass ihre Ports korrekte Verfeinerungen der Rollen darstellen, die Komponente die Rolleninvarianten aller verfeinerter Rollen erfüllt und verklemmungsfrei ist.

Dass ein Port eine korrekte Verfeinerung einer Rolle darstellt, wird durch seine Konstruktion erreicht. Bei dieser Konstruktion darf das Rollenverhalten nur um interne Kommunikation erweitert werden. Es darf jedoch kein Verhalten, das durch die Rolle garantiert wird, entfernt oder neues nach außen sichtbares Verhalten hinzugefügt werden.

Der Nachweis, dass die Komponente die Rolleninvarianten der verfeinerten Rollen erfüllt, erfolgt über Model Checking. Wie bei der Verifikation der Koordinationsmuster reicht es aus, eine Komponente unabhängig von anderen Komponenten und Koordinationsmustern zu betrachten. Allerdings werden bei dieser Verifikation die entry()-, do()- und exit()-Methoden der Zustände sowie die Methodenaufrufe der Transitionen nicht berücksichtigt. Für diese Methoden kann mittels des in [Sei05] vorgestellten Ansatzes die maximale Ausführungszeit bestimmt werden. Diese Information kann dann dazu genutzt werden, um zu bestimmen, wie lange das Schalten einer Transition maximal dauert. Diese Zeit wird dann beim Model Checking berücksichtigt. Die Verifikation der Methoden erfolgt im Rahmen der Verifikation der Kulturen (siehe 2.4.3). Beim Model Checking der Komponente wird davon ausgegangen, dass die Methoden korrekt sind.

Da für die Rollen nur verlangt wird, dass sie verklemmungsfrei sind, muss für die Shuttle-Komponente geprüft werden, ob $M_{frontPort}||M_{Synch}||M_{rearPort} \models \neg\text{deadlock}$ gilt.

Die Verifikation hybrider Komponenten ist Inhalt aktueller Forschung. Burmester liefert in [Bur05] Argumente, die darauf schließen lassen, dass die Verifikation hybrider Komponenten ähnlich zur Verifikation diskreter Komponenten erfolgen kann. Auch bei dieser Verifikation wird für die aufgerufenen Methoden, beschrieben durch Story Patterns, mittels des Ansatz von [Sei05] die maximale Ausführungszeit berechnet. Wie bei den diskreten Komponenten wird aber auch hier angenommen, dass die Methoden korrekt sind. Die Korrektheit der Methoden wird wie bei den diskreten Komponenten mittels der Verifikation der Kulturen sichergestellt. Dies ist möglich da die Story Patterns, die die Methoden beschreiben, diskret sind und auf der gleichen Ontologie arbeiten wie die Story Patterns der diskreten Komponenten.

### 2.4.3 Verifikation der Kulturen

Für jede Kultur wird eine Menge von Eigenschaften spezifiziert, die von Absichtserklärungen, Instanzierungs- und Verhaltensregeln erfüllt werden müssen. Wird eine Kultur instanziert, so

werden die verifizierten Regeln den beteiligten Agenten zugewiesen, je nachdem welche Rolle sie in der resultierenden Community übernehmen.

Die Organisation der Kulturen in einer Hierarchie ermöglicht eine kompositionale Verifikation. Für eine Kultur muss nachgewiesen werden, dass die in ihr angegebenen Regeln sowie die Regeln aller übergeordneter Kulturen, die Sicherheitseigenschaften einhalten, die in der betrachteten und allen übergeordneten Kulturen angegeben sind. Eigenschaften, die in untergeordneten Kulturen verlangt werden oder in Kulturen in einem anderen Zweig der Hierarchie enthalten sind, brauchen von der Kultur nicht eingehalten werden und deshalb bei der Verifikation einer Kultur auch nicht berücksichtigt werden. Somit führt die Verwendung der Kulturhierarchie dazu, dass immer nur ein Ausschnitt des Systems verifiziert werden muss. Wurden alle Kulturen erfolgreich verifiziert, so ist auch das Gesamtsystem korrekt. Eine Verifikation, die zeigt, dass alle Regeln des Systems alle Eigenschaften erfüllen, ist nicht notwenig[1].

Besteht eine Kultur aus einem Koordinationsmuster, so erfolgt die Verifikation des Musters wie in Abschnitt 2.4.1 dargestellt.

Werden die Regeln einer Kultur durch Story Patterns und die Eigenschaften durch verbotene Story Patterns beschrieben, so wird eine Instanzsituation, beschrieben durch ein UML-Objektdiagramm, als Graph aufgefasst (siehe dazu auch Kapitel 4). Jedes Objekt des Objektdiagramms entspricht dann einem Knoten des Graphen und jeder Link einer Kanten. Die Story Patterns, die die Regeln beschreiben, können dann als Graphtransformationsregeln aufgefasst werden.

Für die Verifikation von Graphtransformationsregeln existieren einige Ansätze (siehe Abschnitt 5.2 und 6). Allerdings sind diese Ansätze dahingehend beschränkt, dass sie zum einen einen Initialgraphen (also eine initiale Instanzsituation) benötigen, zum anderen dürfen durch die Anwendung der Graphtransformationsregeln nur endlich viele Graphen erzeugbar sein. Bei den hier betrachteten Systemen ist zum Zeitpunkt der Verifikation der Initialgraph noch nicht bekannt. Darüber hinaus können die Regeln der Kultur unendlich viele Graphen erzeugen. Deshalb wird in Kapitel 3 ein neuer Ansatz zur Verifikation von Graphtransformationsregeln vorgestellt, der nicht diesen Einschränkungen unterliegt.

Die Objekte eines Story Patterns können auch Koordinationsmuster darstellen. In diesem Fall gilt jedoch, dass das in dem Koordinationsmuster beschriebene Verhalten keinen Einfluss auf das Story Pattern hat. Bei der Verifikation der Story Patterns wird davon ausgegangen, dass das zugrunde liegende Koordinationsmuster korrekt ist. Ist dies der Fall, so garantiert die Verifikation der Story Patterns, dass benötigte Koordinationsmuster immer vorhanden sind bzw. nicht benötigte Koordinationsmuster gelöscht werden. Die Realisierung der Koordinationsmuster hat keinen Einfluss auf die Verifikation der Regeln.

### 2.4.4 Korrektheit des Gesamtsystems

Nachdem die Teile verifiziert wurden, aus denen das Gesamtsystem zusammengesetzt wird, ist die Verifikation abgeschlossen. Eine zusätzliche Verifikation des Gesamtsystems ist nicht mehr notwendig.

[1] In zukünftigen Arbeiten soll gezeigt werden, dass bei der Verifiaktion einer Kultur nur die Regeln und Eigenschaften verifiziert werden müssen, die Teil dieser Kultur sind, siehe dazu auch Abschnitt 7.

In [GTB+03, GST+03] wurde gezeigt, dass ein System, das aus erfolgreich verifizierten Koordinationsmuster- und Komponenteninstanzen syntaktisch korrekt zusammengesetzt ist, auch insgesamt korrekt ist.

Die Instanzierung und das Löschen der Koordinationsmuster erfolgt durch die Story Patterns. Für diese Story Patterns wird mit dem in Kapitel 3 vorgestellten Ansatz gezeigt, dass sie die Koordinationsmuster instanzieren, sobald sie benötigt werden bzw. löschen, wenn sie nicht mehr benötigt werden. Allerdings muss noch garantiert werden, dass das komponenteninterne Real-Time Statechart diese Story Patterns immer dann ausführt, wenn die Story Patterns auf die Ontologie der Komponente angewendet werden können. Um dies garantieren zu können, muss in jedem Zustand des komponenteninternen Real-Time Statechart jedes der Story Patterns entweder als do()-Methode oder als Seiteneffekt einer ausgehenden Transition ausgeführt werden.

Die in Abbildung 2.9 dargestellte Shuttle-Komponente erfüllt diese Anforderung noch nicht.

## 2.5 Der Modellierungs- und Verifikationsprozess

In den Abschnitten 2.1.2 bis 2.4.4 wurde gezeigt, wie die Software eines mechatronischen Systems modelliert, strukturiert und verifiziert wird. In diesem Abschnitt soll nun der Modellierungs- und Verifikationsprozess noch einmal zusammenfassend dargestellt werden.

Zunächst (1) wird das Koordinationsverhalten des Systems in Koordinationsmustern spezifiziert. Diese Koordinationsmuster werden (2), wie oben beschrieben, durch Model Checking verifiziert.

Ausgehend von den Koordinationsmusters wird die Kulturhierarchie aufgebaut (3). Die Regeln jeder Kultur, die mittels Story Patterns spezifiziert sind, werden (4) durch den in Kapitel 3 vorgestellten Ansatz verifiziert.

Als nächstes (5) wird das Koordinationsverhalten der Komponenten modelliert. Die Komponenten stellen die Agenten des mechatronischen Systems dar. Dazu wird aus den Koordinationsmustern/Kulturen die Menge der Rollen ausgewählt, die die Komponente realisieren soll. Die Rollen werden in den Ports der Komponente verfeinert. Die Regeln, die in einer Kultur für eine Rolle spezifiziert sind, werden im komponenteninternen Real-Time Statechart als entry()-, do()- und exit()-Methoden bzw. als Methodenaufrufen an den Transitionen verwendet. Mittels Model Checking kann (6) für jede Komponente nachgewiesen werden, dass sie die Rolleninvarianten der von ihr verfeinerten Rollen erfüllt. Nach der Verifikation wird (7) für jede Komponente und jedes Koordinationsmuster der entsprechende Code automatisch generiert.

Das Gesamtsystem wird dann (8) dynamisch zur Laufzeit gebildet.

## 2.6 Zusammenfassung

In diesem Kapitel wurde zu nächst die Architektur des Operator-Controller-Moduls vorgestellt. Diese Architektur ermöglicht eine Strukturierung der Informationsverarbeitung eines mechatronischen Systems und teilt es in die drei Ebenen Controller, reflektorischer Operator und kognitiver Operator ein. Die Software des reflektorischen Operators ist für die Steuerung eines mecha-

tronischen Systems und der Interaktion mit anderen Systemen verantwortlich. Da diese Software komplex und sicherheitskritisch ist wurde ein Ansatz zur kompositionalen Modellierung und Verifikation dieser Software vorgestellt. Die verwendeten Modellierungstechniken basieren auf Ausschnitten der UML. Allerdings wurden die gewählten Ausschnitte auf die Anforderungen bei der Modellierung von mechatronischen Systemen angepasst.

Die Architektur der Software wird mittels Komponentendiagrammen und Echtzeit-Koordinationsmustern beschrieben. Zur Spezifikation der komponenteninternen Struktur werden Klassendiagramme verwendet.

Das Koordinationsverhalten der Komponenten wird in den Koordinationsmustern mittels Real-Time Statecharts beschrieben. Komponenten, die eine bestimmte Rolle innerhalb eines solchen Koordinationsmusters einnehmen sollen, verfeinern die entsprechende Rolle zu einem Port. Ein Port sowie das komponenteninterne Verhalten wird durch Real-Time Statecharts modelliert. Dabei darf das komponenteninterne Real-Time Statechart entry()-,do()- und exit()-Methoden in den Zuständen aufrufen und Methoden als Seiteneffekte der Transitionen ausführen. Diese Methoden werden mittels Story Patterns beschrieben.

Soziale Metaphern wie Kulturen, Communities und Agenten werden verwendet, um die komplexe Software eines mechatronischen Systems zu strukturieren und die benötigten Story Patterns, die in den komponenteninternen Real-Time Statecharts aufgerufen werden, zu finden.

Um garantieren zu können, dass die resultierende Software bestimmte Sicherheitseigenschaften einhält, wird eine kompositionale Verifikation verwendet. Dabei wird jedes Koordinationsmuster und jede Komponente unabhängig von anderen Mustern und Komponenten mittels Model Checking überprüft. Wobei bei der Verifikation der Komponenten angenommen wird, dass die aufgerufenen Methoden korrekt sind. Um nachzuweisen, dass die aufgerufenen Methoden, beschrieben durch Story Patterns, korrekt sind, wird ausgenutzt, dass Story Patterns eine eingeschränkte Form von Graphtransformationsregeln darstellen. Das bedeutet, die Verifikation kann mittels Verifikationstechniken für Graphtransformationssysteme durchgeführt werden.

Für eine solche Verifikation existieren zwar schon einige Ansätze (siehe Abschnitt 6), diese sind jedoch zur Verifikation der Software mechatronischer Systeme nur eingeschränkt einsetzbar. Dafür gibt es zwei Gründe. Zum einen benötigen diese Verifikationstechniken einen Anfangszustand und zum anderen darf der resultierende Zustandsraum nur endlich sein.

Bei der Entwicklung der Software eines mechatronischen Systems soll die Verifikation zum frühestmöglichen Zeitpunkt erfolgen, um Fehler im System entsprechend früh erkennen und beheben zu können. Allerdings muss zu diesem Zeitpunkt der Anfangszustand des Systems noch nicht bekannt sein. Darüber hinaus kann für die Story Patterns, die die Methoden des Systems beschreiben, nicht garantiert werden, dass sie nur endlich viele Zustände erzeugen. Deshalb soll im folgenden Kapitel ein Ansatz vorgestellt werden, der diesen Einschränkungen nicht unterliegt.

# Kapitel 3

# Nachweis induktiver Invarianten

Im vorangegangenen Kapitel wurde gezeigt, wie die Softwarearchitektur eines mechatronischen Systems mittels Komponenten- und Klassendiagrammen beschrieben wird. Zudem wurden Echtzeit-Koordinationsmuster eingeführt, um eine sichere Kommunikation zwischen den Komponenten modellieren zu können. Das Verhalten der Komponenten sowie die Kommunikationsprotokolle der Koordinationsmuster werden durch Real-Time Statecharts spezifiziert. Kulturen und Communities strukturieren die Software eines mechatronischen Systems. Sie ermöglichen es, das Gesamtmodell aus kleinen Teilmodellen zusammenzufügen. In einer Kultur werden Regeln für ihre Teilnehmer festgelegt. Zu diesen Regeln gehören Instanzierungs- und Verhaltensregeln sowie Absichtserklärungen. Die Regeln werden mittels Story Patterns spezifiziert. Sie werden in den Real-Time Statecharts der Komponenten als entry()-, do()- und exit()-Methoden in den Zuständen aber auch als Seiteneffekte der Transitionen ausgeführt.

Damit ein derart spezifiziertes Modell korrekt ist, muss es die in den Kulturen festgelegten Eigenschaften erfüllen. Zu diesen Eigenschaften gehören einerseits die als TCTL-Formeln spezifizierten Musterconstraints und Rolleninvarianten der Koordinationsmuster, andererseits können aber auch verbotene Eigenschaften wie kritische Situationen und Unfälle, in Form von verbotenen Story Patterns, in den Kulturen definiert sein.

In Abschnitt 2.4 wurde gezeigt, wie mit Hilfe von Model Checking verifiziert werden kann, ob die Real-Time Statecharts der Koordinationsmuster und Komponenten die Musterconstraints und Rolleninvarianten einhalten. In diesem Kapitel wird ein Ansatz vorgestellt, der eine Verifikation der durch Story Patterns beschriebenen Regeln ermöglicht.

Die Idee, die der Verifikation zugrunde liegt, ist die folgende: Ein Systemzustand wird durch seine Objekte und deren Links charakterisiert, d.h. ein Systemzustand kann durch ein Objektdiagramm beschrieben werden. Die Story Patterns beschreiben die Zustandsübergänge, also das Instanzieren und Löschen von Objekten sowie das Erzeugen und Löschen von Links. Ein Systemzustand ist korrekt, wenn auf das entsprechende Objektdiagramm keines der verbotenen Story Patterns anwendbar ist. Um überprüfen zu können, ob ein inkorrekter Zustand durch die Anwendung der Regeln erreicht werden kann, werden die Objektdiagramme als Graphen aufgefasst und die Story Patterns als Graphtransformationsregeln. Der Anfangszustand, der durch einen Initialgraphen repräsentiert wird, zusammen mit allen Graphtransformationsregeln bildet dann ein Graphtransformationssystem.

Für die Verifikation von Graphtransformationssysteme existieren bereits einige Ansätze (siehe Abschnitt 5.2 und Kapitel 6). Diese unterliegen aber den Einschränkungen, dass sie einen Initialgraphen benötigen, um die Analyse durchführen zu können. Außerdem sind die meisten dieser Ansätze auf Graphtransformationssysteme beschränkt, die nur eine endliche Menge von Graphen erzeugen können. Diese beiden Einschränkungen sind in mechatronischen Systemen im Allgemeinen jedoch nicht erfüllt.

In diesem Kapitel wird deshalb ein Ansatz vorgestellt, der eine Verifikation der Regeln auch dann erlaubt, wenn kein Initialgraph gegeben ist und die Regeln unendlich viele Graphen erzeugen können. Der Ansatz wird in Abschnitt 3.1 zunächst informal erläutert, bevor in Abschnitt 3.2 ff. die formale Erläuterung erfolgt. Die Beweisskizzen zu den in Abschnitt 3.2 ff. aufgeführten Lemmata und Theoremen sind in Anhang B enthalten.

Heckel und Wagner stellen in [HW95] ein Ansatz vor, bei dem die Systemkorrektheit durch eine automatische Modifikation der Regeln erreicht wird. Systemeigenschaften werden dabei als Konsistenzbedingungen modelliert. Die Konsistenzbedingungen werden dabei als Graphen und Graphhomomorphismen spezifiziert. Die Modifikation erfolgt indem die Nachbedingungen jeder Regel mit einer Konsistenzbedingung verbunden werden. Dabei resultiert dann ein neuer Graph. Auf diesen Graphen wird die Regel in Rückwärtsrichtung angewendet. Der dabei entstandene Graph liefert eine Beschreibung, wann die entsprechende Regel nicht angewendet werden darf, und wird als negative Anwendungsbedigung der Regel bezeichnet. Der im Folgenden vorgestellte Ansatz verwendet eine ähnliche Technik. Auch dieser Ansatz verbindet die Nachbedingung einer Regel mit einem Graphen, der eine Eigenschaft beschreibt. Allerdings werden die Regeln nicht automatisch verändert. Findet der Ansatz eine Regel, die inkorrekt ist, so liefert er ein Gegenbeispiel, das zeigt, wann die Anwendung der Regel einen inkorrekten Graphen erzeugen kann. Eine ausführliche Diskussion der Unterschiede zwischen den beiden Ansätzen wird in Abschnitt 6.1 gegeben.

In diesem Kapitel wird zunächst die Idee des Verifikationsansatzes informal beschrieben (Abschnitt 3.1). In Abschnitt 3.2 werden dann die Grundlagen der Graphtransformationssysteme formal eingeführt. In dieser Arbeit werden Eigenschaften von Graphtransformationssystemen und Mengen von Graphen mittels Graphmustern beschrieben. Diese werden in Abschnitt 3.3 eingeführt. Nachdem diese Grundlagen beschrieben wurden, werden die Graphen und Graphtransformationssysteme um Typen erweitert (Abschnitt 3.4). Der Verifikationsansatz verlangt, dass Graphtransformationsregeln auch rückwärts und auf Graphmuster angewendet werden können. Damit dies möglich ist, sind einige Erweiterungen notwendig, diese werde in Abschnitt 3.5 vorgestellt. Abschnitt 3.6 beschäftigt sich dann mit Systemeigenschaften und beschreibt Einschränkungen, die bei der Verifikation der Graphtransformationssysteme ausgenutzt werden können. Der eigentliche Verifikationsansatz wird dann in Abschnitt 3.7 vorgestellt.

## 3.1 Die Idee

Der in diesem und den folgenden Abschnitten vorgestellte Ansatz ermöglicht die Verifikation von Graphtransformationssystemen. Nach der informalen Einführung von Graphtransformationssystemen werden Graphmuster zur Beschreibung struktureller Eigenschaften solcher Systeme ein-

geführt und die Verifikationsidee erläutert. Dazu werden sowohl die Graphtransformationen als auch die Graphmuster zunächst so vereinfacht, dass sie keine negativen Anwendungsbedingungen enthalten. In Abschnitt 3.1.5 wird dann die Verifikation von Regeln mit negativer Anwendungsbedingung betrachtet.

### 3.1.1 Graphtransformationssysteme

Ein Graph besteht nach [Roz97] aus einer Menge von typisierten Knoten und einer Menge von gerichteten und typisierten Kanten zwischen den Knoten. Bei den hier betrachteten Graphen handelt es sich um typisierte Graphen ohne Attribute.

Eine Graphtransformationsregel beschreibt die Veränderung eines Graphen durch Löschen und Hinzufügen von Knoten und Kanten (im Folgenden kurz als Elemente bezeichnet). Eine solche Regel $L \rightarrow_r R$ besteht aus einer linken Regelseite L, der Anwendungs- oder Vorbedingung und einer rechten Regelseite R, der Nachbedingung. Jede der beiden wird durch einen Graphen beschrieben. r ist der Name der Regel.

Soll eine Graphtransformationsregel auf einen Graphen, den Anwendungsgraphen, angewendet werden, so müssen alle Elemente der Anwendungsbedingung auf entsprechende Elemente aus dem Anwendungsgraphen abgebildet werden können. Ist dies möglich, so werden alle Elemente, die nur in der Anwendungsbedingung enthalten sind, nicht aber in der Nachbedingung, gelöscht. Elemente, die in beiden Graphen enthalten sind, bleiben erhalten. Solche Elemente, die nur in der Nachbedingung enthalten sind, werden neu erzeugt und in den Anwendungsgraphen eingefügt.

In Abbildung 3.1 ist ein Beispiel für eine Graphtransformationsregel gegeben. Diese Regel beschreibt die Vorwärtsbewegung eines Shuttles rs von einem Track $rt_1$ auf einen folgenden Track $rt_2$. Dazu wird die locatedOn-Kante $rlo_1$ zwischen rs und $rt_1$ gelöscht und eine neue locatedOn-Kante zwischen rs und $rt_2$ erzeugt. Die Regel entspricht dem Story Pattern moveSimple in Abbildung 2.10 aus Kapitel 2.

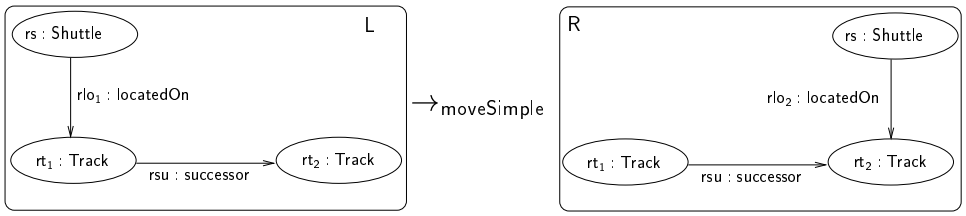



Abbildung 3.1: Die Graphtransformationsregel moveSimple

Ein Graphtransformationssystem besteht aus einer Menge von Initialgraphen sowie einer Menge von Graphtransformationsregeln.

### 3.1.2 Graphmuster und Systemkorrektheit

Strukturelle Eigenschaften, die von einem gegebenen Graphtransformationssystem erfüllt werden sollen, werden durch Graphmuster beschrieben. Solange die negativen Anwendungsbedin-

gungen außer Acht gelassen werden, entsprechen diese Graphmuster einfachen Graphen. Graphmuster beschreiben Teilgraphen. Jeder Graph, der einen solchen Teilgraphen enthält, erfüllt das Graphmuster. Deshalb können Graphmuster dazu verwendet werden, Mengen von Graphen zu beschreiben.

Es gibt zwei Arten von Graphmustern; die geforderten Graphmuster und die verbotenen Graphmuster. Geforderte Graphmuster müssen immer erfüllt werden. Verbotene Graphmuster dürfen nie erfüllt werden. Sie stellen kritische Situationen oder Unfälle dar und sind deshalb in dieser Arbeit von besonderem Interesse.

Abbildung 3.2 zeigt das verbotene Graphmuster impendingCollisionSimplified. Dieses verbotene Graphmuster besagt, dass eine kritische Situation eingetreten ist, wenn sich zwei Shuttles auf benachbarten Tracks befinden. Dabei handelt es sich um eine Vereinfachung der in Abschnitt 2.1.3 beschriebenen kritischen Situation impendingCollision, bei der sich nie zwei Shuttles auf benachbarten Tracks befinden dürfen, die nicht das DistanceCoordination-Koordinationsmuster miteinander ausführen.

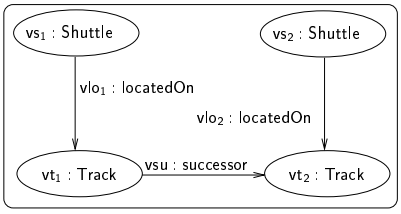


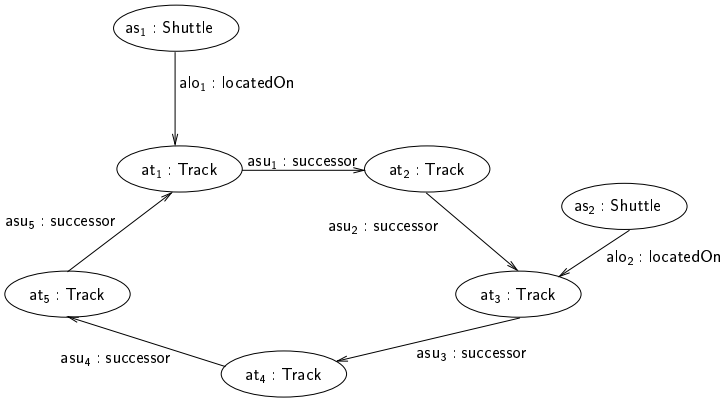

Abbildung 3.2: Verbotenes Graphmuster impendingCollisionSimplified

Ein Graph und somit der durch den Graphen repräsentierte Systemzustand ist korrekt, wenn er kein verbotenes Graphmuster enthält, d.h. wenn keines der verbotenen Graphmuster auf einen Teilgraphen des Graphen abgebildet werden kann. Das gesamte Graphtransformationssystem ist korrekt, wenn alle durch die Anwendung der Graphtransformationsregeln erreichbaren Graphen korrekt sind.

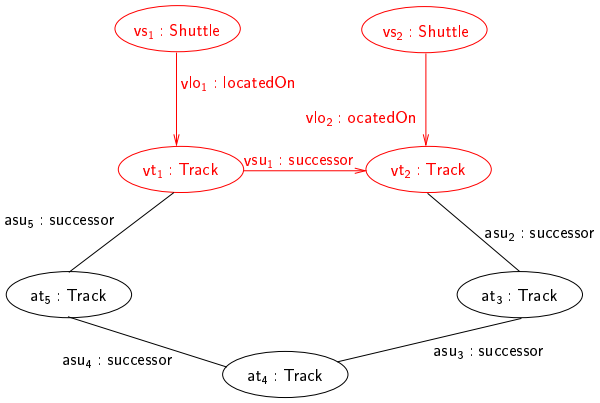
In Abbildung 3.3(a) ist ein Graph dargestellt, der im Bezug auf das verbotene Graphmuster impendingCollisionSimplified korrekt ist. Abbildung 3.3(b) zeigt dagegen einen inkorrekten Graphen, da das verbotene Graphmuster impendingCollisionSimplified auf einen Teilgraphen abgebildet werden kann. Die Elemente, auf die das verbotene Graphmuster abgebildet werden kann, sind zum einen rot dargestellt und zum anderen beginnt ihr Name jeweils mit v. Elemente, auf die kein Element des verbotenen Graphmusters abgebildet wird, sind schwarz dargestellt und ihr Name beginnt mit einem a.

### 3.1.3 Erreichbarkeitsanalyse

Eine erste Idee, die Korrektheit eines Graphtransformationssystems im Bezug auf eine Menge von verbotenen Graphmustern zu prüfen, stellt die Erreichbarkeitsanalyse dar. Dabei wird zunächst für den Anfangszustand geprüft, ob dieser korrekt ist. Ist das nicht der Fall, so kann die Analyse beendet werden, da das System schon bei der Initialisierung inkorrekt ist. Ist der Initialgraph korrekt, so werden alle Graphtransformationsregeln angewendet, deren Anwendungs-




(a) Beispiel für einen Graphen, der im Bezug auf das verbotene Graphmuster impendigCollisionSimplified korrekt ist




(b) Beispiel für einen Graphen, der im Bezug auf das verbotene Graphmuster impendigCollisionSimplified inkorrekt ist

Abbildung 3.3: Beispiel für die Korrektheit bzw. Inkorrektheit von Graphen

bedingung erfüllt ist, d.h. der Graph, der die Anwendungsbedingung beschreibt, kann auf einen Teilgraphen des Initialgraphen abgebildet werden. Die Anwendung aller möglichen Graphtransformationsregeln auf den Anwendungsgraphen resultiert in einer Menge von Graphen. Für jeden dieser Graphen muss wieder geprüft werden, ob er korrekt ist. Ist einer der Graphen inkorrekt, so bricht die Analyse ab, da das System einen inkorrekten Graphen erreichen kann. Sind alle Graphen korrekt, so werden auf jeden der Graphen alle anwendbaren Graphtransformationsregeln angewendet, woraus wieder eine Menge von Graphen resultiert. Auch für diese Menge muss überprüft werden, ob alle in ihr enthaltenen Graphen korrekt sind. Dieses Verfahren wird so lange fortgesetzt bis entweder ein inkorrekter Graph gefunden wurde oder keine weitere Graphtransformationsregel mehr anwendbar ist. Im letzteren Fall ist das System korrekt, da kein inkorrekter

Graph erreichbar ist. Die bei der Erreichbarkeitsanalyse nachgewiesenen Eigenschaften werden als operationale Invarianten bezeichnet.

Bei der Erreichbarkeitsanalyse wird geprüft, ob ein inkorrekter Graph erreicht werden kann. Eine solche Analyse ist jedoch nicht immer möglich. Zum einen ist zur Entwurfszeit eines mechatronischen Systems, wenn die Korrektheit der Graphtransformationsregeln geprüft wird, die Menge der Initialgraphen des Graphtransformationssystems noch nicht bekannt bzw. die Menge kann sich noch verändern. Für jeden neu hinzugefügten Initialgraphen müsste erneut eine Analyse durchgeführt werden. Das zweite Problem, das diese Art der Analyse verhindert, ist, dass durch die Anwendung der Graphtransformationsregeln unendlich viele verschiedene Graphen erzeugt werden können, sodass eine Überprüfung aller erreichbaren Graphen nicht terminiert.

Deshalb wird im folgenden Abschnitt ein Verfahren vorgestellt, das die Korrektheit einer Menge von Graphtransformationsregeln bezüglich einer Menge von Sicherheitseigenschaften, die als Graphmuster gegeben sind, auch dann nachweisen kann, wenn die Regeln unendlich viele Graphen erzeugen können und kein Initialgraph gegeben ist.

### 3.1.4 Nachweis induktiver Invarianten

Eine Erreichbarkeitsanalyse ist nur dann möglich, wenn ein Initialgraph gegeben ist und das betrachtete Graphtransformationssystem nur endlich viele erreichbare Graphen besitzt. Deshalb soll im Folgenden ein Verfahren beschrieben werden, das unabhängig von der Erreichbarkeit eines Graphen überprüft, ob eine Graphtransformationsregel ein verbotenes Graphmuster erzeugen kann.

Bei dieser Verifikation wird geprüft, ob die Anwendung einer Graphtransformationsregel auf einen korrekten Graphen wieder einen korrekten Graphen erzeugt. Ob die korrekten Graphen zur Menge der erreichbaren Graphen gehört, wird dabei nicht betrachtet. Kann für alle Regeln gezeigt werden, dass sie, wenn sie auf korrekte Graphen angewendet werden, niemals einen inkorrekten Graphen erzeugen können, so stellt das „Nicht-Auftreten“eines verbotenen Graphmusters eine induktive Invariante des betrachteten Graphtransformationssystems dar [GS04, GSK+06, BGS05a].

Der Ansatz nutzt drei Eigenschaften der Graphtransformationsregeln: (1.) Es muss mindestens ein Element (Knoten oder Kante) im verbotenen Graphmuster geben, das auf ein Element der rechten Regelseite abgebildet werden kann. Andernfalls würde die Regelanwendung das verbotene Muster nicht erzeugen. (2.) Die Anwendung einer Graphtransformationsregel auf einen Graphen hat nur Auswirkungen auf Elemente, auf die die Elemente der Regel abgebildet werden. Deshalb ist es ausreichend, Ausschnitte von Graphen statt vollständiger beliebig großer Graphen zu betrachten. (3.) Eine Graphtransformationsregel kann auch rückwärts angewendet werden, dadurch kann für einen inkorrekten Graphen geprüft werden, ob dieser durch die Anwendung einer Regel aus einem korrekten Graphen entstanden sein kann.

Zu (1.): Gegeben sind eine Graphtransformationsregel und ein verbotenes Graphmuster. Ist zusätzlich ein Graph gegeben, der im Bezug auf das verbotene Graphmuster korrekt ist, so kann der Graph, der resultiert, wenn man die Graphtransformationsregel auf den gegebenen korrekten Graphen anwendet nur dann inkorrekt sein, wenn die Anwendung der Regel den korrekten Graphen in den inkorrekten überführt. Da das verbotene Graphmuster nicht Teil des Anwendungs-

graphen war, muss durch die Anwendung der Regel mindestens ein Element erzeugt worden sein, das Bestandteil des verbotenen Graphen ist.

Die Abbildungen 3.4 und 3.5 zeigen je zwei Graphen. Der jeweils untere ist inkorrekt, da er das verbotene Graphmuster impendingCollisionSimplified enthält. Beide Graphen sind durch die Anwendung der Graphtransformationsregel moveSimple entstanden. Der Anwendungsgraph, auf den die Regel angewendet wurde, ist jeweils der obere Graph.

In diesem und den folgenden Beispielen wurde die folgende Namens- und Farbkonvention verwendet: Elemente, auf die sowohl die Elemente der Regel als auch die des verbotenen Graphmusters abgebildet werden, sind blau dargestellt und ihr Name beginnt mit einem m. Elemente, auf die nur Elemente der Regel abgebildet werden, sind grün gekennzeichnet und ihr Name beginnt mit einem r. Solche Elemente, auf die nur Elemente des verbotenen Musters abgebildet werden, sind rot und ihr Name beginnt mit einem v. Alle übrigen Elemente sind schwarz und ihr Name beginnt mit a. Wobei m für Merging steht und eine Verbindung von verschiedenen Graphen beschreibt, v steht für verboten, r für Regel und a für Anwendungsgraph.

In Abbildung 3.4 wurde durch die Anwendung der Regel kein Element erzeugt, das zum verbotenen Muster gehört. Betrachtet man den Graphen im oberen Teil der Abbildung, so sieht man, dass das verbotene Graphmuster schon Teil des Anwendungsgraphen war und somit nicht durch die Regelanwendung entstanden ist. Genauer enthält der Anwendungsgraph im oberen Teil der Abbildung das verbotene Graphmuster sogar zweimal. Zum einen kann er auf den Teilgraphen abgebildet werden, der aus den Knoten $vs_1$, $mt_1$, $vt_2$ und $vs_2$ besteht und durch die Regelanwendung nicht beeinflusst wird, zum anderen kann das verbotene Graphmuster aber auch auf den Teilgraphen abgebildet werden, der aus den Knoten $rs_1$, $rt_1$, $mt_1$ und $vs_1$ besteht. Ebenso enthält der resultierende Graph neben dem verbotenen Graphmuster impendingCollision auch ein verbotenes Graphmuster, dass die Kollision von zwei Shuttles und somit einen Unfall beschreibt. Dieser Teilgraph besteht aus den Knoten $vs_1$, $mt_1$ und $vs_2$.

Im Gegensatz dazu wurde in Abbildung 3.5 die Kante $mlo_1$ durch die Anwendung der Regel erzeugt, auf die die Kante $vlo_1 : locatedOn$ abgebildet werden kann. Der Anwendungsgraph war noch korrekt, erst die Anwendung der moveSimple-Regel hat die Kante erzeugt, die das verbotene Graphmuster vervollständigt hat und somit den korrekten Graphen in einen inkorrekten überführt.

Zu (2.): Um eine Graphtransformationsregel auf einen Graphen anwenden zu können, müssen alle Elemente der linken Regelseite auf Elemente des Graphen abgebildet werden. Durch die Anwendung der Regel können nur diese Elemente gelöscht werden. Ebenso können nur solche Elemente zum Graphen hinzugefügt werden, die in der rechten Regelseite enthalten sind. Damit eine neu erzeugte Kante inzident zu einem existierenden Knoten sein kann, muss der existierende Knoten in der rechten Regelseite enthalten sein. Ebenso kann ein neuer Knoten nur dann adjazent zu einem existierenden Knoten sein, wenn die entsprechende Kante neu erzeugt wird und der existierende Knoten Teil der rechten Regelseite ist. Somit hat die Anwendung einer Regel nur Auswirkungen auf Elemente, auf die die Elemente der Regel abgebildet werden.

Zu (3.): Wurde eine Graphtransformationsregel auf einen Graphen angewendet, so kann diese Anwendung wieder rückgängig gemacht werden. Die dazu notwendigen Einschränkungen werden in Abschnitt 3.5 erläutert. Dazu wird die Regel rückwärts auf den entstandenen Graphen angewendet, d.h. Elemente, die in der rechten Regelseite, aber nicht in der linken Seite der Regel

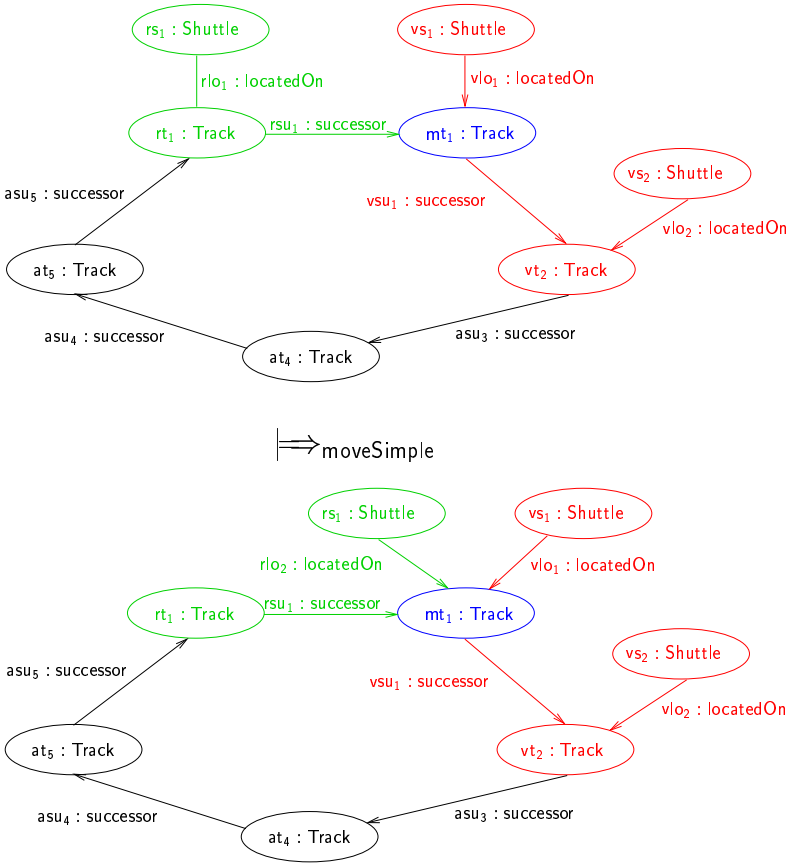



Abbildung 3.4: Die Regel moveSimple transformiert einen inkorrekten Graphen in einen anderen inkorrekten Graphen

enthalten sind, werden gelöscht und Elemente, die in der linken Regelseite enthalten sind, aber nicht in ihrer rechten Seite, werden erzeugt.

Bei der weiter oben beschriebenen Erreichbarkeitsanalyse wird ausgehend von einem Startgraphen überprüft, ob die daraus erzeugbaren Graphen korrekt sind. Im Folgenden wird in umgekehrter Richtung vorgegangen. Das bedeutet, ausgehend von jedem inkorrekten Graphen, der ein verbotenes Graphmuster enthält, wird für alle Regeln überprüft, ob sie diesen inkorrekten Graphen aus einem korrekten erzeugt haben können. Bei der Analyse werden also Graphpaare gesucht, bei denen der Anwendungsgraph korrekt ist, die Anwendung der betrachteten Regel jedoch in einem inkorrekten Graphen resultiert.

Nach (1.) brauchen dazu nur solche Regeln betrachtet werden, die Elemente erzeugen, auf die Elemente eines verbotenen Graphmusters abgebildet werden können. Konnte eine solche Regel gefunden werden, so wird diese rückwärts auf den inkorrekten Graphen angewendet. Der resultierende Graph wird nun auf seine Korrektheit hin überprüft. Enthält dieser Graph kein verbotenes Graphmuster, ist also korrekt, so wurde ein Beispiel gefunden, das zeigt, dass die angewendete Regel das System in einen inkorrekten Zustand überführen kann. Enthält dagegen der resultierende Graph ebenfalls ein verbotenes Graphmuster, so erzeugt die Regel einen inkorrek-

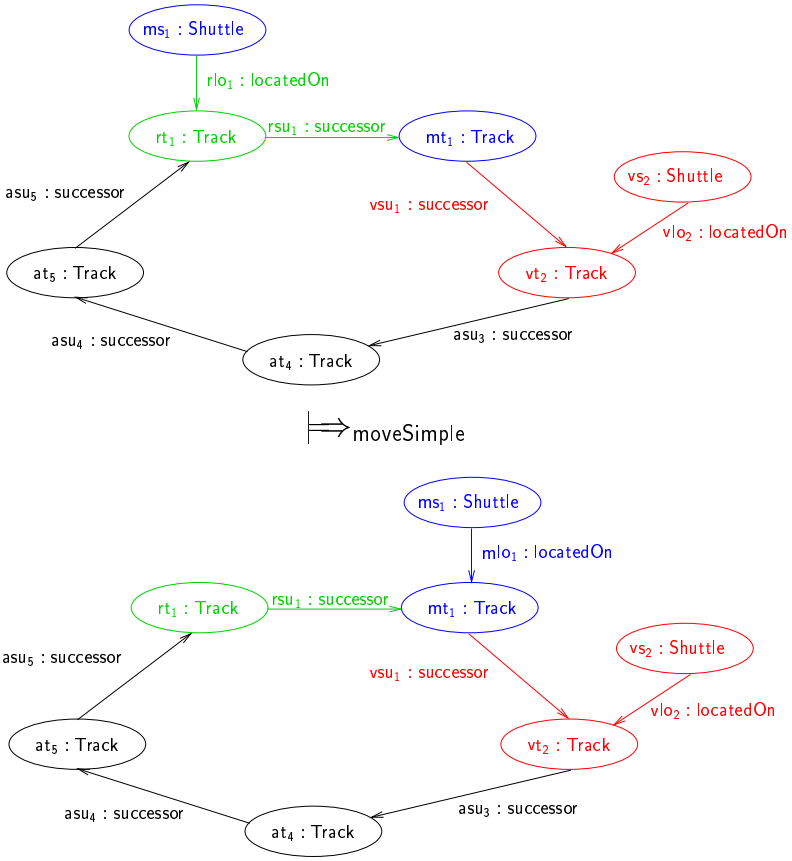



Abbildung 3.5: Die Regel moveSimple transformiert einen korrekten Graphen in einen inkorrekten Graphen

ten Zustand aus einem anderen inkorrekten Zustand. Dieser Fall braucht nicht weiter betrachtet zu werden, da nur die Fälle relevant sind, in denen eine Regel einen korrekten Zustand in einen inkorrekten überführt.

Abbildungen 3.4 und 3.5 zeigen zwei Beispiele, in denen die rechte Regelseite der Graphtransformationsregel moveSimple und das verbotene Graphmuster impendingCollisionSimplified mindestens einen gemeinsamen Knoten haben und somit die Regel das verbotene Graphmuster erzeugt haben kann. Der jeweils untere Graph entspricht dem inkorrekten Graphen, auf den die Regel rückwärts angewendet wird. Der obere Graph stellt jeweils den Graphen dar, der resultiert, wenn die Regel rückwärts angewendet wurde.

In Abbildung 3.4 haben die rechte Regelseite und das verbotene Graphmuster nur den Knoten $\mathtt{mt}_1$ gemeinsam, dieser entspricht dem Knoten $\mathtt{rt}_2$ der rechten Regelseite und dem Knoten $\mathtt{vt}_1$ des verbotenen Graphmusters. Die Knoten $\mathtt{vs}_1$ und $\mathtt{vs}_2$ entsprechen den gleichnamigen Knoten des verbotenen Graphmusters und der Knoten $\mathtt{rs}_1$ dem Knoten rs der Regel. Wird die Regel moveSimple auf diesen Graphen rückwärts angewendet, so resultiert der Graph, der im oberen Teil der Abbildung zu sehen ist. Da auch dieser Graph inkorrekt ist, impendingCollisionSimplified ist auch in ihm enthalten, hat die Regel lediglich einen inkorrekten Graphen in einen anderen

inkorrekten Graphen überführt. Eine Regel ist jedoch nur dann inkorrekt, wenn sie, angewendet auf einen korrekten Graphen, einen inkorrekten Graphen erzeugt. Deshalb kann der Fall, in dem ein inkorrekter Graph in einen anderen inkorrekten überführt wird, ignoriert werden.

Im unteren Graphen von Abbildung 3.5 haben das verbotene Graphmuster und die rechte Regelseite die beiden Knoten $mt_1$ und $ms_1$ sowie die dazwischen verlaufene locatedOn-Kante $mlo_1$ gemeinsam. Wird auf diesen Graphen die moveSimple-Regel rückwärts angewendet, so resultiert der obere der beiden in Abbildung 3.5 dargestellten Graphen. Dieser Graph enthält das verbotene Graphmuster nicht und ist somit korrekt, d.h. es wurde ein Beispiel gefunden, das zeigt, dass die Regel moveSimple einen korrekten Graphen in einen inkorrekten überführen kann.

Bisher wurden immer vollständige Graphen betrachtet, um die Korrektheit der Regel zu überprüfen. Da es jedoch unendlich viele inkorrekte Graphen geben kann, ist es nicht möglich, für jeden inkorrekten Graphen und jede Regel zu zeigen, dass die jeweilige Regel den inkorrekten Graphen nicht aus einem korrekten erzeugt haben kann. Stattdessen sollen die bereits zuvor eingeführten Graphmuster dazu verwendet werden, Mengen von Graphen zu beschreiben. Graphmuster statt vollständiger Graphen zu betrachten ist ausreichend, da die Anwendung von Graphtransformationsregeln nur Auswirkungen auf die Elemente eines Graphen haben, auf die die Elemente der Regel abgebildet werden (2.). Ein solches Graphmuster muss zum einen ein verbotenes Graphmuster enthalten und zum anderen die Elemente der rechten Regelseite. Darüber hinaus muss mindestens eines dieser Element sowohl Teil des verbotenen Graphmusters als auch der rechten Regelseite sein.

In den Abbildungen 3.4 und 3.5 entsprechen die relevanten Ausschnitte genau den Elementen, die blau, grün oder rot gekennzeichnet sind.

Ein solcher Ausschnitt kann Teil von mehreren, unter Umständen unendlich vielen, Graphen sein und kann somit dazu verwendet werden, Mengen von Graphen zu beschreiben. Diese Ausschnitte werden als Graphmuster oder genauer als Ergebnisgraphmuster bezeichnet. Da sowohl die Menge der Graphtransformationsregeln als auch die Menge der verbotenen Graphmuster endlich ist und jeder der Graphen aus endlich vielen Knoten und Kanten besteht, ist die Menge der möglichen Ergebnisgraphmuster ebenfalls endlich.

Zwei mögliche Ergebnisgraphmuster für die Regel moveSimple und das verbotene Graphmuster impendingCollisionSimplified sind jeweils im unteren Teil der Abbildungen 3.6 und 3.7 dargestellt. Zu der Menge der Graphen, die durch das Ergebnisgraphmuster aus Abbildung 3.6 beschrieben werden, gehört auch der untere Graph aus Abbildung 3.4. Der untere Graph aus Abbildung 3.5 gehört zu der Menge, die durch das Ergebnisgraphmuster in Abbildung 3.7 beschrieben wird.

Ein Ergebnisgraphmuster beschreibt eine Menge von Graphen, die alle inkorrekt sind, da sie ein verbotenes Graphmuster enthalten. Da das Ergebnisgraphmuster neben dem verbotenen Graphmuster auch die rechte Regelseite enthält und diese mindestens ein Element enthält, das auch Teil des verbotenen Graphmusters ist, kann die Anwendung der entsprechenden Regel das verbotene Graphmuster erzeugt haben. Um festzustellen, ob dies der Fall ist, wird die Regel rückwärts auf das Ergebnisgraphmuster angewendet. Daraus resultiert wieder ein Graphmuster, das Startgraphmuster. Für dieses Muster muss nun überprüft werden, ob ein beliebiges verbotenes Graphmuster darauf abbildbar ist. Ist dies nicht der Fall, so wurde ein Beispiel dafür gefunden, dass

die angewendete Regel ein korrektes Graphmuster in ein inkorrektes Graphmuster überführen kann.

Für die Graphen bedeutet dies, dass jeder korrekte Graph, auf den das Startgraphmuster abgebildet werden kann, mittels der fehlerhaften Regel in einen inkorrekten Graphen transformiert werden kann, auf den das Ergebnisgraphmuster abbildbar ist.

Der obere Teil der Abbildungen 3.6 und 3.7 zeigt die Rückwärtsanwendung der moveSimple-Regel auf die im unteren Teil gegebenen Ergebnisgraphmuster. Auf das Startgraphmuster aus Abbildung 3.6 ist das verbotene Graphmuster impendingCollisionSimplified abbildbar, während das Startgraphmuster aus Abbildung 3.7 korrekt ist.

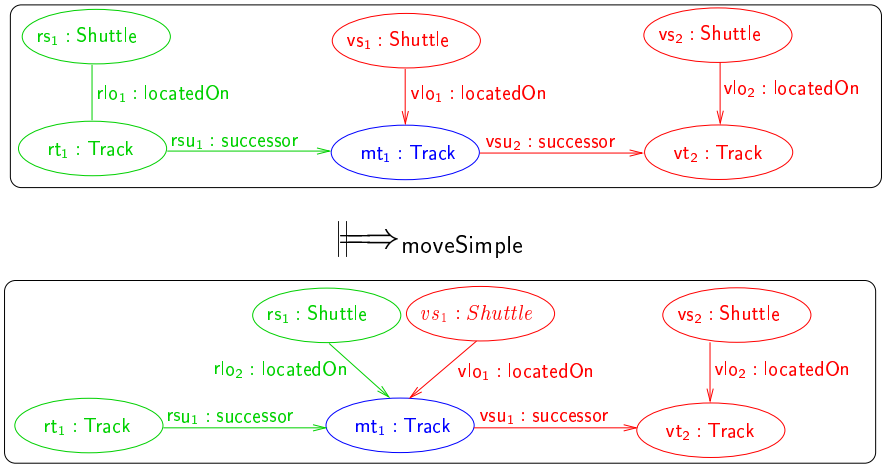



Abbildung 3.6: Start- und Ergebnisgraphmuster für die Regel moveSimple und das verbotene Graphmuster impendingCollisionSimplified, wobei das Startgraphmuster einem inkorrekten Graphmuster entspricht

Um nachzuweisen, dass ein Graphtransformationssystem korrekt ist im Bezug auf eine Menge von verbotenen Graphmustern, muss für jede der Regeln und jedes der verbotenen Graphmuster die Menge aller möglichen Ergebnisgraphmuster gebildet werden. Auf diese Ergebnisgraphmuster wird dann die entsprechende Regel rückwärts angewendet. Für jedes dabei resultierende Startgraphmuster muss geprüft werden, ob es ein verbotenes Graphmuster enthält. Gibt es ein Startgraphmuster, das kein verbotenes Graphmuster enthält, so wurde ein Beispiel gefunden, das zeigt, dass die entsprechende Regel ein korrektes Graphmuster in ein inkorrektes Graphmuster überführen kann. Da dieses Beispiel zeigt, dass eine Regel inkorrekt ist, wird es als Gegenbeispiel bezeichnet. Für Graphen bedeutet ein derartiges Gegenbeispiel, dass die Regel einen korrekten Graphen, der das Startgraphmuster enthält, durch die Anwendung der betrachteten Regel in einen inkorrekten Graphen überführt werden kann, der das Ergebnisgraphmuster enthält.

Wurde eine solche inkorrekte Regel gefunden, so muss sie in einem Graphtransformationssystem mit gegebenen Initialgraphen jedoch nicht zwangsläufig einen inkorrekten Zustand erzeugen. Es kann sein, dass es im System keinen erreichbaren Graphen gibt, der das entsprechende Startgraphmuster enthält. Um festzustellen, ob die Regel in einem Graphtransformationssystem

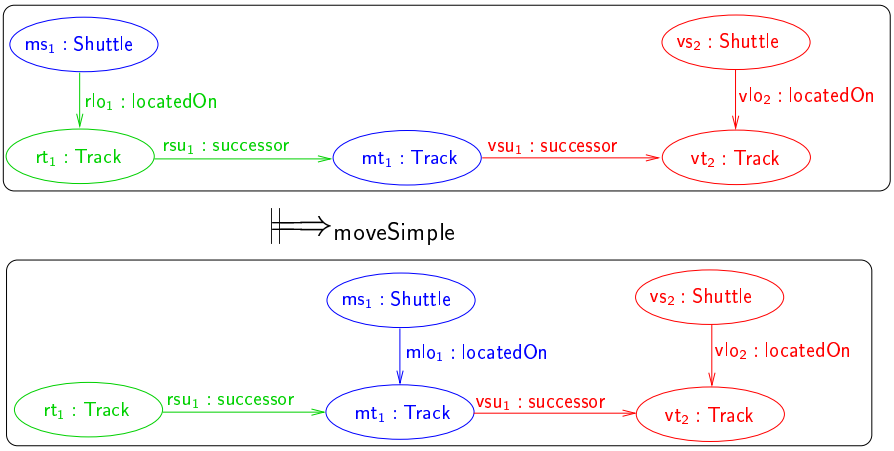



Abbildung 3.7: Start- und Ergebnisgraphmuster für die Regel moveSimple und das verbotene Graphmuster impendingCollisionSimplified, wobei das Startgraphmuster einem korrekten Graphen entspricht

einen inkorrekten Graphen erzeugt, wäre eine Erreichbarkeitsanalyse erforderlich, die jedoch aus den in Abschnitt 3.1.3 genannten Gründen nicht immer möglich ist. In diesem Fall muss die Regel so abgeändert werden, dass ihre Anwendung das verbotene Graphmuster nicht erzeugen kann. Eine solche Korrektur kann beispielsweise mittels des von Heckel und Wagner vorgestellten Ansatzes vorgenommen werden (siehe [HW95] und Abschnitt 6.1).

### 3.1.5 Erweiterung der Idee um negative Anwendungsbedingungen

Wie im vorangegangenen Abschnitt gezeigt wurde, kann die Regel moveSimple zu einer kritischen Situation und im schlechtesten Fall auch zu einem Unfall führen. Ein Unfall tritt dann ein, wenn die Regel auf ein Shuttle angewendet wird, obwohl auf dem Track, auf den das Shuttle als nächstes fährt, bereits ein anderes Shuttle ist. Zu einer kritischen Situation führt die Regel, wenn sich auf dem übernächsten Track ein anderes Shuttle befindet. Um zu vermeiden, dass die Regel in einem der beiden Fälle angewendet wird, wird eine negative Anwendungsbedingung verwendet.

Die negative Anwendungsbedingung wird zur linken Regelseite hinzugefügt. Die derart erweiterte Regel sieht dann folgendermaßen aus $[L, \hat{\mathcal{L}}] \rightarrow_r R$, wobei $\hat{\mathcal{L}}$ die Menge der Graphen der negativen Anwendungsbedingung ist. Während die linke Regelseite $L$ beschreibt, wann die Regel angewendet werden kann, beschreibt die negative Anwendungsbedingung $\hat{\mathcal{L}}$, wann die Regel nicht angewendet werden darf.

Eine negative Anwendungsbedingung besteht aus einer Menge von Graphen. Jeder dieser Graphen enthält den Graphen der linken Regelseite, erweitert ihn aber um mindestens einen Knoten oder eine Kante. Ein solcher Graph wird auch als Constraint der negativen Anwendungsbedingung bezeichnet. Für jeden Knoten eines Constraints gilt, dass er entweder auch zur Anwendungsbedingung gehört oder von ihm ausgehend über endlich viele Kanten ein Knoten aus der Anwendungsbedingung erreicht werden kann.

Ein Beispiel für eine Graphtransformationsregel mit negativer Anwendungsbedingung ist in Abbildung 3.8 dargestellt. Die Regel stellt eine Erweiterung der Regel moveSimple dar und heißt moveSimpleNAC. Sie beschreibt, dass ein Shuttle nur dann auf den nächsten Track fahren darf, wenn sich weder auf dem nächsten, noch auf dem übernächsten Track ein anderes Shuttle befindet. Diese Regel entspricht dem Story Pattern aus Abbildung 2.13. Der Graph $\hat{L}_1$ verhindert die Weiterfahrt des Shuttles, wenn sich auf dem nächsten Track ein anderes Shuttle befindet. $\hat{L}_2$ verhindert die Weiterfahrt, wenn sich auf dem übernächsten Track ein anderes Shuttle befindet. Durch diese Erweiterung der Regel soll zum einen eine Kollision von zwei Shuttles als auch die kritische Situation impendingCollision bzw. impendingCollisionSimplified verhindert werden.

Besteht die negative Anwendungsbedingung, wie im Beispiel, aus mehreren Graphen, so darf die Regel nur dann angewendet werden, wenn die Anwendungsbedingung auf den Anwendungsgraphen abgebildet werden kann, dies jedoch für keinen der Graphen aus der negativen Anwendungsbedingung möglich ist.

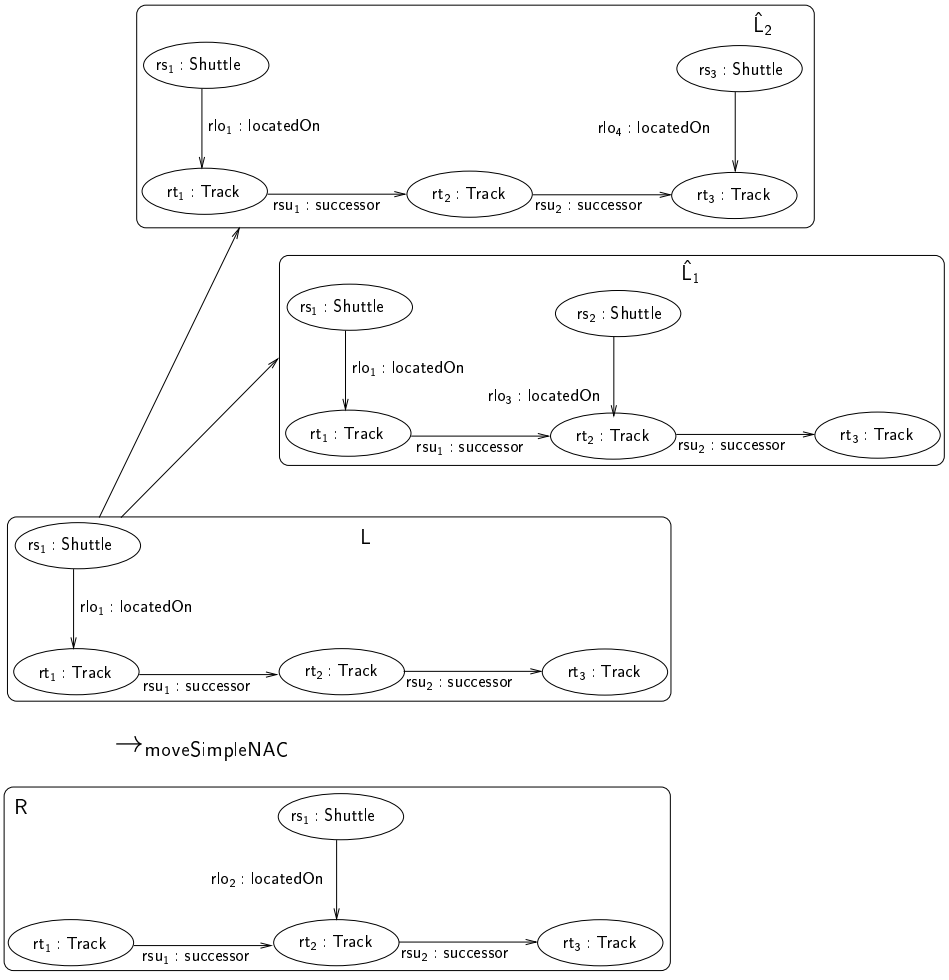



Abbildung 3.8: Graphtransformationsregel moveSimpleNAC mit negativer Anwendungsbedingung

Soll eine Regel mit negativer Anwendungsbedingung rückwärts angewendet werden, so muss die negative Anwendungsbedingung zunächst angepasst werden, da anderenfalls die Nachbedingung der Regel eine negative Anwendungsbedingung hätte, wofür jedoch keine Semantik definiert ist. Die Anwendung einer Regel in Rückwärtsrichtung entspricht dem Rückgängigmachen ihrer Anwendung in Vorwärtsrichtung. Wird die negative Anwendungsbedingung nicht transformiert, so kann bei der Rückwärtsanwendung einer Regel ein Graph resultieren, auf den die Regel in Vorwärtsrichtung auf Grund der negativen Anwendungsbedingung nicht anwendbar ist.

Um die negative Anwendungsbedingung einer Regel $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_r \mathsf{R}$ anzupassen, wird sie zunächst aus der linken Regelseite entfernt. Dann wird die derart reduzierte Regel $\mathsf{L} \rightarrow_{r'} \mathsf{R}$ auf alle Graphen aus der negativen Anwendungsbedingung, also auf alle Graphen aus $\hat{\mathcal{L}}$, angewendet. Da die Graphen der negativen Anwendungsbedingung immer den Graphen enthalten, der die linke Regelseite beschreibt, existiert auch immer eine Anwendungsstelle für die Regel. Die resultierenden Graphen werden als negative Anwendungsbedingung zur rechten Regelseite hinzugefügt. Die konvertierte Regel hat dann die Form $[\mathsf{L}^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{r^{-1}} \mathsf{R}^{-1}$.

Abbildung 3.9 zeigt die modifizierte Regel aus Abbildung 3.8. Dabei entspricht $\mathsf{L}^{-1}$ der ursprünglichen rechten Regelseite $\mathsf{R}$ und $\hat{\mathsf{L}}_1^{-1}$ und $\hat{\mathsf{L}}_2^{-1}$ stellen die transformierte negative Anwendungsbedingung dar. $\mathsf{R}^{-1}$ entspricht der ursprünglichen linken Regelseite $\mathsf{L}$.

Neben den Anwendungsbedingungen der Graphtransformationsregeln können auch die verbotenen Graphmuster um negative Anwendungsbedingungen erweitert werden.

Ein Beispiel für ein verbotenes Graphmuster mit negativer Anwendungsbedingung ist in Abbildung 3.10 gegeben. Dieses verbotene Graphmuster beschreibt eine kritische Situation, bei der zwei Shuttles auf benachbarten Tracks fahren, jedoch das DistanceCoordination-Koordinationsmuster nicht ausführen; in diesem Fall droht eine Kollision der beiden Shuttles.

Eine solche kritische Situation ist eingetreten, d.h. ein Anwendungsgraph ist inkorrekt im Bezug auf dieses verbotene Graphmuster, wenn die Anwendungsbedingung des verbotenen Graphmusters auf ihn abgebildet werden kann, dies jedoch für keinen Graphen der negativen Anwendungsbedingung gilt.

Abbildung 3.11(a) zeigt einen inkorrekten Graphen, denn die Anwendungsbedingung der kritischen Situation kann auf diesen Graphen abgebildet werden (diese Abbildung ist rot dargestellt), der Graph der negativen Anwendungsbedingung aber nicht, da zwischen dem Shuttle $\mathsf{vs}_1$ und dem Shuttle $\mathsf{vs}_2$ kein DistanceCoordination-Muster existiert. Im Gegensatz dazu ist der Graph in Abbildung 3.11(b) korrekt. Zwar kann auch auf diesen Graphen die Anwendungsbedingung des verbotenen Graphmusters abgebildet werden, jedoch koordinieren sich die beiden Shuttles über das DistanceCoordination-Muster, d.h. auch der Knoten, der das DistanceCoordination-Muster im Graphen der negativen Anwendungsbedingung darstellt, kann auf den Anwendungsgraphen abgebildet werden.

Enthält das verbotene Graphmuster eine negative Anwendungsbedingung, so kann die Anwendung einer Graphtransformationsregel auf zwei Arten einen korrekten Graphen in einen inkorrekten überführen. Zum einen kann sie ein Element erzeugen, auf das ein Teil der Anwendungsbedingung des verbotenen Graphmusters abbildbar ist. Dieser Fall ist identisch dazu, dass eine Regel ohne negative Anwendungsbedingung ein verbotenes Graphmuster erzeugt. Der zweite Fall besteht darin, dass auf den Anwendungsgraph die Anwendungsbedingung eines verbotenen Graphmusters abgebildet werden kann und dies ebenfalls für einen Graphen der negativen

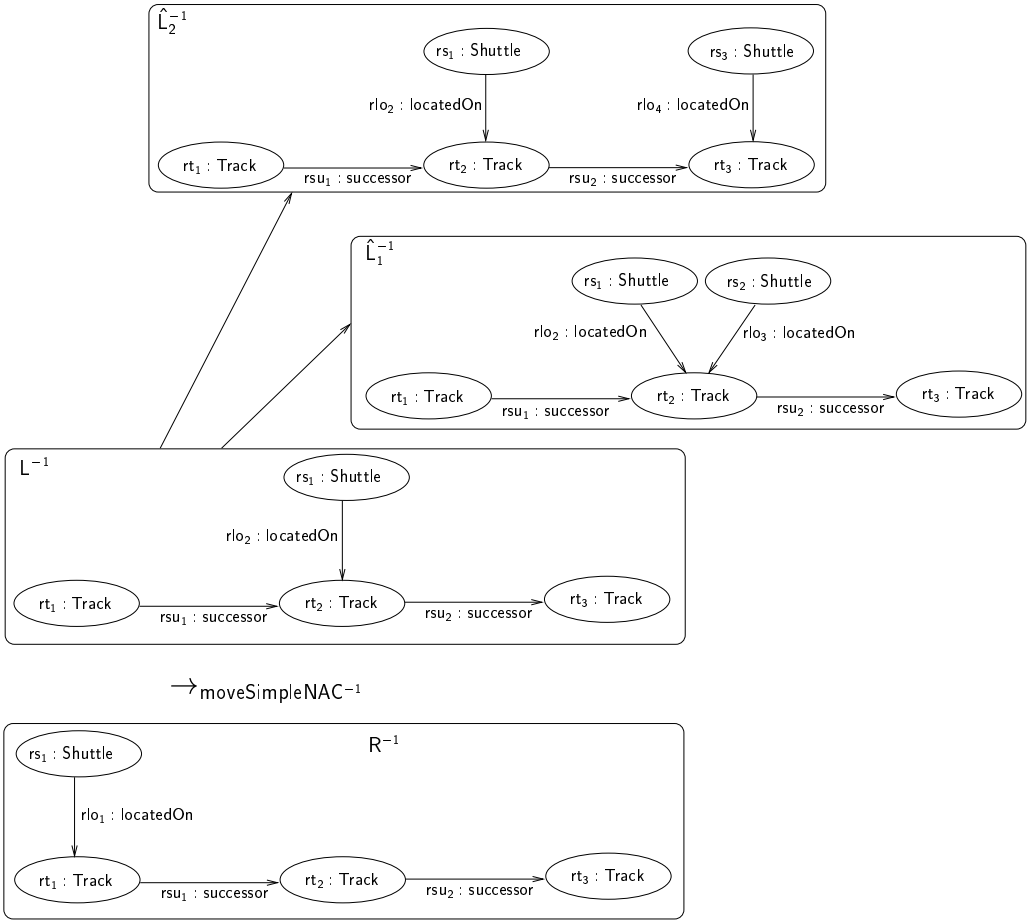



Abbildung 3.9: Konvertierung der Regel moveSimpleNAC in die Regel moveSimpleNAC$^{-1}$, sodass sie rückwärts angewendet werden kann

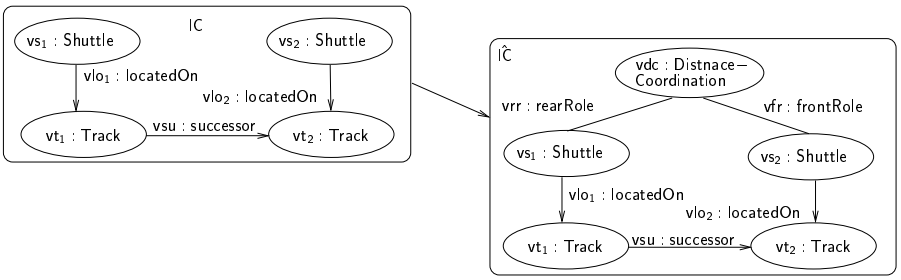
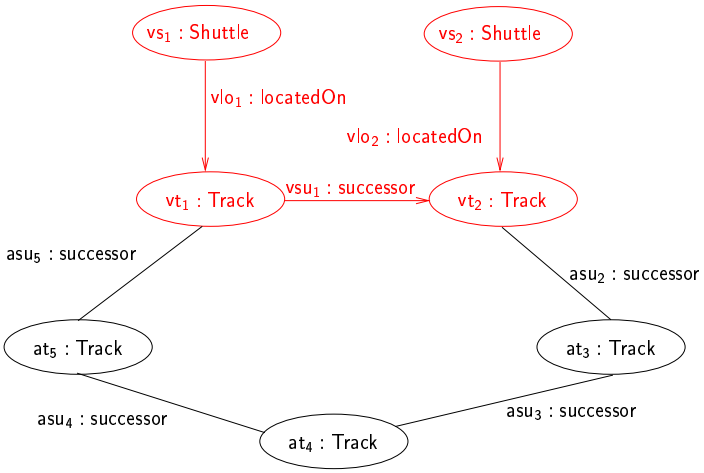
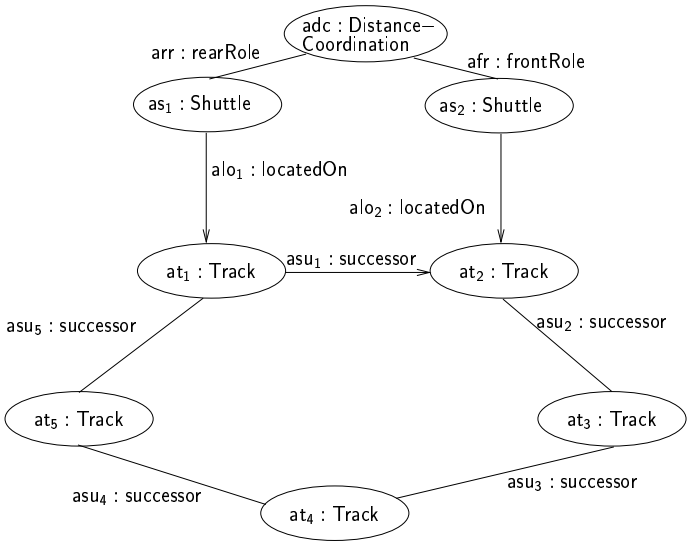



Abbildung 3.10: Verbotenes Graphmuster impendingCollision mit negativer Anwendungsbedingung

Anwendungsbedingung gilt. Wird nun durch die Regelanwendung ein Element gelöscht, auf das ein Element des Graphen der negativen Anwendungsbedingung, jedoch nicht der Anwendungsbedingung, abgebildet werden kann, so ist das verbotene Graphmuster Teil des resultierenden Graphen und dieser ist somit inkorrekt.




(a) Graph, der in Bezug auf das verbotene Graphmuster impendingCollision inkorrekt ist




(b) Graph, der in Bezug auf das verbotene Graphmuster impendingCollision korrekt ist

Abbildung 3.11: Graphen, die in Bezug auf das verbotene Graphmuster impendingCollision inkorrekt bzw. korrekt sind

Die Anwendung der Regel deleteDCSwitch, in Abbildung 3.12 dargestellt, erzeugt auf diese Weise aus einem korrekten Graphen einen inkorrekten. Sie löscht eine Instanz des DistanceCoordination-Musters zwischen zwei Shuttles, falls das vordere Shuttle an einer Weiche einen anderen Weg fährt, als das hintere Shuttle fahren möchte (dargestellt durch die next-Kante). Der Graph

bzw. das Graphmuster auf der linken Regelseite ist korrekt, die Anwendung der Regel löscht jedoch das DistanceCoordination-Muster und erzeugt so einen inkorrekten Graphen bzw. ein inkorrektes Graphmuster, dargestellt im unteren Teil der Abbildung.

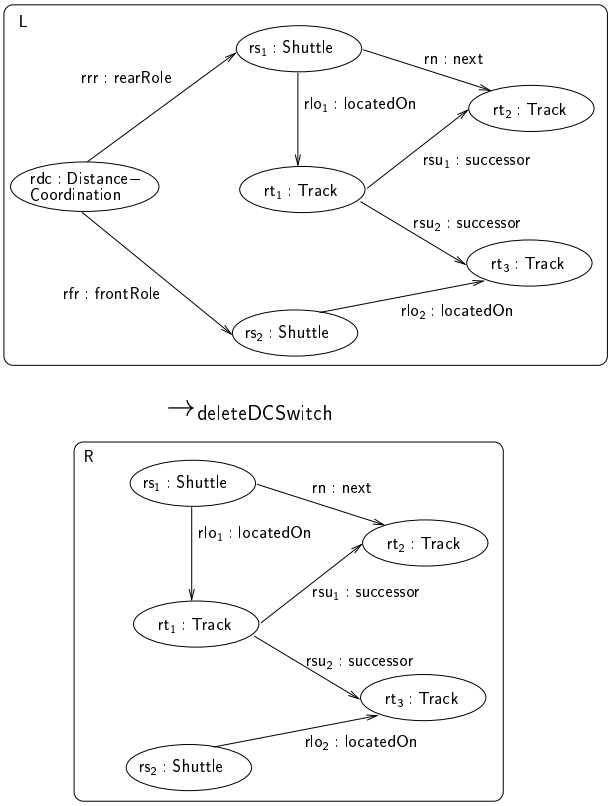



Abbildung 3.12: Regel deleteDCSwitch, die eine Instanz des DistanceCoordination-Musters löscht

Für diesen zweiten Fall muss eine zusätzliche Überprüfung eingefügt werden. Die Graphen der negativen Anwendungsbedingung des verbotenen Musters haben die Eigenschaft, dass jeder ihrer Knoten entweder auch zur Anwendungsbedingung gehört oder ausgehend von diesem Knoten über eine endliche Menge von Kanten ein Knoten der Anwendungsbedingung erreicht werden kann.

Im obigen Beispiel wird der Knoten rdc : DistanceCoordination gelöscht. Durch das Löschen von rdc kann der Graph $\hat{IC}$ der negativen Anwendungsbedingung des verbotenen Musters impendingCollision nicht mehr auf den resultierenden Graphen abgebildet werden. Der Knoten rdc : DistanceCoordination besitzt zwei Kanten; eine beschriftet mit rrr : rearRole zum Knoten $rs_1$ : Shuttle und eine zweite beschriftet mit rfr : frontRole zum Knoten $rs_2$ : Shuttle. Auf diese beiden Knoten können die Shuttle-Knoten der Anwendungsbedingung des verbotenen Musters abgebildet werden.

Das bedeutet, wenn ein Knoten gelöscht werden soll, auf den ein Knoten eines Graphen der negativen Anwendungsbedingung abgebildet werden kann, aber keines der Anwendungsbedingung, dann muss dieser Knoten mindestens eine inzidente Kante haben, die ebenfalls gelöscht

werden muss. Um eine Kante löschen zu können, muss sie mit beiden zu ihr inzidenten Knoten in der linken Regelseite enthalten sein. Wird der zweite inzidente Knoten ebenfalls gelöscht, so gilt auch hier wieder, dass alle inzidenten Kanten durch die Regel entfernt werden müssen. Es muss allerdings eine Kante geben, die inzident zu einem Knoten ist, der explizit durch die Regel erhalten bleibt, da die Anwendungsbedingung des verbotenen Musters durch die Regelanwendung nicht verändert wird.

Im obigen Beispiel hat der zu löschende Knoten rdc zwei inzidente Kanten, rrr und rfr. Sowohl der zweite zu rrr inzidente Knoten $rs_1$ als auch der zweite zu rfr inzidente Knoten $rs_2$ werden durch die Regel explizit erhalten. Daher sind beide Knoten Teil sowohl der linken als auch der rechten Regelseite, sie sind aber auch Teil des verbotenen Graphmusters.

Diese Tatsache kann genutzt werden, um auch für das Löschen von Elementen Ergebnisgraphmuster zu bilden. Ein solches Ergebnisgraphmuster enthält mindestens einen Knoten, der in beiden Regelseiten und in einem Graphen der negativen Anwendungsbedingung des verbotenen Graphmusters enthalten ist.

Abbildung 3.13 zeigt ein mögliches Ergebnisgraphmuster. Das daraus erzeugte Startgraphmuster für das verbotene Graphmuster aus Abbildung 3.10 und die Graphtransformationsregel deleteDCSwitch ist in Abbildung 3.14 abgebildet. Die beiden Graphmuster stellen ein Gegenbeispiel dar, das zeigt, dass die Regel deleteDCSwitch einen korrekten Graphen in einen inkorrekten überführen kann. Die Anwendung der Regel führt genau dann dazu, dass ein korrekter Graph in einen inkorrekten überführt wird, wenn der korrekte Graph, das Startgraphmuster des Gegenbeispiels enthält und die entsprechende Regel auf diesen Teil des Graphen angewendet wird.

## 3.2 Graphtransformationssysteme

In diesem und den folgenden Abschnitten soll die zuvor informal dargestellte Idee formalisiert werden.

Der Zustand der Software eines mechatronischen Systems ist charakterisiert durch die existierenden Objekte und deren Links. Eine solche Instanzsituation wird in einem Objektdiagramm festgehalten. In diesem Kapitel werden statt der Objektdiagramme Graphen verwendet, sodass ein Systemzustand durch diesen Graphen repräsentiert wird. Die Übergänge zwischen den Zuständen werden durch Graphtransformationsregeln beschrieben.

Im Folgenden wird als erstes ein Überblick über die grundlegende Terminologie gegeben. Ein Teil der aufgeführten Definitionen wurden aus anderen Texten, zumeist dem „Handbook of Graphgrammars“[Roz97], übernommen. Dazu gehören die Definitionen von (beschrifteten) Graphen (Definitionen 1 und 2), Graphhomomorphismen und Graphisomorphismen (Definitionen 9 und 10), die Definition eines Matches (Definition 13) und die Definition eines Graphtransformationssystems (Definition 17).

Weitere Definitionen wie die der Graphtransformationsregeln (Definition 12), der direkten Transformation (Definition 14), der negativen Anwendungsbedingung (Definition 15) und des Double Pushout Ansatzes (Definitition 26) wurden im Vergleich zu den Definitionen in der Literatur vereinfacht bzw. die in den Definitionen verwendeten Graphen und Funktionen unterliegen stärkeren Einschränkungen. Durch die Vereinfachungen und Einschränkungen wird der Verifi-


Abbildung 3.13: Ergebnisgraphmuster mit negativer Anwendungsbedingung

kationsansatz in Abschnitt 3.6 ermöglicht. Wurde eine Definition aus der Literatur übernommen und für die vorliegende Arbeit angepasst, so wird im Folgenden auch der Unterschied zur ursprünglichen Definition genannt.

Für den Verifikationsansatz in Abschnitt 3.6 ist es wichtig, dass eine Graphtransformationsregel auch rückwärts angewendet werden kann. Wird zur Anwendung einer Graphtransformationsregel der Double Pushout Ansatz angewendet, so ist dies immer möglich (siehe [Roz97] Kapitel 3). In dieser Arbeit wird zwar eine eingeschränkte Form des Double Pushout Ansatzes verwendet, jedoch sind für die linke Regelseite auch negative Anwendungsbedingungen erlaubt. Die Modifikation einer Regel, um diese korrekt rückwärts anwenden zu können, wird in dieser Arbeit in Definition 29 gegeben.

Um bei der Verifikation Aussagen über Mengen von Graphen machen zu können und strukturelle Eigenschaften von Graphen beschreiben zu können, werden in dieser Arbeit Graphmuster eingeführt (Definition 19) und die Anwendung von Graphtransformationsregeln auf diese Muster definiert (Definition 30). Die Graphmuster sind ähnlich zu den Eigenschaftsgraphen von Rensink (siehe dazu auch Abschnitt 3.6) und den Konsistenzbedingungen, die zum Beispiel von Heckel und Wagner in [HW95] verwendet werden (ein Vergleich zwischen Graphmustern und Konsistenztbedingungen wird in Abschnitt 6.1 gegeben). Allerdings werden weder Eigenschaftsgraphen noch Konsistenzbedingungen dazu verwendet, um Mengen von Graphen zu beschreiben.

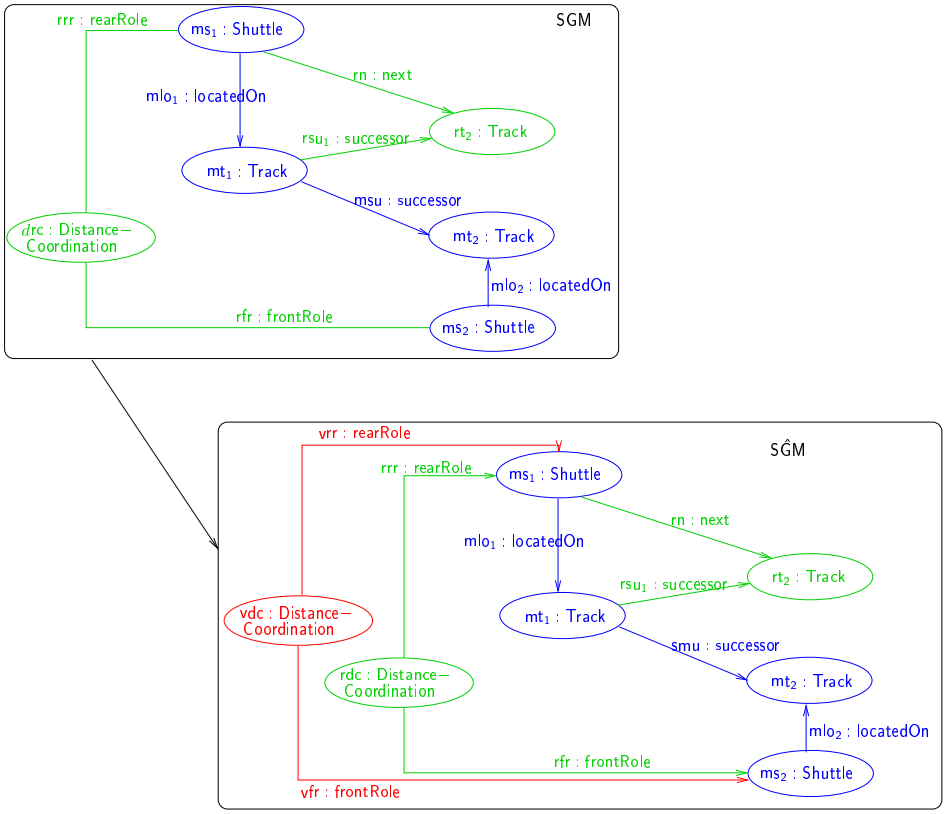



Abbildung 3.14: Startgraphmuster mit negativer Anwendungsbedingung

Abschnitt 3.2.1 gibt eine allgemeine Definition für Graphen und einen Überblick über die wichtigsten Operationen auf Graphen. Graphtransformationsregeln werden in Abschnitt 3.2.2 erläutert. In Abschnitt 3.3 werden Graphmuster eingeführt, mit denen Mengen von Graphen beschrieben werden können. Mit den eingeführten Konstrukten kann dann das Systemmodell beschrieben werden (Abschnitt 3.4). Zur Verifikation ist es notwendig, die Graphtransformationsregeln auch rückwärts und auf Graphmuster anwenden zu können. Deshalb erfolgt in Abschnitt 3.5 eine Erweiterung der Graphtransformationen. Abschnitt 3.6 beschäftigt sich mit strukturellen Eigenschaften eines Graphen.

Die Beweise der hier aufgeführten Lemmata und Theoreme sind im Anhang B skizziert.

### 3.2.1 Graphen und grundlegende Operationen darauf

Nach Rozenberg [Roz97] besteht ein Graph aus endlich vielen Knoten, die durch Kanten verbunden werden. Jede Kante hat einen Start- und einen Zielknoten. Durch die Festlegung des Start- bzw. Zielknotens wird auch gleichzeitig die Richtung einer Kante festgelegt. Formal ist ein solcher Graph definiert als:

**Definition 1.** *(Graph)*
*Ein Graph ist ein Tupel* $\mathsf{G} = (\mathsf{N}, \mathsf{E}, \mathsf{src}, \mathsf{tgt})$*, wobei*

- $\mathsf{N}$ *eine endliche Knotenmenge ist,*
- $\mathsf{E}$ *eine endliche Kantenmenge ist,*
- $\mathsf{src} : \mathsf{E} \mapsto \mathsf{N}$ *eine Funktion ist, die jeder Kante einen Startknoten zuweist und*
- $\mathsf{tgt} : \mathsf{E} \mapsto \mathsf{N}$ *eine Funktion ist, die jeder Kante einen Zielknoten zuweist.*

Um die Knoten und Kanten benennen und unterscheidbar machen zu können, werden zwei Alphabete eingeführt. Eines der Alphabete legt mögliche Beschriftungen für Knoten fest und das andere Beschriftungen für Kanten. Mittels zweier Beschriftungsfunktionen (engl. labeling function) können den Knoten und Kanten dann Beschriftungen zugewiesen werden. Die formale Definition für einen solchen beschrifteten Graphen entspricht der aus [Roz97] und lautet

**Definition 2.** *(Beschrifteter Graph)*
*Gegeben sind zwei Alphabete* $\Omega_\mathsf{N}$ *und* $\Omega_\mathsf{E}$*, für Knoten- bzw. Kantenbeschriftungen. Ein beschrifteter Graph (engl. labeled graph) ist ein Tupel* $\mathsf{G} = (\mathsf{N}, \mathsf{E}, \mathsf{src}, \mathsf{tgt}, \mathsf{l}_\mathsf{N}, \mathsf{l}_\mathsf{E})$*, mit*

- $(\mathsf{N}, \mathsf{E}, \mathsf{src}, \mathsf{tgt})$ *ist ein Graph wie in Definition 1,*
- $\mathsf{l}_\mathsf{N} : \mathsf{N} \mapsto \Omega_\mathsf{N}$ *ist eine Funktion zur Beschriftung von Knoten und*
- $\mathsf{l}_\mathsf{E} : \mathsf{E} \mapsto \Omega_\mathsf{E}$ *ist eine Funktion zur Beschriftung von Kanten.*

Dabei sind $\mathsf{N_G}$, $\mathsf{E_G}$, $\mathsf{src_G}$ und $\mathsf{tgt_G}$ wie in Definition 1 definiert. Die Funktionen $\mathsf{l}_{\mathsf{N_G}}$ und $\mathsf{l}_{\mathsf{E_G}}$ stellen knoten- bzw. kantenbeschriftende Funktionen dar. $\mathsf{G}_\emptyset$ ist der leere Graph für den gilt $\mathsf{N}_{\mathsf{G}_\emptyset} = \mathsf{E}_{\mathsf{G}_\emptyset} = \emptyset$. $\mathcal{G}$ ist die Menge aller Graphen.

In Abbildung 3.15 ist ein Beispiel für einen beschrifteten Graphen $\mathsf{G} := (\mathsf{N},\mathsf{E},\mathsf{src},\mathsf{tgt},\mathsf{l}_\mathsf{N},\mathsf{l}_\mathsf{E})$ gegeben[1]. Die vier Knoten $\mathsf{an}_1,\ldots,\mathsf{an}_4$ bilden die Knotenmenge $\mathsf{N} := \{\mathsf{an}_1,\mathsf{an}_2,\mathsf{an}_3,\mathsf{an}_4\}$, $\mathsf{ae}_1,\ldots,\mathsf{ae}_4$ bilden die Kantenmenge $\mathsf{E} := \{\mathsf{ae}_1,\mathsf{ae}_2,\mathsf{ae}_3,\mathsf{ae}_4\}$. Die Funktion src weist jeder der Kanten aus $\mathsf{E}$ einen Startknoten zu. Beispielsweise ist $\mathsf{an}_1$ der Startknoten von Kante $\mathsf{ae}_1$, src ist gegeben durch $\mathsf{src} := \{(\mathsf{ae}_1 \mapsto \mathsf{an}_1),(\mathsf{ae}_2 \mapsto \mathsf{an}_2),(\mathsf{ae}_3 \mapsto \mathsf{an}_3),(\mathsf{ae}_4 \mapsto \mathsf{an}_4)\}$. Dementsprechend weist tgt den Kanten einen Zielknoten zu $\mathsf{tgt} := \{(\mathsf{ae}_1 \mapsto \mathsf{an}_2),(\mathsf{ae}_2 \mapsto \mathsf{an}_3),(\mathsf{ae}_3 \mapsto \mathsf{an}_4),(\mathsf{ae}_4 \mapsto \mathsf{an}_2)\}$. Zur Beschriftung der Knoten steht das Alphabet $\Omega_\mathsf{N} := \{\mathsf{Shuttle},\mathsf{Track}\}$ zur Verfügung, die Kanten können mit $\Omega_\mathsf{E} := \{\mathsf{locatedOn},\mathsf{successor}\}$ beschriftet werden. Die Beschriftungsfunktion $\mathsf{l}_\mathsf{N}$ weist jedem Knoten aus der Menge $\mathsf{N}$ eine Beschriftung aus $\Omega_\mathsf{N}$ zu, $\mathsf{l}_\mathsf{N} := \{(\mathsf{an}_1 \mapsto \mathsf{Shuttle}),(\mathsf{an}_2 \mapsto \mathsf{Track}),(\mathsf{an}_3 \mapsto \mathsf{Track}),(\mathsf{an}_4 \mapsto \mathsf{Track})\}$. Genauso weist auch die Funktion $\mathsf{l}_\mathsf{E}$ jeder Kante aus $\mathsf{E}$ eine Beschriftung aus $\Omega_\mathsf{E}$ zu, $\mathsf{l}_\mathsf{E} := \{(\mathsf{ae}_1 \mapsto \mathsf{locatedOn}),(\mathsf{ae}_2 \mapsto \mathsf{successor}),(\mathsf{ae}_3 \mapsto \mathsf{successor}),(\mathsf{ae}_4 \mapsto \mathsf{successor})\}$.

[1] Um die Knoten und Kanten des Graphen referenzieren zu können sind sie mit $\mathsf{an}_1,\ldots,\mathsf{an}_4$ bzw. $\mathsf{ae}_1,\ldots,\mathsf{ae}_4$ nummeriert.

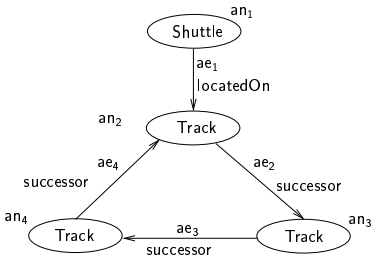



Abbildung 3.15: Beispiel für einen beschrifteten Graphen

### Teilgraphbeziehungen und Graphoperationen

Für zwei gegebene Graphen $G_1$ und $G_2$ kann bestimmt werden, ob die beiden Graphen identisch sind oder einer der beiden Graphen Teilgraph des anderen ist. Die folgenden Definitionen sind jeweils für beschriftete Graphen angegeben; bei unbeschrifteten Graphen können die für die Beschriftungsfunktionen angegebenen Bedingungen ignoriert werden.

Die beiden Graphen $G_1$ und $G_2$ sind identisch, wenn ihre Knoten- und Kantenmengen identisch sind, gleiche Kanten auch die gleichen Start- und Zielknoten haben und die Funktionen zur Knoten- und Kantenbeschriftung gleichen Knoten bzw. Kanten die gleichen Beschriftungen zuweisen.

**Definition 3.** *(Identische Graphen)*
*Zwei Graphen* $G_1, G_2 \in \mathcal{G}$ *sind identisch, wenn gilt:*

- $N_{G_1} = N_{G_2}$,
- $E_{G_1} = E_{G_2}$,
- $src_{G_1} = src_{G_2}$,
- $tgt_{G_1} = tgt_{G_2}$,
- $l_{N_{G_1}} = l_{N_{G_2}}$ *und*
- $l_{E_{G_1}} = l_{E_{G_2}}$.

Der Graph $G_1$ ist ein Teilgraph von $G_2$, wenn die Knoten- und Kantenmengen von $G_1$ Teilmengen der Mengen von $G_2$ sind und die Funktionen zur Bestimmung des Start- und Zielknoten einer Kante sowie die beiden Beschriftungsfunktionen eingeschränkt auf die Knoten und Kanten aus $G_1$ identisch sind.

**Definition 4.** *(Teilgraph)*
*Für zwei Graphen* $TG, G \in \mathcal{G}$ *gilt, dass* $TG$ *ein Teilgraph von* $G$ *ist, geschrieben* $TG \leq G$, *falls gilt*

- $N_{TG} \subseteq N_G$,
- $E_{TG} \subseteq E_G$,
- $src_{TG} = src_G \mid_{E_{TG}}$,
- $tgt_{TG} = tgt_G \mid_{E_{TG}}$,

- $\mathsf{l}_{\mathsf{N}_{\mathsf{TG}}} = \mathsf{l}_{\mathsf{N}_{\mathsf{G}}} |_{\mathsf{N}_{\mathsf{TG}}}$ *und*
- $\mathsf{l}_{\mathsf{E}_{\mathsf{TG}}} = \mathsf{l}_{\mathsf{E}_{\mathsf{G}}} |_{\mathsf{E}_{\mathsf{TG}}}$.

Für zwei Graphen $\mathsf{TG}, \mathsf{G} \in \mathcal{G}$ gilt, dass $\mathsf{TG}$ ein echter Teilgraph von $\mathsf{G}$ ist (geschrieben $\mathsf{TG} < \mathsf{G}$), wenn $\mathsf{G}$ mindestens einen Knoten oder eine Kante mehr besitzt als $\mathsf{TG}$, d.h. $\mathsf{TG} \leq \mathsf{G} \wedge (|\mathsf{N}_{\mathsf{TG}}| < |\mathsf{N}_{\mathsf{G}}| \vee |\mathsf{E}_{\mathsf{TG}}| < |\mathsf{E}_{\mathsf{G}}|)$.

Für zwei gegebene Graphen kann nicht nur bestimmt werden, ob sie identisch sind oder in einer Teilgraphbeziehung zueinander stehen, sondern sie können auch dazu verwendet werden, einen neuen Graphen zu beschreiben. Dazu ist es jedoch erforderlich, dass für die beiden Graphen dieselben Alphabete zur Beschriftung der Knoten und der Kanten verwendet werden. In diesem Fall werden die beiden Graphen als beschriftungskompatibel bezeichnet.

Die Vereinigung bietet eine Möglichkeit, aus zwei beschriftungskompatiblen Graphen $\mathsf{G}_1$ und $\mathsf{G}_2$ einen neuen Graphen zu erzeugen. Dazu werden die Knoten- und Kantenmengen zu einer Knoten- bzw. Kantenmenge vereinigt. Die Funktionen src, tgt, $\mathsf{l}_{\mathsf{N}}$ und $\mathsf{l}_{\mathsf{E}}$ des neuen Graphen entsprechen für Knoten und Kanten, die in $\mathsf{G}_1$ enthalten sind, den Funktionen von $\mathsf{G}_1$ und für alle Knoten und Kanten, die nur in $\mathsf{G}_2$, nicht aber in $\mathsf{G}_1$ auftreten, den Funktionen aus $\mathsf{G}_2$.

**Definition 5.** *(Vereinigung von Graphen)*
*Die Vereinigung von zwei beschriftungskompatiblen Graphen* $\mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}$ *ist definiert als* $\mathsf{G}_1 \cup \mathsf{G}_2 := (\mathsf{N}', \mathsf{E}', \mathsf{src}', \mathsf{tgt}', \mathsf{l}'_{\mathsf{N}}, \mathsf{l}'_{\mathsf{E}})$ *mit*

- $\mathsf{N}' := \mathsf{N}_{\mathsf{G}_1} \cup \mathsf{N}_{\mathsf{G}_2}$,
- $\mathsf{E}' := \mathsf{E}_{\mathsf{G}_1} \cup \mathsf{E}_{\mathsf{G}_2}$,
- $\mathsf{src}' := \mathsf{src}_{\mathsf{G}_1} \oplus \mathsf{src}_{\mathsf{G}_2} |_{(\mathsf{E}_{\mathsf{G}_2} \setminus \mathsf{E}_{\mathsf{G}_1})}$,
- $\mathsf{tgt}' := \mathsf{tgt}_{\mathsf{G}_1} \oplus \mathsf{tgt}_{\mathsf{G}_2} |_{(\mathsf{E}_{\mathsf{G}_2} \setminus \mathsf{E}_{\mathsf{G}_1})}$,
- $\mathsf{l}'_{\mathsf{N}} := \mathsf{l}_{\mathsf{N}_{\mathsf{G}_1}} \oplus \mathsf{l}_{\mathsf{N}_{\mathsf{G}_2}} |_{(\mathsf{N}_{\mathsf{G}_2} \setminus \mathsf{N}_{\mathsf{G}_1})}$ *und*
- $\mathsf{l}'_{\mathsf{E}} := \mathsf{l}_{\mathsf{E}_{\mathsf{G}_1}} \oplus \mathsf{l}_{\mathsf{E}_{\mathsf{G}_2}} |_{(\mathsf{E}_{\mathsf{G}_2} \setminus \mathsf{E}_{\mathsf{G}_1})}$.

*Es gilt* $(\mathsf{G}_1 \cup \mathsf{G}_2) = (\mathsf{G}_2 \cup \mathsf{G}_1)$.

Für zwei Mengen $\mathsf{X}_1$ und $\mathsf{X}_2$ sowie zwei Funktionen $\mathsf{f}_1$ und $\mathsf{f}_2$ ist $\oplus$ folgendermaßen definiert:

$$\mathsf{f}_1 \oplus \mathsf{f}_2 |_{(\mathsf{X}_2 \setminus \mathsf{X}_1)} (\mathsf{x}) := \begin{cases} f_1(x) & : \quad x \in X_1 \\ f_2(x) & : \quad x \in X_2 \setminus X_1 \end{cases}$$

Ähnlich dazu kann auch durch den Schnitt von $\mathsf{G}_1$ und $\mathsf{G}_2$ ein neuer Graph erzeugt werden. In diesem Fall werden die Knoten- und Kantenmengen geschnitten und die Funktionen src, tgt, $\mathsf{l}_{\mathsf{N}}$ und $\mathsf{l}_{\mathsf{E}}$ des neuen Graphen entsprechen den Funktionen aus $\mathsf{G}_1$, wobei diese auf die Knoten und die Kanten aus dem Schnitt eingeschränkt werden.

**Definition 6.** *(Schnitt von Graphen)*
*Der Schnitt von zwei beschriftungskompatiblen Graphen* $\mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}$ *ist definiert als* $\mathsf{G}_1 \cap \mathsf{G}_2 := (\mathsf{N}', \mathsf{E}', \mathsf{src}', \mathsf{tgt}', \mathsf{l}'_{\mathsf{N}}, \mathsf{l}'_{\mathsf{E}})$ *mit*

- $\mathsf{N}' := \mathsf{N}_{\mathsf{G}_1} \cap \mathsf{N}_{\mathsf{G}_2}$,

- $E' := E_{G_1} \cap E_{G_2}$,
- $src' := src_{G_1} |_{E'}$,
- $tgt' := tgt_{G_1} |_{E'}$,
- $l'_N := l_{N_{G_1}} |_{N'}$ *und*
- $l'_E := l_{E_{G_1}} |_{E'}$.

*Es gilt* $(G_1 \cap G_2) = (G_2 \cap G_1)$.

Aus den beiden beschriftungskompatiblen Graphen $G_1$ und $G_2$ kann auch durch Subtraktion ein neuer Graph gebildet werden. Bei den Knoten wird die Knotenmenge von $G_2$ von der Knotenmenge von $G_1$ abgezogen. Würde die neue Kantenmenge auf die gleiche Weise berechnet, so könnten in der neuen Kantenmenge Kanten enthalten sein, die keinen Start- oder Zielknoten besitzen. Dies ist genau dann der Fall, wenn der Start- oder Zielknoten in beiden ursprünglichen Graphen enthalten ist, die entsprechende Kante aber nur in $G_1$. Damit der neue Graph der Definition eines Graphen (siehe Definitionen 1 und 2) entspricht, müssen diese Kanten aus der Menge entfernt werden. Die Funktionen src, tgt, $l_N$ und $l_E$ des neuen Graphen werden vom Graph $G_1$ übernommen und auf die neue Knoten- und Kantenmenge eingeschränkt.

**Definition 7.** *(Subtraktion von Graphen)*
*Die Subtraktion von zwei beschriftungskompatiblen Graphen* $G_1, G_2 \in \mathcal{G}$ *ist definiert als* $G_1 \setminus G_2 := (N', E', src', tgt', l'_N, l'_E)$ *mit*

- $N' := N_{G_1} \setminus N_{G_2}$,
- $E' := \{e \in (E_{G_1} \setminus E_{G_2}) |_{(src_{G_1}(e) \in N' \wedge tgt_{G_1}(e) \in N')}\}$,
- $src' := src_{G_1} |_{E'}$,
- $tgt' := tgt_{G_1} |_{E'}$,
- $l'_N := l_{N_{G_1}} |_{N'}$ *und*
- $l'_E := l_{N_{G_1}} |_{E'}$.

*Im Allgemeinen gilt* $(G_1 \setminus G_2) \neq (G_2 \setminus G_1)$.

In späteren Definitionen und Sätzen wird verlangt, dass zwischen zwei bestimmten Knoten eines Graphen ein Pfad existiert. Dabei besteht ein Pfad aus einer Menge von Knoten, die durch eine Menge von Kanten verbunden sind. Beginnt man beim Startknoten des Pfades und folgt den Kanten, ohne deren Richtung zu berücksichtigen, so gelangt man zum Zielknoten.

**Definition 8.** *(Pfad)*
*Für einen gegebenen Graphen* $G$ *ist ein Pfad definiert als eine endlich Sequenz* $\pi \in N_G^*$ *mit der Länge* $k$, *wobei* $\pi = n_1, n_2, \ldots, n_k \in N_G$ *und für alle* $i \in 1, \ldots, k-1$ *existiert ein* $e_i \in E_G$ *mit* $(src(e_i) = n_i \wedge tgt(e_i) = n_{i+1}) \vee (tgt(e_i) = n_i \wedge src(e_i) = n_{i+1})$. *Ein Pfad, der am Knoten* $n_1$ *beginnt und im Knoten* $n_k$ *endet, wird mit* $n_1 \mapsto_\pi n_k$ *beschrieben. Die Länge eines Pfades* $\pi$ *wird als* $length(\pi) : N_G^* \mapsto \mathbb{N}$ *bezeichnet und entspricht* $length(\pi) := | E_\pi |$, $E_\pi$ *stellt die Kantenmenge des Pfades* $\pi$ *dar.*

### Matching von beschrifteten Graphen

Nicht immer ist es für eine Operation erforderlich, dass die beiden beteiligten Graphen identisch sind oder der eine Graph ein Teilgraph des anderen ist. Oft reicht es aus, wenn die beiden Graphen strukturgleich sind. Eine solche Strukturgleichheit wird durch einen totalen Graphhomomorphismus, wie in [Roz97] definiert, beschrieben.

**Definition 9.** *(Totaler Graphhomomorphismus)*
*Für zwei gegebene Graphen* $G_1, G_2 \in \mathcal{G}$ *wird der totale Graphhomomorphismus* $m : G_1 \mapsto G_2$ *beschrieben durch ein Funktionenpaar* $m := \langle m_N : N_{G_1} \mapsto N_{G_2}, m_E : E_{G_1} \mapsto E_{G_2} \rangle$. *Dieser Graphhomomorphismus erhält Start- und Zielknoten von Kanten sowie die Beschriftungen der Knoten und Kanten, d.h. es gilt* $m_N \circ src_{G_1} = src_{G_2} \circ m_E$, $m_N \circ tgt_{G_1} = tgt_{G_2} \circ m_E$, $l_{N_{G_2}} \circ m_N = l_{N_{G_1}}$ *und* $l_{E_{G_2}} \circ m_E = l_{E_{G_1}}$.

Für zwei Funktionen $f$ und $g$ beschreibt $\circ$ ihre Hintereinanderausführung, wobei gilt $g \circ f(x) = g(f(x))$.

Für zwei Graphen $G_1, G_2 \in \mathcal{G}$ bezeichnet die Menge $\mathcal{M}[G_1, G_2]$ die Menge aller Graphhomomorphismen zwischen $G_1$ und $G_2$, d.h. $G_1, G_2 \in \mathcal{G} : \mathcal{M}[G_1, G_2] := \{m \mid m : G_1 \mapsto G_2\}$. Wenn aus dem Kontext hervorgeht, zwischen welchen Graphen die Graphhomomorphismen existieren, wird abkürzend $\mathcal{M}$ verwendet.

Abbildung 3.16 zeigt einen beschrifteten Graphen $G' := (N', E', src', tgt', l'_N, l'_E)$, bestehend aus der Knotenmenge $N' := \{rn'_a, rn'_b, rn'_c\}$, der Kantenmenge $E' := \{re'_a, re'_b\}$, und den Funktionen $src' := \{(re'_a \mapsto rn'_a), (re'_b \mapsto rn'_b)\}$, $tgt' := \{(re'_a \mapsto rn'_b), (re'_b \mapsto rn'_c)\}$, $l'_{N'} := \{(rn'_a \mapsto Shuttle), (rn'_2 \mapsto Track), (rn'_3 \mapsto Track)\}$ und $l'_{E'} := \{(re'_a \mapsto locatedOn), (re'_b \mapsto successor)\}$. Ein Beispiel für einen Graphhomomorphismus $m \in \mathcal{M}[G', G]$ zwischen $G'$ und dem Graphen $G$ aus Abbildung 3.15 ist $m := \langle m_N, m_E \rangle$. Die Funktionen $m_N$ und $m_E$ sind definiert als $m_N := \{(rn'_a \mapsto an_1), (rn'_b \mapsto an_2), (rn'_c \mapsto an_3)\}$ und $m_E := \{(re'_a \mapsto ae_1), (re'_b \mapsto ae_2)\}$. Betrachtet man zum Beispiel den Knoten $rn'_b$ und die Kante $re'_a$ aus $G'$ so gilt: $m_N(src'_{G'}(re'_a)) = m_N(rn'_a) = an_1 = src_G(ae_1) = src_G(m_E(re'_a))$, $m_N(tgt'_{G'}(re'_a)) = m_N(rn'_b) = an_2 = tgt(ae_1) = tgt(m_E(re'a))$, $l_{N_G}(m_N(rn'_b)) = l_{N_G}(an_2) = Track = l'_{N_{G'}}(rn'_b)$ und $l_{E_G}(m_E(re'_a)) = l_{E_G}(ae_1) = locatedOn = l'_{E_{G'}}(ae_a)$.

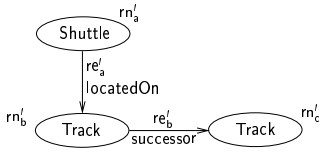



Abbildung 3.16: Graph $G'$, für den ein Graphhomomorphismus in Graph $G$ aus Abbildung 3.15 bestimmt werden soll

Ein totaler Graphhomomorphismus $m : G_1 \mapsto G_2$ bildet alle Knoten und Kanten aus $G_1$ auf Knoten und Kanten aus $G_2$ ab. Dabei muss jedoch nicht auf jeden Knoten und jede Kante aus $G_2$ ein Knoten oder eine Kante aus $G_1$ abgebildet werden. Darüber hinaus ist es möglich, dass zwei oder mehr Knoten oder Kanten aus $G_1$ auf den gleichen Knoten oder die gleiche Kante aus $G_2$

abgebildet werden. Soll dies verhindert werden, so muss man statt des Graphhomomorphismus einen Graphisomorphismus verwenden.

**Definition 10.** *(Graphisomorphismus)*
*Ein Graphisomorphismus* $\mathsf{iso}$ *ist ein totaler Graphhomomorphismus für den gilt, dass die beiden Funktionen* $\mathsf{m_N}$ *und* $\mathsf{m_E}$ *bijektiv sind.*

Im Folgenden wird für zwei Graphen $\mathsf{G_1}, \mathsf{G_2} \in \mathcal{G}$ und einen Graphisomorphismus $\mathsf{iso} : \mathsf{G_1} \mapsto \mathsf{G_2}$ die Kurzschreibweise $\mathsf{G_1} \approx_{\mathsf{iso}} \mathsf{G_2}$ verwendet. Existiert für die beiden Graphen ein beliebiger Graphisomorphismus, der jedoch nicht näher spezifiziert werden soll, so wird dafür $\mathsf{G_1} \approx \mathsf{G_2}$ verwendet.

Für zwei Graphen $\mathsf{G_1}, \mathsf{G_2} \in \mathcal{G}$ bezeichnet die Menge $\mathcal{ISO}[\mathsf{G_1}, \mathsf{G_2}]$ die Menge aller Graphisomorphismen zwischen $\mathsf{G_1}$ und $\mathsf{G_2}$, d.h. $\mathsf{G_1}, \mathsf{G_2} \in \mathcal{G} : \mathcal{ISO}[\mathsf{G_1}, \mathsf{G_2}] := \{\mathsf{iso} \mid \mathsf{iso} \in \mathcal{M} \wedge \text{iso bijektiv}\}$.

Bildet der Graphisomorphismus $\mathsf{iso}$ den Graphen $\mathsf{G_1}$ auf einen Teilgraphen $\mathsf{SG} \in \mathcal{G}$ von $\mathsf{G_2}$ ab ($\mathsf{SG} \leq \mathsf{G_2}$), so wird $\mathsf{iso}$ als Teilgraphisomorphismus bezeichnet.

**Definition 11.** *(Teilgraphisomorphismus)*
*Für drei Graphen* $\mathsf{G_1}, \mathsf{G_2}, \mathsf{TG_2} \in \mathcal{G}$ *mit* $\mathsf{TG_2} \leq \mathsf{G_2}$ *stellt ein Isomorphismus* $\mathsf{iso} \in \mathcal{ISO}[\mathsf{G_1}, \mathsf{TG_2}]$ *einen Teilgraphisomorphismus zwischen* $\mathsf{G_1}$ *und* $\mathsf{G_2}$ *dar.*

Für zwei Graphen $\mathsf{G_1}, \mathsf{G_2} \in \mathcal{G}$ bezeichnet die Menge $\mathcal{TISO}[\mathsf{G_1}, \mathsf{G_2}]$ die Menge aller Teilgraphisomorphismen, d.h. $\mathcal{TISO}[\mathsf{G_1}, \mathsf{G_2}] := \{\ \mathsf{tiso} \mid \exists \mathsf{TG_2} \in \mathcal{G} : \mathsf{TG_2} \leq G_2 \wedge \mathsf{tiso} \in \mathcal{ISO}[\mathsf{G_1}, \mathsf{TG_2}]\}$.

Existiert für zwei Graphen $\mathsf{G_1}, \mathsf{G_2} \in \mathcal{G}$ ein Teilgraphisomorphismus $\mathsf{tiso} \in \mathcal{TISO}[\mathsf{G_1}, \mathsf{G_2}]$ so wird dies kurz notiert als $\mathsf{G_1} \preceq_{\mathsf{tiso}} \mathsf{G_2}$ oder $\mathsf{G_1} \preceq \mathsf{G_2}$.

### 3.2.2 Graphtransformationen

Mit den im vorangegangenen Abschnitt eingeführten Operationen ist es möglich, Graphen zu transformieren, indem Knoten und Kanten aus einem existierenden Graphen gelöscht bzw. neue Knoten und Kanten eingefügt werden. Zur Beschreibung einer solchen Graphtransformation werden Graphtransformationsregeln verwendet.

**Definition 12.** *(Graphtransformationsregel)*
*Eine Graphtransformationsregel* $\mathsf{L} \rightarrow_{\mathsf{r}} \mathsf{R}$ *besteht aus*

- *dem Regelnamen* $\mathsf{r}$*,*
- *einem Graphen* $\mathsf{L} \in \mathcal{G}$*, der als linke Regelseite bezeichnet wird und*
- *einem Graphen* $\mathsf{R} \in \mathcal{G}$*, der als rechte Regelseite bezeichnet wird.*

*Für die beiden Graphen* $\mathsf{L}$ *und* $\mathsf{R}$ *gilt* $\exists \mathsf{TL} \in \mathcal{G} : \mathsf{TL} \leq \mathsf{L} \wedge \mathsf{TL} \leq \mathsf{R} \wedge \mathsf{TL} \neq \mathsf{G}_\emptyset$.

Die linke Regelseite einer Graphtransformationsregel beschreibt, wann die Regel angewendet werden darf, und wird deshalb auch als Vor- oder Anwendungsbedingung bezeichnet. Die rechte

Regelseite beschreibt die Nachbedingung der Regel. Beide Regelseiten zusammen beschreiben, wann die Regel angewendet werden kann und welche Änderungen gemacht werden sollen.

Dabei stellt die Definition 12 der Graphtransformationsregeln eine Vereinfachung der in [Roz97] eingeführten Definition dar. Nach Rozenberg wird eine Regel gegeben durch $r : L \rightarrow_m R$, wobei $r$ der Regelname ist, $L$ die linke und $R$ die rechte Regelseiten sind. $m$ ist ein Graphhomomorphismus, der einen Teilgraphen von $L$ auf $R$ abbildet. Diese Homomorphismus wurde in der Definition 12 durch eine Teilgraphbeziehung ersetzt.

Eine Graphtransformationsregel kann auf einen existierenden Graphen, den so genannten Anwendungsgraphen, angewendet werden, wenn es einen Graphhomomorphismus gibt, der die linke Regelseite auf einen Teilgraphen des Anwendungsgraphen abbildet. In diesem Fall erfüllt der Anwendungsgraph die Vorbedingung der Regel. Ein solcher Graphhomomorphismus wird als Match bezeichnet [Roz97].

**Definition 13.** *(Match)*
*Ein Match* $m$ *für einen Graphen* $L \in \mathcal{G}$ *in einem anderen Graphen* $G \in \mathcal{G}$ *ist ein totaler Graphhomomorphismus mit* $m : L \mapsto G$.

Ein Beispiel für eine solche Graphtransformationsregel ist in Abbildung 3.17 gegeben. Diese Regel entspricht der moveSimple-Regel aus Abbildung 2.10.

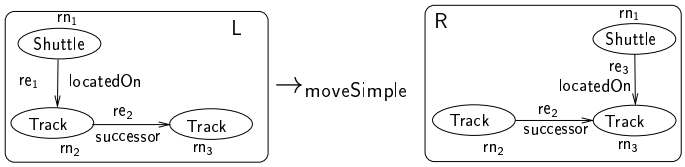
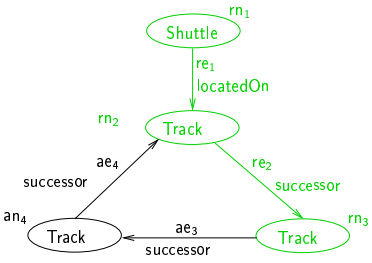


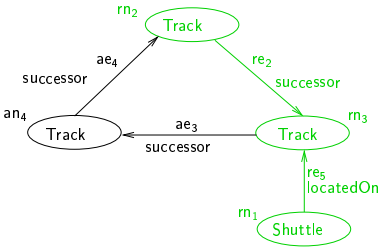

Abbildung 3.17: Beispiel für eine Graphtransformationsregel

Die Regel kann angewendet werden, wenn die drei Knoten $nr_1, nr_2$ und $nr_3$, sowie die beiden Kanten $re_1$ und $re_2$ auf Knoten des entsprechenden Anwendungsgraphen abgebildet werden können. Ein mögliches Matching $m_{ex} : L \mapsto G$ für die Regel moveSimple und den Graphen $G$ aus Abbildung 3.18(a) ist $m_{ex} := \langle m_{ex_N}, m_{ex_E} \rangle$. Dabei ist $m_{ex_N} := \{(rn_1 \mapsto rn_1), (rn_2 \mapsto rn_2), (rn_3 \mapsto rn_3)\}$ und $m_{ex_E} := \{(re_1 \mapsto re_1), (re_2 \mapsto re_2)\}$. Das Matching ist in Abbildung 3.18(a) grün hervorgehoben.

Existiert ein solcher Match für eine Regel $r$ und einen Anwendungsgraphen $G$, so werden alle Knoten und Kanten, auf die der Match die Elemente der linken Regelseite von $r$ abbildet, aber nicht die Elemente der rechten Regelseite, aus $G$ gelöscht. Die Knoten und Kanten, die nur in der Nachbedingung der Regel enthalten sind, aber nicht in der Vorbedingung, werden erzeugt und in den Anwendungsgraphen eingefügt. Der resultierende Graph wird als Zielgraph bezeichnet. Die Regelanwendung wird als direkte Transformation bezeichnet und unterliegt verschiedenen Einschränkungen, die in den Pushout Ansätzen beschrieben sind (siehe Abschnitt 3.4.2). Die folgende Definition für eine direkte Transformation stellt eine Anpassung der Definition aus [HE00] dar. Ein Algorithmus, der die Anwendung einer Graphtransformationsregel beschreibt, ist in Abbildung C.3 in Anhang C.1.2 in Pseudocode-Notation gegeben.




(a) Graph G auf den die Regel aus Abbildung3.17 angewendet werden soll




(b) Graph G′, der resultiert, wenn die Regel aus Abbildung 3.17 auf G angewendet wurde

Abbildung 3.18: Zwei Graphen $G, G'$, für die ein Auftreten der Regel aus Abbildung 3.17 existiert, sodass $o(L) \subseteq G$ und $o(R) \subseteq G'$ gilt

**Definition 14.** *(Direkte Transformation)*

*Gegeben ist eine Graphtransformationsregel* $L \rightarrow_r R$ *und ein Match* $m$ *für* $r$ *in einem Graphen* $G_1 \in \mathcal{G}$*, d.h.* $m : L \mapsto G_1$*. Eine direkte Transformation von* $G_1$ *nach* $G_2$ *(*$G_1$ *und* $G_2$ *sind beschriftungskompatibel) durch eine Graphtransformationsregel* $r$ *ist definiert als ein Graphhomomorphismus* $o : L \cup R \mapsto G_1 \cup G_2$ *mit* $o|_L = m$*, sodass gilt:*

- $o(L) \leq G_1$ *und* $o(R) \leq G_2$*, d.h. die linke Regelseite kann auf einen Teilgraphen von* $G_1$ *abgebildet werden und die rechte Regelseite auf einen Teilgraphen von* $G_2$ *und*
- $o(L \setminus R) = G_1 \setminus G_2$ *und* $o(R \setminus L) = G_2 \setminus G_1$*, d.h. die Anwendung der Regel* $r$ *löscht genau den Teil von* $G_1$*, auf den Elemente von* $L$ *aber nicht von* $R$ *durch* $o$ *abgebildet werden. Umgekehrt werden durch die Anwendung von* $r$ *die Elemente aus* $G_2$ *erzeugt auf die* $R$ *durch* $o$ *abgebildet wird.*

$o$ *wird als Auftreten (engl. occurence) bezeichnet.*

In Abbildung 3.18 sind zwei Graphen $G$ und $G'$ gegeben. Das Auftreten $o$ der Regel aus Abbildung 3.17 ist in den beiden Graphen grün dargestellt. Das oben vorgestellte Matching $m_{ex}$ wird von $o$ verwendet, um die linke Seite der Regel auf $G$ abzubilden.

Wird ein Graph $G_1$ durch eine Regel $r$ und dem Auftreten $o$ von $r$ in $G_1$ in den Graphen $G_2$ transformiert, so wird dies beschrieben durch $G_1 \rightarrow_{r,o} G_2$ oder kurz $G_1 \rightarrow_r G_2$.

Eine Sequenz von direkten Transformationen der Form $\rho = (G_0 \Longrightarrow_{r_0,o_0} G_1 \Longrightarrow_{r_1,o_1} \ldots \Longrightarrow_{r_{n-1},o_{n-1}} G_n)$ beschreibt die Transformation eines Graphen $G_1$ in einen Graphen $G_n$ durch die Regeln $r_1, \ldots, r_{n-1}$ und ihre Auftreten $o_1, \ldots, o_{n-1}$. Die Kurzschreibweise für die Anwendung von einer solchen Sequenz von Regeln ist $G_0 \Longrightarrow^*_{(r_0,o_0),\ldots,(r_{n-1},o_{n-1})} G_n$, $G_0 \Longrightarrow^*_{r_0,\ldots,r_{n-1}} G_n$ oder $G_0 \Longrightarrow^* G_n$.

### 3.2.3 Graphtransformationsregeln mit negativer Anwendungsbedingung

Die linke Seite einer Graphtransformationsregel gibt an, wann die Regel angewendet werden darf. Mittels einer negativen Anwendungsbedingung kann die Anwendbarkeit einer Regel genau-

er spezifiziert bzw. eingeschränkt werden. Solche negativen Anwendungsbedingungen werden in diesem Abschnitt eingeführt.

In Abbildung 3.17 ist die Regel moveSimple angegeben. Diese Regel beschreibt, dass das Shuttle vom ersten Track $rn_2$ auf den nächsten Track $rn_3$ weitergesetzt wird. Wird diese Regel auf den Anwendungsgraphen in Abbildung 3.19 so angewendet, dass die Kante locatedOn $ae_1$ zwischen dem mit Shuttle beschrifteten Knoten $an_1$ und dem mit Track beschrifteten Knoten $an_2$ gelöscht und zwischen $an_1$ und Track-Knoten $an_5$ eine neue locatedOn-Kante erzeugt wird, so würde die kritische Situation impendingCollisionSimplified (siehe Abbildung 3.2) auftreten, da die beiden Shuttles $an_1$ und $an_5$ auf benachbarten Tracks fahren würden. Um die Regelanwendung in einem solchen Fall zu verhindern, muss die Regel um eine so genannte negative Anwendungsbedingung erweitert werden.

Während die bisher betrachteten Anwendungsbedingungen beschreiben, wann eine Regel angewendet werden darf, beschreiben negative Anwendungsbedingungen, wann eine Regel nicht angewendet werden darf. Eine negative Anwendungsbedingung besteht aus einer Menge von Graphen. Jeder dieser Graphen enthält den Graphen der Anwendungsbedingung und erweitert diesen um mindestens einen Knoten oder eine Kante. Gibt es ein Match für die Anwendungsbedingung in einem Anwendungsgraphen, so muss geprüft werden, ob es für einen Graphen der negativen Anwendungsbedingung ebenfalls einen Match gibt. Ist dies der Fall, so darf die Regel nicht angewendet werden.

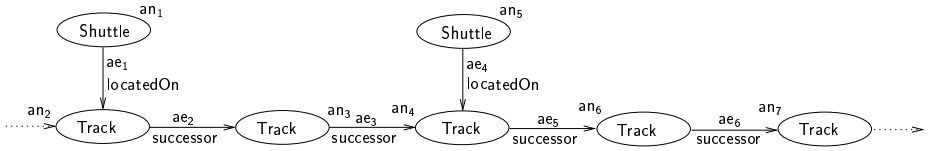



Abbildung 3.19: Beispiel für einen Graphen, bei dem die Anwendung von moveSimple zu einer kritischen Situation führen kann

**Definition 15.** *(Negative Anwendungsbedingung)*

*Eine negative Anwendungsbedingung (NAB) für einen Graphen* $\mathsf{G} \in \mathcal{G}$ *ist eine endliche Menge von Graphen* $\hat{\mathcal{G}}$*, für die gilt* $\forall \hat{\mathsf{G}}_\mathsf{i} \in \hat{\mathcal{G}} : \mathsf{G} \leq \hat{\mathsf{G}}_\mathsf{i}$ *und für alle Knoten* $\mathsf{n} \in (\mathsf{N}_{\hat{\mathsf{G}}_\mathsf{i}} \setminus \mathsf{N}_\mathsf{G})$ *gibt es einen Pfad* $\pi_\mathsf{i}$ *von* $\mathsf{n}$ *zu einem Knoten* $\mathsf{n}' \in \mathsf{G}$*, d.h.* $\forall \mathsf{n} \in (\mathsf{N}_{\hat{\mathsf{G}}_\mathsf{i}} \setminus \mathsf{N}_\mathsf{G}) \exists \mathsf{n}' \in \mathsf{N}_\mathsf{G}, \mathit{Pfad}\ \ \pi : \mathsf{n} \mapsto_\pi \mathsf{n}'$*. Die Graphen aus* $\hat{\mathcal{G}}$ *werden als Constraints bezeichnet. Ein solches Constraint* $\hat{\mathsf{G}}_\mathsf{i}$ *wird durch einen Graphen* $\mathsf{AG} \in \mathcal{G}$ *und einen Teilgraphisomorphismus* $\mathsf{tiso} \in \mathcal{TISO}[\mathsf{G}, \mathsf{AG}]$ *erfüllt (geschrieben* $\mathsf{AG}, \mathsf{G}, \mathsf{tiso} \vdash \hat{\mathsf{G}}_\mathsf{i}$*), wenn gilt* $\mathsf{G} \preceq_\mathsf{tiso} \mathsf{AG}$ *aber* $\nexists \mathsf{tiso}' \in \mathcal{TISO}[\mathsf{G}, \mathsf{AG}] : \mathsf{tiso}' \mid_\mathsf{G} = \mathsf{tiso} \wedge \hat{\mathsf{G}}_\mathsf{i} \preceq_{\mathsf{tiso}'} \mathsf{AG}$*. Ein Graph* AG *und ein Teilgraphisomorphismus erfüllen eine negative Anwendungsbedingung* $\hat{\mathcal{G}}$ *(geschrieben* $\mathsf{AG}, \mathsf{G}, \mathsf{tiso} \vdash \hat{\mathcal{G}}$*), falls* AG *alle Constraints* $\hat{\mathsf{G}}_\mathsf{i} \in \hat{\mathcal{G}}$ *erfüllt, d.h.* $\forall \hat{\mathsf{G}}_\mathsf{i} \in \hat{\mathcal{G}} : (\mathsf{AG}, \mathsf{G}, \mathsf{tiso} \vdash \hat{\mathsf{G}}_\mathsf{i})$.

Auf Grund der Einschränkung, dass ein Auftreten durch einen Teilgraphisomorphismus beschrieben wird, anstelle des sonst üblichen Graphhomomorphismus, kann die Definition der negativen Anwendungsbedingung im Vergleich zur Definition in [Roz97] vereinfacht werden.

Die erweiterte Graphtransformationsregel, moveSimpleNAC, ist in Abbildung 3.20 abgebildet. Die Graphen $\hat{L}_1$ und $\hat{L}_2$ stellen die negative Anwendungsbedingung dar. $\hat{L}_1$ verbietet die Anwendung der Regel, falls sich auf dem nächsten Track ein weiteres Shuttle befindet, da es sonst zu einer Kollision käme. $\hat{L}_2$ verbietet, das Shuttle vorwärts zu setzen, wenn sich auf dem übernächsten Track ein anderes Shuttle befindet, da andernfalls die kritische Situation auftreten würde. Die so veränderte Regel kann nicht mehr auf den Knoten $an_1$ angewendet und das entsprechende Shuttle somit nicht weiter gesetzt werden. Sie kann jedoch auf den Knoten $an_5$ angewendet werden, da sich auf den beiden Tracks vor diesem Shuttle kein weiteres Shuttle befindet.

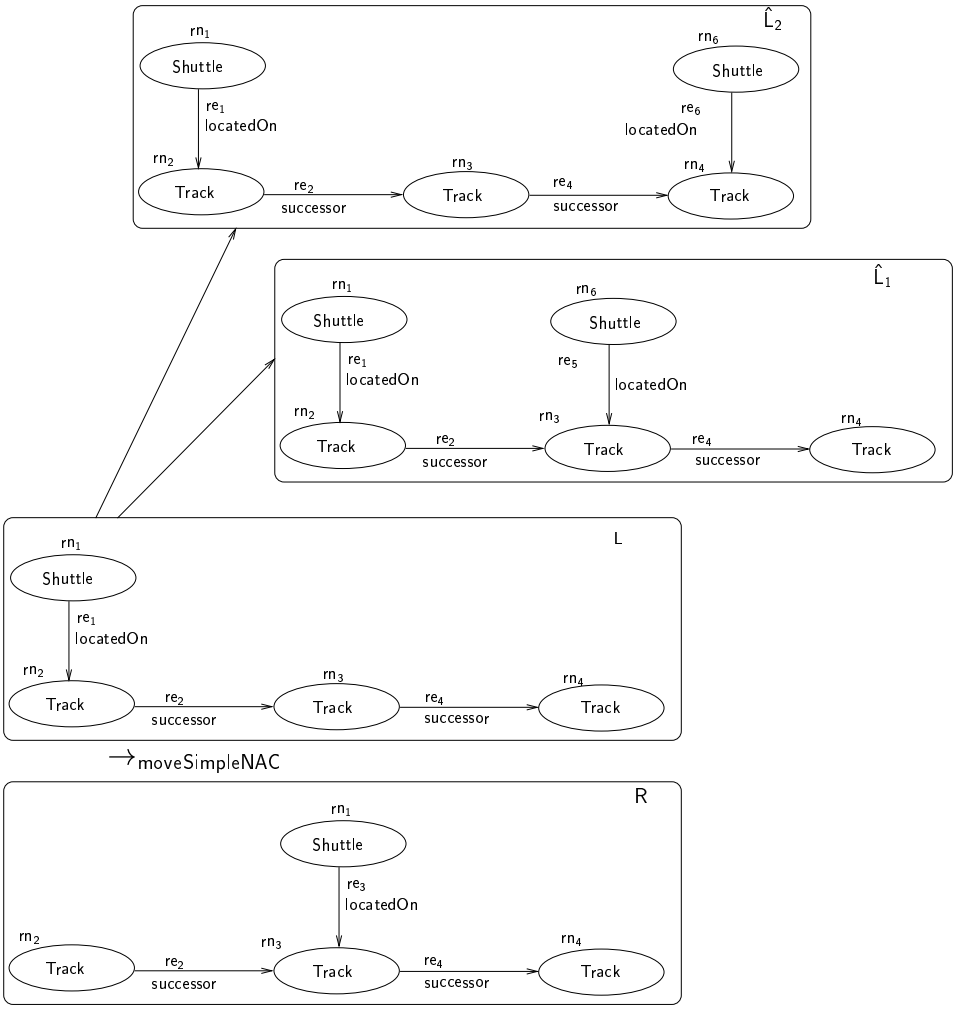



Abbildung 3.20: Regel moveSimpleNAC, die die Regel moveSimple um negative Anwendungsbedingungen erweitert

Um redundante Informationen zu vermeiden, werden im Folgenden minimale negative Anwendungsbedingungen betrachtet. Angenommen, eine negative Anwendungsbedingung enthält zwei Graphen $\hat{G}_1$ und $\hat{G}_2$ und $\hat{G}_1 \preceq \hat{G}_2$. Wenn es einen Match von $\hat{G}_2$ in einem Anwendungsgra-

phen $\mathsf{G}$ gibt, dann kann dieser Match so verändert werden, dass er auch $\hat{\mathsf{G}}_1$ auf $\mathsf{G}$ abbildet. In einem solchen Fall kann $\hat{\mathsf{G}}_2$ aus der negativen Anwendungsbedingung entfernt und die negative Anwendungsbedingung dadurch minimiert werden. Enthält eine negative Anwendungsbedingung kein Graphpaar für das gilt, dass einer der beiden Graphen durch einen Isomorphismus auf einen Teilgraph des anderen Graphen abgebildet werden kann, so ist die negative Anwendungsbedingung minimal.

**Definition 16.** *(Minimale Negative Anwendungsbedingung)*
*Eine negative Anwendungsbedingung $\hat{\mathcal{G}}$ eines Graphen $\mathsf{G}$ ist eine minimale Menge von Graphen, wenn gilt* $\forall \hat{\mathsf{G}}, \hat{\mathsf{G}}' \in \hat{\mathcal{G}} : \hat{\mathsf{G}} \neq \hat{\mathsf{G}}' \Rightarrow (\nexists \mathsf{tiso} \in \mathcal{TISO}[\hat{\mathsf{G}}, \hat{\mathsf{G}}'] : \mathsf{tiso} \mid_{\mathsf{G}} = \mathsf{id}_{\mathsf{G}} \wedge \hat{\mathsf{G}} \preceq_{\mathsf{tiso}} \hat{\mathsf{G}}')$.

$\mathsf{id}_{\mathsf{G}}$ entspricht der Identitätsfunktion.

Ein Algorithmus zur Minimierung von negativen Anwendungsbedingungen ist in Abbildung C.2 in Anhang C.1.1 in Pseudocode-Notation angegeben.

### 3.2.4 Graphtransformationssysteme

Nach [Roz97] wird ein Graphtransformationssystem durch eine Menge von initialen Graphen sowie einer Menge von Graphtransformationsregeln gegeben.

**Definition 17.** *(Graphtransformationssystem (GTS))*
*Ein Graphtransformationssystem* $\mathsf{S} := (\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$ *ist ein Tupel mit:*

- $\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}$ *ist die Menge der Initial- oder Startgraphen des Systems und*
- $\mathcal{R}_{\mathsf{S}}$ *ist eine endliche Menge von Graphtransformationsregeln.*

Für ein gegebenes Graphtransformationssystem $\mathsf{S} := \{\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}}\}$ bezeichnet $\mathsf{REACH}(\mathsf{S}) := \{\mathsf{G}_1 \in \mathcal{G} \mid \exists \mathsf{G}_2 \in \mathcal{G}^{\mathsf{i}}_{\mathsf{S}} : \mathsf{G}_2 \models\!\Rightarrow^* \mathsf{G}_1\}$ die Menge aller in $\mathsf{S}$ erreichbaren Zustände.

Die Definition 17 erlaubt Systeme mit unendlich großem Zustandsraum zu spezifizieren, die als unendliche Graphtransformationssysteme bezeichnet werden. Ein GTS $\mathsf{S} := \{\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}}\}$ ist genau dann unendlich, wenn gilt, dass $\mathsf{REACH}(\mathsf{S})$ unendlich viele Graphen enthält, die unter Berücksichtigung des Graphisomorphismus verschieden sind, d.h. $\forall \mathsf{G}_1 \in \mathsf{REACH}(\mathsf{S}), \nexists \mathsf{G}_2 \in (\mathsf{REACH}(\mathsf{S}) \setminus \mathsf{G}_1) : \mathsf{G}_1 \approx \mathsf{G}_2$.

## 3.3 Graphmuster

Mit den bisher eingeführten Konstrukten ist es möglich Graphtransformationssysteme zu beschreiben. Darüber hinaus wurde gezeigt, wie eine Regel auf einen Graphen angewendet werden kann, wenn es ein gültiges Matching für die linke Regelseite im Graphen gibt. Es ist allerdings noch nicht möglich, strukturelle Eigenschaften zu beschreiben, die bei der Anwendung einer Graphtransformationsregel erhalten bleiben sollen. Zur Beschreibung struktureller Eigenschaften sollen im Folgenden Graphmuster eingeführt werden.

Graphmuster wurden bereits im informalen Abschnitt dieses Kapitels, Abschnitt 3.1, dazu verwendet, die kritische Situation impendingCollision bzw. impendingCollisionSimplified zu spezifizieren.

Ein einfaches Graphmuster besteht aus einem Graphen $\mathsf{G}$. Ein Anwendungsgraph $\mathsf{AG}$ erfüllt dieses Muster, falls es einen Teilgraphisomorphismus $\mathsf{tiso} \in \mathcal{TISO}[\mathsf{G}, \mathsf{AG}]$ gibt, der $\mathsf{G}$ auf einen Teilgraphen von $\mathsf{AG}$ abbildet, d.h. $\mathsf{G} \preceq_{\mathsf{tiso}} \mathsf{AG}$. Ein solches Graphmuster kann dazu verwendet werden, eine Menge von Graphen zu beschreiben. Dies ist genau die Menge von Graphen, die das entsprechende Muster erfüllen und somit alle einen gemeinsamen Teilgraphen haben.

**Definition 18.** *(Einfache Graphmuster)*
*Ein einfaches Graphmuster* $\mathsf{p} := [\mathsf{P}]$ *besteht aus einem nicht leeren Graphen* $\mathsf{P} \in \mathcal{G}$.

Die Ausdrucksstärke von Graphmustern kann erhöht werden, indem negative Anwendungsbedingungen hinzugenommen werden. Ein solches Graphmuster besteht dann aus einem Graphen $\mathsf{P}$ und einer negativen Anwendungsbedingung $\hat{\mathcal{P}}$, wobei die Menge der Graphen der negativen Anwendungsbedingung leer sein kann. Ein Graphmuster mit negativer Anwendungsbedingung beschreibt eine Menge von Graphen, die einen Teilgraphen haben, auf den einerseits $\mathsf{P}$ durch einen Teilgraphisomorphismus tiso abgebildet werden kann. Andererseits aber gilt, dass kein Graph der negativen Anwendungsbedingung $\hat{\mathsf{P}} \in \hat{\mathcal{P}}$ durch eine Erweiterung von tiso auf die Graphen der Menge abgebildet werden kann.

**Definition 19.** *(Graphmuster)*
*Ein Graphmuster* $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ *besteht aus einem nicht leeren Graphen* $\mathsf{P} \in \mathcal{G}$ *und einer negativen Anwendungsbedingung, die durch eine möglicherweise leeren Menge von Graphen* $\hat{\mathcal{P}}$ *beschrieben wird. Dabei gilt:* $\forall \hat{\mathsf{P}} \in \hat{\mathcal{P}} : \mathsf{P} \leq \hat{\mathsf{P}}$.

$\mathcal{GP}$ bezeichnet die Menge aller Graphmuster.

Im vorangegangenen Abschnitt wurde die linke Regelseite einer Graphtransformationsregel um eine negative Anwendungsbedingung erweitert. Die Anwendungsbedingung der Regel zusammen mit der negativen Anwendungsbedingung entspricht einem Graphmuster.

Auch die kritische Situation impendingCollision kann als ein Graphmuster aufgefasst werden. In diesem Fall beschreibt das Muster die Menge der Graphen, in denen eine kritische Situation eingetreten ist und eine Kollision droht.

Ein einfaches Graphmuster $[\mathsf{G}]$ entspricht einem Graphmuster $[\mathsf{G}, \hat{\mathcal{G}}]$, wobei $\hat{\mathcal{G}}$ eine leere Menge ist.

Sind zwei Graphmuster $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ und $\mathsf{p}' := [\mathsf{P}', \hat{\mathcal{P}}']$ gegeben, dann ist $\mathsf{p}$ ein Teilgraphmuster von $\mathsf{p}'$, wenn die folgenden beiden Bedingungen erfüllt sind. Erstens muss es einen Teilgraphisomorphismus $\mathsf{tiso} \in \mathcal{TISO}[\mathsf{P}, \mathsf{P}']$ geben, der $\mathsf{P}$ auf einen Teilgraphen von $\mathsf{P}'$ abbildet. Es muss also gelten: $\mathsf{P} \preceq \mathsf{P}'$. Allerdings darf kein Graph der negativen Anwendungsbedingung $\hat{\mathcal{P}}$ durch einen Teilgraphisomorphismus auf einen Teilgraphen von $\mathsf{P}'$ abgebildet werden können. Zweitens, wenn $\mathsf{p}$ nicht erfüllt ist, weil einer der Graphen aus $\hat{\mathcal{P}}$ auf einen Anwendungsgraphen abgebildet werden kann, muss es auch einen Graphen in $\hat{\mathcal{P}}'$ geben, der auf den Anwendungsgraphen abgebildet werden kann. Das bedeutet, dass es für jeden Graphen aus $\hat{\mathcal{P}}$ einen Graphen in $\hat{\mathcal{P}}'$ gibt, der auf den Anwendungsgraphen abgebildet werden kann. Dies ist genau dann der Fall, wenn es für jeden Graphen $\hat{\mathsf{P}} \in \hat{\mathcal{P}}$ einen Graphen $\hat{\mathsf{P}}' \in \hat{\mathcal{P}}'$ gibt, der die folgende Eigenschaft hat. Es gilt, $\mathsf{P} \preceq_{\mathsf{tiso}} \mathsf{P}'$. Entfernt man aus $\hat{\mathsf{P}}'$ alle Elemente, auf die der Teilgraphisomorphismus tiso keine Elemente von $\mathsf{P}$ abbildet, so gibt es für den verbleibenden Graphen einen Teilgraphisomorphismus, der den Restgraphen auf einen Teilgraph von $\hat{\mathsf{P}}$ abbildet.

**Definition 20.** *(Teilgraphmuster)*
*Gegeben sind zwei Graphmuster* $\mathsf{p} = [\mathsf{P}, \hat{\mathcal{P}}]$ *und* $\mathsf{p}' = [\mathsf{P}', \hat{\mathcal{P}}']$. $\mathsf{p}$ *ist ein Teilgraphmuster von* $\mathsf{p}'$ *(geschrieben* $\mathsf{p} \subseteq \mathsf{p}'$*) wenn gilt:*

- $\exists \mathsf{tiso} \in \mathcal{TISO} : \mathsf{P} \preceq_{\mathsf{tiso}} \mathsf{P}' \wedge \mathsf{P}', \mathsf{P}, \mathsf{tiso} \vdash \hat{\mathcal{P}}$,
- $\forall \hat{\mathsf{P}} \in \hat{\mathcal{P}}, \exists \hat{\mathsf{P}}' \in \hat{\mathcal{P}}', \mathsf{tiso}' \in \mathcal{TISO} : \mathsf{tiso}'^{-1} \mid_{\mathsf{P}} = \mathsf{tiso} \wedge (\hat{\mathsf{P}}' \setminus (\mathsf{P}' \setminus \mathsf{tiso}(\mathsf{P}))) \preceq_{\mathsf{tiso}'} \hat{\mathsf{P}}$.

Abbildung 3.21 zeigt ein Graphmuster, impendingCollisionExtended, das eine kritische Situation beschreibt. Dieses Muster besagt, dass eine kritische Situation eingetreten ist, wenn sich zwei Shuttles auf benachbarten Tracks befinden, die beiden Shuttles das DistanceCoordination-Muster nicht miteinander ausführen, obwohl beide Shuttles das Publication-Muster mit der gleichen BaseStation ausführen. Ist ein Shuttle bei einer BaseStation registriert, so muss es in regelmäßigen Abständen seine Position an diese senden und erhält die Position der anderen gemeldeten Shuttles. Das bedeutet, dass die beiden Shuttles wissen, dass das jeweils andere Shuttle sich in ihrer Nähe befindet, koordinieren sich aber dennoch nicht.

Das Graphmuster impendingCollision ist ein Teilgraphmuster von impendingCollisionExtended. Zum einen gilt, dass die Anwendungsbedingung von impendingCollision mittels eines Teilgraphisomorphismus auf die Anwendungsbedingung von impendingCollisionExtended abgebildet werden kann. Zum anderen ist aber auch die zweite Bedingung für ein Teilgraphmuster erfüllt. Für den Graph der negativen Anwendungsbedingung $\hat{\mathsf{IC}}$ existiert ein Graph in der negativen Anwendungsbedingung $\hat{\mathsf{ICE}}$, sodass gilt, kann $\hat{\mathsf{IC}}$ auf einen Anwendungsgraphen abgebildet werden, so kann auch $\hat{\mathsf{IC}}$ auf den Anwendungsgraphen abgebidelt werden. Entfernt man aus $\hat{\mathsf{ICE}}$ alle Elemente, die in ICE vorkommen, aber nicht in IC (dies sind die Knoten $\mathsf{vn}'_5$, $\mathsf{vn}'_6$ und $\mathsf{vn}'_7$ sowie die Kanten $\mathsf{ve}'_4$, $\mathsf{ve}'_5$, $\mathsf{ve}'_6$, $\mathsf{ve}'_7$, $\mathsf{ve}'_8$ und $\mathsf{ve}'_9$) so erhält man einen Graphen, der isomorph zum Graphen $\hat{\mathsf{IC}}$ ist[2].

Für ein Graphmuster $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ beschreibt $\mathcal{G}[\mathsf{p}]$ die Menge aller Graphen, die das Muster $\mathsf{p}$ erfüllen, d.h. $\mathcal{G}[\mathsf{p}] := \{\mathsf{G} \in \mathcal{G} \mid \exists \mathsf{tiso} \in \mathcal{TISO} : \mathsf{P} \preceq_{\mathsf{tiso}} \mathsf{G} \wedge \mathsf{G}, \mathsf{P}, \mathsf{tiso} \vdash \hat{\mathcal{P}}\}$. Die Kurzschreibweise dafür, dass ein Graph $\mathsf{G}$ ein Muster $\mathsf{p}$ erfüllt, ist $\mathsf{G} \models \mathsf{p}$.

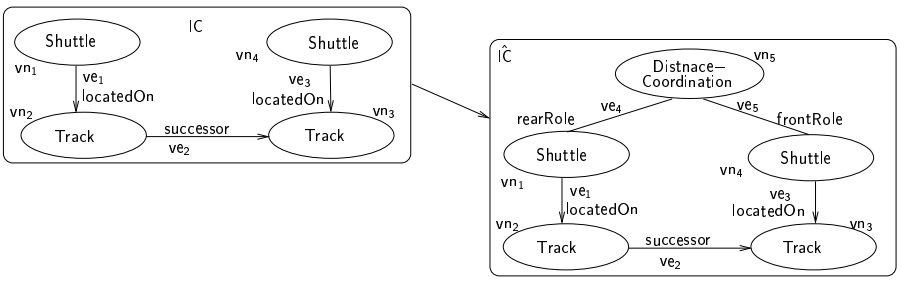
Gegeben sind zwei Graphmuster $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ und $\mathsf{p}' := [\mathsf{P}', \hat{\mathcal{P}}']$. Ist $\mathsf{p}$ ein Teilgraphmuster von $\mathsf{p}'$, so erfüllt ein Graph $\mathsf{G} \in \mathcal{G}$, der das Graphmuster $\mathsf{p}'$ erfüllt, auch $\mathsf{p}$. Dies ist in der Implikation B.1 von Lemma 1 festgehalten. Gibt es andererseits einen Graphen $\mathsf{G} \in \mathcal{G}$, der $\mathsf{p}'$ erfüllt, $\mathsf{p}$ aber nicht, so folgt daraus, dass $\mathsf{p}$ kein Teilgraphmuster von $\mathsf{p}'$ sein kann. Dieses Ergebnis wird in der Implikation 3.2 zusammengefasst. Eine Beweisskizze für dieses und die folgenden Lemmata und Theoreme ist in Anhang B gegeben.

**Lemma 1.** *(Implikation von Teilgraphmustern)*
*Für zwei Graphmuster* $\mathsf{p}$ *und* $\mathsf{p}'$ *gilt:*

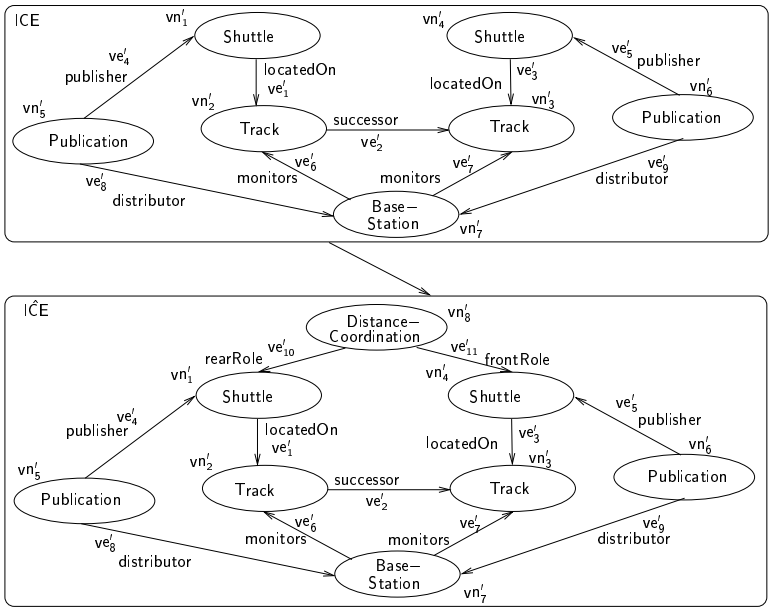
$$(\mathsf{p} \subseteq \mathsf{p}') \Rightarrow (\forall \mathsf{G} \in \mathcal{G}[\mathsf{p}'] : \mathsf{G} \models \mathsf{p}) \quad \textit{und} \tag{3.1}$$

$$(\exists \mathsf{G} \in \mathcal{G}[\mathsf{p}'] : \mathsf{G} \not\models \mathsf{p}) \Rightarrow (\mathsf{p} \not\subseteq \mathsf{p}') \tag{3.2}$$

[2]Es gilt auch, dass impendingCollisionSimplified ein Teilgraphmuster von impendingCollision ist




(a) Graphmuster impendingCollision




(b) Graphmuster impendingCollisionExtended

Abbildung 3.21: Beispiel für ein Graphmuster und ein Teilgraphmuster

## 3.4 Das Systemmodell

Erweitert man die in Definition 17 eingeführten Graphtransformationssysteme um Typen, so lassen sich damit UML-Modelle formalisieren.

### 3.4.1 Graphen zur Zustandsbeschreibung

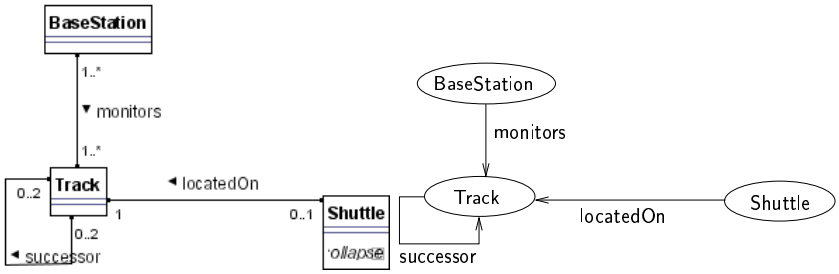
In dieser Arbeit werden UML-Klassendiagramme verwendet, um die Architektur der Software eines mechatronischen Systems zu beschreiben. Ein Objektdiagramm beschreibt einen bestimmten Zustand des Systems. Um ein System, das durch Klassendiagramme und Objektdiagramme

beschrieben ist, zu formalisieren, wird das Klassendiagramm auf einen Typgraphen abgebildet und jedes Objektdiagramm auf einen typisierten Graphen. Die Übergänge von einem Zustand zu einem anderen werden mittels Graphtransformationsregeln gegeben.

Das Klassendiagramm wird formalisiert, indem die Klassen auf Knoten und die Assoziationen auf Kanten abgebildet werden. Klassennamen und Assoziationsnamen werden mittels der Knoten- und Kantenbeschriftungsfunktionen den Knoten und Kanten zugewiesen. Um die Graphtransformationssysteme möglichst einfach zu halten, werden Kardinalitäten und Vererbung hier nicht berücksichtigt.

Ein Klassendiagramm kann auf einen Graphen $G_\Omega := (N_\Omega, E_\Omega, src_\Omega, tgt_\Omega)$ abgebildet werden. Dieser Graph $G_\Omega$ dient dann als Typgraph. Die Menge der Knotentypen, bzw. das Alphabet zur Beschriftung von Knoten, $\Omega_N$ ist definiert als $\Omega_N = N_\Omega$, ebenso ist die Menge der Kantentypen $\Omega_E$ definiert als $\Omega_E = E_\Omega$.

In Abbildung 4.1(a) ist noch einmal das einfache UML-Klassendiagramm aus Abschnitt 2.1.3 abgebildet. Dieses Diagramm zeigt die vereinfachte Ontologie des Shuttle-Systems. Der Typgraph, der diesem Klassendiagramm entspricht, besteht aus drei Knoten, beschriftet mit BaseStation, Track und Shuttle. Diese Knoten entsprechen den gleichnamigen Klassen des Klassendiagramms. Ebenso enthält der Typgraph drei Kanten, die mit locatedOn, monitors und successor beschriftet sind. Die Kanten entsprechen den Assoziationen des Klassendiagramms, wobei die Assoziationsnamen nun als Kantenbeschriftungen dienen. Die Kardinalitäten, die im Klassendiagramm angegeben sind, fallen im Typgraphen weg.




(a) Klassendiagramm, das die vereinfachte Systemontologie beschreibt

(b) Typgraph für das Klassendiagramm in Abbildung 4.1(a)

Abbildung 3.22: Einfaches Klassendiagramm und der dazugehörige Typgraph

Nach der Abbildung des Klassendiagramms auf einen Typgraphen entspricht ein UML-Objektdiagramm einem beschrifteten Graphen. Jeder Knoten des beschrifteten Graphen entspricht einem Objekt des Objektdiagramms und jede Kante einem Link. Für diese Graphen muss garantiert werden, dass ihre Beschriftung konform zum Typgraphen und somit zum ursprünglichen Klassendiagramm ist. Ein Knoten ist typkonform, wenn seine Beschriftung einem Typ des Typgraphen entspricht. Eine Kante ist typkonform, wenn ihre Beschriftung mit einer Kantenbeschriftung aus dem Typgraphen übereinstimmt und wenn die

Beschriftung des Start- und Zielknotens mit den Typen übereinstimmen, die vom Typgraphen verlangt werden. Um Knoten und Kanten besser referenzieren zu können, erhalten sie zusätzlich einen eindeutigen Namen; dieser hat jedoch keinen Einfluss auf die auf den Graphen angewendeten Operationen. Der Name eines Knoten oder einer Kante steht immer vor einem „:“, während der Typ dem „:“folgt. Die Menge der typkonformen Kanten ist als $\mathsf{correctTypedEdges}(\mathsf{G}_\Omega) := \{(\mathsf{src}_\Omega(\mathsf{e}), \mathsf{e}, \mathsf{tgt}_\Omega(\mathsf{e})) \in \mathsf{N}_\Omega \times \mathsf{E}_\Omega \times \mathsf{N}_\Omega\}$ definiert. Ein Graph ist typkonform, wenn die Knoten- und Kantenbeschriftungsfunktionen nur Beschriftungen zuweisen, die im Typgraph definiert sind und wenn alle Kanten typkonform sind. Formal ist die Typkonformität definiert als:

**Definition 21.** *(Typkonformität von Graphen)*
*Die Beschriftung eines Graphen* $\mathsf{G} := (\mathsf{N}, \mathsf{E}, \mathsf{src}, \mathsf{tgt}, \mathsf{l}_\mathsf{n}, \mathsf{l}_\mathsf{E})$ *ist genau dann typkonform zu einem Typgraph* $\mathsf{G}_\Omega := (\mathsf{N}_\Omega, \mathsf{E}_\Omega, \mathsf{src}_\Omega, \mathsf{tgt}_\Omega)$*, wenn gilt:*

$$\mathsf{l}_\mathsf{N} : \mathsf{N} \mapsto \Omega_\mathsf{N} \wedge \Omega_\mathsf{N} \subseteq \mathsf{N}_\Omega \quad \textit{und} \quad \mathsf{l}_\mathsf{E} : \mathsf{E} \mapsto \Omega_\mathsf{E} \wedge \Omega_\mathsf{E} \subseteq \mathsf{E}_\Omega \tag{3.3}$$

$$\forall \mathsf{e} \in \mathsf{E} : (\mathsf{l}_\mathsf{N}(\mathsf{src}(\mathsf{e})), \mathsf{l}_\mathsf{E}(\mathsf{e}), \mathsf{l}_\mathsf{N}(\mathsf{tgt}(\mathsf{e}))) \in \mathsf{correctTypedEdges}(\mathsf{G}_\Omega). \tag{3.4}$$

Die Menge aller Graphen, die typkonform zum Typgraphen $\mathsf{G}_\Omega$ sind, wird mit $\mathcal{G}[\mathsf{G}_\Omega]$ bezeichnet. Anhand des Typgraphen $\mathsf{G}_\Omega$ können UML-Objektdiagramme auf typkonforme beschriftete Graphen abgebildet werden. Wird eine Graphtransformationsregel auf einen typkonformen Graphen angewendet, so muss garantiert werden, dass auch der resultierende Graph typkonform ist. Die folgende Definition 22 sowie das Theorem 1 zeigen, dass eine Regelanwendung die Typkonformität erhält, wenn der Anwendungsgraph, der Graph der linken Regelseite, inklusive der negativen Anwendungsbedingung sowie der Graph der rechten Regelseite typkonform sind.

**Definition 22.** *(Typkonformität von Graphtransformationsregeln)*
*Eine Graphtransformationsregel* $\mathsf{r} \in \mathcal{R}_\mathsf{S}$ *eines Graphtransformationssystems* $\mathsf{S} := (\mathcal{G}_\mathsf{S}, \mathcal{G}^\mathsf{i}_\mathsf{S}, \mathcal{R}_\mathsf{S})$ *ist genau dann typkonform zu* $\mathsf{G}_\Omega$*, wenn alle in* $\mathsf{r}$ *enthaltenen Graphen (*$\mathsf{L}$, $\hat{\mathcal{L}}$ *und* $\mathsf{R}$*) typkonform zu* $\mathsf{G}_\Omega$ *sind.*

Damit eine Regelanwendung typkonform erfolgen kann, muss die Vereinigung, der Schnitt, die Subtraktion und das Matching von Graphen die Typkonformität erhalten. Da bei keiner der vier Operationen neue Knoten oder Kanten entstehen, erfüllen alle diese Anforderungen. Die entsprechenden Lemmata (Lemma 2, 3 und 4) sind in Anhang B.2 enthalten.

Diese Ergebnisse werden in Theorem 1 verwendet, um zu zeigen, dass die Anwendung einer typkonformen Graphtransformationsregel auf einen typkonformen Graphen wieder in einem typkonformen Graphen resultiert.

**Theorem 1.** *(Erhaltung der Typkonformität)*
*Für jeden typkonformen Graphen* $\mathsf{G}_1 \in \mathcal{G}$ *und jede typkonforme Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_\mathsf{r} \mathsf{R}$ *gilt: falls ein* $\mathsf{G}_2 \in \mathcal{G}$ *existiert, mit* $\mathsf{G}_1 \models\Rightarrow_\mathsf{r} \mathsf{G}_2$*, so ist auch* $\mathsf{G}_2$ *typkonform.*

### 3.4.2 Graphtransformationen zur Beschreibung von Zustandsänderungen

Nach Definition 14 kann eine Graphtransformationsregel $[\mathsf{L}, \hat{\mathcal{L}}]\rightarrow_{\mathsf{r}}\mathsf{R}$ auf einen Graphen $\mathsf{G}$ angewendet werden, falls es einen Match $\mathsf{m} \in \mathcal{M}[\mathsf{L}, \mathsf{G}]$ gibt, der $\mathsf{L}$ auf $\mathsf{G}$ abbildet, der aber nicht so erweitert werden kann, dass er ein $\hat{\mathsf{L}} \in \mathcal{L}$ auf $\mathsf{G}$ abbildet. Zusätzlich zu dieser Bedingung kann es zwei weitere Bedingungen geben, die die Anwendung einer Regel einschränken. Dies sind die Identifikationsbedingung (engl. identification condition) und die Lose-Kanten-Bedingung (engl. dangling edge condition (siehe [Roz97] Seite 189).

Damit eine Regel angewendet werden darf, wird in dieser Arbeit verlangt, dass es einen Teilgraphisomorphismus gibt, der die linke Seite der Regel auf den Anwendungsgraphen abbildet. Rozenberg verlangt in [Roz97] dagegen nur einen Graphhomomorphismus. Bei einem Homomorphismus kann es vorkommen, dass zwei Knoten der linken Regelseite auf einen Knoten des Anwendungsgraphen abgebildet werden. Wenn einer der beiden Knoten durch die Regel gelöscht, der andere jedoch durch die Regel erhalten werden soll, so entsteht ein Konflikt. Nach [Roz97] verlangt die Identifikationsbedingung für einen Knoten, der gelöscht werden soll, dass er auf einen Knoten abgebildet werden muss, auf den kein anderer Knoten abgebildet wird.

**Definition 23.** *(Identifikationsbedingung)*
*Gegeben ist eine Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}]\rightarrow_{\mathsf{r}}\mathsf{R}$*, ein Anwendungsgraph* $\mathsf{G} \in \mathcal{G}$ *und ein Match* $\mathsf{m} : \mathsf{L} \mapsto \mathsf{G}$*, wobei für* $\mathsf{m}$ *gilt* $\nexists\hat{\mathsf{L}} \in \hat{\mathcal{L}}, \nexists\mathsf{m}' : \hat{\mathsf{L}} \mapsto \mathsf{G} \wedge \mathsf{m}' \mid_{\mathsf{L}} = \mathsf{m}$.
*Die Identifikationsbedingung (engl. identification condition) verlangt, dass* $\forall \mathsf{n} \in (\mathsf{N}_{\mathsf{L}} \setminus \mathsf{N}_{\mathsf{R}}), \mathsf{n}' \in \mathsf{N}_{\mathsf{L}} : \mathsf{n} \neq \mathsf{n}' \Rightarrow \mathsf{m}(\mathsf{n}) \neq \mathsf{m}(\mathsf{n}')$.

Die starke Identifikationsbedingung verlangt, dass grundsätzlich keine zwei Knoten der linken Seite auf den selben Knoten des Anwendungsgraphen abgebildet werden dürfen.

**Definition 24.** *(Starke Identifikationsbedingung)*
*Gegeben ist eine Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}]\rightarrow_{\mathsf{r}}\mathsf{R}$*, ein Anwendungsgraph* $\mathsf{G} \in \mathcal{G}$ *und ein Match* $\mathsf{m} : \mathsf{L} \mapsto \mathsf{G}$*, wobei für* $\mathsf{m}$ *gilt* $\nexists\hat{\mathsf{L}} \in \hat{\mathcal{L}}, \nexists\mathsf{m}' : \hat{\mathsf{L}} \mapsto \mathsf{G} \wedge \mathsf{m}' \mid_{\mathsf{L}} = \mathsf{m}$.
*Die starke Identifikationsbedingung verlangt für* $\mathsf{r}$ *und* $\mathsf{m}$*, dass gilt* $\forall \mathsf{n}, \mathsf{n}' \in \mathsf{N}_{\mathsf{L}} : \mathsf{n} \neq \mathsf{n}' \Rightarrow \mathsf{m}(\mathsf{n}) \neq \mathsf{m}(\mathsf{n}')$.

Die Verwendung von Teilgraphisomorphismen zur Beschreibung der Matchings und des Auftretens einer Regel in einem Anwendungsgraphen führt dazu, dass die starke Identifikationsbedingung erfüllt ist. Auf diese Weise wird der oben beschriebene Konflikt beim Löschen eines Elements verhindert.

Auch die Lose-Kanten-Bedingung dient dazu, Konflikte beim Löschen von Knoten zu vermeiden. Diese Bedingung verlangt, dass durch das Löschen von Knoten keine Kanten entstehen, denen der Start- oder Zielknoten fehlt (Kanten, die keinen Start- oder Zielknoten besitzen, werden als lose Kanten[3] bezeichnet). D.h. die Lose-Kanten-Bedingung fordert, dass Kanten, die zu einem zu löschenden Knoten inzident sind, ebenfalls durch die Regel gelöscht werden.

[3] Im Deutschen wird der Begriff meistens mit hängende Kanten übersetzt. In dieser Arbeit werden die Kanten als lose Kanten bezeichnet, da dies ihren Charakter meiner Meinung nach besser beschreibt.

**Definition 25.** *(Lose-Kanten-Bedingung)*

*Gegeben sind ein Anwendungsgraph* $\mathsf{G} \in \mathcal{G}$*, eine Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *und ein Auftreten* $\mathsf{o}$ *mit* $\mathsf{L} \preceq_{\mathsf{o}} \mathsf{G}$*, außerdem gilt* $\forall \hat{\mathsf{L}} \in \hat{\mathcal{L}}, \nexists \mathsf{o}' : \mathsf{o}' \mid_{\mathsf{L}} = \mathsf{o} \wedge \hat{\mathsf{L}} \preceq_{\mathsf{o}'} \mathsf{G}$.

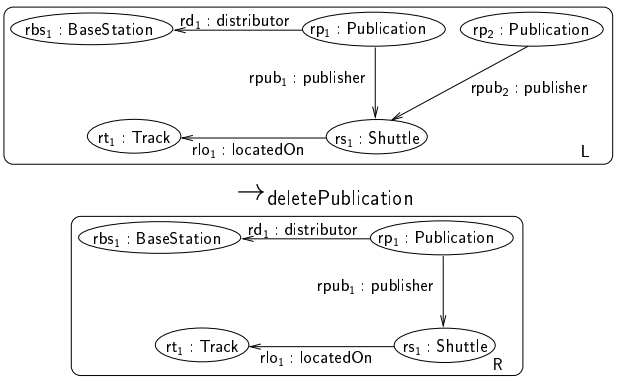
*Die Lose-Kanten-Bedingung verlangt, dass für die Regel* $\mathsf{r}$*, den Anwendungsgraphen* $\mathsf{G}$ *und das Auftreten* $\mathsf{o}$ *gilt* $\nexists \mathsf{e} \in \mathsf{E}_{\mathsf{G}}, \mathsf{e}' \in \mathsf{E}_{\mathsf{L}} : \mathsf{o}_{\mathsf{E}}(\mathsf{e}') = \mathsf{e} \wedge (\mathsf{src}(\mathsf{e}') \in (\mathsf{N}_{\mathsf{L}} \setminus \mathsf{N}_{\mathsf{R}}) \vee \mathsf{tgt}(\mathsf{e}') \in (\mathsf{N}_{\mathsf{L}} \setminus \mathsf{N}_{\mathsf{R}}))$.

Ein Ansatz, in dem sowohl die Identifikationsbedingung als auch die Lose-Kanten-Bedingung erfüllt sein müssen, damit eine Regel angewendet werden darf, ist der Double Pushout Ansatz(DPO, [Roz97] Kapitel 3). Damit die Lose-Kanten-Bedingung erfüllt ist, darf eine Regel nur dann angewendet werden, wenn alle Kanten, die zu einem zu löschenden Knoten inzident sind, ebenfalls von der Regel gelöscht werden. Dem gegenüber steht der Single Pushout Ansatz (SPO, siehe [Roz97] Kapitel 4). Im SPO muss weder die Identifikationsbedingung noch die Lose-Kanten-Bedingung erfüllt sein, damit eine Regel angewendet werden darf. Damit bei der Regelanwendung wieder ein Graph resultiert, der keine losen Kanten enthält, müssen solche Kanten nach der Regelanwendung gelöscht werden. Zusätzlich bietet der Single Pushout Ansatz die Möglichkeit, negative Anwendungsbedingungen zu formulieren, was im Double Pushout Ansatz nach [Roz97] nicht möglich ist. Nach Habel, Heckel und Taentzer [HHT96] können negative Anwendungsbedingungen aber auch im Double Pushout Ansatz verwendet werden.

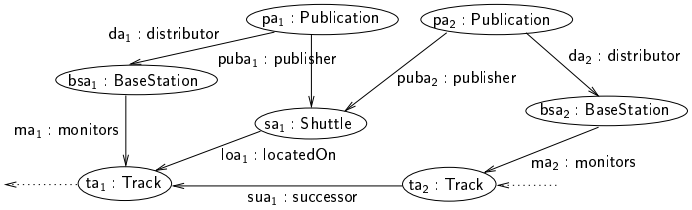
Wenn ein Shuttle zwei Publication-Muster ausführt, so wird durch die Graphtransformationsregel in Abbildung 3.23(a) eine Instanz des Musters gelöscht. Für die zu löschende Instanz ist in der Regel jedoch nicht spezifiziert, mit welcher BaseStation dieses Muster ausgeführt wird. Abbildung 3.23(b) zeigt einen Graphen, auf den die Regel angewendet werden soll. Ein Matching $\mathsf{m} := \langle \mathsf{m}_{\mathsf{N}}, \mathsf{m}_{\mathsf{E}} \rangle$, das die (starke) Identifikationsbedingung erfüllt, sieht folgendermaßen aus: die Funktion $\mathsf{m}_{\mathsf{N}}$ bildet keine zwei Knoten der linken Regelseite auf den selben Knoten des Anwendungsgraphen ab, $\mathsf{m}_{\mathsf{N}} := \{(\mathsf{rbs}_1 \mapsto \mathsf{abs}_1), (\mathsf{rp}_1 \mapsto \mathsf{ap}_1), (\mathsf{rt}_1 \mapsto \mathsf{at}_1), (\mathsf{rs}_1 \mapsto \mathsf{as}_1), (\mathsf{rp}_2 \mapsto \mathsf{ap}_2)\}$. Ebenso bildet die Funktion $\mathsf{m}_{\mathsf{E}}$ keine zwei Kanten der linken Regelseite auf eine Kante des Anwendungsgraphen ab, $\mathsf{m}_{\mathsf{E}} := \{(\mathsf{rd}_1 \mapsto \mathsf{ad}_1), (\mathsf{rpub}_1 \mapsto \mathsf{apub}_1), (\mathsf{rlo}_1 \mapsto \mathsf{alo}_1), (\mathsf{rpub}_2 \mapsto \mathsf{apub}_2)\}$. Die Anwendung der Regel unter Verwendung dieses Matchings würde dazu führen, dass der Knoten $\mathsf{ap}_2$ gelöscht wird. Allerdings wird mit $\mathsf{apub}_2$ nur eine der zu $\mathsf{ap}_2$ inzidenten Kanten gelöscht. Die Kante $\mathsf{ad}_2$ bleibt als lose Kante erhalten. Von daher ist eine Anwendung der Regel im Double Pushout Ansatz nicht erlaubt. Im Single Pushout Ansatz ist die Anwendung erlaubt, allerdings muss die Kante $\mathsf{ad}_2$ noch gelöscht werden.

Ein weiteres Matching $\mathsf{m}' := \langle \mathsf{m}'_{\mathsf{N}}, \mathsf{m}'_{\mathsf{E}} \rangle$, das die Identifikationsbedingung jedoch nicht erfüllt, bildet die beiden Knoten $\mathsf{rp}_1$ und $\mathsf{rp}_2$ der linken Regelseite auf den Knoten $\mathsf{ap}_1$ des Anwendungsgraphen ab, $\mathsf{m}'_{\mathsf{N}} := \{(\mathsf{rp}_1 \mapsto \mathsf{ap}_1), (\mathsf{rp}_2 \mapsto \mathsf{ap}_1), (\mathsf{rs}_1 \mapsto \mathsf{as}_1), (\mathsf{rbs}_1 \mapsto \mathsf{abs}_1), (\mathsf{rt}_1 \mapsto \mathsf{at}_1)\}$. Auch die Funktion $\mathsf{m}'_{\mathsf{E}}$ bildet zwei Kanten der linken Regelseite auf die selbe Kante des Anwendungsgraphen ab, $\mathsf{m}'_{\mathsf{E}} := \{(\mathsf{rpub}_1 \mapsto \mathsf{apub}_1), (\mathsf{rpub}_2 \mapsto \mathsf{apub}_1), (\mathsf{rd}_1 \mapsto \mathsf{ad}_1), (\mathsf{rlo}_1 \mapsto \mathsf{alo}_1)\}$. In diesem Fall ist die Identifikationsbedingung nicht erfüllt und eine Regelanwendung im Double Pushout Ansatz nicht erlaubt. Im Single Pushout Ansatz ist die Anwendung zwar erlaubt, jedoch muss der Konflikt behoben werden, dass der Knoten $\mathsf{ap}_1$ und die Kante $\mathsf{ap}_2$ durch die Regelanwendung sowohl erhalten, als auch gelöscht werden sollen.

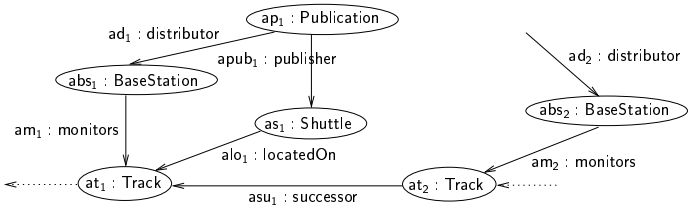
Dadurch, dass im DPO sowohl die Identifikations- als auch die Lose-Kanten-Bedingung erfüllt sind, können Regeln auch rückwärts angewendet werden, was einem Rückgängigmachen




(a) Regel delPublication




(b) Anwendungsgraph auf den die Regel deletePublication im Single bzw. Double Pushout Ansatz angewendet werden soll.




(c) Graph, der resultiert, wenn deletePublication unter dem Match m im Single Pushout Ansatz auf den Anwendungsgraphen angewendet wird. Die lose Kante $da_2$ wurde noch nicht entfernt.

Abbildung 3.23: Regel deletePublication und ihre Anwendung auf einen Graphen im Single bzw. Double Pushout Ansatz

einer Regelanwendung entspricht. Im SPO ist dies nicht immer möglich. Dies gilt zum Beispiel nicht, wenn nach der Anwendung der Regel lose Kanten gelöscht werden. Wird auf den daraus resultierenden Graphen die entsprechende Regel rückwärts angewendet, würden die gelöschten losen Kanten nicht wieder erzeugt, da sie nicht Teil der Regel waren.

In dieser Arbeit sollen Graphtransformationsregeln dazu verwendet werden, die Zustandsübergänge der Software eines mechatronischen Systems zu modellieren. Dazu müssen die Regeln einige Anforderungen erfüllen. Zum einen werden ausdrucksstarke Regeln benötigt, d.h. neben der Anwendungsbedingung muss für eine Regel auch eine negative Anwendungsbedingung spezifiziert werden können. Andererseits bedeutet das Löschen einer Kante in einem mechatronischen System, dass die Verbindung zwischen zwei Objekten gelöscht wird. Um ein versehentliches Löschen einer Verbindung zu verhindern, muss die Regelanwendung die Lose-Kanten-Bedingung erfüllen. Im Verifikationsansatz, der in Abschnitt 3.7 vorgestellt wird, müssen die Regeln auch rückwärts angewendet werden können. Dies ist aber nur möglich, wenn die Anwendung der Regeln sowohl die starke Lose-Kanten- als auch die starke Identifikationsbedingung erfüllt. Deshalb wird in dieser Arbeit der Double Pushout$^{\text{iso}}$ Ansatz verwendet. Dieser Ansatz schränkt den DPO so ein, dass im Gegensatz zum normalen DPO eine Regel nur dann anwendbar ist, wenn das Auftreten o der Regelseite einem Teilgraphisomorphismus entspricht. Zusätzlich zum DPO aus [Roz97] sind auch negative Anwendungsbedingungen erlaubt.

**Definition 26.** *(Double Pushout Ansatz $^{\text{iso}}$ (DPO$^{\text{iso}}$))*

*Eine direkte Transformation von einem Graphen* $\mathsf{AG} \in \mathcal{G}$ *zu einem Graphen* $\mathsf{TG} \in \mathcal{G}$ *(*$\mathsf{AG}$ *und* $\mathsf{TG}$ *sind beschriftungskompatibel) im Double Pushout$^{\text{iso}}$ Ansatz entspricht einer direkten Transformation (siehe Definition 14) mit den folgenden Einschränkungen:*

1. *Jedes korrekte Match einer Regel* $\mathsf{r}$ *erfüllt die Lose-Kanten-Bedingung.*
2. *Für jedes Match* $\mathsf{m}$ *einer Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *in* $\mathsf{AG}$ *gilt, dass* $\mathsf{m}$ *ein Teilgraphisomorphismus ist,* $\mathsf{L} \preceq_{\mathsf{m}} \mathsf{G}$ *und* $\forall \hat{\mathsf{L}} \in \hat{\mathcal{L}}, \nexists \mathsf{m}' \in \mathcal{TISO}[\hat{\mathsf{L}}, \mathsf{AG}] : \mathsf{m}' \mid_{\mathsf{L}} = \mathsf{m} \wedge \hat{\mathsf{L}} \preceq_{\mathsf{m}'} \mathsf{AG}$. *Für alle Auftreten* $\mathsf{o}$ *von* $\mathsf{r}$ *in* $\mathsf{AG}$ *gilt, dass* $\mathsf{o}$ *ein Teilgraphisomorphismus ist.*

Bedingung 2 garantiert, dass die starke Identifikationsbedingung per Konstruktion erfüllt ist. Durch Bedingung 1 der Definition 26 wird erreicht, dass die Regelanwendungen keine losen Kanten erzeugen, wenn der Match der linken Regelseite korrekt ist.

Wann ein gefundener Match korrekt ist, kann durch zusätzliche Graphen in der negativen Anwendungsbedingung beschrieben werden. Dazu wird die negative Anwendungsbedingung der linken Regelseite so erweitert, dass die Regel nur dann angewendet werden kann, wenn alle Kanten, die zu einem zu löschenden Knoten inzident sind, ebenfalls durch die Regel gelöscht werden. Um dies zu erreichen, müssen für jeden zu löschenden Knoten zusätzliche Graphen zur negativen Anwendungsbedingung hinzugefügt werden. In Anhang C.1.1 ist in Abbildung C.1 der Algorithmus extendNAC in Pseudocode-Notation gegeben, der diese Erweiterung durchführt.

Wie in Abbildung 3.23(c) zu sehen ist, kann die Anwendung der Regel deletePublication zu losen Kanten führen. Um dies zu verhindern, muss die Anwendung der Regel durch eine negative Anwendungsbedingung eingeschränkt werden. Die Regel deletePublication löscht den Knoten $\mathsf{rp}_2$ vom Typ Publication. Laut Typgraph kann ein Knoten vom Typ Publication durch eine Kante vom Typ distributor mit einem Knoten vom Typ BaseStation verbunden sein und durch eine Kante vom Typ publisher mit einem Knoten vom Typ Shuttle. Für beide Fälle muss jeweils ein zusätzlicher Graph zur negativen Anwendungsbedingung hinzugefügt werden. Beide zusätzlichen Graphen der negativen Anwendungsbedingung enthalten die Anwendungsbedingung als Teilgraphen. $\hat{\mathsf{L}}_1$ enthält darüber hinaus einen Knoten vom Typ BaseStation, der adjazent zu $\mathsf{rp}_2$

ist. Die Kante zwischen den beiden Knoten hat den Typ distributor. Der andere Graph, $\hat{L}_2$, enthält zusätzlich einen Knoten vom Typ Shuttle. Zwischen diesem neuen Knoten und dem Knoten $rp_2$ verläuft eine Kante vom Typ publisher. Die erweiterte Regel ist in Abbildung 3.24 dargestellt.

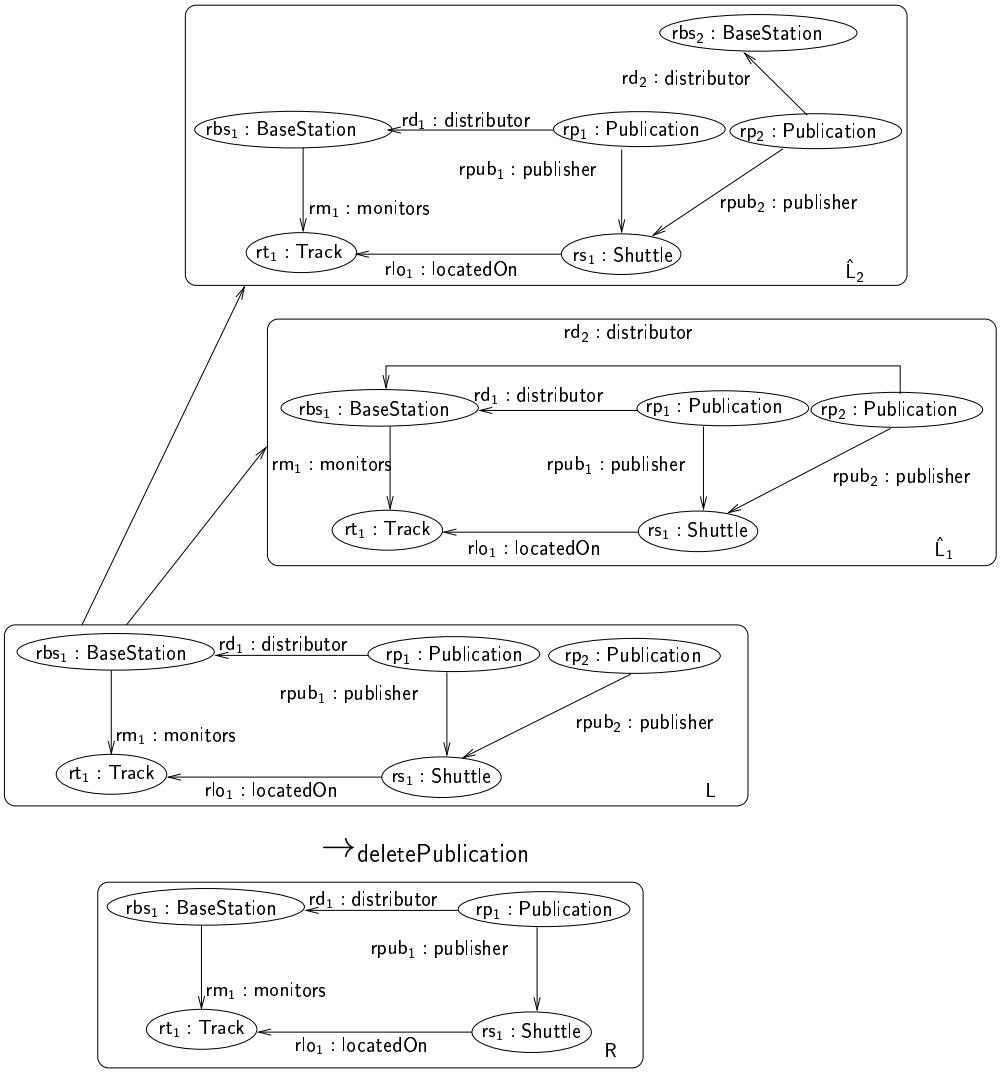



Abbildung 3.24: Graphtransformationsregel deletePublication, die so angepasst wurde, dass ihre Anwendung im $DPO^{iso}$ keine losen Kanten erzeugen kann

Mit dem Algorithmus extendNAC wird die negative Anwendungsbedingung einer Regel um zusätzliche Graphen erweitert, sodass die Regelanwendung die Lose-Kanten-Bedingung einhält. Durch die Anwendung des Algorithmus können jedoch redundante Graphen entstehen. Deshalb wird bei der Regelerweiterung zusätzlich die Funktion minimizeNAC (siehe Abbildung C.2 in Anhang C.1.1) verwendet. Enthält eine negative Anwendungsbedingung zwei Graphen $\hat{G}_1$ und $\hat{G}_2$, mit $\hat{G}_1 \preceq \hat{G}_2$, dann wird $\hat{G}_2$ durch minimizeNAC gelöscht. Dies ist möglich, da es immer, wenn es einen Teilgraphisomorphismus gibt, der $\hat{G}_2$ auf einen Anwendungsgraphen abbildet, auch einen Teilgraphisomorphismus gibt, der $\hat{G}_1$ auf den Anwendungsgraphen abbildet.

**Definition 27.** *(Erweiterte Graphtransformationsregel)*
*Für eine Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *und einen Typgraph* $\mathsf{G}_\Omega$ *wird die negative Anwendungsbedingung der erweiterten Regel* $[\mathsf{L}, \hat{\mathcal{L}}_\mathsf{e}] \rightarrow_{\mathsf{r}_\mathsf{e}} \mathsf{R}$ *gebildet durch* $\hat{\mathcal{L}}_\mathsf{e} := \mathsf{minimizeNAC}(\hat{\mathcal{L}} \cup \mathsf{extendNAC}(\mathsf{L} \rightarrow_{\mathsf{r}} \mathsf{R}, \mathsf{G}_\Omega))$.

Das folgende Lemma 5 zeigt, dass die Anwendung einer erweiterten Graphtransformationsregel aus Definition 27 die Lose-Kanten-Bedingung erfüllt.

**Lemma 5.** *(Einhaltung der Lose-Kanten-Bedingung)*
*Wird eine typkonforme Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *wie in Definition 27 erweitert und die erweiterte Regel auf einen typkonformen Graphen* $\mathsf{G}$ *im DPO*$^{\mathsf{iso}}$ *angewendet, so erfüllt die Regelanwendung die Lose-Kanten-Bedingung.*

### 3.4.3 Typisierte Graphtransformationssysteme

In Abschnitt 3.4.1 wurde gezeigt, wie UML-Modelle auf Graphen abgebildet werden können. Dabei wurde ein Klassendiagramm auf einen Typgraphen und Objektdiagramme auf beschriftete Graphen abgebildet, wobei die Typen erhalten wurden. Im anschließenden Abschnitt 3.4.2 wurden die Übergänge zwischen den Zuständen durch Graphtransformationen beschrieben und garantiert, dass eine Regel nur dann angewendet werden kann, wenn sie die Bedingungen des DPO$^{\mathsf{iso}}$ erfüllt. Ein typisiertes Graphtransformationssystem besteht aus einem Typgraphen, einer Menge von initialen Graphen und einer Menge von Graphtransformationsregeln.

**Definition 28.** *(Typisiertes Graphtransformationssystem)*
*Ein typisiertes Graphtransformationssystem* $\mathsf{S}$ *ist ein Tupel* $\mathsf{S} := (\mathcal{G}[\mathsf{G}_\Omega], \mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$ *mit*

- $\mathcal{G}[\mathsf{G}_\Omega]$ *ist der Typgraph,*
- $(\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$ *ist ein Graphtransformationssystem wie in Definition 17 definiert,*
- $\forall \mathsf{r} \in \mathcal{R}_{\mathsf{S}}$ *gilt:* $\mathsf{r}$ *ist typkonform zu* $\mathsf{G}_\Omega$.

Nach Theorem 1 ist ein derart spezifiziertes Graphtransformationssystem typkonform. Das bedeutet, dass jeder der erreichbaren Zustände typkonform sind.

## 3.5 Erweiterte Graphtransformationen

Bisher wurden Graphtransformationsregeln immer in Vorwärtsrichtung auf Graphen angewendet. Um den Ansatz zur Überprüfung von Graphtransformationssystemen, der in Abschnitt 3.7 eingeführt wird, zu ermöglichen, müssen die Regeln auch rückwärts angewendet werden können. Darüber hinaus müssen die Regeln auch auf Graphmuster anwendbar sein.

Die Transformation von Graphmustern sowie die Rückwärtsanwendung von Regeln werden deshalb in diesem Abschnitt vorgestellt.

### 3.5.1 Rückwärtsanwendung von Graphtransformationsregeln

Bisher wurden Regeln so angewendet, dass die linke Regelseite die Anwendungsbedingung der Regel beschrieben hat und die rechte Regelseite die Nachbedingung. Elemente, die Teil der linken, aber nicht der rechten Regelseite sind, werden bei der Regelanwendung gelöscht. Elemente, die zur rechten aber nicht zur linken Regelseite gehören, werden durch eine Regelanwendung neu erzeugt.

Im Folgenden sollen Graphtransformationsregeln auch rückwärts angewendet werden können, d.h. die rechte Regelseite stellt die Anwendungsbedingung dar und die linke Regelseite die Nachbedingung. Dementsprechend werden Elemente, die durch die Vorwärtsanwendung erzeugt werden, bei der Rückwärtsanwendung gelöscht. Solche Elemente, die in der Vorwärtsrichtung gelöscht werden, werden in der Rückwärtsrichtung neu erzeugt.

Wendet man eine Regel $r$ auf einen Graphen $G$ an, so erhält man einen Graphen $G'$. Wird auf $G'$ die Regel $r$ rückwärts angewendet, so resultiert wieder der ursprüngliche Graph $G$. Die Rückwärtsanwendung von $r$ hat somit die Vorwärtsanwendung von $r$ rückgängig gemacht.

Damit eine Regel $[L, \hat{\mathcal{L}}] \rightarrow_r R$ im DPO$^{iso}$ korrekt rückwärts angewendet werden kann, muss ihre negative Anwendungsbedingung angepasst werden. Die negative Anwendungsbedingung muss aus der linken Regelseite entfernt werden. Dies ist notwendig, da bei der Rückwärtsanwendung aus der linken Regelseite die Nachbedingung der Regel wird und im DPO$^{iso}$ die Nachbedingung keine negative Anwendungsbedingung haben darf. Die zweite Modifikation besteht darin, dass eine negative Anwendungsbedingung zu $R$ hinzugefügt werden muss. Dadurch wird erreicht, dass die Regel nur dann rückwärts auf einen Graphen anwendbar ist, wenn dieser Graph durch Anwendung der Regel in Vorwärtsrichtung aus einem anderen Graphen entstanden sein kann.

Die Anpassung der negativen Anwendungsbedingung erfolgt mittels des Algorithmus convertNAC (siehe Abbildung C.4 in Anhang C.2.1). Der Algorithmus bestimmt alle Graphen der negativen Anwendungsbedingung, die die Regelanwendung verhindern, wenn sonst lose Kanten erzeugt werden. Diese Graphen, so wie alle, auf die ein solcher Graph abgebildet werden kann, werden aus der negativen Anwendungsbedingung entfernt. Auf die verbleibenden Graphen wird jeweils die Regel $L \rightarrow_r R$ angewendet, d.h. die negative Anwendungsbedingung von $r$ wird nicht berücksichtigt. Die resultierenden Graphen bilden einen Teil der neuen negativen Anwendungsbedingung. Durch das Löschen von Graphen aus $\hat{\mathcal{L}}$ zu Beginn der Anpassung wird erreicht, dass nur solche Graphen erhalten bleiben, auf die die Regel $L \rightarrow_r R$ angewendet werden kann, ohne dass die resultierenden Graphen lose Kanten enthalten. In einem letzten Schritt wird $\hat{\mathcal{L}}^{-1}$, die negative Anwendungsbedingung der rechten Regelseite, um Graphen erweitert, die eine Rückwärtsanwendung der Regel verhindern, wenn ansonsten lose Kanten entstehen. Dies ist aus dem folgenden Grund notwendig. Kanten, die bei der Rückwärtsanwendung gelöscht werden, sind solche, die bei der Vorwärtsanwendung erzeugt werden. Damit eine Kante bei der Rückwärtsanwendung zu einer losen Kante werden kann, muss einer ihrer Knoten durch die Regel gelöscht werden, d.h. der Knoten ist Teil von $R$, aber nicht von $L$. Die Kante selber darf jedoch nicht Teil von $R$ sein. In diesem Fall würde sie aber auch nicht bei der Vorwärtsanwendung von $r$ erzeugt.

In Abbildung 3.25 ist die Regel moveSimpleNAC dargestellt. Die negative Anwendungsbedingung der Regel besagt, dass das Shuttle nur dann von einem Track zum nächsten weiterfahren darf, wenn sich weder auf dem nächsten noch auf dem übernächsten Track ein anderes Shuttle befindet. Um die Graphen dieser negativen Anwendung zu invertieren, wird die Regel darauf angewendet. Abbildung 3.26 zeigt die Regel mit invertierter negativer Anwendungsbedingung.

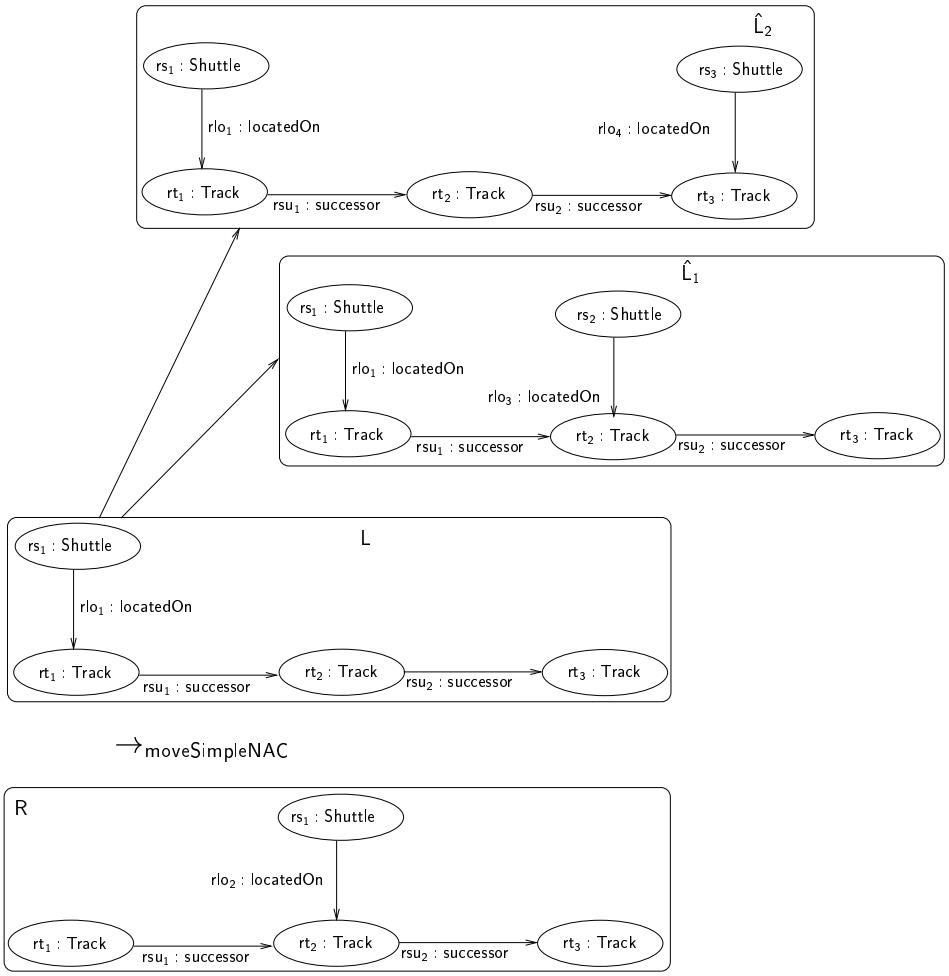



Abbildung 3.25: Regel moveSimpleNAC

Die Invertierung einer Graphtransformationsregel wird folgendermaßen durchgeführt (der entsprechende Algorithmus reverse ist in Abbildung C.5 in Anhang C.2.1 abgebildet): Zunächst wird die negative Anwendungsbedingung mittels convertNAC transformiert. Um redundante Graphen in der negativen Anwendungsbedingung auszuschließen, wird die negative Anwendungsbedingung mittels minimizeNAC minimiert. Anschließend werden die linke und die rechte Regelseite vertauscht, sodass die rechte Regelseite zusammen mit der invertierten negativen Anwendungsbedingung die Anwendungsbedingung der invertierten Regel darstellt und die linke Regelseite der ursprünglichen Regel als Nachbedingung der invertierten Regel fungiert.

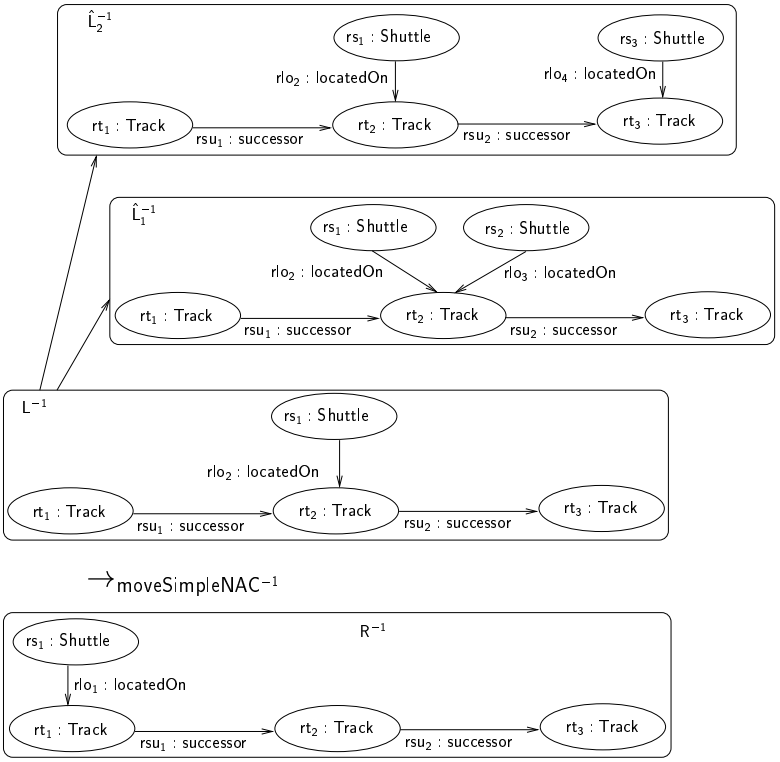



Abbildung 3.26: Regel moveSimpleNAC$^{-1}$ stellt die inverse Regel moveSimpleNAC dar

Definition 29 verwendet den Algorithmus reverse (siehe Abbildung C.5 in Anhang C.2.1), um die inverse Regel $r^{-1}$ einer Regel r zu definieren.

**Definition 29.** *(Invertierte Graphtransformationsregel)*
*Die inverse Regel* $[L^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{r^{-1}} R^{-1}$ *einer Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *im DPO*$^{iso}$ *ist definiert als* $r^{-1} = \text{reverse}(r, G_\Omega)$.

Die in Abbildung 3.26 dargestellte Regel moveSimpleNAC$^{-1}$ entspricht der inversen Regel von moveSimpleNAC. Dabei entspricht der Graph $L^{-1}$ der ursprünglichen rechten Regelseite und $R^{-1}$ der ursprünglichen linken Regelseite. Da keiner der beiden Graphen aus der negativen Anwendungsbedingung durch einen Teilgraphisomorphismus auf einen Teilgraphen des anderen abgebildet werden kann, ist die negative Anwendungsbedingung minimal.

Gegeben sind zwei Graphen $G, G' \in \mathcal{G}$, wobei $G'$ resultiert, wenn die Graphtransformationsregel r auf G angewendet wird, d.h. $G \models\!\Rightarrow_r G'$. Ist $r^{-1}$ das Inverse von r, so resultiert die Anwendung von $r^{-1}$ auf $G'$ wieder in G, d.h. $G' \models\!\Rightarrow_{r^{-1}} G$. Die Anwendung der inversen Regel hat also die Anwendung der ursprünglichen Regel rückgängig gemacht. Diese Eigenschaft inverser Regeln ist in Theorem 2 festgehalten.

**Theorem 2.** *(Rückwärtsanwendung einer Graphtransformationsregel im DPO*$^{iso}$*)*
*Für jede Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$*, ihr Inverses* $[L^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{r^{-1}} R^{-1}$ *und ein Auftreten* o *von* r *in* $G_1$ *gilt im DPO*$^{iso}$*:*
$G_1 \models\!\Rightarrow_{(r,o)} G_2 \Leftrightarrow G_2 \models\!\Rightarrow_{(r^{-1},o^{-1})} G_1$.

### 3.5.2 Transformationen von Graphmustern

Graphmuster können dazu verwendet werden, um Mengen von Graphen zu beschreiben. Soll für diese Graphmengen geprüft werden, welche Auswirkung die Anwendung einer bestimmten Graphtransformationsregel hat, so ist es notwendig, die Regel statt auf jeden der Graphen auf das entsprechende Graphmuster anzuwenden. Die Anwendung einer Regel $\mathsf{r}$ unter dem Auftreten $\mathsf{o}$ auf ein Graphmuster $\mathsf{p}$ resultiert wieder in einem Graphmuster, $\mathsf{p}'$. Wird die Regel unter $\mathsf{o}$ statt auf $\mathsf{p}$ auf einen Graphen angewendet, der $\mathsf{p}$ erfüllt, also in der Menge $\mathcal{G}[\mathsf{p}]$ enthalten ist, so resultiert die Anwendung in einem Graphen, der $\mathsf{p}'$ erfüllt und somit zur Menge $\mathcal{G}[\mathsf{p}']$ gehört.

Um eine Graphtransformationsregel $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ auf ein Graphmuster $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ anwenden zu können, muss die Regelanwendung etwas modifiziert werden. Die linke Regelseite muss immer durch ein Muster beschrieben werden, wobei die Menge $\hat{\mathcal{L}}$ leer sein darf. Die Regel ist anwendbar, wenn das Muster der linken Regelseite ein Teilgraphmuster des zu verändernden Graphmusters ist. Ist diese Bedingung erfüllt, so wird die Regel $\mathsf{L} \rightarrow_{\mathsf{r}'} \mathsf{R}$ auf den Graphen $\mathsf{P}$ angewendet. Die negative Anwendungsbedingung $\hat{\mathcal{P}}$ des Graphmusters wird durch den Aufruf der Methode convertNAC (s.o. und Abbildung C.4 in Anhang C.2.1) transformiert.

**Definition 30.** *(Direkte Transformationen von Graphmustern im DPO$^{\mathsf{iso}}$)*
*Für eine Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *und ein Graphmuster* $\mathsf{p} = [\mathsf{P}, \hat{\mathcal{P}}]$*, wobei* $[\mathsf{L}, \hat{\mathcal{L}}]$ *ein Teilgraphmuster von* $\mathsf{p}$ *ist (*$[\mathsf{L}, \hat{\mathcal{L}}] \subseteq [\mathsf{P}, \hat{\mathcal{P}}]$*), gilt im DPO$^{\mathsf{iso}}$, dass die Anwendung von* $\mathsf{r}$ *auf* $\mathsf{p}$ *in einem Graphmuster* $\mathsf{p}' = [\mathsf{P}, \hat{\mathcal{P}}']$ *resultiert mit* $\mathsf{P} \models\!\Rightarrow_{\mathsf{r}} \mathsf{P}'$ *und* $\hat{\mathcal{P}}' = \mathsf{convertNAC}([\mathsf{P}, \hat{\mathcal{P}}], [\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}, \mathsf{G}_\Omega)$.

Im Folgenden wird die Schreibweise $\mathsf{p} \nRightarrow_{\mathsf{r}} \mathsf{p}'$ verwendet, wenn die Anwendung der Regel $\mathsf{r}$ auf ein Graphmuster $\mathsf{p}$ in einem Graphmuster $\mathsf{p}'$ resultiert.

Theorem 3 zeigt, dass Regelanwendungen, die auf Mustern möglich sind, auch auf Graphen möglich sind, die diese Muster erfüllen. Außerdem wird gezeigt, dass die Anwendung einer Regel auf ein Muster den gleichen Effekt hat wie bei der Anwendung auf einen Graphen, der dieses Muster erfüllt.

**Theorem 3.** *(Transformation von Graphmustern)*
*Für eine Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ *und zwei Graphmuster* $\mathsf{p} := [\mathsf{P}, \mathsf{hat}\mathcal{P}], \mathsf{p}' := [\mathsf{P}', \hat{\mathcal{P}}']$ *gilt im DPO$^{\mathsf{iso}}$:*

$$
\begin{aligned}
&(\mathsf{p} \nRightarrow_{\mathsf{r}} \mathsf{p}') \\
&\Leftrightarrow \\
&(\forall \mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}, \mathsf{iso} \in \mathcal{ISO} : \mathsf{P} \cup \mathsf{P}' \rightarrow \mathsf{G}_1 \cup \mathsf{G}_2 : (\mathsf{G}_1 \models \mathsf{p} \wedge \mathsf{G}_1 \models\!\Rightarrow_{\mathsf{r},\mathsf{iso}|_{\mathsf{L}\cup\mathsf{R}}} \mathsf{G}_2) \Rightarrow (\mathsf{G}_2 \models \mathsf{p}')).
\end{aligned}
$$

In Theorem 2 wurde für eine Regel $\mathsf{r}$ und ihr Inverses $\mathsf{r}^{-1}$ gezeigt, dass die Anwendung von $\mathsf{r}$ auf einen Graphen $\mathsf{G}$ durch die Anwendung von $\mathsf{r}^{-1}$ auf den durch $\mathsf{r}$ erzeugten Graphen wieder in $\mathsf{G}$ resultiert. Unter Verwendung der Theoreme 2 und 3 kann gezeigt werden, dass dieses Ergebnis auch für Graphmuster gilt.

**Theorem 4.** *(Rückwärtsanwendung von Regeln auf Graphmuster)*
*Für eine Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$*, ihr Inverses* $[\mathsf{L}^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{\mathsf{r}^{-1}} \mathsf{R}^{-1}$*, zwei Graphmuster* $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ *und* $\mathsf{p}' := [\mathsf{P}', \hat{\mathcal{P}}']$ *und ein Auftreten* $\mathsf{o} : \mathsf{L} \cup \mathsf{R} \mapsto \mathsf{P} \cup \mathsf{P}'$ *gilt im DPO*$^{\mathsf{iso}}$*:*

$$(\mathsf{p} \Longmapsto_{(\mathsf{r},\mathsf{o})} \mathsf{p}') \Leftrightarrow (\mathsf{p}' \Longmapsto_{(\mathsf{r}^{-1},\mathsf{o}^{-1})} \mathsf{p}).$$

## 3.6 Systemeigenschaften und Invarianten

In Abschnitt 3.3 wurden Graphmuster eingeführt. Diese Graphmuster sollen dazu verwendet werden, strukturelle Eigenschaften eines Graphtransformationssystems zu beschreiben.

Die Idee, Grapheigenschaften mittels einfacher Graphmuster zu formulieren, haben Rensink et al. in [RSV04] beschrieben. Sie nennen die Graphmuster Eigenschaftsgraphen und haben gezeigt, dass ihre Ausdrucksstärke äquivalent zum $\exists$ $\not\exists$-Teil von $\forall$-freien logischen Formeln erster Ordnung mit binären Prädikaten ist.

Graphmuster können entweder gefordert oder verboten sein. Geforderte Graphmuster beschreiben Eigenschaften, die immer erfüllt sein müssen. Dem gegenüber stehen die verbotenen Graphmuster; sie beschreiben Eigenschaften, die nie erfüllt sein dürfen.

In Grapheigenschaftsformeln stellen die Graphmuster die atomaren Eigenschaften dar.

**Definition 31.** *(Grapheigenschaftsformel)*
*Eine Grapheigenschaftsformel* $\phi$ *ist entweder*

- *eine atomare Eigenschaft, die durch ein Graphmuster* $\mathsf{p}$ *beschrieben und als gefordertes Graphmuster bezeichnet wird oder*
- *eine atomare Eigenschaft, die durch das negierte Graphmuster* $\neg\mathsf{p}$ *des Graphmusters* $\mathsf{p}$ *definiert ist. Diese Eigenschaft wird als verbotenes Graphmuster bezeichnet oder*
- *eine zusammengesetzte Eigenschaft, gegeben durch* $\phi \wedge \psi$ *oder* $\phi \vee \psi$*, wobei* $\phi$ *und* $\psi$ *Grapheigenschaftsformeln sind.*

Wann eine solche Grapheigenschaftsformel von einem Graphen erfüllt ist, ist folgendermaßen definiert:

**Definition 32.** *(Semantik von Grapheigenschaftsformeln)*
*Ob ein Graph* $\mathsf{G} \in \mathcal{G}$ *eine Grapheigenschaftsformel* $\theta$ *erfüllt, ist über die Struktur der Formel, bestehend aus ihren atomaren Eigenschaften* $\mathsf{p_i} := [\mathsf{P_i}, \hat{\mathcal{P}}_\mathsf{i}]$ *und Teilformeln* $\phi$ *und* $\psi$*, folgendermaßen definiert:*

- $\mathsf{G} \models \mathsf{p}$ *gdw.* $\exists \mathsf{tiso} \in \mathcal{TISO}[\mathsf{P_i}, \mathsf{G}] : \mathsf{P_i} \preceq_{\mathsf{tiso}} \mathsf{G}$ *und* $\mathsf{G}, \mathsf{P_i}, \mathsf{tiso} \vdash \hat{\mathsf{P}}_\mathsf{i}$,
- $\mathsf{G} \models \neg\phi$ *gdw.* $\mathsf{G} \not\models \phi$,
- $\mathsf{G} \models \phi \wedge \psi$ *gdw.* $\mathsf{G} \models \phi$ *und* $\mathsf{G} \models \psi$ *und*
- $\mathsf{G} \models \phi \vee \psi$ *gdw.* $\mathsf{G} \models \phi$ *oder* $\mathsf{G} \models \psi$.

Für eine Menge $\mathcal{G}$ von Graphen und eine Menge $\mathcal{R}$ von Graphtransformationsregeln können die Eigenschaften dazu verwendet werden, die Graphen und Transformationen in verschiedene Mengen zu unterteilen. Bei den Graphen bezeichnet beispielsweise $\mathcal{G}[\phi]$ die Menge aller Graphen, die die Eigenschaft $\phi$ erfüllen, d.h. $\mathcal{G}[\phi] := \{\mathsf{G} \in \mathcal{G} \mid \mathsf{G} \models \phi\}$. Überführt eine Graphtransformation einen Graphen, der $\phi$ erfüllt, in einen Graphen, der $\psi$ erfüllt, so gehört die Transformation zur Menge $\mathcal{T}[\phi, \psi]$. Diese Menge ist definiert als $\mathcal{T}[\phi, \psi] := \{\mathsf{r} \in \mathcal{R} \mid \exists \mathsf{G}_\phi \in \mathcal{G}[\phi], \mathsf{G}_\psi \in \mathcal{G}[\psi] : \mathsf{G}_\phi \Longrightarrow_{\mathsf{r}} \mathsf{G}_\psi\}$.

Ein Systemzustand, gegeben als (typisierter) Graph, ist korrekt im Bezug auf eine Systemeigenschaftsformel $\phi$, wenn gilt $\mathsf{G} \models \phi$, d.h. geforderte Graphmuster sind in $\mathsf{G}$ enthalten und verbotene Graphmuster sind nicht in $\mathsf{G}$ enthalten. Das gesamte System $\mathsf{S} := (\mathcal{G}[\mathsf{G}_\Omega], \mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$ ist korrekt ($\mathsf{S} \models \phi$), wenn alle Startgraphen aus $\mathcal{G}^{\mathsf{i}}_{\mathsf{S}}$ sowie alle erreichbaren Graphen von $\mathsf{S}$ korrekt sind. Ist $\mathsf{S}$ korrekt bezüglich $\phi$, so ist $\phi$ eine operationale Invariante des Systems. Die formale Definition von operationalen Invarianten stammt aus [Cha03] und lautet[4]:

**Definition 33.** *(Operationale Invariante)*
*Eine Eigenschaft $\phi$ ist eine operationale Invariante eines Graphtransformationssystems $\mathsf{S} := (\mathcal{G}[\mathsf{G}_\Omega], \mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$ wenn gilt:*

- $\forall \mathsf{G}^{\mathsf{i}} \in \mathcal{G}^{\mathsf{i}}_{\mathsf{S}} : \mathsf{G}^{\mathsf{i}} \models \phi$ *und*
- $\forall \mathsf{G} \in \mathsf{REACH}(\mathsf{S}) : \mathsf{G} \models \phi$.

Die Überprüfung operationaler Invarianten ist jedoch für den allgemeinen Fall aus verschiedenen Gründen nicht möglich. Ein Grund dafür ist, dass die hier betrachteten Systeme unendlich viele Zustände haben können. Ein zweiter Grund ist, dass die Menge der Initialgraphen zum Zeitpunkt der Verifikation noch nicht bekannt sein muss, bzw. dass sie sich noch verändern kann. Sobald die Verifikation abgeschlossen ist, kann jedoch nur dann ein neuer Startgraph zur Menge hinzugefügt werden, wenn auch ausgehend von diesem alle erreichbaren Graphen korrekt sind, d.h. eine weitere Erreichbarkeitsanalyse ist erforderlich. Dadurch entsteht ein erheblicher Aufwand.

Aus diesen Gründen sollen im Folgenden statt der operationalen Invarianten induktive Invarianten betrachtet werden. Bei der Überprüfung, ob eine Eigenschaft eine induktive Invariante ist, wird die Erreichbarkeit eines Zustands außer Acht gelassen. Stattdessen wird überprüft, ob jeder korrekte Zustand durch die Anwendung einer beliebigen Regel wieder in einen korrekten Zustand überführt wird.

**Definition 34.** *(Induktive Invariante)*
*Eine Eigenschaft $\phi$ ist eine induktive Invariante eines Systems $\mathsf{S} := (\mathcal{G}[\mathsf{G}_\Omega], \mathcal{G}^{\mathsf{i}}_{\mathsf{S}}, \mathcal{R}_{\mathsf{S}})$, falls gilt: $\forall \mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}[\mathsf{G}_\Omega], \mathsf{r} \in \mathcal{R}_{\mathsf{S}} : (\mathsf{G}_1 \Longrightarrow_{\mathsf{r}} \mathsf{G}_2 \wedge \mathsf{G}_1 \models \phi) \Rightarrow (\mathsf{G}_2 \models \phi)$.*

Ist $\phi$ eine induktive Invariante und wird $\phi$ zusätzlich von allen Startgraphen erfüllt, d.h. $\forall \mathsf{G}^{\mathsf{i}} \in \mathcal{G}^{\mathsf{i}}_{\mathsf{S}} : \mathsf{G}^{\mathsf{i}} \models \phi$, dann ist $\phi$ auch eine operationale Invariante. Umgekehrt ist jedoch nicht jede operationale Invariante auch eine induktive.

[4]Die Bezeichnungen der Eigenschaften sind in der Literatur nicht eindeutig. So werden beispielsweise die operationalen Invarianten von Kindler in [Kin95] als invariante Eigenschaften bezeichnet und die induktiven Invarianten als induktive Eigenschaften.

Um zu zeigen, dass $\phi$ eine induktive Invariante eines Graphtransformationssystems $\mathsf{S} := (\mathcal{G}[\mathsf{G}_\Omega], \mathcal{G}^i_\mathsf{S}, \mathcal{R}_\mathsf{S})$ ist, muss gezeigt werden, dass die Anwendung einer Regel $\mathsf{r} \in \mathcal{R}_\mathsf{S}$ auf einen beliebigen korrekten Graphen $\mathsf{G}_\phi \in \mathcal{G}[\phi]$ einen korrekten Graphen $\mathsf{G}'_\phi \in \mathcal{G}[\phi]$ erzeugt. Das bedeutet, $\phi$ ist genau dann eine induktive Invariante von $\mathsf{S}$, wenn es keine Transformation gibt, die einen Graphen $\mathsf{G}_\phi \in \mathcal{G}[\phi]$ in einen Graphen $\mathsf{G}_{\neg\phi} \in \mathcal{G}[\neg\phi]$ transformiert und somit die Menge $\mathcal{T}[\phi, \neg\phi]$ leer ist. Dieses Ergebnis ist in Lemma 6 festgehalten.

**Lemma 6.** *(Eigenschaften induktiver Invarianten)*
*Eine Eigenschaft $\phi$ ist eine induktive Invariante eines Systems $\mathsf{S} := (\mathsf{G}_\Omega, \mathcal{G}^i_\mathsf{S}, \mathcal{R}_\mathsf{S})$, falls $\mathcal{T}[\phi, \neg\phi] = \emptyset$.*

### 3.6.1 Sicherheitseigenschaften

Eine spezielle Art von Grapheigenschaftsformeln, die im Folgenden eine besondere Rolle spielen, sind die Sicherheitseigenschaften. Eine solche Sicherheitseigenschaft $\phi_\mathsf{S}$ wird durch eine Menge von verbotenen Graphmustern beschrieben, wobei jedes dieser verbotenen Graphmuster einen Unfall oder eine kritische Situation des Systems beschreibt. Die Normalform einer Sicherheitseigenschaft sieht folgendermaßen aus:

$$\phi_\mathsf{s} \quad := \bigwedge_{\mathsf{k} \in \mathsf{K}} (\neg \mathsf{p}_\mathsf{k}).$$

Ein Beispiel für eine solche Sicherheitseigenschaft ist $\phi_\mathsf{s} := \neg$impendingCollision, wobei das Graphmuster impendingCollision dem in Abbildung 3.27 gegebenen Graphmuster entspricht. Die Eigenschaft beschreibt, dass sich niemals zwei Shuttles auf benachbarten Tracks befinden dürfen, wenn sie nicht eine Instanz des DistanceCoordination-Musters ausführen.

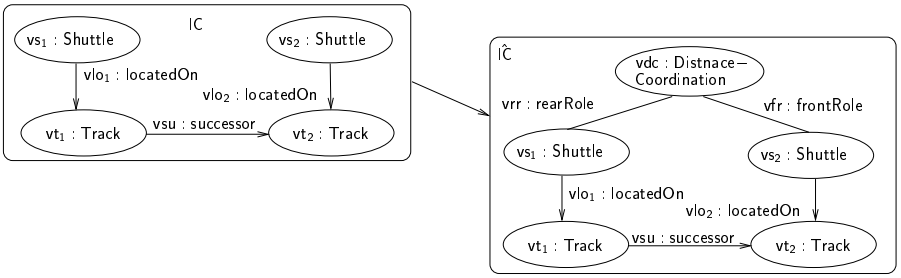



Abbildung 3.27: Sicherheitseigenschaft impendingCollision

Jede so dargestellte Sicherheitseigenschaft kann auch durch $\neg\phi_\mathsf{S} = \bigvee_{\mathsf{j} \in \mathsf{J}} (\mathsf{p}_\mathsf{j})$ ausgedrückt werden. Um diese Formel zu erfüllen, genügt ein einzelnes lokales Auftreten einer solchen kritischen Situation, spezifiziert durch ein verbotenes Graphmuster. Ein solches Auftreten eines verbotenen Graphmusters wird im Folgenden als Zeuge bezeichnet.

### 3.6.2 Beschränkung auf Gegenbeispiele

Um zu zeigen, dass eine Sicherheitseigenschaft der Form $\neg\phi := \bigvee_{j\in J} p_j$ keine induktive Invariante eines Systems ist, ist es ausreichend eine Transformation zu finden, die zur Menge $\mathcal{T}[\phi, \neg\phi]$ gehört. Das bedeutet, es existiert ein Graph $G_{\neg\phi}$, der eines der verbotenen Graphmuster $p_i$ der Sicherheitseigenschaft enthält ($G_{\neg\phi} \models p_i$) und dieser Graph ist das Resultat der Anwendung einer Regel $r$ auf einen Graphen $G_\phi$, der keines der verbotenen Graphmuster enthält ($\forall j \in J : G_\phi \not\models p_j$). In diesem Fall bilden die beiden Graphen $G_\phi$ und $G_{\neg\phi}$ ein Gegenbeispiel, das zeigt, dass die angewendete Regel die Eigenschaft $\phi$ verletzen kann. Für die Überprüfung der Korrektheit eines Systems bedeutet dies, dass die Verifikation abgebrochen werden kann, sobald ein solches Gegenbeispiel gefunden wurde; dies ist in Lemma 7 festgehalten.

**Lemma 7.** *(Zeugen widerlegen Sicherheitseigenschaften)*
*Für eine Sicherheitseigenschaft der Form $\bigwedge_{j\in J}(\neg p_j)$ und ein System $S := (\mathcal{G}[G_\Omega], \mathcal{G}^i_S, \mathcal{R}_S)$ ist die Menge $\mathcal{T}[\phi, \neg\phi]$ genau dann nicht leer und somit das System inkorrekt, wenn gilt:*

$$\exists G_\phi, G_{\neg\phi} \in \mathcal{G}[G_\Omega], r \in \mathcal{R}_S, o \in \mathcal{ISO} : \\ (G_\phi \Longrightarrow_{(r,o)} G_{\neg\phi}) \wedge (\exists i \in J : G_{\neg\phi} \models p_i) \wedge (\forall j \in J : G_\phi \not\models p_j). \quad (3.5)$$

## 3.7 Nachweis von induktiven Invarianten

Unter Zuhilfenahme der bisher vorgestellten Ergebnisse kann jetzt gezeigt werden, dass es ausreicht, eine endliche Menge von Repräsentanten zu betrachten, um nachzuweisen, dass eine Sicherheitseigenschaft eine induktive Invariante eines Graphtransformationssystems ist. Die theoretischen Ergebnisse werden in einem Algorithmus umgesetzt, der automatisch für eine Eigenschaft überprüft, ob die Eigenschaft eine induktive Invarianten des Systems ist (siehe dazu Abschnitt 5.3).

### 3.7.1 Theoretische Ergebnisse

Für den Nachweis, dass eine Sicherheitseigenschaft $\phi_S$ eine induktive Invariante eines Graphtransformationssystems ist, wurde in Lemma 7 gezeigt, dass die Menge der Transformationen, die einen korrekten Zustand in einen inkorrekten überführen, leer sein muss. Um nachzuweisen, dass diese Menge leer ist, müssten jedoch unter Umständen unendliche viele Transformationen betrachtet werden. Im Folgenden wird das Ergebnis aus Lemma 7 ausgenutzt, um mittels einer endlichen minimalen Menge von Repräsentanten die Systemkorrektheit nachzuweisen.

Als Repräsentanten dienen dabei die in Definition 19 eingeführten Graphmuster. Mit Hilfe der Graphmuster wird für jede der Graphtransformationsregeln einzeln geprüft, ob sie korrekt ist.

Eine wichtige Voraussetzung, um diese Art der Überprüfung durchführen zu können, ist, dass die Anwendung einer Graphtransformationsregel nur Auswirkungen auf den Teil des Graphen hat, auf den sie durch den Match abgebildet wird. Deshalb ist es nicht notwendig, vollständige Graphen zu überprüfen. Stattdessen reicht es aus, wenn nur der Ausschnitt des Anwendungs-

und Zielgraphen betrachtet wird, der durch die Regelanwendung verändert wird und dadurch die Gültigkeit der Eigenschaft beeinflussen kann. Die Anwendung einer Regel $r$ kann nur dann die Gültigkeit der Eigenschaft beeinflussen, wenn sie einen Zeugen für eines der verbotenen Graphmuster erzeugt.

Der Teil des Anwendungsgraphen, der durch die Regelanwendung verändert wird, also auf den die linke Regelseite abgebildet wird, kann als Graphmuster kodiert werden. Ebenso ist es möglich, den Teil des Zielgraphen, der sich durch die Regelanwendung verändert hat, d.h. auf den die rechte Regelseite abgebildet werden kann, als Graphmuster zu kodieren. Verwendet man solche Graphmuster statt vollständiger Graphen, so kann die Frage, ob die Menge $\mathcal{T}[\phi_S, \neg\phi_S]$ leer und das System somit korrekt bezüglich $\phi_S$ ist, lokal beantwortet werden.

Die Idee besteht darin, ein Graphmuster, das so genannte Ergebnisgraphmuster, so zu wählen, dass es als Teilgraphmuster ein verbotenes Graphmuster enthält und somit inkorrekt ist. Für dieses Ergebnisgraphmuster wird geprüft, ob es das Ergebnis einer Regelanwendung auf ein korrektes Graphmuster ist. Dazu wird die Regel invertiert und das Inverse auf das Ergebnisgraphmuster angewendet. Das resultierende Graphmuster wird als Startgraphmuster bezeichnet. Ist dieses Startgraphmuster korrekt, so konnte ein Gegenbeispiel gefunden werden, das zeigt, dass die Regel ein korrektes Graphmuster in ein inkorrektes überführen kann. Dies bedeutet aber auch, dass ein korrekter Graph, der das Startgraphmuster als Teilgraphmuster enthält, durch die betrachtete Regel in einen inkorrekten Graphen transformiert wird.

Ein Ergebnisgraphmuster erfüllt zum einen ein verbotenes Graphmuster $p_j := [P_j, \hat{\mathcal{P}}_j]$ und enthält zum anderen die rechte Seite einer Graphtransformationsregel $r$. Damit die Anwendung von $r$ das verbotene Graphmuster $p_j$ erzeugt haben kann, muss entweder $P_j$ oder ein $\hat{P}_j \in \hat{\mathcal{P}}_j$ durch die Regel beeinflusst werden. Im ersten Fall muss durch die Regel mindestens ein Element erzeugt werden, das Teil von $P_j$ ist. Im zweiten Fall gibt es vor der Regelanwendung einen Teilgraphisomorphismus der $P_j$ auf das Anwendungsgraphmuster abbildet. Zusätzlich kann dieser Teilgraphisomorphismus so erweitert werden, dass er auch einen Graphen $\hat{P}_j$ aus der negativen Anwendungsbedingung von $p_j$ auf die Anwendungsbedingung des Anwendungsgraphmusters abbildet. Die Regelanwendung löscht dann ein Element aus $\hat{P}_j \setminus P_j$, sodass nach der Regelanwendung ein Teilgraphisomorphismus existiert, der $P_j$ auf den resultierenden Graphen abbildet. Dieser Teilgraphisomorphismus kann aber nicht so erweitert werden, dass er auch $\hat{P}_j$ auf das resultierende Graphmuster abbildet.

Für den zweiten Fall gilt: Ist das gelöschte Element eine Kante, so bleiben ihre inzidenten Knoten bei der Regelanwendung erhalten. Diese Knoten sind dann sowohl Teil der linken als auch der rechten Regelseite und auch in $\hat{P}_j$ enthalten. Wird ein Knoten gelöscht, so müssen alle seine inzidenten Kanten ebenfalls explizit durch die Regel gelöscht werden. Für jeden Knoten $n \in N_{\hat{P}_j} \setminus N_{P_j}$ gilt nach Definition 15, dass er über einen Pfad $\pi$ mit einem Knoten $n' \in P_j$ verbunden ist. Da die Regelanwendung keinen Einfluss auf $P_j$ hat, muss $n'$ auch nach der Regelanwendung erhalten sein. Ist $n$ adjazent zu $n'$, so wird $n'$ explizit durch die Regelanwendung erhalten und ist somit Teil sowohl der linken als auch der rechten Regelseite. Andernfalls gibt es entweder einen Knoten $n''$, der auf dem Pfad zwischen $n$ und $n'$ liegt, d.h. $n \mapsto_{\pi'} n'' \mapsto_{\pi''} n'$, und bei der Regelanwendung erhalten bleibt. Dann ist dieser Knoten Teil der linken und rechten Regelseite. Oder alle Knoten, die auf dem Pfad $\pi$ liegen und Teil von $\hat{P}_j \setminus P_j$ sind, werden gelöscht.

In diesem Fall ist $n''$ adjazent zu $n'$, wobei $n'$ durch die Regel erhalten bleibt. Dieses Ergebnis wird in Lemma 8 festgehalten.

**Lemma 8.** *(Gemeinsame Elemente von Graphtransformationsregeln und Graphmustern)*
*Für eine Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *und ein Graphmuster* $p := [P, \hat{\mathcal{P}}]$ *gilt im DPO*$^{iso}$*:*

$$\forall G, G' \in \mathcal{G}, o \in \mathcal{ISO} : G \models\!\Rightarrow_{(r,o)} G' \wedge G \not\models_o p \wedge G' \models_{iso} p$$
$$\Rightarrow$$
$$((R \setminus L) \cap o(P) \neq \emptyset) \vee (\exists iso \in \mathcal{ISO}, \hat{P} \in \hat{\mathcal{P}} : iso \mid_P = o \mid_P \wedge L \cap R \cap \hat{P} \neq \emptyset). \quad (3.6)$$

Das Ergebnis aus Lemma 8 wird im Folgenden dazu genutzt, um Ergebnisgraphmuster zu bilden. Da auf ein Ergebnisgraphmuster die entsprechende Regel rückwärts angewendet werden soll, wird zur Bildung des Musters statt der Regel $[L, \hat{\mathcal{L}}] \rightarrow_r R$ ihre inverse Regel $[L^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{r^{-1}} R^{-1}$ verwendet. Das Ergebnisgraphmuster setzt sich dann zusammen aus den beiden Graphmustern $[L^{-1}, \hat{\mathcal{L}}^{-1}]$ und einem Zeugen $p := [P, \hat{\mathcal{P}}]$ für ein verbotenes Graphmuster aus der Sicherheitseigenschaft $\phi := \bigwedge_{k \in K} p_k$. Nach Lemma 8 gibt es zwei Möglichkeiten, ein solches Ergebnisgraphmuster $egm := [EGM, \mathcal{EG}\hat{\mathcal{M}}]$ zu erzeugen.

Die erste Möglichkeit betrachtet den Fall, dass die Anwendung von $r$ $P$ erzeugt. Das bedeutet, dass die inverse Regel ein Element löscht, das Teil von $P$ ist. Deshalb werden die Graphen $L^{-1}$ und $P$ so zu einem Graphen $EGM$ zusammengefügt, dass mindestens eines der durch $r^{-1}$ gelöschten Elemente auch Teil von $P$ ist, d.h. es gibt einen Teilgraphisomorphismus $tiso \in \mathcal{TISO}$, der Elemente von $P$ auf Elemente von $L^{-1}$ abbildet, wobei mindestens ein Knoten aus $P$ auf einen Knoten aus $(N_{L^{-1}} \setminus N_{R^{-1}})$ oder eine Kante aus $P$ auf eine Kante aus $(E_{L^{-1}} \setminus E_{R^{-1}})$ abgebildet wird. Zusätzlich darf es weder einen Graphen in der negativen Anwendungsbedingung $\hat{\mathcal{L}}^{-1}$ von $r^{-1}$ noch in der negativen Anwendungsbedingung $\hat{\mathcal{P}}$ von $p$ geben, der auf einen Teil von $EGM$ durch einen Teilgraphisomorphismus abgebildet werden kann.

Um die negative Anwendungsbedingung $\mathcal{EG}\hat{\mathcal{M}}$ zu bilden, wird sowohl für jeden Graphen aus $\hat{\mathcal{L}}^{-1}$ als auch für jeden Graphen aus $\hat{\mathcal{P}}$ mindestens ein neuer Graph für die negative Anwendungsbedingung erzeugt. Betrachtet man einen Graphen $\hat{L}^{-1} \in \hat{\mathcal{L}}^{-1}$ und den Graph $P$, so wird für diese beiden Graph folgendermaßen eine Menge von Graphen der negativen Anwendungsbedingung gebildet. Es werden alle Teilgraphisomorphismen $tiso'$ bestimmt, die einen nicht leeren Teilgraphen von $P$ auf einen Teilgraphen von $\hat{L}^{-1}$ abbilden. Für jeden dieser Isomorphismen gilt, dass er alle Elemente von $P$, die durch $tiso$ auf Elemente auf $L^{-1}$ abgebildet werden, auf dieselben Elemente abgebildet, d.h. $tiso'^{-1} \mid_L = tiso^{-1}$. Mit Hilfe dieser Teilgraphisomorphismen werden dann die beiden Graphen zu einem vereinigt. Dabei besteht ein Graph $E\hat{G}M$ zunächst aus allen Elementen von $\hat{L}^{-1}$. Alle Elemente aus $P$, die nicht durch $tiso'$ auf Elemente aus $\hat{L}^{-1}$ abgebildet werden können, werden hinzugefügt. Auf diese Weise entsteht eine Menge von Graphen.

Diese Menge kann unter Umständen noch reduziert werden. Nach Definition 15 muss jeder Graph der negativen Anwendungsbedingung die Anwendungsbedingung enthalten und diese um mindestens ein Element erweitern. Durch die Verwendung von $tiso'$ können Graphen resultieren, die isomorph zur Anwendungsbedingung sind. Diese Graphen müssen aus der negativen Anwendungsbedingung entfernt werden. Außerdem gilt: Enthält die Menge ein Graphenpaar $E\hat{G}M_1$

und $\hat{EGM}_2$ und existiert ein Teilgraphisomorphismus, der $\hat{EGM}_1$ auf einen Teilgraphen von $\hat{EGM}_2$ abbildet ($\hat{EGM}_1 \preceq \hat{EGM}_2$), dann kann $\hat{EGM}_2$ aus der Menge entfernt werden. Dies ist möglich, da immer, wenn es einen Teilgraphisomorphismus gibt, der $\hat{EGM}_2$ auf einen Graphen oder ein Graphmuster abbildet, dieser Teilgraphisomorphismus auch dazu verwendet werden kann, um $\hat{EGM}_1$ auf den Graphen/das Graphmuster abzubilden. Die Bildung der Graphen bestehend aus einem Graphen $\hat{P} \in \hat{\mathcal{P}}$ und der Anwendungsbedingung $L^{-1}$ wird analog durchgeführt. Auf diese Weise wird erreicht, dass EGM ein Teilgraph von jedem $\hat{EGM} \in \hat{\mathcal{EGM}}$ ist und $\hat{\mathcal{EGM}}$ somit eine korrekte negative Anwendungsbedingung ist.

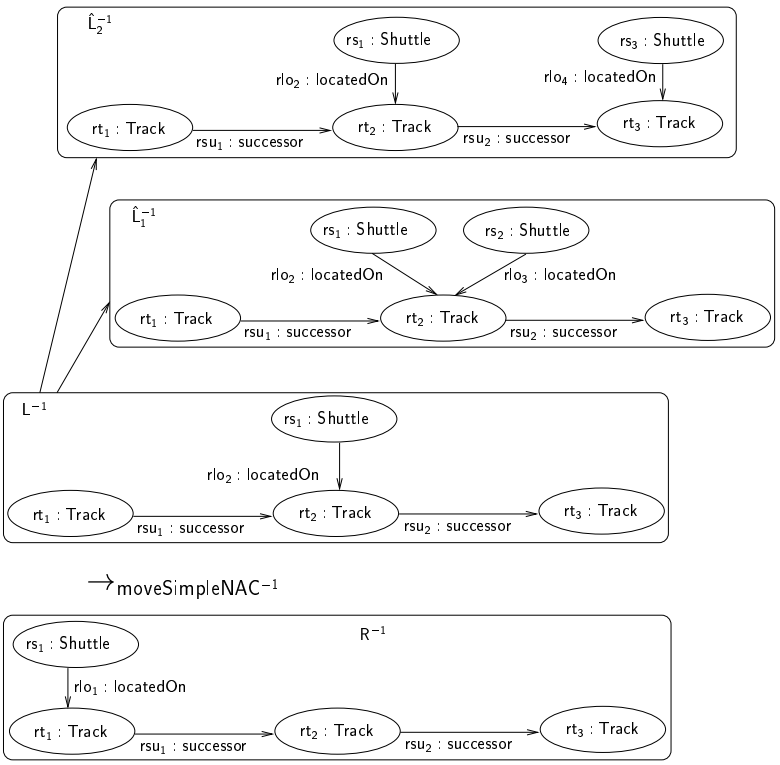
Abbildung 3.28(a) zeigt das Inverse der Regel moveSimpleNAC ($[L^{-1}, \hat{L}^{-1}] \rightarrow_{\text{moveSimpleNAC}^{-1}} R^{-1}$), Abbildung 3.28(b) zeigt das verbotene Muster impendingCollision $:= [IC, \hat{IC}]$.

Ein mögliches Ergebnisgraphmuster für die Regel und das verbotene Muster ist in Abbildung 3.29 dargestellt. Dieses Ergebnisgraphmuster beschreibt den Fall, dass die Anwendung der Regel moveSimpleNAC das verbotene Muster erzeugt, indem die Anwendungsbedingung des verbotenen Musters vervollständigt wird. Das bedeutet, dass es Knoten oder Kanten in der Nachbedingung der Regel gibt, die bei der Regelanwendung neu erzeugt werden (und somit zu $(N_R \setminus N_L)$ bzw. $(E_R \setminus E_L)$ gehören) und zusätzlich auch durch einen Teilgraphisomorphismus $tiso \in \mathcal{TISO}$ auf einen Teil von P abgebildet werden können. Da zur Bildung des Ergebnisgraphmusters die inverse Regel verwendet wird, müssen die Elemente durch die Regelanwendung gelöscht werden; sie sind somit Teil von $N_{L^{-1}} \setminus N_{R^{-1}}$ bzw. $E_{L^{-1}} \setminus E_{R^{-1}}$.

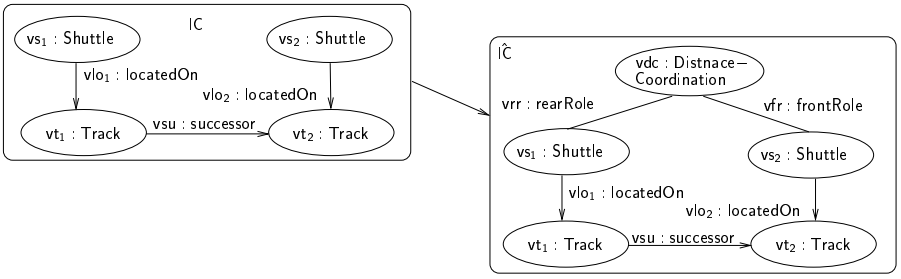
Im Beispiel wird durch r nur die $rlo_2 : locatedOn$ erzeugt. Auf diese Kante kann durch einen Teilgraphisomorphismus die Kante $vlo_1$ des verboten Graphmusters abgebildet werden. Da für eine Kante der Start- und der Zielknoten eindeutig festgelegt ist, muss der Teilgraphisomorphismus auch den Knoten $vs1_1 : Shuttle$ auf $rs_1 : Shuttle$ und den Knoten $vt_1 : Track$ auf $rt_2 : Track$ abbilden. Der im Beispiel verwendete Teilgraphisomorphismus bildet zusätzlich noch den Knoten $vt2 : Track$ auf den Knoten $rt_3 : Track$ ab und die Kanten $vsu : successor$ auf $rsu_2 : successor$.

Das Ergebnisgraphmuster $egm := [EGM, \hat{\mathcal{EGM}}]$ wird nun folgendermaßen gebildet. Für EGM werden mit Hilfe des Teilgraphisomorphismus tiso die beiden Graphen $L^{-1}$ und IC der Regel bzw. des verbotenen Musters vereint. Das bedeutet, dass EGM zunächst aus dem Graph $L^{-1}$ besteht. Alle Elemente, die durch tiso nicht auf Elemente aus $L^{-1}$ abgebildet werden, werden zu EGM hinzugefügt. Das Ergebnisgraphmuster ist in Abbildung 3.29 dargestellt.

Um die Graphen der negativen Anwendungsbedingung zu bilden, müssen alle Graphen der negativen Anwendungsbedingung der inversen Graphtransformationsregel sowie des verbotenen Graphmusters betrachtet werden. In diesem Beispiel besteht die negative Anwendungsbedingung von $r^{-1}$ aus den beiden Graphen $\hat{L}_1^{-1}$ und $\hat{L}_2^{-1}$ und die negative Anwendungsbedingung des verbotenen Graphmusters aus dem Graphen $\hat{IC}$. Für jeden dieser drei Graphen müssen neue Graphen gebildet werden. Der Graph $\hat{EGM}_1$ wird aus den Graphen $\hat{L}_1^{-1}$ und IC gebildet. Dazu werden alle Teilgraphisomorphismen bestimmt, die einen Teilgraph von IC auf einen Teilgraphen von $\hat{L}_1^{-1}$ abbilden. Diese Teilgraphisomorphismen müssen alle Elemente von IC, die durch den zuvor bestimmten tiso auf Elemente aus $L^{-1}$ abgebildet werden, auf dieselben Elemente abbilden. In diesem Beispiel ist das nur der Teilgraphisomorphismus tiso. $\hat{EGM}_1$ besteht dann zunächst aus $\hat{L}_1^{-1}$. Alle Elemente aus IC, die nicht durch tiso auf Elemente aus $\hat{L}_1^{-1}$ abgebildet werden, werden




(a) Die inverse Regel moveSimpleNAC$^{-1}$




(b) Verbotenes Graphmuster impendingCollision

Abbildung 3.28: Regel moveSimpleNAC, deren Korrektheit im Bezug auf das verbotene Graphmuster impendingCollision überprüft werden soll

zu $E\hat{G}M_1$ hinzugefügt. Für $\hat{L}_2^{-1}$ und IC gibt es zwei Teilgraphisomorphismen, die die geforderten Eigenschaften erfüllen. Deshalb entstehen die beiden Graphen $E\hat{G}M_2$ und $E\hat{G}M_3$. $E\hat{G}M_2$ ist in Abbildung 3.29 mit gestricheltem Rand dargestellt. Der Grund dafür ist, dass $E\hat{G}M_3$ ein Teilgraph von $E\hat{G}M_2$ ist, somit braucht $E\hat{G}M_2$ nicht in die negative Anwendungsbedingung aufgenommen werden. $E\hat{G}M_4$ ist der Graph, der aus der Vereinigung von $L^{-1}$ und $\hat{IC}$ resultiert.

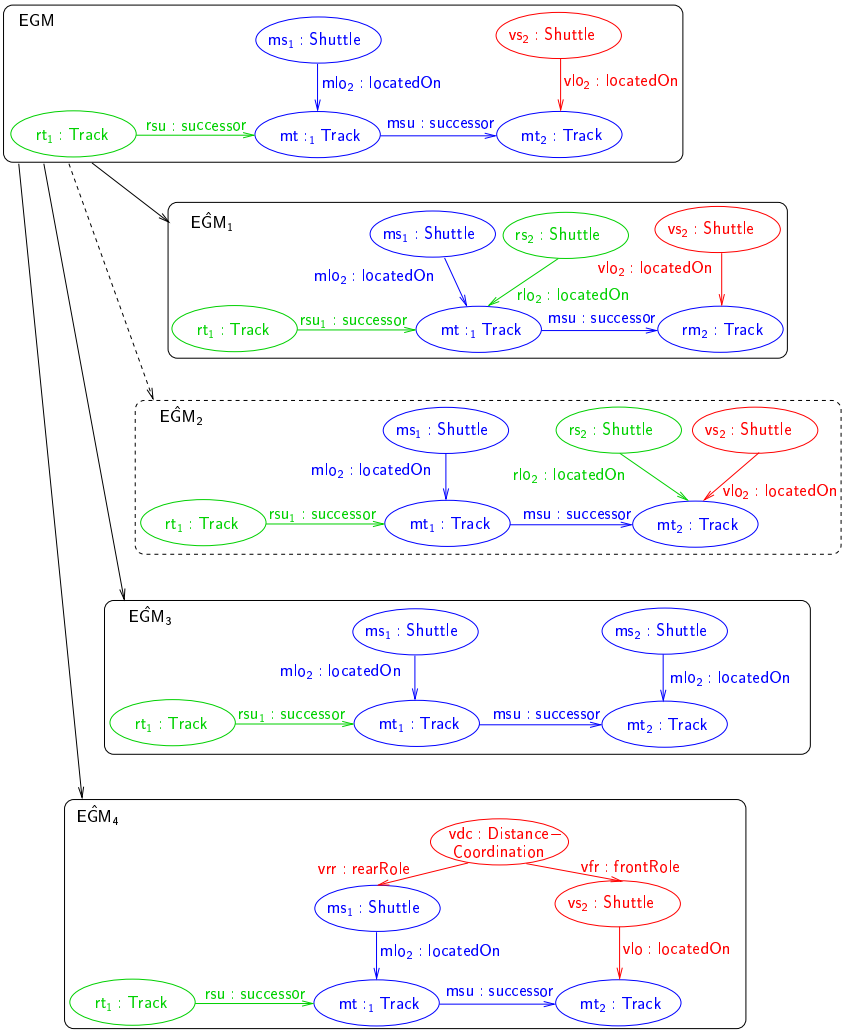



Abbildung 3.29: Ein mögliches Ergebnisgraphmuster für die Regel moveSimpleNAC und das verbotene Graphmuster impendingCollision

Auf diese Weise muss für jeden Teilgraphisomorphismus, der ein oder mehrere durch die Regel neu erzeugte Elemente auf ein verbotenes Graphmuster abbildet, ein Ergebnisgraphmuster gebildet werden.

Die zweite Möglichkeit betrachtet den Fall, dass die Anwendung der Regel $r$ ein Element löscht (das Element wird dann durch die Anwendung von $r^{-1}$ erzeugt), sodass anschließend ein $\hat{P} \in \hat{\mathcal{P}}$ Teilgraphmuster des resultierenden Graphen ist. Das entsprechende Ergebnisgraphmuster $egm' := [EGM', \mathcal{EG}\hat{\mathcal{M}}']$ wird folgendermaßen gebildet. $EGM'$ wird durch die Vereinigung von $L^{-1}$ und $P$ gebildet, wobei es in diesem Fall nicht unbedingt einen Teilgraphisomorphismus geben muss, der einen Teilgraph von $L^{-1}$ auf einen Teilgraph von $P$ abbildet. Allerdings muss es einen Teilgraphisomorphismus $tiso' \in \mathcal{TISO}$ und ein $\hat{P} \in \hat{\mathcal{P}}$ geben, sodass $tiso'$ mindestens einen Knoten, der sowohl in der linken als auch rechten Regelseite auftritt, auf einen

Knoten aus $\hat{\mathsf{P}}$ abbildet. Dieser Teilgraphisomorphismus wird dann dazu verwendet, die Graphen $\mathsf{E}\hat{\mathsf{G}}\mathsf{M}_\mathsf{i}^{'} \in \mathcal{E}\hat{\mathcal{G}}\mathcal{M}^{'}$ der negativen Anwendungsbedingung zu bilden. Dabei wird für jedes Paar von Graphen $\hat{\mathsf{P}} \in \hat{\mathcal{P}}$ und $\hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}^{-1}$ ein solches $\mathsf{E}\hat{\mathsf{G}}\mathsf{M}_\mathsf{i}^{'}$ gebildet. Das $\mathsf{E}\hat{\mathsf{G}}\mathsf{M}_\mathsf{i}^{'}$ entspricht zunächst $\hat{\mathsf{P}}$. Der Teilgraphisomorphismus $\mathsf{tiso}'$ wird dazu verwendet, alle möglichen Elemente von $\hat{\mathsf{L}}^{-1}$ auf Elemente aus $\hat{\mathsf{P}}$ abzubilden. Elemente für die eine solche Abbildung unter $\mathsf{tiso}'$ nicht möglich ist werden zum Graphen hinzugefügt.

In Abbildung 3.30 ist die Regel deleteDCSwitch abgebildet. Die Regel beschreibt, dass die Instanz des DistanceCoordination-Musters zwischen zwei Shuttles gelöscht wird, wenn die Shuttles an einer Weiche in unterschiedliche Richtungen weiterfahren wollen. Die Regel löscht also einen Knoten, der auch Teil der negativen Anwendungsbedingung des verbotenen Graphmusters impendingCollision aus Abbildung 3.28(b) ist.

Da die Regel nur einen Knoten und seine inzidenten Kanten löscht, jedoch keine Elemente erzeugt, braucht bei der Bildung der Ergebnisgraphmuster nur die zweite Möglichkeit betrachtet werden. Die Regel löscht den Knoten rdc, seine adjazenten Knoten $\mathsf{rs}_1$ und $\mathsf{rs}_2$ bleiben bei der Anwendung erhalten. Zusätzlich dazu erhält die Regel die Knoten $\mathsf{rt}_1$, $\mathsf{rt}_2$ und $\mathsf{rt}_3$ sowie die Kanten $\mathsf{rlo}_1$, $\mathsf{rn}_1$, $\mathsf{rsu}_1$, $\mathsf{rsu}_2$ und $\mathsf{rlo}_2$. Für alle diese Elemente gilt, dass sie sowohl Teil der linken als auch der rechten Regelseite sind. Für den Graphen $\mathsf{I}\hat{\mathsf{C}}$ wird nun ein Teilgraphisomorphismus $\mathsf{tiso}' \in \mathcal{ISO}$ gesucht, der einen Teil der Elemente von $\mathsf{I}\hat{\mathsf{C}}$ auf die Elemente abbildet, die bei der Regelanwendung erhalten bleiben. Ein möglicher Teilgraphisomorphismus ist $\mathsf{tiso}' := \langle \mathsf{tiso}'_\mathsf{N}, \mathsf{tiso}'_\mathsf{E} \rangle$, mit $\mathsf{tiso}'_\mathsf{N} := \{(\mathsf{vs}_1 \mapsto \mathsf{rs}_1), (\mathsf{vs}_2 \mapsto \mathsf{rs}_2), (\mathsf{vt}_1 \mapsto \mathsf{rt}_1), (\mathsf{vt}_2 \mapsto \mathsf{rt}_2)\}$ und $\mathsf{tiso}'_\mathsf{E} := \{(\mathsf{vlo}_1 \mapsto \mathsf{rlo}_1), (\mathsf{vlo}_2 \mapsto \mathsf{rlo}_2), (\mathsf{vsu} \mapsto \mathsf{rsu}_1)\}$. Mit Hilfe dieses Teilgraphisomorphismus kann nun $\mathsf{EGM}'$ gebildet werden. Dazu wird $\mathsf{L}^{-1}$ mit $\mathsf{tiso}'(\mathsf{IC})$ vereinigt. Das heißt, dass der Graph $\mathsf{L}^{-1}$ um alle Elemente von IC erweitert wird, die nicht durch $\mathsf{tiso}'$ auf Elemente von $\mathsf{L}^{-1}$ abgebildet werden. Die negative Anwendungsbedingung $\mathcal{E}\hat{\mathcal{G}}\mathcal{M}$ besteht nur aus dem einen Graphen $\mathsf{E}\hat{\mathsf{G}}\mathsf{M}^{'}$. $\mathsf{E}\hat{\mathsf{G}}\mathsf{M}^{'}$ besteht zunächst aus $\mathsf{L}^{-1}$. Jedes Element aus $\mathsf{I}\hat{\mathsf{C}}$, das nicht durch $\mathsf{tiso}'$ auf ein Element aus $\mathsf{L}^{-1}$ abgebildet werden kann, wird zusätzlich in diesen Graphen eingefügt. Das Ergebnisgraphmuster $\mathsf{egm}'$ ist in Abbildung 3.31 dargestellt.

**Definition 35.** *(Ergebnisgraphmuster (EGM))*

*Für eine Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_\mathsf{r} \mathsf{R}$*, ihr Inverses* $[\mathsf{L}^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{\mathsf{r}^{-1}} \mathsf{R}^{-1}$ *und ein verbotenes Muster* $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ *werden die Ergebnisgraphmuster* $\mathsf{egm} := [\mathsf{EGM}, \mathcal{E}\hat{\mathcal{G}}\mathcal{M}]$ *unter Berücksichtigung von Lemma 8 auf die folgenden zwei Arten gebildet.*

*(1) Ein Teil der Menge wird Berücksichtigung der ersten Bedingung von Lemma 8 gebildet. Für jedes Ergebnisgraphmuster dieser Teilmenge wird jeweils ein Graph* $\mathsf{P}'$ *und ein Graphisomorphismus* $\mathsf{iso} \in \mathcal{ISO}$ *gewählt, mit*

$$
\begin{aligned}
&\mathsf{P}' \approx_{\mathsf{iso}} \mathsf{P} : \\
&\quad \mathsf{N}_{\mathsf{P}'} \cap (\mathsf{N}_{\mathsf{L}^{-1}} \setminus \mathsf{N}_{\mathsf{R}^{-1}}) \neq \emptyset \quad \vee \\
&\quad \mathsf{E}_{\mathsf{P}'} \cap (\mathsf{E}_{\mathsf{L}^{-1}} \setminus \mathsf{E}_{\mathsf{R}^{-1}}) \wedge \forall \mathsf{e} \in \mathsf{E}_{\mathsf{P}'} \cap (\mathsf{E}_{\mathsf{L}^{-1}} \setminus \mathsf{E}_{\mathsf{R}^{-1}}) : \\
&\qquad\qquad \mathsf{src}(\mathsf{e}) \in (\mathsf{N}_{\mathsf{P}'} \cap \mathsf{N}_{\mathsf{L}^{-1}}) \wedge \mathsf{tgt}(\mathsf{e}) \in (\mathsf{N}_{\mathsf{P}'} \cap \mathsf{N}_{\mathsf{L}^{-1}}) \quad \wedge \\
&\forall \hat{\mathsf{R}}_\mathsf{i} \in \hat{\mathcal{L}}^{-1}, \nexists \mathsf{tiso}' \in \mathcal{TISO} : \\
&\quad \mathsf{tiso}'^{-1} \mid_{\mathsf{L}^{-1}} = \mathsf{iso}^{-1} \wedge \hat{\mathsf{L}}_\mathsf{i}^{-1} \preceq_{\mathsf{tiso}'} \mathsf{P}' \cup \mathsf{L}^{-1} \quad \wedge \\
&\forall \hat{\mathsf{P}}_\mathsf{i} \in \hat{\mathcal{P}}, \nexists \mathsf{tiso}'' \in \mathcal{TISO} : \\
&\quad \mathsf{tiso}'' \mid_{\mathsf{P}} = \mathsf{iso} \wedge \hat{\mathsf{P}}_\mathsf{i} \preceq_{\mathsf{tiso}''} \mathsf{P}' \cup \mathsf{L}^{-1}.
\end{aligned}
$$

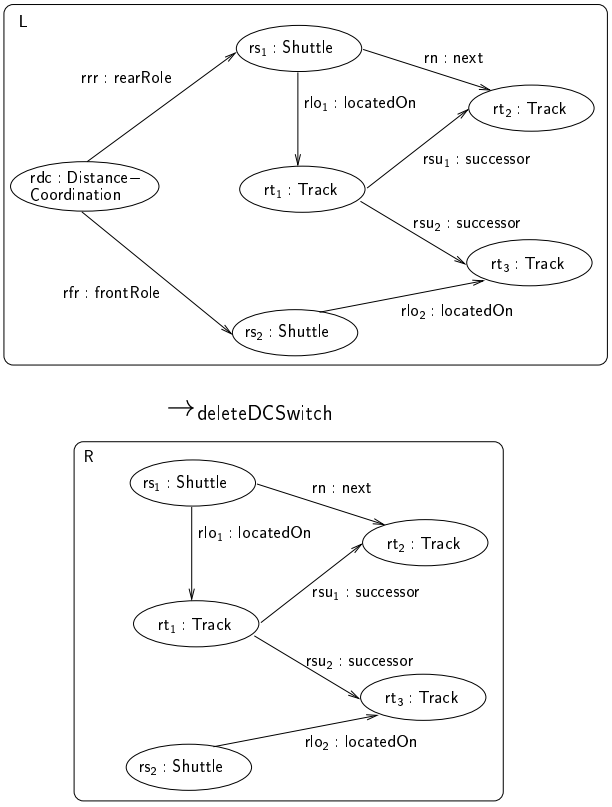



Abbildung 3.30: Die Regel deleteDCSwitch

*Das Ergebnisgraphmuster wird dann folgendermaßen gebildet:*

$$\mathsf{EGM} := \quad \mathsf{P}' \cup \mathsf{L}^{-1}$$

*und*

$$\begin{aligned}
\mathcal{EG\hat{G}M} := \quad & \{\hat{\mathsf{L}}^{-1} \cup \mathsf{iso}'(\mathsf{P}) \mid \exists \mathsf{iso}' \in \mathcal{ISO}, \hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}^{-1}, \nexists \mathsf{iso}'' \in \mathcal{ISO} : \\
& \quad \mathsf{iso}'^{-1} \mid_{\mathsf{L}^{-1}} = \mathsf{iso}^{-1} \wedge \mathsf{EGM} < (\hat{\mathsf{L}}^{-1} \cup \mathsf{iso}'(\mathsf{P})) \qquad \wedge \\
& \quad \mathsf{iso}''^{-1} \mid_{\mathsf{L}'} = \mathsf{iso}^{-1} \wedge \mathsf{EGM} < (\hat{\mathsf{L}}^{-1} \cup \mathsf{iso}''(\mathsf{P})) \wedge \\
& \qquad\qquad (\hat{\mathsf{L}}^{-1} \cup \mathsf{iso}''(\mathsf{P})) \preceq (\hat{\mathsf{L}}^{1} - 1 \cup \mathsf{iso}'(\mathsf{P})) \\
& \} \\
& \cup \\
& \{\mathsf{L}^{-1} \cup \mathsf{iso}'(\hat{\mathsf{P}}) \mid \exists \mathsf{iso}' \in \mathcal{ISO}, \hat{\mathsf{P}} \in \mathcal{P}, \nexists \mathsf{iso}'' \in \mathcal{ISO} : \\
& \quad \mathsf{iso}' \mid_{\mathsf{P}} = \mathsf{iso} \wedge \mathsf{EGM} < (\mathsf{L}^{-1} \cup \mathsf{iso}'(\hat{\mathsf{P}})) \qquad \wedge \\
& \quad \mathsf{iso}'' \mid_{\mathsf{P}} = \mathsf{iso} \wedge \mathsf{EGM} < (\mathsf{L}^{-1} \cup \mathsf{iso}''(\hat{\mathsf{P}})) \wedge \\
& \qquad\qquad (\mathsf{L}^{-1} \cup \mathsf{iso}''(\hat{\mathsf{P}})) < (\mathsf{L}^{-1} \cup \mathsf{iso}'(\hat{\mathsf{P}})) \\
& \}.
\end{aligned}$$

*(2) Der zweite Teil der Menge wird unter Berücksichtigung der zweiten Bedingung aus Lemma 8 gebildet. Dabei wird für jedes Ergebnisgraphmuster dieser Teilmenge jeweils ein Graph* $\hat{\mathsf{P}}$ *und ein Graphisomorphismus* $\mathsf{iso}' \in \mathcal{ISO}$ *gewählt, sodass gilt:*


Abbildung 3.31: Ein mögliches Ergebnisgraphmuster für die Regel deleteDCSwitch und das verbotene Muster impendingCollision

$$(N_{R^{-1}} \cap N_{L^{-1}} \cap N_{iso'(\hat{P})}) \neq \emptyset. \qquad \wedge$$
$$\exists n \in (N_{R^{-1}} \cap N_{L^{-1}} \cap N_{iso'(\hat{P})}), e \in (E_{R^{-1}} \setminus E_{L^{-1}}) : (n = src(e)) \vee (n = tgt(e)).$$

*In diesem Fall wird* egm *folgendermaßen gebildet:*

$$EGM := \quad L^{-1} \cup iso'(P) \qquad \wedge$$
$$\hat{\mathcal{EGM}} := \quad \{L^{-1} \cup iso''(\hat{P}') \mid \exists iso'' \in \mathcal{ISO}, \hat{P}' \in \hat{\mathcal{P}}, \nexists iso''' \in \mathcal{ISO} :$$
$$iso'' \mid_P = iso' \wedge EGM < (L^{-1} \cup iso''(\hat{P}')) \qquad \wedge$$
$$iso''' \mid_P = iso' \wedge EGM < (L^{-1} \cup iso'''(\hat{P}')) \wedge$$
$$(L^{-1} \cup iso'''(\hat{P}')) \preceq (L^{-1} \cup iso''(\hat{P}'))$$
$$\}.$$

In den Abbildungen 3.32, 3.33 und 3.34 ist schematisch die Bildung der Graphen der negativen Anwendungsbedingung dargestellt. Dabei gehören die Abbildungen 3.32 und 3.33 zu dem Fall, dass die Regelanwendung ein Element des verbotenen Graphmusters erzeugt. Abbildung 3.32 beschreibt, dass ein Graph $\hat{L}^{-1} \in \hat{\mathcal{L}}^{-1}$ um Elemente aus P erweitert wird. Die Erweiterung eines Graphen $\hat{P} \in \mathcal{P}$ um Elemente aus $L^{-1}$ ist in Abbildung 3.33 zu sehen. Abbildung 3.34 gehört dagegen zu dem Fall, dass die Regelanwendung ein Element löscht, das Teil eines $\hat{P} \in \hat{\mathcal{P}}$ ist. Die Abbildung zeigt schematisch die Erweiterung des Graphen $\hat{P}$ um Elemente aus $L^{-1}$.

Die Menge aller korrekten Ergebnisgraphmuster für eine Regel r und ein verbotenes Muster p wird mit $\mathcal{EGM}(r, p)$ bezeichnet. Die Menge

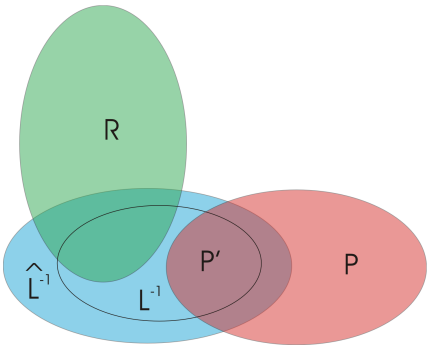



Abbildung 3.32: Schematische Darstellung der Erweiterung eines Graphen $\hat{L}^{-1} \in \hat{\mathcal{L}}^{-1}$ um Elemente aus $P$

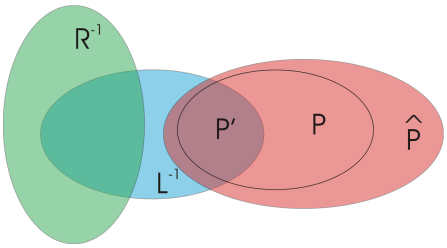



Abbildung 3.33: Schematische Darstellung der Erweiterung eines Graphen $\hat{P} \in \hat{\mathcal{P}}$ um Elemente aus $L^{-1}$

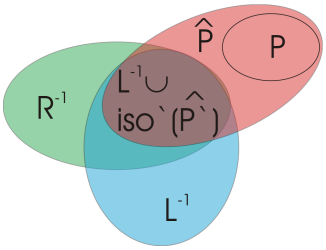



Abbildung 3.34: Schematische Darstellung der Erweiterung eines Graphen $\hat{P} \in \hat{\mathcal{P}}$ um Elemente aus $L^{-1}$

$\mathcal{EGM}_{\approx}(r, p)$ ist eine Teilmenge von $\mathcal{EGM}(r, p)$. Diese Menge ist minimal, d.h. es gilt $\forall egm \in \mathcal{EGM}(r, p), \exists egm' \in \mathcal{EGM} :_{\approx} (r, p) : egm' \subseteq egm$ und $\forall egm_1, egm_2 \in \mathcal{EGM}_{\approx}(r, p) : egm_1 \not\subseteq egm_2$.

Durch die Konstruktion des Ergebnisgraphmusters ist das Graphmuster $[L^{-1}, \hat{\mathcal{L}}^{-1}]$ der inversen Regel $r^{-1}$ immer ein Teilgraphmuster des Ergebnisgraphmusters und eine Rückwärtsanwendung der Regel im DPO$^{iso}$ immer möglich. Das Graphmuster, das aus der Anwendung resultiert, wird als Startgraphmuster bezeichnet.

Das Startgraphmuster, das resultiert, wenn die inverse Regel moveSimpleNAC$^{-1}$ auf das Ergebnisgraphmuster egm aus Abbildung 3.29 angewendet wird, ist in Abbildung 3.35 dargestellt.

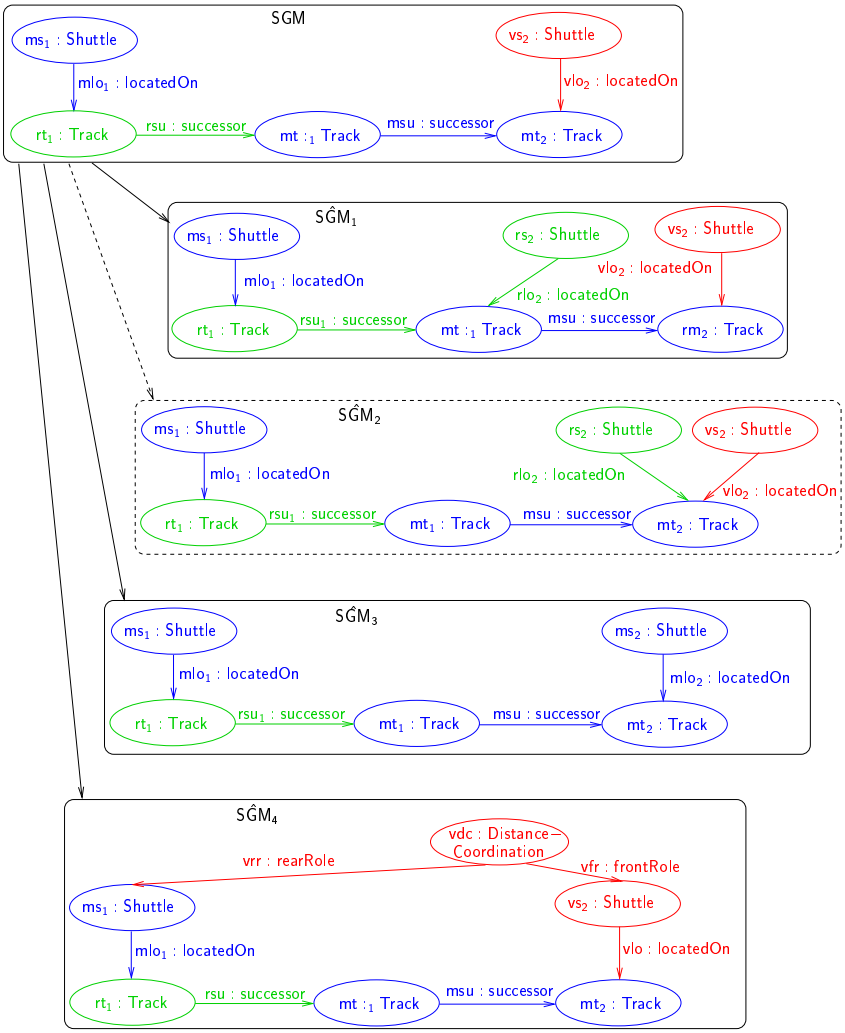



Abbildung 3.35: Startgraphmuster, das aus der Anwendung der inversen Regel $\mathsf{moveSimpleNAC}^{-1}$ auf das Ergebnisgraphmuster aus Abbildung 3.29 resultiert

**Definition 36.** *(Startgraphmuster)*
*Für eine Regel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$*, ihr Inverses* $[\mathsf{L}^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{\mathsf{r}^{-1}} \mathsf{R}^{-1}$*, das verbotene Muster* $\mathsf{p} := [\mathsf{P}, \hat{\mathcal{P}}]$ *und ein dazugehöriges Ergebnisgraphmuster* $\mathsf{egm} := [\mathsf{EGM}, \mathcal{EG}\hat{\mathcal{M}}]$*, ist das Startgraphmuster* $\mathsf{sgm} := [\mathsf{SGM}, \mathcal{SG}\hat{\mathcal{M}}]$ *definiert als* $\mathsf{tgm} \not\Longrightarrow_{\mathsf{r}^{-1}} \mathsf{sgm}$.

Die Menge $\mathcal{SGM}_{\approx}(\mathsf{r}, \mathsf{p})$ bezeichnet die minimale Menge der Startgraphmuster. Diese Menge wird aus der Menge $\mathcal{TGM}_{\approx}(\mathsf{r}, \mathsf{p})$ durch Anwendung der inversen Regel von r erzeugt. Für jeden der Graphen aus $\mathcal{SGM}_{\approx}(\mathsf{r}, \mathsf{p})$ muss geprüft werden, ob er ein verbotenes Graphmuster enthält. Kann ein Startgraphmuster $\mathsf{sgm} \in \mathcal{SGM}_{\approx}(\mathsf{r}, \mathsf{p})$ gefunden werden, das kein verbotenes Graphmuster enthält, so wurde ein Gegenbeispiel gefunden, dass zeigt, dass die Regel r einen korrekten Graphen in einen inkorrekten überführen kann, d.h. die Menge $\mathcal{T}[\phi_{\mathsf{S}}, \neg\phi_{\mathsf{S}}]$ nicht leer ist.

Lemma 9 zeigt für ein System $\mathsf{S}$ und eine Sicherheitseigenschaft $\phi_s := \bigwedge_{k\in K}(\neg p_i)$, dass die Menge $\mathcal{T}[\phi_S, \neg\phi_S]$ leer und $\phi_S$ somit eine induktive Invariante von $\mathsf{S}$ ist, falls für jedes Ergebnisgraphmuster egm das dazugehörigen Startgraphmuster sgm einen Zeugen für ein verbotenes Muster $p_i$ aus $\phi_S$ enthält. Ein Graph, der eines der Ergebnisgraphmuster enthält und somit inkorrekt ist, kann nur entstanden sein, wenn die entsprechende Regel auf einen Graphen angewendet wurde, der eines der Startgraphmuster enthalten hat. Da auch die Startgraphmuster einen Zeugen für ein verbotenes Muster aufweisen, ist auch der ursprüngliche Graph inkorrekt.

**Lemma 9.** *(Zeuge im Startgraphmuster impliziert induktive Invariante)*
*Für ein System* $S := (G_\Omega, \mathcal{G}^i, \mathcal{R}_S)$ *und eine Sicherheitseigenschaft* $\phi := \bigwedge_{j\in J}(\neg p_j)$ *gilt im DPO*$^{iso}$ $\forall i \in J, r \in \mathcal{R}$*:*

$$(\forall egm \in \mathcal{EGM}_{\approx}(r, p_i), sgm, o \in \mathcal{ISO} : (egm \not\Longrightarrow_{(r^{-1},o^{-1})} sgm) \wedge (\exists j \in J : p_j \subseteq sgm)) \Rightarrow (\forall G_1, G_2 \in \mathcal{G}[G_\Omega] : (G_1 \Longrightarrow_{(r,o)} G_2 \wedge G_2 \models p_i) \Rightarrow (\exists j \in J : G_1 \models p_j)).$$

Mittels dieses Lemmas kann der Nachweis, dass eine Sicherheitseigenschaft eine induktive Invariante eines Systems ist, auf die Betrachtung von Start- und Ergebnisgraphmustern reduziert werden. Eine Sicherheitseigenschaft ist genau dann eine induktive Invariante eines Systems, wenn die Transformation eines Ergebnisgraphmusters durch die dazugehörige Graphtransformationsregel immer in einem Startgraphmuster resultiert, das einen Zeugen für ein verbotenes Muster enthält. Dieses Ergebnis ist in Theorem 5 festgehalten.

**Theorem 5.** *(Sicherheitseigenschaft ist induktive Invariante)*
*Für ein System* $S := (G_\Omega, \mathcal{G}^i_S, \mathcal{R}_S)$ *ist die Sicherheitseigenschaft* $\phi := \bigwedge_{j\in J}(\neg p_j)$ *genau dann eine induktive Invariante, wenn gilt:*

$$\forall i \in J, r \in \mathcal{R}_S, egm \in \mathcal{EGM}_{\approx}(r, p_i), sgm, o \in \mathcal{ISO} : (egm \not\Longrightarrow_{(r^{-1},o^{-1})} sgm) \Rightarrow (\exists j \in J : p_j \subseteq sgm) \tag{3.7}$$

## 3.7.2 Der Algorithmus

Die im voran gegangenen Abschnitt vorgestellten theoretischen Ergebnisse können nun dazu verwendet werden, um einen Algorithmus zu beschreiben, der automatisch überprüft, ob ein Graphtransformationssystem korrekt bezüglich einer Sicherheitseigenschaft $\phi_S$ ist. D.h. der Algorithmus prüft, ob jede der Eigenschaften eine induktive Invariante des Systems ist. Der Algorithmus check ist in Abbildung 3.36 in Pseudocode angegeben. Er erwartet als Eingabe eine Menge von Graphtransformationsregeln $\mathcal{R}_S$ und eine Menge von verbotenen Mustern $\mathcal{P}$, wobei $\mathcal{P}$ der Menge der verbotenen Graphmuster aus $\phi_S$ entspricht. Zusätzlich dazu benötigt der Algorithmus noch den Typgraphen $G_\Omega$.

In einem ersten Schritt (Zeile 03 bis 06) ruft der Algorithmus für jedes Paar, bestehend aus dem Inversen $r^{-1}$ einer Regel $r \in \mathcal{R}_S$ und ein verbotenes Graphmuster $p_i \in \mathcal{P}$, den Algorithmus buildTGP auf, der in Abschnitt C.3.1 beschrieben wird. Dieser berechnet für das jeweilige

Paar von Regel und verbotenem Graphmuster die Menge aller gültigen Ergebnisgraphmuster $\mathcal{EGM}(\mathsf{r}, \mathsf{p_i})$. Diese Menge wird reduziert, sodass eine minimale Menge von Ergebnisgraphmustern $\mathcal{EGM}_{\approx}(\mathsf{r}, \mathsf{p_i})$ resultiert (Zeile 07). Der zur Reduktion aufgerufene Algorithmus reduceTGP wird in Abschnitt C.3.1 vorgestellt.

In den Zeilen 08 bis 16 wird die Gültigkeit von Bedingung 3.7 aus Theorem 5 überprüft. Dazu wird zunächst auf jedes Ergebnisgraphmuster egm aus der Menge $\mathcal{EGM}_{\approx}(\mathsf{r}, \mathsf{p_i})$ die entsprechende inverse Graphtransformationsregel $\mathsf{r}^{-1}$ angewendet (Zeile 10). Für das jeweils resultierende Startgraphmuster sgm muss überprüft werden, ob es irgendein verbotenes Muster enthält. Kann kein Zeuge für ein verbotenes Muster in sgm gefunden werden, so liefert der Algorithmus das Startgraphmuster sgm, das Ergebnisgraphmuster egm und die Regel r als Gegenbeispiel zurück. Andernfalls gibt der Algorithmus aus, dass die Regelmenge $\mathcal{R}_\mathsf{S}$ korrekt ist.

```
01: Boolean check(Set<GraphRule> R, Set<GraphPattern>P, Graph G_Ω) begin
02:   Set<Graph> TGP := ∅
03:   forall (r ∈ R) do
04:     GraphRule r^-1 := reverse(r)
05:     forall (p ∈ P) do
06:       TGP := buildTGP(r^-1, p)
07:       TGP := reduceTGP(TGP)
08:       forall tgp ∈ TGP do
09:         //determine source graph
10:         GraphPattern sgp := applyRuleToPattern(tgp, r^-1, id_R, G_Ω)
11:         //check if sgp fulfills any p' ∈ P
12:         if (∄ p' ∈ P : p' ⊆ sgp) then
13:           //report counterxample sgp ⊭⟹_r tgp
14:           return false
15:         fi
16:       end
17:     end
18:   end
19:   return true
20: end
```

Abbildung 3.36: Algorithmus check

Das System $\mathsf{S} := (\mathsf{G}_\Omega, \mathcal{G}^{\mathsf{i}}_\mathsf{S}, \mathcal{R}_\mathsf{S})$ ist korrekt bezüglich der Sicherheitseigenschaft $\phi_\mathsf{S}$, wenn der Algorithmus check angibt, dass die Menge der Regeln korrekt ist und zusätzlich alle Initialgraphen korrekt sind, d.h. $\forall \mathsf{G} \in \mathcal{G}^{\mathsf{i}}_\mathsf{S} : \mathsf{G} \models \phi_\mathsf{S}$.

### 3.7.3 Aufwandsabschätzung

Im voran gegangenen Abschnitt wurde der Algorithmus zum Nachweis induktiver Invarianten in Graphtransformationssystemen vorgestellt. In diesem Abschnitt wird für diesen Algorithmus eine Abschätzung für seine Komplexität gegeben.

In der Abschätzung werden die folgenden Variablen verwendet: $\mathsf{n_G}$ ist die maximale Anzahl von Elementen (Knoten und Kanten) im größten Graphen. Dabei kann dieser Graph eine linke oder rechte Regelseite, die Anwendungsbedingung eines verbotenen Graphmusters oder ein Graph einer negativen Anwendungsbedingungen einer Regel oder eines verbotenen Graphmusters sein. Die Anzahl aller Regeln wird mit $\mathsf{n_R}$ bezeichnet. $\mathsf{n_P}$ entspricht der Anzahl aller verbo-

tenen Graphmuster. Die maximale Anzahl von Graphen in der negativen Anwendungsbedingung einer Regel wird mit $n_{\hat{\mathcal{L}}}$ bezeichnet. Dementsprechend bezeichnet $n_{\hat{\mathcal{P}}}$ die maximale Anzahl von Graphen in der negativen Anwendungsbedingung eines verbotenen Graphmusters.

Der Algorithmus bestimmt mehrfach (Teil-)Graphisomorphismen zwischen zwei Graphen. Dabei muss für alle Elemente des einen Graphen eine Abbildung auf ein Element des anderen Graphen gefunden werden, wodurch ein Aufwand entsteht, der exponentiell in den Größen der beiden Graphen ist. Um die Aufwandsabschätzung zu vereinfachen, wird die Größe eines Graphen als die Summe seiner Knoten und Kanten angegeben. Da alle Graphen eine maximale Größe von $n_G$ haben, beträgt der Aufwand zum Bestimmen eines Graphisomorphismus zwischen zwei Graphen $O(\exp(2n_G))$.

Als erstes wird die Regel invertiert. Dazu muss jeder Graph der negativen Anwendungsbedingung konvertiert werden und dann die minimale negative Anwendungsbedingung berechnet werden. Zudem müssen Isomorphismen bestimmt werden. sodass sich insgesamt für die Invertierung der Regel eine Komplexität von $O({n_{\hat{\mathcal{L}}}}^2 * \exp(2n_G))$ ergibt.

Anschließend bildet der Algorithmus alle möglichen Ergebnisgraphmuster. Dazu müssen zunächst alle Teilgraphen der linken Regelseite und der Anwendungsbedingung des verbotenen Graphmuster bestimmt werden. Für jeden Teilgraphen der linken Regelseite und jeden Teilgraphen des verbotenen Graphmusters wird dann geprüft, ob es einen Isomorphismus gibt, der die beiden Teilgraphen aufeinander abbildet. Um die negative Anwendungsbedingung der Ergebnisgraphmuster zu bestimmen, müssen alle Graphen der negativen Anwendungsbedingungen aller linken Regelseiten sowie aller verbotenen Graphmuster betrachtet werden. Auch beim Bilden der negativen Anwendungsbedingungen müssen wieder Isomorphismen bestimmt werden, wodurch sich für die Funktion buildTGP eine Gesamtkomplexität von $O(\exp(6n_G))$ ergibt.

Nachdem die Menge aller Ergebnisgraphmuster erzeugt wurde, werden redundante Muster wieder aus dieser Menge entfernt. Dazu muss jedes Paar von Ergebnisgraphmustern $P_1$ und $P_2$ betrachtet werden und bestimmt werden, ob es einen Isomorphismus gibt, der die Anwendungsbedingung des Musters $P_1$ auf einen Teil der Anwendungsbedingung des Musters $P_2$ abbilden kann. Ist dies der Fall, muss geprüft werden, ob dieser Isomorphismus so erweitert werden kann, dass er einen Graphen der verbotenen Anwendungsbedingung von $P_1$ auf die Anwendungsbedingung von $P_2$ abbilden kann. Ist dies der Fall, ist das Muster $P_1$ kein Teilgraphmuster von $P_2$. Andernfalls muss noch geprüft werden, ob der gefundene Isomorphismus so erweitert werden kann, dass jeder der Graphen der negativen Anwendungsbedingung von Muster $P_1$ auf einen Graphen der negativen Anwendungsbedingung von Muster $P_2$ abgebildet werden kann. Die Graphen (sowohl die Anwendungsbedingung als auch die Graphen der negativen Anwendungsbedingung) eines Ergebnisgraphmusters setzen sich jeweils aus zwei Graphen zusammen. Im schlechtesten Fall haben diese beiden Graphen nur einen gemeinsamen Knoten, sodass die maximale Größe der Graphen eines Ergebnisgraphmusters $2n_G$ beträgt. Für die Bestimmung eines Isomorphismus zwischen zwei Graphen eines Ergebnisgraphmusters bedeutet dies einen Aufwand von $O(4n_G)$. Insgesamt hat die Minimierung der Menge der Ergebnisgraphmuster deshalb einen Aufwand von $O(\exp(8n_G))$.

Wurde die Menge der Ergebnisgraphmuster berechnet und reduziert, kann die Regel auf jedes der Muster angewendet werden und dadurch die Menge der Startgraphmuster bestimmt werden. Da sich ein Ergebnisgraphmuster aus zwei Graphen zusammensetzt, die jeweils eine maximale

Größe von $n_G$ haben, gibt es maximal $exp(2n_G)$ Ergebnisgraphmuster. Deshalb wird auch die Schleife in Zeile 08 maximal $exp(2n_G)$-mal ausgeführt. Bei der Anwendung der Regel müssen wieder Isomorphismen bestimmt werden, was zu einer Komplexität von $O(exp(2n_G))$ führt.

Als letztes wird für jedes Startgraphmuster bestimmt, ob es ein verbotenes Graphmuster gibt, das ein Teilgraphmuster des Startgraphmusters ist. Dabei wird wie bei der Reduktion der Menge der Ergebnisgraphmuster vorgegangen. Daraus ergibt sich dann eine Komplexität von $O(exp(8n_G))$.

Fügt man diese Komplexitäten zusammen, so erhält man den folgenden Ausdruck, wobei die Zahl auf der linken Seite der Zeilennummer im Algorithmus entspricht.

```
03  O(n_R * (
04          n^e_L̂ xp(2n_G)+
05          n_P * (
06                  exp(6n_G)+
07                  exp(8n_G)+
08                  exp(2n_G) * (
10                          exp(2n_G)+
12                          exp(8n_G)
                    )
              )
        )
```

Löst man die innerste Klammer auf, so erhält man $O(n_R * (n^e_{\hat{\mathcal{L}}}xp(2n_G) + n_P * (exp(6n_G) + exp(8n_G) + exp(4n_G) + exp(10n_G))$. Das O-Kalkül stellt eine obere Schranke für den Aufwand dar. Deshalb ist es möglich die Summe $(n^e_{\hat{\mathcal{L}}}xp(2n_G) + n_P * (exp(6n_G) + exp(8n_G) + exp(4n_G) + exp(10n_G))$ durch $4exp(10n_G)$ zu ersetzen. Da die 4 eine Konstante ist, mit der die Exponentialfunktion multipliziert wird, kann sie weggelassen werden. Damit ergibt sich $O(n_R * (n^e_{\hat{\mathcal{L}}}xp(2n_G) + n_P * exp(10n_G))$. Zudem gilt für große $n_G$, dass der Term $n^e_{\hat{\mathcal{L}}}xp(2n_G)$ kleiner ist als $n_P * exp(10n_G)$ und somit ebenfalls wegfällt. Daraus ergibt sich eine Komplexität des gesamten Algorithmus von $O(n_R * n_P * exp(10n_G))$.

Die Abschätzung hat somit ergeben, dass der Aufwand des Algorithmus exponentiell in der Größe der betrachteten Graphen, aber nur linear in der Anzahl der Regeln und verbotenen Graphmuster ist. Damit hat der Algorithmus zwar eine sehr hohe Komplexität, allerdings gilt diese nur für den schlechtesten Fall. In der Praxis gilt jedoch, dass die Graphen der Regeln und verbotenen Graphmuster relativ klein sind, sodass die hohe Komplexität eine Verifikation nicht verhindert. Zudem wurde bei der Komplexitätsberechnung davon ausgegangen, dass alle Elemente denselben Typ haben. Auch dies trifft in der Praxis nicht zu. In Abschnitt 5.3 wird die Umsetzung des Ansatzes in der Fujaba Real-Time Tool Suite beschrieben und in Abschnitt 5.4 eine Evaluierung anhand eines kleinen Beispiels durchgeführt. Diese Evaluierung zeigt, dass der Algorithmus trotz seiner Komplexität anwendbar ist.

## 3.8 Zusammenfassung

Das Kapitel 2 stellt einen Ansatz vor, in dem Methoden mechatronischer Systeme mittels Story Patterns beschrieben werden. Für diese Story Patterns wird ein Verfahren benötigt, dass die Verifikation der Story Patterns ermöglicht.

Da Story Patterns eine eingeschränkte Form von Graphtransformationsregeln darstellen, ist es möglich zu ihrer Überprüfung Verfahren zur Verifikation von Graphtransformationssystemen einzusetzen. Allerdings benötigen existierende Ansätze zum einen einen Initialgraphen, zum anderen sind die meisten existierenden Ansätze nur anwendbar, wenn der durch die Graphtransformationsregeln aufgespannte Zustandsraum endlich ist (siehe dazu Kapitel 6). Diese beiden Einschränkungen können für die Story Patterns der mechatronischen Systeme nicht gewährleistet werden. Deshalb wurde in diesem Kapitel ein Ansatz vorgestellt, der nicht diesen Einschränkungen unterliegt.

Im vorgestellten Ansatz werden die zu verifizierenden strukturellen Eigenschaften als (verbotene) Graphmuster beschrieben. Um auch unendliche Graphtransformationssysteme verifizieren zu können, wird weder eine Erreichbarkeitsanalyse durchgeführt, noch werden konkreten Graphen analysiert. Stattdessen werden Mengen von Graphen durch Ergebnisgraphmuster (bestehend aus der rechten Seite einer Regel und einem verbotenen Graphmuster) beschrieben. Für ein solches Ergebnisgraphmuster wird geprüft, ob es das Resultat einer Regelanwendung auf ein korrektes Graphmuster (das so genannte Startgraphmuster) ist. Ist dies der Fall, so wurde gezeigt, dass die betrachtete Regel einen korrekten Graphen in einen inkorrekten überführen kann. Dies gilt immer dann, wenn ein Graph das Startgraphmuster enthält und auf dieses Startgraphmuster die betrachtete Regel angewendet wird. Durch die Anwendung der Regel auf das Startgraphmuster resultiert ein Graph, der das Ergebnisgraphmuster enthält. Das Startgraphmuster zusammen mit der Regel und dem Ergebnisgraphmuster bilden in einem solchen Fall ein Gegenbeispiel.

Am Ende des Kapitels wurde ein Algorithmus vorgestellt, der den Ansatz in Pseudocode-Notation beschreibt. Die Aufwandsabschätzung ergab, dass der vorgestellte Algorithmus linear in der Anzahl der Regeln und verbotenen Graphmuster ist, aber exponentiell in der Größe der Graphen. Das folgende Kapitel zeigt informal, dass Story Patterns eine eingeschränkte Form der in diesem Kapitel vorgestellten Graphtransformationsregeln, bzw. verbotenen Graphmuster sind. In Kapitel 5 wird die prototypische Umsetzung des Ansatzes für Story Patterns in der Fujaba Real-Time Tool Suite beschrieben. In der Evaluierung wird dann mittels eines kleinen Beispiels gezeigt, dass die Laufzeit in der Praxis nicht so schlecht ist, wie die Aufwandsabschätzung ergeben hat. Dies gilt deshalb, da in der Praxis die Graphen der Regeln sowie der verbotenen Graphmuster relativ klein und vor allem typisiert sind, wodurch weniger Ergebnisgraphmuster gebildet werden können.

# Kapitel 4

# Story Patterns und Graphtransformationssysteme

Im voran gegangenen Kapitel wurden Graphtransformationssysteme eingeführt, wie sie auch in der Literatur (beispielsweise in [Roz97]) verwendet werden. Die Anwendbarkeit der Graphtransformationsregeln wurde im $DPO^{iso}$ eingeschränkt. Zudem wurde definiert, wie eine Regel mit negativer Anwendungsbedingung rückwärts angewendet werden kann. Sicherheitseigenschaften wurden in Form von verbotenen Graphmustern beschrieben. In Abschnitt 3.7 wurde dann ein Verfahren vorgestellt, mit dem man nachweisen kann, dass die Anwendung der Regeln keinen Graphen erzeugen kann, der eines der verbotenen Graphmuster als Teilgraphmuster enthält und somit inkorrekt ist.

In Kapitel 2 wurden Story Patterns vorgestellt und dazu verwendet, um die von den Real-Time Statecharts aufgerufenen Methoden modellieren zu können. Außerdem wurden verbotene Story Patterns eingeführt, um kritische Situationen und Unfälle zu modellieren.

Die Story Patterns stellen eine spezielle Art von Graphtransformationsregeln dar. Sie unterscheiden sich von den in Kapitel 3 eingeführten Graphtransformationsregeln hauptsächlich in ihrer Syntax. Während Graphtransformationsregeln aus zwei Graphen, der linken und der rechten Regelseite, bestehen, besteht ein Story Patterns nur aus einem Graphen, der beide Regelseiten enthält. Zudem werden in den zuvor beschriebenen Graphtransformationssystemen negative Anwendungsbedingungen als zusätzliche Graphen zur linken Regelseite hinzugefügt, während sie in Story Patterns direkt in den Graphen eingefügt werden, der die linke und die rechte Regelseite beschreibt. Daraus resultiert eine schwächere Ausdrucksstärke der negativen Anwendungsbedingungen von Story Patterns. Schränkt man die Anwendbarkeit der Story Pattern zudem noch so ein, dass sie im $DPO^{iso}$ angewendet werden, so ist es möglich, den im vorangegangenen Kapitel vorgestellten Verifikationsansatz dazu zu verwenden, um für die Story Patterns nachzuweisen, dass ihre Anwendung kein Objektdiagramm erzeugt, auf das ein verbotenes Story Pattern angewendet werden kann. Das bedeutet, die Anwendung eines Story Patterns auf ein korrektes Objektdiagramm resultiert immer wieder in einem korrekten Objektdiagramm.

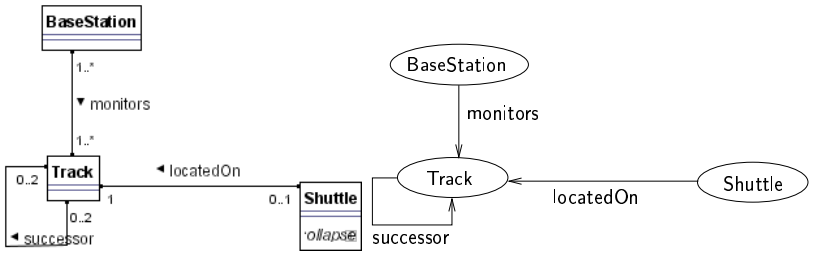
In diesem Kapitel soll informal gezeigt werden, dass Story Patterns auf die in Kapitel 3 eingeführten Graphtransformationsregeln abgebildet werden können und somit der Verifikationsansatz auch zur Verifikation von Story Pattern anwendbar ist.

## 4.1 Objektdiagramme und Graphen

Wie bereits in Abschnitt 3.4.1 erläutert wurde, kann ein Klassendiagramm auf einen Typgraphen abgebildet werden. Dabei wird jede Klasse auf einen Knoten und jede Assoziation auf eine Kante abgebildet. Klassen- und Assoziationsnamen werden den Knoten und Kanten durch die entsprechenden Beschriftungsfunktionen zugewiesen. In den hier verwendeten Klassendiagrammen können ungerichtete Assoziationen enthalten sein. Bei der Abbildung auf einen Graphen wird für diese Assoziationen eine Richtung festgelegt.

Dementsprechend kann ein Objektdiagramm auf einen typisierten Graphen abgebildet werden. Dabei wird jedes Objekt des Objektdiagramms auf einen Knoten des Graphen abgebildet. Die Beschriftung des Knotens entspricht dem Typ sowie dem Namen des Objekts. Ebenso wird jeder Link des Objektdiagramms auf eine Kante abgebildet, deren Beschriftung dem Objekttyp und -namen entspricht. Enthält ein Objektdiagramm ungerichtete Links, so werden diese auf gerichtete Kanten abgebildet, wobei ihre Richtungen den im Typgraph definierten Richtungen entsprechen müssen.

Abbildung 4.1(a) zeigt noch einmal das einfache Klassendiagramm aus Abschnitt 2.1.3, Abbildung 4.1(b) zeigt den entsprechenden Typgraphen. Das Objektdiagramm in Abbildung 4.2(a) stellte eine mögliche Instanzierung des Klassendiagramms dar, der zu ihm äquivalente typisierte Graph ist in Abbildung 4.2(b) zu sehen. Gleiche Namen von Objekten und Knoten sowie Links und Kanten stellen den Zusammenhang zwischen den beiden Diagrammen dar.



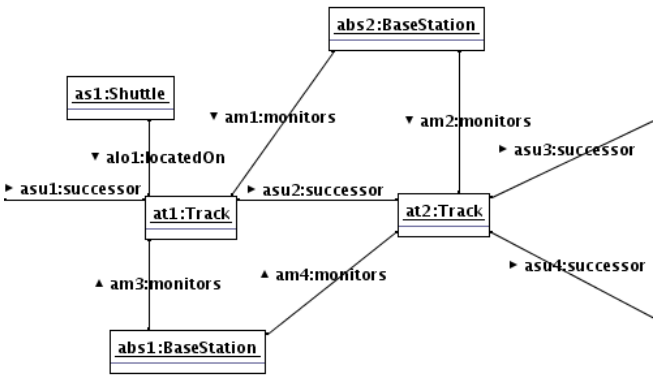

(a) Einfaches Klassendiagramm

(b) Typgraph, der dem obigen Klassendiagramm entspricht

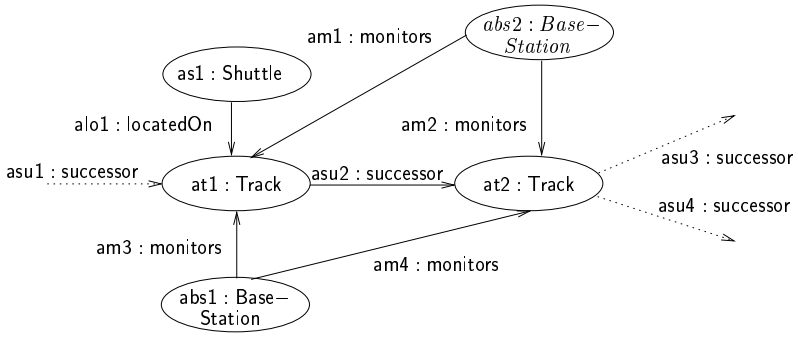
Abbildung 4.1: Klassendiagramm und entsprechender Typgraph

## 4.2 Story Patterns und Graphtransformationsregeln

Ein Story Pattern (siehe [FNTZ98, Zün01] und Abschnitt 2.2.2) ist eine Graphtransformationsregel, bei der die linke und die rechte Regelseite in einem Graphen zusammengefasst sind. Die Objekte des Story Patterns entsprechen dabei den Knoten der Graphtransformationsregel und die Links den Kanten. Die Objekte und Links eines Story Patterns können, zusätzlich zum Namen und zum Typ, mit ≪ destroy ≫ oder mit ≪ create ≫ beschriftet sein. Alle Elemente, die nicht




(a) Ausschnitt eines Objektdiagramms, das eine mögliche Instanzierung des Klassendiagramms aus Abbildung 4.1(a) darstellt




(b) Typisierter Graph, der dem Objektdiagramm aus Abbildung 4.2(a) entspricht

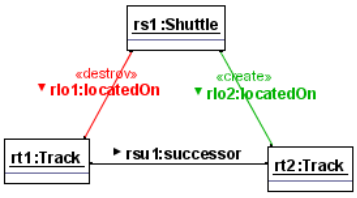
Abbildung 4.2: Objektdiagramm und entsprechender typisierter Graph

mit ≪ create ≫ beschriftet sind, stellen die Anwendungsbedingung der Graphtransformationsregel dar. Ebenso entsprechen alle Elementen, die nicht mit ≪ destroy ≫ beschriftetet sind, der Nachbedingung der Graphtransformationsregel.

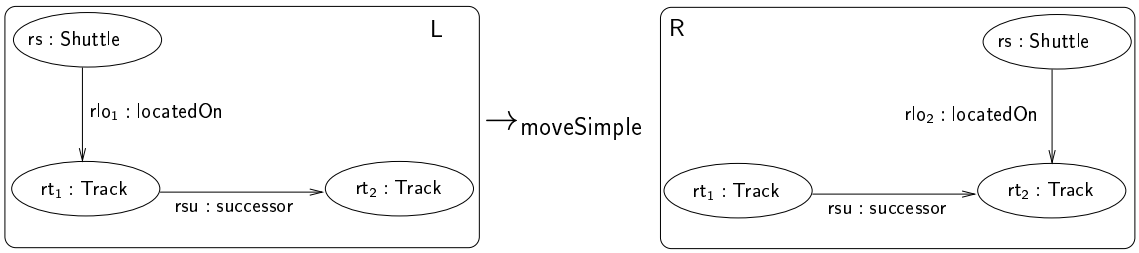
Abbildung 4.3 zeigt die Vorwärtsbewegung eines Shuttles, moveSimple, zum einen als Story Pattern und zum anderen als Graphtransformationsregel.

### 4.2.1 Anwendung von Story Patterns und Graphtransformationsregeln

Die Anwendung eines Story Patterns auf ein Objektdiagramm erfolgt im Single Pushout Ansatz. Allerdings muss das Matching, dass die Anwendungsbedingung auf einen Teil des Objektdiagramms abbildet, ein Teilgraphisomorphismus sein. Auf diese Weise wird die Identifikationsbedingung erfüllt und Konflikte vermieden, bei denen zu löschende Objekte auf dasselbe Objekt




(a) Vorwärtsbewegung als Story Pattern




(b) Vorwärtsbewegung als Graphtransformationsregel

Abbildung 4.3: Vorwärtsbewegung, moveSimple, eines Shuttles als Story Pattern und als Graphtransformationsregel

abgebildet werden wie Objekte, die durch die Anwendung des Story Patterns erhalten bleiben. Entstehen bei der Anwendung eines Story Patterns lose Kanten, so werden sie implizit gelöscht. Aufgrund dieses impliziten Löschens von losen Kanten ist eine Rückwärtsanwendung eines Story Patterns nicht immer möglich.

Im oben vorgestellten Verifikationsansatz wurde der DPO$^{iso}$ Ansatz für die Anwendung von Graphtransformationsregeln eingeführt. Wie bei der Anwendung von Story Patterns verlangt auch der DPO$^{iso}$ Ansatz, dass das Matching der linken Regelseite auf einen Teilgraphen des Anwendungsgraphen ein Graphisomorphismus ist. Im Gegensatz zu den Story Patterns muss die Lose-Kanten-Bedingung im DPO$^{iso}$ Ansatz jedoch explizit erfüllt werden. Das bedeutet, ein Knoten darf nur dann gelöscht werden, wenn dadurch keine losen Kanten entstehen.

Da sowohl das Matching der Story Patterns als auch der Graphtransformationsregeln ein Graphisomorphismus sein muss, kann ein Story Pattern, das keine Objekte löscht, genau dann angewendet werden, wenn auch die entsprechende Graphtransformationsregel angewendet werden kann.

Um zu erzwingen, dass dies auch gilt, wenn das Story Pattern ein Objekt löscht, muss das Story Pattern um zusätzliche negative Objekte und Links erweitert werden. Im Klassendiagramm ist definiert, welche Klassen adjazent zur Klasse des zu löschenden Objektes sind. Für jede dieser Klassen wird ein negatives Objekt in das Story Pattern eingefügt. Jedes der neu erzeugten negativen Objekte wird über eine Kante mit dem zu löschenden Objekt verbunden. Diese Erweiterung entspricht der Erweiterung der Graphtransformationsregeln aus Definition 27.

Durch die Erweiterung der Story Patterns um zusätzliche negative Knoten und Kanten wird erreicht, dass auch diese im $DPO^{iso}$ Ansatz angewendet werden und somit immer eine Rückwärtsanwendung der Story Patterns möglich ist.

### 4.2.2 Negative Anwendungsbedingungen

Sowohl bei Story Patterns als auch bei Graphtransformationsregeln können negative Anwendungsbedingungen angegeben werden. Ihre Syntax und Semantik unterscheiden sich zwar, allerdings können die negativen Anwendungsbedingungen von Story Patterns auf negative Anwendungsbedingungen von Graphtransformationsregeln abgebildet werden.

In Story Patterns wird die negative Anwendungsbedingung direkt in Form von negativen Elementen in den Graph eingefügt, der sowohl die Anwendungs- als auch die Nachbedingung des Story Patterns spezifiziert.

Kann die Anwendungsbedingung eines Story Patterns auf ein Objektdiagramm abgebildet werden, jedoch keines der negativen Elemente, so kann das Story Pattern angewendet werden. Enthält ein Story Pattern mehrere negative Elemente und es gibt ein Match für mindestens eines der negativen Elemente im Objektdiagramm, so ist die Anwendung des Story Patterns verboten. Enthält ein Story Pattern einen negativen Knoten mit mehreren inzidenten Kanten, so ist seine Anwendung genau dann verboten, wenn sowohl der negative Knoten als auch alle seine inzidenten Kanten auf das Objektdiagramm abgebildet werden können.

Abbildung 4.4 zeigt ein Beispiel für ein Story Pattern mit negativer Anwendungsbedingung. Dieses Story Pattern, createDC, erzeugt für zwei Shuttles eine Instanz des DistanceCoordination-Musters, falls die beiden Shuttles auf zwei Tracks sind, zwischen denen sich ein weiterer Track befindet. Die negative Anwendungsbedingung verlangt, dass zwischen den beiden Shuttles noch kein DistanceCoordination-Muster existiert, bei dem das Shuttle rs1 die rearRole und das Shuttle rs2 die frontRole spielt.

Das Story Pattern darf genau dann auf ein Objektdiagramm angewendet werden, wenn die Anwendungsbedingung des Story Patterns auf das Objektdiagramm abgebildet werden kann. Es darf jedoch kein Objekt vom Typ DistanceCoordination geben, auf das rdc2 oder rdc3 abgebildet werden können und auf dessen inzidenten Link der zu rdc2 bzw. rdc3 inzidente Link abgebildet werden kann. Das bedeutet, dass das Muster nur dann erzeugt wird, wenn es noch kein DistanceCoordination-Muster gibt, in dem das Shuttle rs1 die rearRole übernommen hat und kein Muster, in dem rs2 die frontRole spielt.

In Graphtransformationsregeln werden die negativen Anwendungsbedingungen durch zusätzliche Graphen beschrieben. Eine Graphtransformationsregel mit negativer Anwendungsbedingung darf angewendet werden, wenn ihre Anwendungsbedingung auf einen Teilgraphen des Anwendungsgraphen abgebildet werden kann, eine Abbildung eines kompletten Graphen der negativen Anwendungsbedingung jedoch nicht möglich ist.

Enthält ein Story Pattern mehrere negative Elemente, so muss bei der Abbildung auf Graphtransformationsregeln für jedes der Elemente ein Graph zur negativen Anwendungsbedingung hinzugefügt werden. Jeder dieser Graphen erweitert die Anwendungsbedingung, d.h. die Anwendungsbedingung stellt einen Teilgraphen des neu erzeugten Graphen dar. Bei der Abbildung

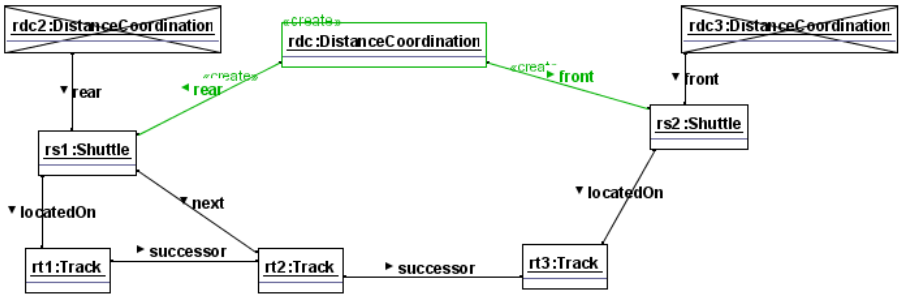



Abbildung 4.4: Story Pattern createDC, das die Erzeugung eines DistanceCoordination-Musters zeigt

eines negativen Links wird eine entsprechende Kanten in den Graphen eingefügt. Bei einem negativen Objekt wird der Graph um einen entsprechenden Knoten erweitert. Zusätzlich wird für jeden inzidenten Link eine Kante in den Graphen eingefügt.

Abbildung 4.5 zeigt die Graphtransformationsregel createDC, die äquivalent zu dem gleichnamigen Story Pattern ist.

## 4.3 Kritische Situationen und Unfälle

Auch kritische Situationen und Unfälle, die als verbotene Story Patterns gegeben sind, können mittels verbotenen Graphmustern dargestellt werden. Dazu wird die Anwendungsbedingung des verbotenen Story Patterns auf die Anwendungsbedingung eines Graphmusters abgebildet. Enthält das verbotene Story Pattern negative Elemente, so wird die negative Anwendungsbedingung des Graphmusters genau so gebildet, wie im Fall der Graphtransformationsregeln.

Abbildung 4.6(a) zeigt die kritische Situation impendingCollision als Story Pattern und Abbildung 4.6(b) das dazu äquivalente Graphmuster.

## 4.4 Zusammenfassung

In Kapitel 2 wurden Story Patterns eingeführt, um die Methoden, die von einem Real-Time Statechart aufgerufen werden, modellieren zu können. Zudem wurden verbotene Story Patterns eingeführt, mit denen kritische Situationen und Unfälle modelliert werden können. In Kapitel 3 wurde ein Verfahren erläutert, dass eine Verifikation von Graphtransformationsregeln ermöglicht. Bei diesem Verfahren werden die nachzuweisenden Eigenschaften als verbotene Graphmuster beschrieben.

In diesem Kapitel wurde informal gezeigt, dass die in Kapitel 2 eingeführten Story Patterns eine eingeschränkte Form der in Kapitel 3 definierten Graphtransformationsregeln sind. Ebenso stellen die verbotenen Story Patterns eine eingeschränkte Form der verbotenen Graphmuster dar. Somit ist es durch den Ansatz aus Kapitel 3 möglich, nachzuweisen, dass die Anwendung

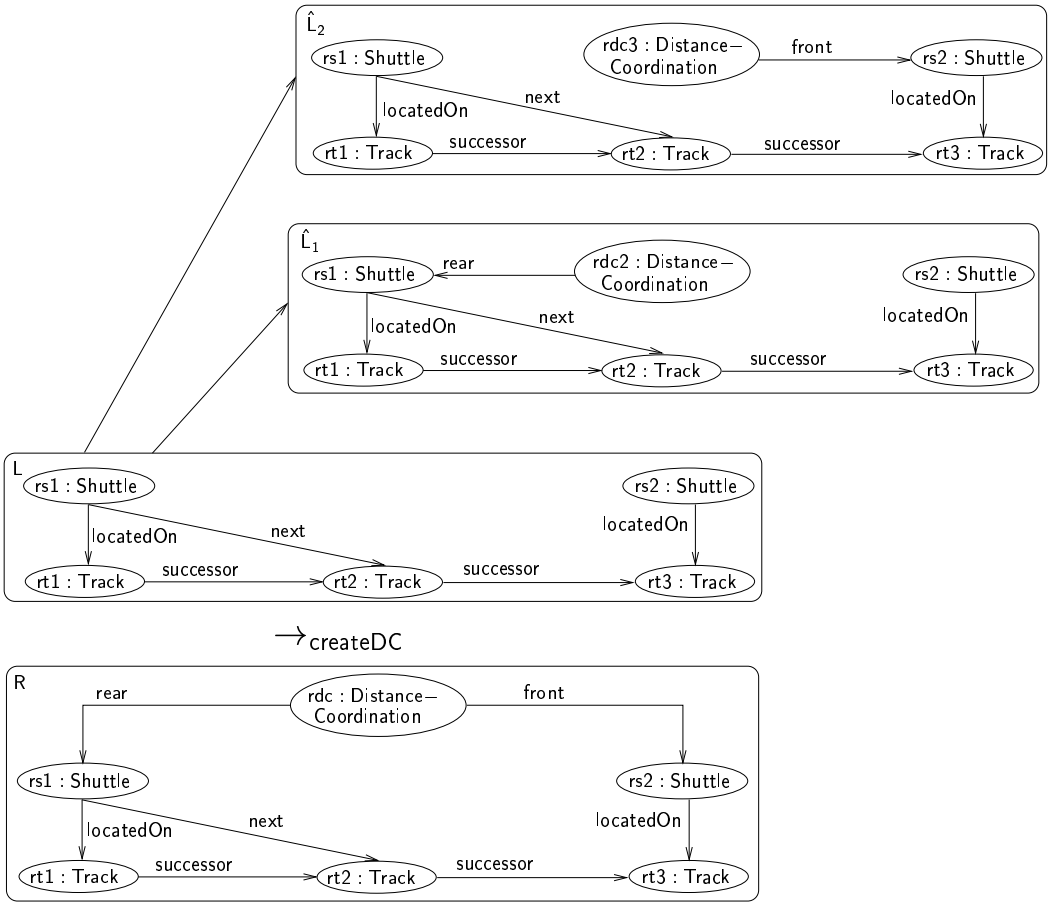
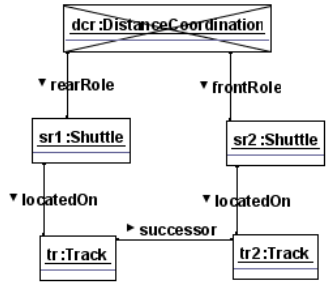


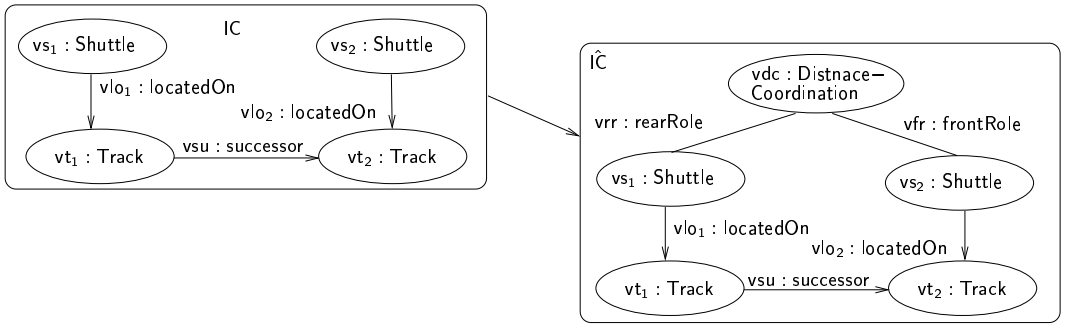

Abbildung 4.5: Graphtransformationsregel createDC

der Story Patterns nicht zu einer kritischen Situation oder einem Unfall (beschrieben durch ein verbotenes Story Pattern) führen kann.

Da eine manuelle Verifikation sowohl zeitaufwändig als auch fehleranfällig ist, wird im nachfolgenden Kapitel gezeigt, wie der Ansatz automatisiert werden kann.




(a) Kritische Situation impendingCollision als Story Pattern




(b) Kritische Situation impendingCollision als verbotenes Graphmuster

Abbildung 4.6: Die kritische Situation impendingCollision als Story Pattern und äquivalentes verbotenes Graphmuster

# Kapitel 5

# Werkzeugunterstützung

In den vorangegangenen Kapiteln wurde die Modellierung und Verifikation der Software mechatronischer Systeme vorgestellt. In diesem Kapitel soll nun die Umsetzung der Konzepte in der Fujaba Real-Time Tool Suite[1] erläutert werden.

Fujaba wird seit 1997 im Fachgebiet für Softwaretechnik an der Universität Paderborn entwickelt. Das ursprüngliche Ziel von Fujaba bestand in einem Roundtrip Engineering, daher auch der Name Fujaba - From UML to Java and Back Again. Die Modellierung eines Softwaresystems erfolgt mittels verschiedener UML-Diagramme. Aus den derart spezifizierten Modellen kann lauffähiger Code erzeugt und zum Testen verwendet werden [Gei02]. Fujaba ermöglicht es jedoch auch, Java-Code einzulesen und als UML-Diagramme darstellen zu lassen [NNWZ00, NNZ00].

2003 fand ein Redesign von Fujaba statt und Fujaba wurde zur Fujaba Tool Suite umstrukturiert. Die Fujaba Tool Suite unterstützt ein Plugin-Konzept, das es ermöglicht, neue Diagrammarten und Algorithmen in die Tool Suite einzufügen [Wen03]. Dieser Plugin-Mechanismus wurde dazu verwendet, um die Fujaba Real-Time Tool Suite zu ermöglichen, in der die Modellierung und Verifikation der Software von mechatronischen Systemen umgesetzt ist.

Im folgenden Abschnitt wird gezeigt, wie die Fujaba Real-Time Tool Suite den Prozess zur Modellierung und Verifikation der Software mechatronischer Systeme unterstützt und einzelne Phasen automatisiert. Im Kontext dieser Arbeit sind zwei neue Plugins entwickelt worden, die den Entwickler dabei unterstützen, korrekte Story Patterns zu modellieren. Das erste Plugin bindet dazu den Simulator und Model Checker Groove an Fujaba an (Abschnitt 5.2). Das Plugin in Abschnitt 5.3 nutzt die Tatsache aus, dass es sich bei Story Patterns um eine eingeschränkte Form der in Kapitel 3 vorgestellten Graphtransformationsregeln handelt (siehe Kapitel 4). Es stellt eine Umsetzung des in Kapitel 3 vorgestellten Verifikationsansatzes für Story Patterns dar. In Abschnitt 5.4 wird das bereits eingeführte Shuttle-Beispiel mittels des zweiten Plugins verifiziert.

[1] www.fujaba.de

## 5.1 Werkzeugunterstützung des Modellierungs- und Verifikationsprozesses

In Abschnitt 2.5 wurde der Prozess zur Modellierung und Verifikation der Software mechatronischer Systeme vorgestellt. In diesem Abschnitt wird nun gezeigt, wie die Fujaba Real-Time Tool Suite die Phasen dieses Prozesses unterstützt bzw. automatisiert. Der Prozess lässt sich grob in die drei Aufgaben Modellierung, Verifikation und Codegenerierung unterteilen.

### 5.1.1 Modellierung

Wie in Kapitel 2 beschrieben, wird die Architektur der Software eines mechatronischen Systems zum einen mittels Komponentendiagrammen beschrieben. Die Kommunikation zwischen den Komponenten wird mittels Koordinationsmustern modelliert. Der interne Aufbau einer Komponente sowie ihre Datenstrukturen werden mittels Klassendiagrammen spezifiziert.

Die Kommunikationsprotokolle, die durch die Muster festgelegt werden sowie das komponenteninterne Verhalten werden durch Real-Time Statecharts angegeben. Die komponenteninternen Real-Time Statecharts können Methoden aufrufen (entry()-, do()-, exit()-Methoden in den Zuständen und Methoden als Seiteneffekte an den Transitionen). Diese Methoden werden mit Hilfe von Story Patterns beschrieben.

Klassendiagramme und Story Patterns gehören zum Kern von Fujaba. Koordinationsmuster, Komponenten und Real-Time Statecharts wurden in der Fujaba Real-Time Tool Suite umgesetzt [BGHS04, BGH+05, HG03].

Somit wird durch die Fujaba Real-Time Tool Suite die Modellierung der Softwarearchitektur und des Koordinationsverhaltens unterstützt.

### 5.1.2 Verifikation

Nachdem die Softwarearchitektur und das Koordinationsverhalten spezifiziert wurden, erfolgt die Verifikation des Systems.

Das Model Checking von Koordinationsmustern und Komponenten mit den dazugehörigen Real-Time Statecharts wurde in verschiedenen Plugins der Fujaba Real-Time Tool Suite umgesetzt. Dazu wurde zunächst eine Schnittstelle entwickelt, die eine Anbindung verschiedener Model Checker zulässt. In [Hir04, BGHS04, BGH+05, HG03] wurde die Anbindung des Model Checkers UPPAAL eingeführt und in [Ste05] die Anbindung von RAVEN. Neben der Anbindung verschiedener Model Checker ermöglicht diese Schnittstelle auch, den Model Checking Prozess in den Hintergrund zu verlegen. Das heißt, der Konsistenzmechanismus von Fujaba erkennt, dass am Modell eine Änderung vorgenommen wurde und stellt fest, welche Eigenschaften dadurch beeinflusst werden. Fujaba stößt dann automatisch den Model Checking Prozess für das entsprechende Koordinationsmuster oder die entsprechende Komponente und die betroffene Eigenschaft an. Das model Checking wird dann im Hintergrund durchgeführt.

In den Real-Time Statecharts der Koordinationsmustern und Komponenten werden Methoden ausgeführt. Diese werden durch Story Patterns beschrieben. Die Story Patterns müssen ei-

ne Menge von Sicherheitseigenschaften, beschrieben durch verbotene Story Patterns, einhalten. Um die Entwicklung der Story Patterns zu unterstützen, wurde ein Plugin entwickelt, das Groove (einen Model Checker und Simulator für Graphtransformationssysteme) an Fujaba anbindet. Dieses Plugin bietet die Möglichkeit, das Verhalten des durch die Story Patterns beschriebenen Systems für einen gegebenen Anfangszustand zu beobachten. Können vom Anfangszustand aus nur endlich viele verschiedene Zustände erreicht werden, so kann Groove mittels einer Erreichbarkeitsanalyse prüfen, ob die Story Patterns einen inkorrekten Zustand erzeugen können. Dieses Plugin in ist in Abschnitt 5.2 beschrieben. Für die eigentliche Verifikation von Story Patterns wurde im Kontext dieser Arbeit ein Plugin entwickelt, das für eine Menge von Story Patterns und eine Menge von verbotenen Story Patterns eine automatische Überprüfung durchführt. Dabei wird festgestellt, ob das „Nicht-Auftreten“der verbotenen Story Patterns eine induktive Invariante des Systems darstellt. Das Plugin setzt somit den Ansatz aus Kapitel 3 für Story Patterns um und ist in Abschnitt 5.3 beschrieben.

### 5.1.3 Codegenerierung

Durch die Verifikation wird garantiert, dass das System eine Menge von Sicherheitseigenschaften erfüllt. Allerdings wurde bisher immer das Modell des Systems betrachtet und nicht der Code, der im mechatronischen System wirklich ausgeführt wird. Um sicherstellen zu können, dass auch der Code die für das Modell nachgewiesenen Eigenschaften erfüllt, ist es notwendig, den Code automatisch aus dem verifizierten Modell zu generieren.

Die Codegenerierung für Klassendiagramme und Story Patterns ist Bestandteil des Fujaba Kerns [FNT98, FNTZ98, NNWZ00, NNZ00]. Für Koordinationsmuster und Komponenten sowie deren Real-Time Statecharts wurde die Codegenerierung in der Fujaba Real-Time ToolSuite umgesetzt [Bur02].

## 5.2 Das Groove-Plugin

In [Ren04] stellt Rensink Groove, ein Werkzeug zur Simulation und Verifikation von Graphtransformationssystemen, vor. Dabei steht Groove für GRaph-based Object-Oriented VErification. Zurzeit unterstützt Groove eine Erreichbarkeitsanalyse für strukturelle Eigenschaften. Die zu überprüfenden Eigenschaften werden dabei als Graphtransformationsregel angegeben. Der Simulator von Groove kann so konfiguriert werden, dass er für eine derart spezifizierte Eigenschaft überprüft, ob sie in jedem erreichbaren Zustand des Systems erfüllt ist oder ob sie in keinem erreichbaren Zustand erfüllt ist. Dazu prüft der Simulator für alle Graphen, die die erreichbaren Zustände des Systems beschreiben, ob die entsprechende Regel auf sie anwendbar ist. Regeln, die auf jeden Zustand anwendbar sein müssen, werden im Folgenden als geforderte Regeln oder Eigenschaften bezeichnet und solche, die nie anwendbar sein dürfen, als verbotene Regeln oder Eigenschaften.

Groove benötigt als Eingabe einen Startgraphen, eine Menge von Graphtransformationsregeln sowie entweder eine geforderte oder eine verbotene Regel.

Die Graphen in Groove sind gerichtete Graphen mit beschrifteten Kanten. Die Notation der Regeln ist ähnlich zu denen von Story Patterns. Die linke und rechte Regelseite einer Graphtransformationsregel werden in einem Graphen zusammengefasst. Elemente, die durch die Anwendung einer Regel gelöscht oder erzeugt werden, werden entsprechend gekennzeichnet. Wie in Story Patterns wird die negative Anwendungsbedingung in Form von negativen Knoten und Kanten zum Diagramm hinzugefügt.

Im Rahmen dieser Arbeit wird Groove dazu verwendet, um für eine Menge von Story Patterns deren Anwendbarkeit zu untersuchen und zu prüfen, ob ihre Anwendung ausgehend von einem gegebenen Anfangszustand eine kritische Situation oder Unfall verursachen kann. Dazu ist es notwendig, die Story Patterns in die Eingabesprache von Groove zu transformieren.

Da Fujaba keine Objektdiagramme unterstützt, werden zu Evaluierungszwecken Story Patterns verwendet, um den Startgraphen eines Graphtransformationssystems zu modellieren. Diese Story Patterns dürfen weder zu löschende oder zu erzeugende Elemente enthalten, noch dürfen sie negative Knoten oder Kanten besitzen. Die zu verifizierenden Regeln werden in Fujaba mittels Story Patterns beschrieben. Ebenso werden die verbotenen oder geforderten Eigenschaften als Story Patterns gegeben.

Der in Fujaba realisierte Export von Story Patterns in die Eingabesprache von Groove, die Simulation und die Verifikation mittels Groove und die Darstellung eines von Groove erzeugten Gegenbeispiels in Fujaba soll anhand des folgenden Beispiels erläutert werden. Der Startgraph entspricht dem in Abbildung 5.1 gegebenen Story Pattern. Dieses Story Pattern beschreibt ein Shuttle-System bestehend aus fünf Tracks, die im Kreis angeordnet sind sowie zwei Shuttles, die sich auf jeweils einem Track befinden. Die zu verifizierende Regel ist die Regel moveSimple, die in Abbildung 5.2 geben ist. Die Regel darf niemals dazu führen, dass sich zwei Shuttles auf dem selbem Track befinden. Diese Eigenschaft ist als verbotenes Story Pattern in Abbildung 5.3 gegeben und entspricht einem verbotenen Graphmuster.

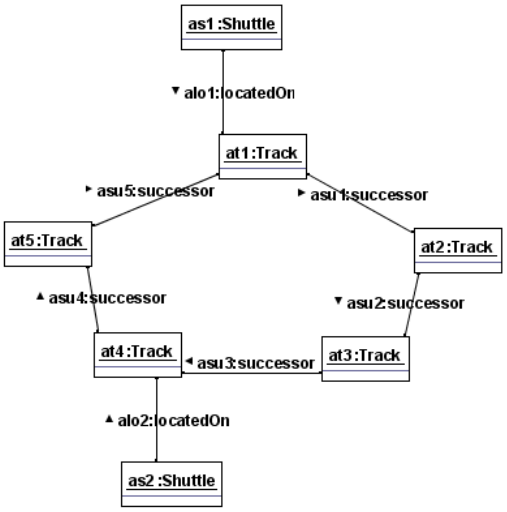



Abbildung 5.1: Startgraph des zu verifizierenden Graphtransformationssystems

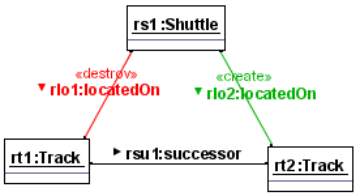



Abbildung 5.2: Zu verifizierende Regel moveSimple

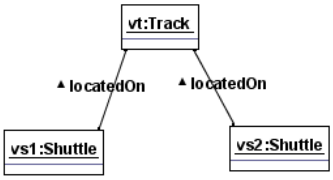



Abbildung 5.3: Verbotenes Story Pattern collision

### 5.2.1 Export von Story Patterns

Bevor der Export der Story Patterns in die Eingabesprache von Groove erfolgen kann, muss für die einzelnen Story Patterns angegeben werden, ob es sich dabei um einen Startgraphen, eine Regel oder ein gefordertes oder verbotenes Story Pattern handelt. Wird für ein Story Pattern angegeben, dass es sich dabei um eine Regel handelt, so besteht die Möglichkeit, für diese Regel noch eine Priorität anzugeben. Das bedeutet, wenn auf ein Objektdiagramm zwei Story Patterns angewendet werden können, so wird immer das Story Pattern mit der höheren Priorität angewendet. Haben beide Story Patterns die gleiche Priorität, so wird nichtdeterministisch eines der beiden Story Patterns angewendet.

Da die Transformation für Startgraphen, Regeln und verbotene Graphmuster gleich ist, soll sie hier am Beispiel der Regel moveSimple aus Abbildung 5.2 erläutert werden.

Groove unterscheidet zwischen einer internen Darstellung und einer graphischen Darstellung. Im Folgenden werden jeweils beide Darstellungen angegeben.

Für jedes Objekt im Story Pattern wird ein Knoten in Groove erzeugt. In Story Patterns können sowohl die Objekte, als auch die Links beschriftet sein. In Groove können dagegen nur die Kanten beschriftet sein. Um auch einem Knoten eine Beschriftung zuweisen zu können, muss eine Selbstkante eingefügt werden. Die Beschriftung des Knotens entspricht dann der Beschriftung dieser Kante. Soll einem Knoten sowohl ein Name als auch ein Typ zugewiesen werden, so müssen dafür zwei Kanten in das Modell eingefügt werden. Daraus resultieren jedoch Unterschiede bei der Anwendung eines Story Patterns und der entsprechenden Regel in Groove. Soll das Story Pattern angewendet werden, so wird überprüft, ob die Objekte des Story Patterns auf Objekte im Objektdiagramm abgebildet werden können, die den gleichen Typ haben. Der Name der Objekte wird bei dieser Abbildung jedoch nicht berücksichtigt. Soll dagegen die Regel in Groove angewendet werden, so ist das nur möglich, wenn alle Knoten und alle Kanten auf den Anwendungsgraphen abbildbar sind. Das bedeutet, dass die Regel nur dann anwendbar ist, wenn es einen Knoten im Anwendungsgraphen gibt, der den gleichen Namen wie der Knoten in

der Regel besitzt. Dadurch wird die Anwendbarkeit einer Regel im Vergleich zum dazugehörigen Story Pattern eingeschränkt. Deshalb wird hier nur der jeweilige Objekttyp nach Groove exportiert.

Für jeden Link des Story Patterns wird eine Kante in den Graph von Groove eingefügt, wobei die Kantenbeschriftung dem Linktypen entspricht.

Wird ein Element durch das Story Pattern gelöscht, erzeugt oder ist im Story Pattern als negatives Element angegeben, so wird die Kantenbeschriftung in Groove um das Präfix „del:“, „new:“bzw. „not:“erweitert. In der graphischen Darstellung wird das Präfix nicht angezeigt. Stattdessen werden zu löschende Elemente mit gestrichelten blauen Linien und blauer Schrift, zu erzeugende Elemente mit dicken grünen Linien und grüner Schrift und negative Elemente mit gestrichelten dicken roten Linien und roter Schrift dargestellt.

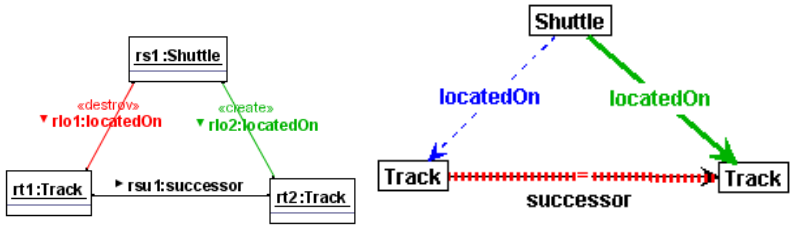
In Groove werden die Regeln im Single Pushout Ansatz angewendet. Resultieren aus einer Regelanwendung lose Kanten, so werden sie durch Groove gelöscht. Außerdem kann eine Regel auf einen Graphen angewendet werden, wenn es einen Graphhomomorphismus gibt, der die Anwendungsbedingung auf den Anwendungsgraphen abbildet. Ein solcher Homomorphismus darf auch zwei Knoten der Anwendungsbedingung auf einen Knoten des Anwendungsgraphen abbilden. Um zu erzwingen, dass die Regelanwendung im $DPO^{iso}$ Ansatz erfolgt, müssen zwei Modifikationen an den Regeln erfolgen. 1. Bevor ein Story Pattern, das eine Regel darstellt, nach Groove exportiert werden kann, muss es um zusätzliche negative Knoten erweitert werden. Diese zusätzlichen negativen Knoten verhindern eine Anwendung, sollten anderenfalls lose Kanten entstehen. Die Erweiterung der Story Patterns entspricht der Erweiterung von Graphtransformationsregeln, die in Definition 27 gegeben ist. 2. Muss verhindert werden, dass zwei Knoten der Regel auf den selben Knoten im Anwendungsgraphen abgebildet werden, d.h. es muss erzwungen werden, dass der Match der linken Regelseite einem Teilgraphisomorphismus entspricht. Dazu werden in Groove negative Kanten verwendet. Für jedes Paar von Knoten, die den selben Typ haben und entweder durch die Regelanwendung gelöscht oder explizit erhalten bleiben und nicht negativ sind, muss eine zusätzliche Kante eingefügt werden, die mit „not:“beschriftet ist. Für zwei negative Knoten mit dem selben Typen bzw. für einen negativen Knoten und einen positiven Knoten, der nicht durch die Regel erzeugt wird, wird eine zusätzliche mit „!not“beschriftete Kante eingefügt. In der graphischen Darstellung werden dann negative Kanten angezeigt, die mit „=“bzw. „!not“beschriftet sind.

Abbildung 5.4 zeigt die Regel moveSimple zum einen als Story Pattern und zum anderen als Groove-Regel. Dabei wurde die successor-Kante und die Kante, die anzeigt, dass die beiden Track-Knoten nicht auf den selben Knoten abgebildet werden dürfen, als eine Kanten dargestellt.

### 5.2.2 Simulation und Verifikation mit Groove

Nachdem der Startgraph, die Transformationsregeln und die geforderten bzw. verbotenen Story Patterns auf die Eingabesprache von Groove abgebildet wurden, wird Groove gestartet. Simulation und Verifikation erfolgen in Groove in Abhängigkeit davon, ob die Eingabe geforderte oder verbotene Regeln enthält.

Verbotene Regeln stellen kritische Situationen oder Unfälle dar. Eine kritische Situation oder ein Unfall ist eingetreten, wenn die entsprechende verbotene Regel auf einen Systemzustand,




(a) Die Regel moveSimple als Story Pattern

(b) Die Regel moveSimple dargestellt in Groove

Abbildung 5.4: Die Regel moveSimple als Story Pattern und als Groove-Regel

beschrieben durch einen Graphen, angewendet werden kann. Der Groove-Simulator erzeugt ausgehend vom Startzustand den Zustandsraum des Systems, in dem er alle möglichen Regeln zunächst auf den Anfangszustand und dann auf alle resultierenden Zustände anwendet. Der Aufbau des Zustandsraums endet an einem Graphen, wenn auf diesen entweder eine verbotene Regel angewendet werden kann und der Graph somit inkorrekt ist oder wenn keine Regel anwendbar ist. Der Zustandsraum wird als Baum dargestellt, wobei die Knoten den Zuständen entsprechen und die Kanten den Regelanwendungen. Aus Effizienzgründen wird für mehrere isomorphe Graphen nur ein Zustand erzeugt, sodass aus dem Baum ein Graph wird. Nach [HEWC97] wird dieser Graph als Graphtransitionssystem bezeichnet.

Wählt man einen der Knoten des Graphen aus, so wird der entsprechende Zustand angezeigt und die Anwendungsstelle der nächsten Regel hervorgehoben. Eine weitere Hilfe, die Groove bietet, besteht darin, dass der Simulator eine Liste enthält, in der die Namen aller angewendeten Regeln aufgeführt werden. Aus dieser Liste können dann Rückschlüsse auf die Anwendbarkeit der Regeln gezogen werden. Wurde eine Regel nicht angewendet, so kann dies am gewählten Initialgraphen liegen, es kann aber auch ein Indiz dafür sein, dass die Anwendungsbedingung zu stark ist und eine Anwendung der Regel verhindert.

### 5.2.3 Darstellung von Gegenbeispielen

Nachdem die Analyse des Systems abgeschlossen ist, kann die Darstellung des Zustandsraums in Groove dazu verwendet werden, um Gegenbeispiele zu finden, die zeigen, dass das System inkorrekt ist. Für eine geforderte Eigenschaft ist ein solches Gegenbeispiel ein Graph, auf den die entsprechende Regel nicht angewendet werden kann. Ein Gegenbeispiel für eine verbotene Eigenschaft ist ein Graph, auf den die entsprechende verbotene Regel angewendet werden kann.

Ziel ist es nun, solche Gegenbeispiele aus Groove zu exportieren und in der Fujaba Real-Time Tool Suite darzustellen. Da bei großen Systemen mit vielen verschiedenen Zuständen die Darstellung des Zustandsraums unübersichtlich wird, soll statt des gesamten Zustandsraums nur der kürzeste Pfad vom Startzustand zu einem inkorrekten Endzustand betrachtet werden. Ein solcher Pfad besteht aus einer Menge von Graphen und den darauf angewendeten Transformationsregeln. Da die Graphen groß und somit unübersichtlich werden können und es für eine Regel

unter Umständen mehrere Anwendgungsstellen in einem Graphen gibt, soll Groove so angepasst werden, dass neben den Zuständen des Systems auch die verwendeten Matchings der Regeln gespeichert werden[2].

Das Fenster, das das Gegenbeispiel anzeigt, ist dreigeteilt. Links oben wird eine Liste mit den Namen aller verbotenen Story Patterns des Systems angegeben, für die ein Gegenbeispiel gefunden wurde. Darunter ist eine Tabelle abgebildet, in der von oben nach unten der Pfad vom Startzustand zu einem Zustand aufgeführt ist, auf den das ausgewählte verbotene Story Pattern angewendet werden kann. Links steht der jeweilige Zustand, daneben steht die auf diesen Zustand angewendete Regel und der resultierende Zustand steht rechts. Wählt man eine Zeile aus, so wird der links angegeben Graph im rechten Teil des Fensters angezeigt und die Anwendungsstelle des Story Patterns wird farblich hervorgehoben. Die Anwendungsstelle eines Story Patterns wird grün hervorgehoben, die Anwendungsstelle eines verbotenen Story Patterns rot.

In Abbildung 5.5 ist ein Gegenbeispiel für das Story Pattern moveSimple und das verbotene Story Pattern collision gegeben.

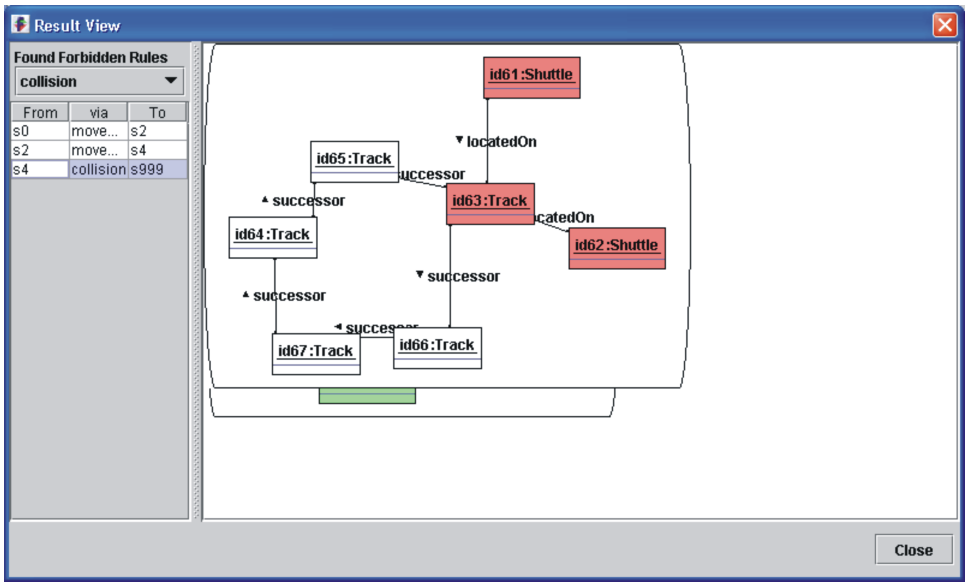



Abbildung 5.5: Von Groove erzeugtes Gegenbeispiel in Fujaba dargestellt

## 5.3 Plugin zum Nachweis von induktiven Invarianten

Existierende Ansätze zur Verifikation von Graphtransformationssystemen, wie zum Beispiel Groove, benötigen einen Startgraphen. Zudem muss sichergestellt werden, dass ausgehend von diesem Startgraphen nur endlich viele Graphen erreichbar sind (siehe Kapitel 6). In Kapitel 3 wurde dagegen ein Ansatz vorgestellt, der die Korrektheit eines Graphtransformationssystems

[2]Mit Rensink wurde das Austauschformat zwischen Fujaba und Groove definiert, sodass eine Umsetzung der Idee in Fujaba bereits erfolgen konnte.

auch dann sicher stellt, wenn kein Startgraph gegeben ist oder die Regeln unendlich viele Graphen erzeugen können. Dazu wird gezeigt, dass das „Nicht-Auftreten“eines verbotenen Graphmusters eine induktive Invariante des Systems ist. Wie in Kapitel 4 gezeigt wurde, stellen Story Patterns eine eingeschränkte Form der in Kapitel 3 eingeführten Graphtransformationssysteme dar. Deshalb ist es möglich, den Verifikationsansatz aus Kapitel 3 zur Verifikation von Story Patterns zu verwenden. In diesem Abschnitt soll das Fujaba Plugin vorgestellt werden, dass den Ansatz für Story Patterns umsetzt und vollautomatisch ausführt (siehe dazu auch [Bec05, BGS05a]).

Als Eingabe erwartet das Plugin eine Menge von Story Patterns sowie eine Menge von verbotenen Story Patterns. Im Gegensatz zu dem in Abschnitt 5.2 vorgestellten Groove-Plugin wird jedoch kein Initialgraph benötigt.

Im Folgenden werden das Story Pattern moveSimple (Abbildung 5.4(a)) und das verbotene Story Pattern collision (Abbildung 5.3) verwendet, um die Arbeitsweise der Funktionen zu beschreiben.

### 5.3.1 Verifikation

Story Patterns werden in Fujaba im Single Pushout Ansatz angewendet, wobei das Matching der Anwendungsbedingung ein Isomorphismus sein muss. Damit der Verifikationsansatz aus Kapitel 3 verwendet werden kann, müssen die Story Patterns im $DPO^{iso}$ Ansatz angewendet werden. Die Verwendung eines Isomorphismus zur Beschreibung des Matchings garantiert, dass die Identifikationsbedingung immer erfüllt ist. Damit auch die Lose-Kanten-Bedingung immer erfüllt wird, müssen die Story Patterns um negative Elemente erweitert werden, die eine Anwendung verhindern, wenn andernfalls lose Kanten resultieren. Die Erweiterung der Story Patterns erfolgt durch die Funktion RuleExtension.

Der Verifikationsalgorithmus bestimmt für jedes Story Pattern und jedes verbotene Story Pattern alle möglichen Abbildungen der Nachbedingung auf das verbotene Story Pattern. Diese Berechnung erfolgt mittels der Funktion Matching.

Für jede der durch Matching bestimmten Abbildungen wird ein neues Story Pattern erzeugt, das sowohl die Nachbedingung des Story Patterns als auch das verbotene Story Pattern enthält. Die Bildung dieses neuen Story Patterns ist die Aufgabe der Funktion Merge. Das neue Story Pattern entspricht einem Ergebnisgraphmuster.

Auf das neue Story Pattern wird das Story Pattern rückwärts angewendet. Das resultierende Story Pattern entspricht dann einem Startgraphmuster. Für dieses Story Pattern muss überprüft werden, ob es korrekt ist, d.h. ob es kein verbotenes Story Pattern enthält und es kein Match für die Anwendungsbedingung eines höher priorisierten Story Patterns im Startgraphmuster gibt. Für diese Überprüfung wird wieder die Funktion Matching verwendet.

#### Matching

Die Matching-Funktion berechnet, ob es für ein Story Pattern G1 ein Matching, in Form eines Teilgraphisomorphismus, in einem anderen Story Pattern G2 gibt, d.h. $G1 \preceq G2$. Ist dies der Fall, so wird das gefundene Matching zurückgegeben.

Die Funktion berechnet das Matching sukzessive. Sie beginnt mit einem Objektpaar, einem Objekt n1 aus G1 und einem Objekt n2 aus G2. Sie prüft, ob diese beiden Objekte denselben Typ haben und ob sie entweder beide positiv oder beide negativ sind. Ist dies nicht der Fall, so wird ein neues Objektpaar bestimmt. Andernfalls wird diese Abbildung in das Matching aufgenommen. Für alle Links, die inzident zu n1 sind, wird geprüft, ob es einen Link in G2 gibt, auf den sie abgebildet werden können und der inzident zu n2 ist. Bei dieser Abbildung muss zum einen geprüft werden, ob die Links vom selben Typ sind, beide entweder positiv oder beide negativ sind und ob auf den Link aus G2 noch kein anderer Link aus G1 abgebildet wurde. Erfüllt eine Abbildung diese Eigenschaften, so wird sie ebenfalls in das Matching aufgenommen. Konnten alle zu n1 inzidenten Links abgebildet werden, wird das jeweils zweite inzidente Objekt des Links betrachtet und geprüft, ob dieses auf ein Objekt aus G2 abgebildet werden kann, auf das bisher noch kein anderes Objekt abgebildet wurde.

Können alle Objekte und Links aus G1 auf Objekte und Links aus G2 abgebildet werden, so wird das berechnete Matching zurückgegeben.

Die Matching-Funktion wird unter anderem zur Bildung der Ergebnisgraphmuster benötigt. Dazu muss die Funktion ein Matching eines Teil-Story Patterns[3] der Nachbedingung eines Story Patterns in einem verbotenen Story Pattern bestimmen.

Betrachtet man das Story Pattern aus Abbildung 5.4(a), so stellen die Objekte rt2 und rs1 sowie der dazwischen verlaufende locatedOn-Link ein Teil-Story Pattern der Nachbedingung dar. Wird für dieses Teil-Story Pattern und das verbotene Story Pattern aus Abbildung 5.3 die Matching-Funktion aufgerufen, so liefert sie zwei mögliche Matchings zurück. Der erste bildet rt2 auf vt ab und rs1 auf vs1 und den zwischen rs1 und rt2 verlaufenden locatedOn-Link auf den locatedOn-Link, der zwischen vs1 und vt verläuft. Der zweite Match bildet rs1 auf vs2, rt2 auf vt und den locatedOn-Link auf den Link zwischen vs2 und vt ab.

### RuleExtension

Die Funktion RuleExtension wird dazu verwendet, ein Story Pattern um zusätzliche negative Objekte zu erweitern, sodass die Anwendung des Story Patterns verhindert wird, wenn andernfalls lose Kanten resultieren sollten. Diese Erweiterung erfolgt nur intern und ist für den Benutzer des Plugins nicht sichtbar. Die Funktion erweitert sowohl die Vor- als auch die Nachbedingung des Story Patterns um zusätzliche negative Elemente, damit weder bei der Vorwärts- noch bei der Rückwärtsanwendung des Story Patterns lose Kanten entstehen können. Die Erweiterung entspricht der Regelerweiterung aus Definition 27 in Kapitel 3.

Bei der Erweiterung der Vorbedingung werden alle Objekte betrachtet, die durch die Anwendung des Story Patterns gelöscht werden. Dazu wird im Klassendiagramm nachgesehen, welche Typen adjazent zu dem Typ des zu löschenden Objektes sind. Für jeden dieser adjazenten Typen wird ein negatives Objekt in das Story Pattern eingefügt, falls dieses noch nicht existiert und durch einen Link mit dem zu löschenden Objekt verbunden ist. Bei der Erweiterung der Nachbedingung wird ebenso verfahren. Statt der zu löschenden Objekte werden jedoch die zu

[3]Ein Teil-Story Pattern besteht aus einer Teilmenge der Objekte und Links des ursprünglichen Story Patterns. Dabei muss jeder Link mit zwei Objekten verbunden sein.

erzeugenden betrachtet. Diese zweite Erweiterung ist notwendig, damit auch bei der Rückwärtsanwendung des Story Patterns die Bedingungen des DPO$^{iso}$ Ansatzes erfüllt sind. Die Funktion liefert das derart erweiterte Story Pattern zurück.

Da eine Erweiterung eines Story Patterns nur dann notwendig ist, wenn das Story Pattern ein Objekt erzeugt oder löscht, das moveSimple Story Pattern jedoch nur einen locatedOn-Link löscht und dann einen neuen erzeugt, ist eine Erweiterung dieses Story Patterns nicht notwendig.

### Merging

Die Merging-Funktion erzeugt aus zwei Story Patterns G1 und G2 ein neues Story Pattern. Dazu benötigt sie ein von der Matching-Funktion berechnetes Matching, das einen Teil des Story Patterns G1 auf einen Teil von G2 abbildet.

Sind die beiden Story Patterns und das Matching gegeben, so wird das neue Story Pattern gebildet, indem das Story Pattern G1 durch alle Elemente aus G2 erweitert wird, die nicht durch das Matching auf G1 abgebildet werden können. Das so entstandene Story Pattern wird von der Funktion zurück gegeben.

Wird der Funktion das Story Pattern moveSimple, das verbotene Story Pattern collision und der erste von der Matching-Funktion berechnete Match (rs1 wird auf vs1 abgebildet und rt2 auf vt) übergeben, so erzeugt die Funktion das in Abbildung 5.6 dargestellte Story Pattern. Da dieses Story Pattern aus der Nachbedingung des Story Patterns moveSimple und dem verbotenen Story Pattern collision besteht, stellt es ein Ergebnisgraphmuster dar.

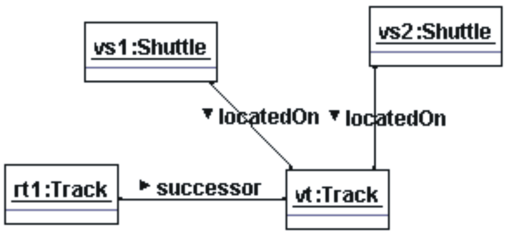



Abbildung 5.6: Durch die Merging-Funktion bestimmtes Story Pattern

### Der Verifikationsalgorithmus

Der Verifikationsalgorithmus erwartet als Eingabe eine Menge von Story Patterns sowie eine Menge von verbotenen Story Patterns. Für Story Patterns können Prioritäten angegeben werden. Das heißt, sind auf eine Instanzsituation zwei oder mehr Story Patterns anwendbar, wird das Story Pattern mit der höchsten Priorität angewendet. Haben alle Story Patterns die gleiche Priorität, so wird nichtdeterministisch eines der Story Patterns angewendet. Diese Prioritäten müssen bei der Verifikation berücksichtigt werden.

In einem ersten Schritt wird für jede Nachbedingung aller Story Patterns die Menge aller Teil-Story Patterns bestimmt. Für jedes der Teil-Story Patterns werden dann mittels der Matching-Funktion alle Matchings mit allen verbotenen Story Patterns gebildet. Jedes gefundenen

Matching wird anschließend zusammen mit dem Story Pattern und dem verbotenen Story Pattern an die Merge-Funktion übergeben, die daraus ein neues Story Pattern, das Ergebnisgraphmuster, bildet.

Das entsprechende Story Pattern wird durch die Funktion RuleExtension erweitert und rückwärts auf jedes der Ergebnisgraphmuster angewendet. Daraus resultieren die Startgraphmuster.

Für jedes dieser Startgraphmuster prüft die Matching-Funktion, ob es ein Matching für ein verbotenes Story Pattern in dem Startgraphmuster gibt. Kann kein solches Matching gefunden werden, so wird geprüft, ob es ein Story Pattern mit höherer Priorität gibt, das auf das Startgraphmuster angewendet werden kann, d.h. für dessen Anwendungsbedingung es ein Matching im Startgraphmuster gibt. Ist dies der Fall, so verhindert das höher priorisierte Story Pattern die Anwendung des zuvor betrachteten Story Patterns. Kann kein solches höher priorisiertes Story Pattern auf das Startgraphmuster angewendet werden, so wurde ein Gegenbeispiel gefunden, das zeigt, dass die Anwendung des betrachteten Story Patterns zu einer kritischen Situation oder einem Unfall führen kann.

Die Menge der zu betrachtenden Ergebnisgraphmuster kann im Vergleich zu der Menge aus Kapitel 3 reduziert werden. Diese Reduktion resultiert aus der geringeren Ausdrucksstärke der negativen Elemente im Vergleich zu den negativen Anwendungsbedingungen (siehe Abschnitt 4.2.2).

Wie in Graphtransformationssystemen gibt es auch bei Story Patterns zwei Möglichkeiten einen korrekten Graphen in einen inkorrekten zu überführen. Die erste besteht darin, dass ein Element erzeugt wird, das Teil der Anwendungsbedingung des verbotenen Story Patterns ist. Die zweite Möglichkeit ist, dass ein Element gelöscht wird und dieses Element zur negativen Anwendungsbedingung des verbotenen Graphmusters gehört, aber nicht zu dessen Anwendungsbedingung.

Im ersten Fall unterscheiden sich die in Kapitel 3 betrachteten Graphtransformationssysteme und Story Patterns nicht. Da in Story Patterns negative Kanten nur zwischen positiven Objekten verlaufen dürfen und negative Knoten nicht miteinander verbunden sein dürfen, kann der zweite Fall nur dann eintreten, wenn ein Element gelöscht wird, das inzident bzw. adjazent zu einem positiven Objekt ist. Deshalb brauchen im zweiten Fall bei den Story Patterns nur solche Ergebnisgraphmuster betrachtet werden, bei denen die Matching-Funktion mindestens ein Objekt der Nachbedingung des Story Patterns auf ein Objekt des verbotenen Story Patterns abbildet. Eine Abbildung der Elemente der Nachbedingung auf die negativen Elemente des verbotenen Story Patterns ist nicht notwendig. Für Graphtransformationsregeln müssen dagegen auch solche Fälle betrachtet werden, in denen Knoten der rechten Regelseite auf Knoten eines Graphen der negativen Anwendungsbedingung abgebildet werden können.

Hat der Algorithmus ein Gegenbeispiel gefunden, das zeigt, dass in einem System eine kritische Situation oder ein Unfall auftreten kann, so terminiert er. In diesem Fall wird das gefundene Gegenbeispiel zurückgegeben. Terminiert der Algorithmus nachdem für jedes Story Pattern und jedes verbotene Story Pattern alle möglichen Start- und Ergebnisgraphmuster betrachtet wurden, ohne das ein Gegenbeispiel gefunden wurde, so sind die Story Pattern korrekt im Bezug auf die betrachtete Menge von verbotenen Story Patterns.

In manchen Fällen kann es vorkommen, dass der Algorithmus „unknown“zurückliefert. Dies bedeutet, dass das Startgraphmuster die Anwendungsbedingung eines verbotenen Story Patterns oder eines höher priorisierten Story Patterns enthält. Allerdings kann keine der negativen Kanten auf das Startgraphmuster abgebildet werden, bzw. für keinen der negativen Knoten mit allen inzidenten Kanten existiert eine Abbildung.

### 5.3.2 Darstellung von Gegenbeispielen

Ein Gegenbeispiel besteht aus einem Start- und einem Ergebnisgraphmuster zusammen mit einem Story Pattern, dessen Anwendung auf das Startgraphmuster im Ergebnisgraphmuster resultiert. Terminiert der Algorithmus, nachdem er ein solches Gegenbeispiel gefunden hat, so werden die beiden Graphmuster, die Instanzen des internen Metamodells darstellen, in Story Patterns transformiert und können dann wieder in Fujaba dargestellt werden.

Für das Story Pattern moveSimple aus Abbildung 5.2 und das verbotene Story Pattern collision liefert der Algorithmus das in Abbildung 5.7 dargestellte Gegenbeispiel.

Das Gegenbeispiel besteht aus dem Startgraphmuster, dem Ergebnisgraphmuster sowie dem angewendeten Story Pattern. Das obere der beiden dargestellten Graphmuster ist das Startgraphmuster, das untere das Ergebnisgraphmuster. Das angewendete Story Pattern wird nicht abgebildet. Von ihm wird nur der Name in der linken oberen Ecke des Fensters angegeben.

Das Gegenbeispiel in Abbildung 5.7 zeigt, dass die Anwendung des Story Patterns moveSimple zu einem Objektdiagramm führen kann, das den Unfall collision enthält. Dies ist genau dann der Fall, wenn zwei Shuttles existieren, die auf benachbarten Tracks positioniert sind. Wird das Story Pattern moveSimple nun auf das hintere der beiden Shuttles angewendet, so wird dieses auf den nächsten Track gesetzt und die beiden Shuttles befinden sich auf demselben Track.

## 5.4 Evaluierung

Im voran gegangenen Abschnitt wurde die prototypische Umsetzung des Verifikationsansatzes aus Kapitel 3 für Story Patterns beschrieben. Die Aufwandsabschätzung am Ende von Kapitel 3 ergab, dass der Verifikationsansatz exponentiell von der Größe der Graphen, die die linke und rechte Regelseite, die Graphmuster und die negativen Anwendungsbedingungen repräsentieren, abhängt, aber nur linear von der Anzahl der Regeln und verbotenen Graphmuster. In diesem Abschnitt soll anhand eines kleinen Beispiels gezeigt werden, dass die Laufzeit des Algorithmus im Allgemeinen jedoch nicht so schlecht ist. Die Differenz zwischen dem berechneten Aufwand und den gemessenen Zeiten resultiert daraus, dass die Aufwandsabschätzung den schlechtesten Fall betrachtet. Dieser tritt ein, wenn weder die Elemente der Graphtransformationsregeln noch der verbotenen Graphmuster typisiert sind. Dass die Elemente einer Regel oder eines verbotenen Graphmusters nicht typisiert sind, trifft in der Realität jedoch selten ein.

Bevor in Abschnitt 5.4.2 die Evaluierung des Plugins zum automatischen Nachweis von induktiven Invarianten beschrieben wird, werden in Abschnitt 5.4.1 die wichtigsten Charakteristiken des evaluierten Modells genannt.

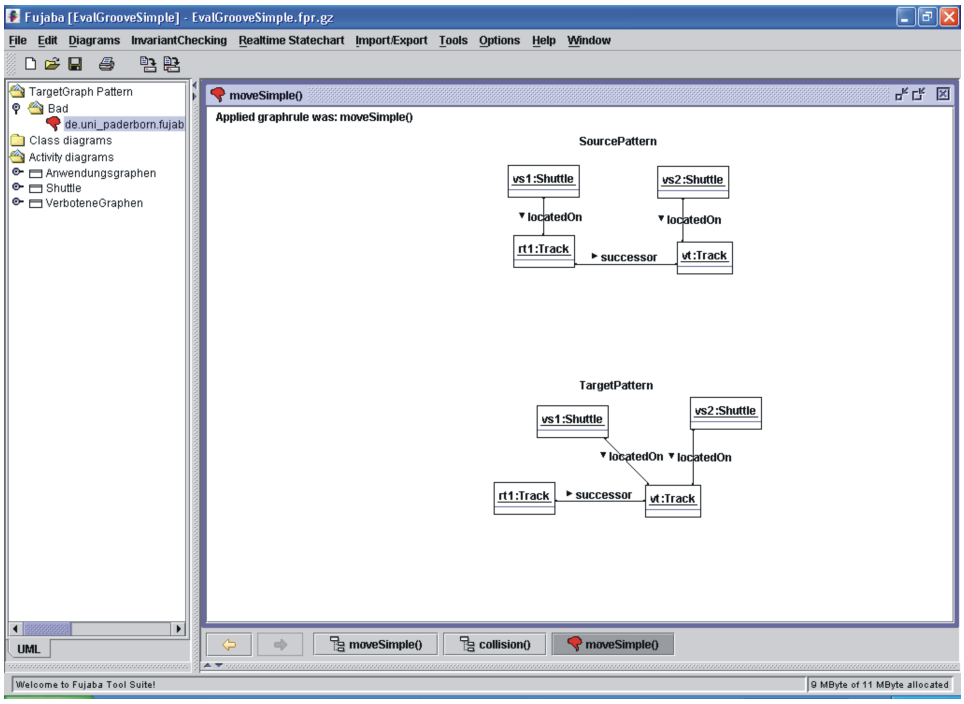



Abbildung 5.7: Von Fujaba generiertes Gegenbeispiel, das zeigt, dass das Story Pattern moveSimple das verbotene Graphmuster collision erzeugen kann

### 5.4.1 Charakteristiken des Evaluierungsbeispiels

Zur Evaluierung des Plugins zum automatischen Nachweis von induktiven Invarianten wurden die Graphtransformationsregeln und verbotenen Graphmuster aus dem Shuttle-Beispiel als Story Patterns bzw. verbotene Story Patterns dargestellt. Wie in Abschnitt 2.3.2 beschrieben, werden die Story Patterns und verbotenen Story Patterns hierarchisch angeordneten Kulturen zugewiesen. Zur Evaluierung wurden alle Story Patterns und verbotenen Graphmuster der ControlledMovement-Kultur sowie der übergeordneten Movement-Kultur verwendet.

Insgesamt wurden 7 Story Patterns verifiziert. Davon haben 4 Story Patterns Instanzierungsregeln dargestellt und 3 Verhaltensregeln. Bei der Verifikation wurde nachgewiesen, dass die Story Patterns 12 verbotene Story Patterns erfüllen. Von diesen 12 verbotenen Story Patterns hat eins einen Unfall modelliert, 2 kritische Situationen und weitere 9 verbotene Story Patterns wurden dazu verwendet, um die Kardinalitäten des zugehörigen Klassendiagramms nachzubilden. Die Nachbildung der Kardinalitäten ist notwendig, da weder der Ansatz aus Kapitel 3 noch dessen im voran gegangenen Abschnitt vorgestellte prototypische Umsetzung die im Klassendiagramm vorgegebenen Kardinalitäten noch nicht berücksichtigen (siehe dazu auch Abschnitt 7.2). Lässt man die vorgegebenen Kardinalitäten bei der Verifikation jedoch außer Acht, so liefert das Plugin unter Umständen ungültige Gegenbeispiele. Alle verifizierten (verbotenen) Story Patterns werden in Anhang A dargestellt und erläutert.

Die Aufwandsabschätzung in Kapitel 3 hat ergeben, dass der Verifikationsaufwand von der Größe der Graphen abhängig ist. Für Story Patterns und verbotene Story Patterns wird bei der

Evaluierung das folgende Größenmaß verwendet: Die Größe eines (verbotenen) Story Patterns entspricht der Anzahl der Objekte und Links, die bei seiner Anwendung erhalten bleiben oder neu erzeugt werden sowie der Anzahl der negativen Elemente. Die Wahl dieses Maßes erfolgte auf Grund der Überlegung, dass der aufwändigste Schritt bei der Verifikation die Bildung aller möglichen Ergebnisgraphmuster ist. Ein solches Ergebnisgraphmuster setzt sich genau aus den Elementen zusammen, die zum verbotenen Story Pattern gehören oder durch das zu verifizierende Story Patterns erhalten bleiben bzw. erzeugt werden. Wurden die positiven Elemente zu einem Ergebnisgraphmuster zusammengefügt, so wird die negative Anwendungsbedingung durch Hinzufügen der negativen Elemente gebildet.

Eine Übersicht über die Größe der verifizierten Story Patterns und verbotenen Story Patterns ist in Tabelle 5.1 gegeben. Dabei gibt #O jeweils die Anzahl der Objekte an und #L die Anzahl der Links.

| Größe der Story Patterns | | | | | | Größe der verbotenen Story Patterns | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Minimum | | Maximum | | Durchschnitt | | Minimum | | Maximum | | Durchschnitt | |
| #O | #L | #O | #L | #O | #L | #O | #L | #O | #L | #O | #L |
| 5 | 8 | 10 | 11 | 7 | 9 | 2 | 2 | 4 | 4 | 3 | 3 |

Tabelle 5.1: Größe der verifizierten (verbotenen) Story Patterns

### 5.4.2 Evaluierung des Plugins zum automatischen Nachweises von induktiven Invarianten

Die Aufwandsabschätzung für den Verifikationsansatz in Abschnitt 3.7.3 hat ergeben, dass der Aufwand exponentiell in der Größe der (verbotenen) Story Patterns, aber nur linear in deren Anzahl ist. Die Abschätzung erfolgte für den schlechtesten Fall. Dieser tritt ein, wenn alle Objekte und Links der (verbotenen) Story Patterns denselben Typ haben. In diesem Abschnitt soll zunächst anhand eines kleinen Beispiels gezeigt werden, welche Auswirkungen die Größe der (verbotenen) Story Patterns auf den Aufwand hat. Im Anschluss daran wird dann gezeigt, welche Auswirkungen die Verwendung von Typen auf den Verifikationsaufwand hat. In beiden Fällen wird der Aufwand in Zeit gemessen.

Zur Evaluierung wurde ein Intel Pentium 4 mit 2.40GHz CPU und 512 KB Cache sowie 1GB Hauptspeicher eingesetzt. Die Evaluierung wurde unter Linux durchgeführt.

Um die Auswirkung der Größe der (verbotenen) Story Patterns zu veranschaulichen, wird das im vorgegangenen Abschnitt charakterisierte Beispiel verifiziert. Die Verifikation des Beispiels benötigte insgesamt 84 Sekunden.

Bei der Verifikation wird für jedes Paar, bestehend aus einem Story Pattern und einem verbotenen Story Pattern, überprüft, ob die Anwendung des Story Patterns das verbotene Story Pattern erzeugen kann. Das Diagramm in Abbildung 5.8 zeigt die Verifikationszeiten für jedes dieser Paare. Die Zeiten sind in Sekunden angegeben. Statt der Namen der (verbotenen) Story Patterns sind die jeweiligen Größen an den Achsen notiert. An der x-Achse sind die Größen der verifizierten Story Patterns in aufsteigender Folge notiert. Die y-Achse stellt die Größen der verwendeten

verbotenen Story Patterns in aufsteigender Folge dar. Wie an der Graphik zu sehen ist, steigt der Aufwand für die Verifikation mit der Größe der (verbotenen) Story Patterns an. Der Grund dafür, dass die Verifikation der beiden größten verbotenen Story Patterns deutlich schneller war als die Vorangegangenen liegt darin, dass die Objekte dieser beiden verbotenen Story Patterns mehr unterschiedliche Typen haben als die der zuvor verifizierten.

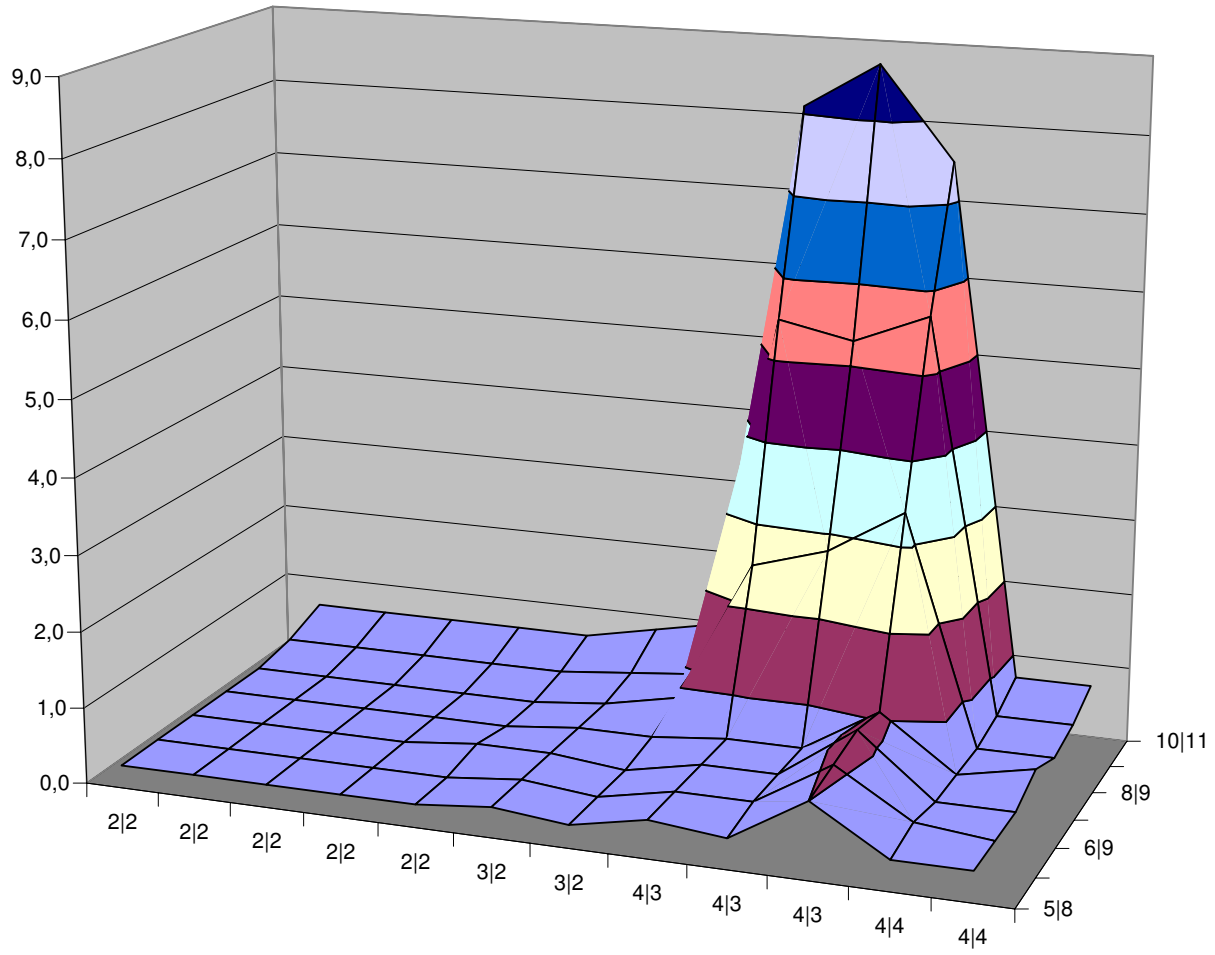



Abbildung 5.8: Benötigte Verifikationszeiten

In Story Patterns haben alle Objekte einen Typ. Um den Einfluss der Verwendung von Typen auf den Verifikationsaufwand zeigen zu können, wurden allen Objekten die gleichen Typen zugewiesen.

Verifiziert man die so modifizierten Story Patterns, so ergibt die Analyse, dass die Story Patterns einen inkorrekten Zustand erzeugen können. In diesem Fall bricht die Analyse ab, sobald ein Gegenbeispiel gefunden wurde. Um die Verifikationszeiten dennoch mit denen des korrekten, typisierten Beispiels vergleichen zu können, wurde der Evaluierungsmodus des Plugins verwendet. Dieser Modus führt die Analyse so durch, als wären die zu verifizierenden Story Patterns korrekt, d.h. in diesem Modus wird die Analyse nicht abgebrochen, wenn ein Gegenbeispiel ge-

funden wurde. Zudem stellt das Plugin in diesem Modus auch die gefundenen Gegenbeispiele nicht dar, sodass das Plugin keine Zeit für die graphische Darstellung benötigt.

Die Verifikation des gesamten nicht typisierten Beispiels musste nach etwa 12 Stunden erfolglos abgebrochen werden. Eine weitere Analyse der 7 Story Patterns sowie der 3 verbotenen Story Patterns, die im ursprünglichen Beispiel den Unfall und die beiden kritischen Situationen modelliert haben, konnte erfolgreich in 1755,8 Sekunden durchgeführt werden. Tabelle 5.2 stellt die benötigten Verifikationszeiten für diese (verbotenen) Story Patterns aus dem typisierten und dem nicht typisierten Beispiel gegenüber. Wie schon in Abbildung 5.8, werden auch hier statt der Namen der (verbotenen) Story Patterns deren Größen angegeben. Die Zeiten sind jeweils in Sekunden angegeben.

| Verbotene Story Patterns/ Story Patterns | 3 \| 2 | | 4 \| 4 | | 4 \| 4 | |
|---|---|---|---|---|---|---|
| | Typen | keine Typen | Typen | keine Typen | Typen | keine Typen |
| 5 \| 5 | 0,4 | 1,1 | 0,4 | 7,8 | 0,4 | 6,4 |
| 6 \| 6 | 0,4 | 1,4 | 0,4 | 12,3 | 0,5 | 12,4 |
| 6 \| 4 | 0,4 | 2,0 | 0,4 | 12,9 | 0,4 | 15,7 |
| 6 \| 10 | 0,4 | 1,4 | 0,4 | 26,5 | 0,4 | 10,6 |
| 5 \| 9 | 0,4 | 2,1 | 0,4 | 1207,4 | 0,4 | 22,4 |
| 6 \| 10 | 0,5 | 4,0 | 0,5 | 131,7 | 0,5 | 61,5 |
| 6 \| 11 | 0,8 | 2,8 | 0,7 | 187,8 | 0,7 | 25,6 |
| gesamt | 3,3 | 14,8 | 3,4 | 1586,4 | 3,3 | 154,6 |

Tabelle 5.2: Vergleich der Verifikationszeiten mit und ohne Typen

Wie der Tabelle 5.2 zu entnehmen ist, benötigt die Verifikation ohne Typen deutlich länger als die Verifikation mit Typen.

Die verifizierten (verbotenen) Story Patterns stammen aus der in Abschnitt 2.3.2 vorgestellten ControlledMovement-Kultur. Die Verifikation der CoordinatedMovement-Kultur musste jedoch ergebnislos nach mehr als 12 Stunden abgebrochen werden. Die CoorinatedMovement-Kultur erweitert die ControlledMovement-Kultur um weitere 4 Story Patterns und 2 verbotene Story Patterns. Die 6 zusätzlichen (verbotenen) Story Patterns sind größer als die (verbotenen) Story Patterns der ControlledMovement-Kultur und auch die Anzahl von Objekten mit demselben Typ ist größer.

Ein Grund für die beiden fehlgeschlagenen Verifikationsversuche besteht darin, dass es sich bei dem Plugin um einen Prototypen handelt. Zurzeit berechnet das Plugin beispielsweise noch zu viele Ergebnisgraphmuster. In Abschnitt 3.6 wurde gezeigt, dass nur solche Ergebnisgraphmuster relevant sind, bei denen mindestens ein Element, das durch das Story Pattern neu erzeugt wird, auf ein Element des verbotenen Story Patterns abgebildet wird. Das Plugin berechnet jedoch jedes mögliche Ergebnisgraphmuster. Das hat besonders dann Auswirkungen, wenn (verbotene) Story Patterns mit vielen Objekten vom selben Typ verifiziert werden sollen.

Das Plugin wird zurzeit weiterentwickelt, um dieses Problem zu beheben. Darüber hinaus wurde in [GSK+06] eine erste Idee präsentiert, die es ermöglicht die (verbotenen) Story Patterns symbolisch zu codieren. Auf diese Weise brauchen die Ergebnisgraphmuster nicht mehr

einzeln aufgezählt werden, sondern können zu Mengen zusammengefasst werden. Um die Ergebnisgraphmuster zu bestimmen, die Rückwärtsanwendung der Regeln durchzuführen und für die resultierenden Startgraphmuster zu prüfen, ob diese eines der verbotenen Story Patterns oder die Anwendungsbedingung eines höher priorisierten Story Patterns enthalten, wird der BDD-basierte Interpreter CroCoPat [BNL05] verwendet.

## 5.5 Zusammenfassung

In diesem Kapitel wurde die Werkzeugunterstützung für die Modellierung und Verifikation der Software mechatronischer Systeme vorgestellt.

Die Fujaba Real-Time Tool Suite unterstützt die Modellierung der Softwarearchitektur sowohl durch Klassendiagramme als auch durch Komponentendigramme. Für die Modellierung des Koordinationsverhaltens bietet Fujaba Real-Time Statecharts. Die in den Statecharts aufgerufenen Story Patterns sind ebenfalls Teil von Fujaba.

Zur Verifikation der Real-Time Statecharts existieren zwei Plugins, die die Model Checker UPPAAL und RAVEN an Fujaba anbinden. Ein Plugin zur Verifikation von Story Patterns wurde im Rahmen dieser Arbeit entwickelt. Dieses Plugin wurde in Abschnitt 5.3 vorgestellt und in Abschnitt 5.4 evaluiert. Die Evaluierung hat die Aufwandsabschätzung aus Kapitel 3 bestätigt. Sie hat aber auch gezeigt, dass die Verwendung von Typen die Verifikation deutlich beschleunigt.

Ist die Verifikation abgeschlossen, so bietet die Fujaba Real-Time Tool Suite die Möglichkeit, automatisch Code zu generieren. Dies bietet den Vorteil, dass Eigenschaften, die durch die formale Verifikation nachgewiesen wurden, auch durch die Implementierung erfüllt werden.

# Kapitel 6

# Verwandte Arbeiten

Der in den Kapiteln 2 und 3 vorgestellte Ansatz ermöglicht eine kompositionale Modellierung und Verifikation der Software mechatronischer Systeme. Außerdem wurde in Kapitel 5 gezeigt, wie der Modellierungs- und Verifikationsprozess durch die Fujaba Real-Time Tool Suite unterstützt und automatisiert wird.

In diesem Kapitel sollen nun verwandte Arbeiten betrachtet werden. Ein ausführlicher Vergleich des gesamten Modellierungs- und Verifikationsprozesses und seiner Umsetzung in der Fujaba Real-Time Tool Suite mit verwandten Ansätzen wurde von Burmester in [Bur05] vorgenommen. An dieser Stelle soll ein Vergleich des in Kapitel 3 vorgestellten Ansatzes zur Verifikation von Graphtransformationssystemen mit anderen existierenden Ansätzen erfolgen.

Es werden drei verschiedene Vorgehensweisen vorgestellt. Als erstes wird in Abschnitt 6.1 ein Ansatz beschrieben, der die Regeln eines Graphtransformationssystems so modifiziert, dass ihre Anwendungen nicht zu einem inkorrekten Zustand führen können. Die zweite betrachtete Kategorie stellen Ansätze dar, bei denen Model Checking Techniken dazu eingesetzt werden, um bestimmte Eigenschaften für ein Graphtransformationssystem nachzuweisen. Diese Ansätze werden in Abschnitt 6.2 beschrieben. In Abschnitt 6.3 werden Verifikationstechniken für Graphtransformationssysteme erläutert, die auf Petrinetzanalysetechniken basieren.

## 6.1 Korrektheit per Konstruktion

Der von Heckel und Wagner in [HW95] vorgestellte Ansatz diente als Basis für den in Kapitel 3 vorgestellten Ansatz zur Verifikation von Graphtransformationssystemen. Der Ansatz von Heckel und Wagner ermöglicht es, Graphtransformationsregeln so zu modifizieren, dass bestimmte strukturelle Eigenschaften eines beliebigen Graphen bei der Regelanwendung erhalten bleiben. In der AGG Tool Suite[1] wurde dieser Ansatz prototypisch umgesetzt [Mat01].

Die Regeln werden im Single Pushout Ansatz angewendet. Ihre Anwendung kann durch Conditional Constraints, die zur linken Regelseite hinzugefügt werden, eingeschränkt werden.

Die Eigenschaften, die durch die Anwendung einer Regel erhalten bleiben sollen, werden als Konsistenzbedingung bezeichnet. Eine Konsistenzbedingung besteht dabei aus einer Menge von graphischen Konsistenzconstraints.

[1] http://tfs.cs.tu-berlin.de/agg

Ein graphisches Konsistenzconstraint beschreibt eine strukturelle Eigenschaft eines Graphen, wie beispielsweise das Vorhandensein oder die Abwesenheit von bestimmten Elementen oder die Gleichheit zweier Elemente. Eine graphisches Konsistenzconstraint besteht aus zwei Graphen, der Prämisse P und der Konklusion Q; Wobei es einen totalen Homomorphismus gibt, der die Prämisse auf die Konklusion abbildet. Ein Graph erfüllt ein Konsistenzconstraint, wenn es einen Homomorphismus gibt, der die Prämisse P auf den Graphen abbildet und der so erweitert werden kann, dass er neben P auch die Konklusion Q auf den Graphen abbildet.

Ein Graph wird als konsistent bezeichnet, wenn er die Konsistenzbedingung erfüllt. Ein Graphtransformationssystem ist konsistent, wenn jede Anwendung einer Regel auf einen konsistenten Graphen wieder einen konsistenten Graphen erzeugt.

Das Ziel des Ansatzes aus [HW95] besteht darin, die Regeln eines Graphtransformationssystems so zu modifiziert, dass das System konsistent ist. Die Modifikation der Regeln erfolgt, indem die rechte Regelseite durch eine so genannte Verklebung (engl. gluing) mit jeder der Konsistenzbedingungen verbunden wird. Dabei werden jedoch nur die Fälle berücksichtigt, bei denen mindestens ein durch die Regel erzeugtes Element mit einem Element der Konsistenzbedingung verklebt wird. Auf den bei der Verklebung entstehenden Graphen wird die inverse Regel angewendet. Daraus resultiert wieder ein Graph. Dieser Graph stellt dann ein zusätzliches Conditional Constraint der linken Regelseite dar. Auf diese Weise wird verhindert, dass die Regel anwendbar ist, wenn andernfalls die Konsistenzbedingung verletzt wird.

Die Definition der von Heckel und Wagner eingeführten Konsistenzbedingungen entspricht der Definition der induktiven Invarianten. Beide Begriffe bezeichnen Eigenschaften, die, wenn sie von einem Graphen erfüllt sind, durch die Anwendung einer beliebigen Regel erhalten bleiben.

Ein Unterschied zwischen den beiden Ansätzen besteht in der Verwendung von negativen Anwendungsbedingungen in Regeln und verbotenen Graphmustern bzw. graphischen Konsistenzconstraints. Die negative Anwendungsbedingung einer Regel oder eines verbotenen Graphmusters aus Kapitel 3 kann aus einer Menge von Graphen bestehen, die Alternativen beschreiben. Dagegen kann die Prämisse einer Regel oder eines Konsistenzconstraints in [HW95] nur aus einem Graphen bestehen. Der Ansatz von Heckel und Wagner bietet zwar die Möglichkeit zusätzlich zur Prämisse noch eine Konklusion anzugeben, diese kann bei Regeln aus einer Menge von Graphen bestehen, bei Konsistenzbedingungen nur aus genau einem Graphen, allerdings haben die Prämissen eine andere Semantik als die negativen Anwendungsbedingungen aus Kapitel 3. Eine Regel ist nach [HW95] anwendbar bzw. ein Konsistenzconstraint erfüllt, wenn die Prämisse mittels eines Homomorphismus auf den Anwendungsgraphen abbildbar ist und der verwendete Homomorphismus so erweitert werden kann, dass er auch mindestens einen Graphen der Konklusion auf den Anwendungsgraphen abbildet.

In Kapitel 3 wurde gezeigt, dass ein korrekter Graph auf zwei Arten in einen inkorrekten Graphen überführt werden kann. Zum einen indem bei einer Regelanwendung ein Element erzeugt wird, das Teil der Anwendungsbedingung des verbotenen Graphmusters ist. Zum andere kann eine Regel ein Element löschen, das zur negativen Anwendungsbedingung, aber nicht zur Anwendungsbedingung des verbotenen Graphmusters gehört. Beim Nachweis induktiver Invarianten, wie er in dieser Arbeit vorgestellt wurde, werden beide Fälle berücksichtigt. Dagegen wird in [HW95] nur der erste Fall betrachtet.

Bei der Modifikation der Graphtransformationsregeln nach [HW95] wird jedes Konsistenzconstraint unabhängig von den anderen Constraints betrachtet. Das kann dazu führen, dass eine Regel unnötig um zusätzliche Conditional Constraints erweitert wird. Das ist genau dann der Fall, wenn ein Konsistenzconstraint $c_1$ nur verletzt werden kann, wenn die betrachtete Regel auf einen Graphen angewendet wird, der zwar das Constraint $c_1$ erfüllt, aber inkonsistent in Bezug auf ein zweites Konsistenzconstraint $c_2$ ist. Im schlimmsten Fall kann die Anwendung einer Regel auch ohne Modifikation keine inkonsistenten Graphen erzeugen, der Ansatz fügt aber für jedes Konsistenzconstraint und jede seiner möglichen Verklebungen mit der rechten Regelseite ein neues Conditional Constraint zur linken Regelseite hinzu. Um eine so erweiterte Regel anwenden zu können, muss für jedes Conditional Constraint überprüft werden, ob es erfüllt ist. Dies kann einen erheblichen Mehraufwand für die spätere Regelanwendung bedeuten.

Im Gegensatz dazu werden beim Nachweis von induktiven Invarianten die verbotenen Graphmuster nicht unabhängig voneinander betrachtet. Nachdem das Ergebnisgraphmuster gebildet wurde, wird die entsprechende Regel rückwärts auf das Muster angewendet. Für das resultierende Startgraphmuster wird dann überprüft, ob es ein verbotenes Muster enthält. Kann ein beliebiges verbotenes Graphmuster im Startgraphmuster gefunden werden, so hat die Regelanwendung einen inkorrekten Graphen in einen anderen inkorrekten Graphen überführt. Die Zahl der gebildeten Ergebnisgraphmuster entspricht der Zahl der Verklebungen (zumindest für den Fall, dass die Regel ein Element erzeugt, das Teil der Anwendungsbedingung des verbotenen Graphmusters ist). Im Gegensatz zum Ansatz aus [HW95] müssen diese Ergebnisgraphmuster jedoch nur einmal zur Verifikation betrachtet werden, während die Konsistenzconstraints bei jeder Regelanwendung wieder überprüft werden müssen.

Die Verwendung des Single Pushout Ansatzes kann sogar dazu führen, dass Conditional Constraints zur linken Regelseite hinzugefügt werden, die eine Regelanwendung verhindern, obwohl diese nicht in einem inkonsistenten Graphen enden. Dies ist genau dann der Fall, wenn bei der Rückwärtsanwendung der Regel Kanten implizit gelöscht werden, d.h. es werden lose Kanten entfernt. Mindestens ein inzidenter Knoten dieser Kante ist Teil der Regel. Die Kante selber ist aber weder Teil der linken noch der rechten Regelseite. Wendet man eine solche Regel zunächst rückwärts auf einen Graphen $G_1$ an und dieselbe Regel dann wieder in Vorwärtsrichtung auf den resultierenden Graphen, so erhält man nicht den Graphen $G_1$, da die implizit gelöschte Kante nicht wieder erzeugt wird. Der Ansatz hat also zuviele Conditional Constraints erzeugt. Um solche Probleme zu verhindern, wurde in Kapitel 3 der $\mathrm{DPO}^{\mathrm{iso}}$ eingeführt.

Insgesamt gibt es einige Parallelen zwischen dem Ansatz von Heckel und Wagner und dem in Kapitel 3 vorgestellten Ansatz. Allerdings führt das Hinzufügen von zusätzlichen Graphen zur negativen Anwendungsbedingung dazu, dass die Überprüfung zur Laufzeit, ob die entsprechende Regel anwendbar ist, länger dauert als zuvor. Dies ist dann kritisch, wenn eine Regelanwendung auch ohne die zusätzlichen Graphen nicht anwendbar wäre, bzw. nicht zu einer Inkonsistenz führen kann. Es ist jedoch vorstellbar, die beiden Ansätze zu kombinieren, sodass zunächst eine Überprüfung der Regeln durchgeführt wird und nur im Fehlerfall eine Modifikation der Regeln erfolgt. Dazu müsste jedoch der Ansatz aus [HW95] zunächst an die Formalisierung aus Kapitel 3 angepasst werden.

Eine solche Erweiterung und Kombination des Ansatzes von Heckel und Wagner wird auch von Koch, Mancini und Parisi-Presicce in [KPP03, KMPP02] vorgeschlagen.

Die Autoren erweitern den Ansatz von Heckel und Wagner um den Fall, dass die Anwendung einer Regel ein Element löscht und dadurch eine Sicherheitseigenschaft verletzt. Zudem erfolgt bei Koch et al. eine Modifikation einer Regel nur dann, wenn ihre Anwendung zu einem inkorrekten Graphen führen kann.

Mit dem Verfahren, das Koch et al. in [KMPP02] vorstellen, ist es möglich eine obere Schranke zu bestimmen, die angibt, wann die Anwendungen der Regeln ausgehend von einem gegebenen Startgraphen spätestens zu einem inkorrekten Graphen führen müssen. Die maximale Pfadlänge wird in Abhänigkeit vom gegebenen Startgraphen sowie der Anzahl der Regeln bestimmt. Mittels dieser maximalen Pfadlänge kann dann ein beschränktes Model Checking (engl. bounded Model Checking) durchgeführt werden, d.h. ausgehend vom Startgraphen wird der Zustandsraum des Systems durch Anwendung der Regeln aufgebaut (siehe dazu auch Abschnitt 6.2). Für jeden erzeugten Graphen wird überprüft, ob er korrekt ist. Kann ein Graph erzeugt werden, der inkorrekt ist, bricht die Analyse ab. Sonst wird die Analyse für jeden der Pfade solange durchgeführt bis er zu Ende ist, d.h. keine Regelanwendung mehr möglich ist, oder bis der betrachtete Pfad die maximale Pfadlänge erreicht hat. Kann im letzteren Fall kein inkorrekter Graph erreicht werden, so ist das betrachtete System korrekt.

Wird ein Graph erreicht, der inkorrekt ist, so wird die zuletzt angewendete Regel durch die Erweiterung des Ansatzes von Heckel und Wagner modifiziert.

Der Ansatz von Koch et al. erweitert den von Heckel und Wagner zwar für den Fall, dass die Anwendung einer Regel ein Element löscht und dadurch eine Eigenschaft verletzt und modifiziert eine Regel nur dann, wenn sie einen inkorrekten Graphen erzeugt, unterliegt aber ansonsten den gleichen Einschränkungen. Um die maximal zu betrachtende Pfadlänge bestimmen zu können, müssen die Regeln im Vergleich zu denen von Heckel und Wagner deutlich eingeschränkt werden. So darf eine Regel entweder Elemente erzeugen oder löschen, aber nicht beides. Zudem dürfen nur Regeln, die Elemente löschen, eine negative Anwendungsbedingung besitzen. Diese beiden Einschränkungen der Regeln und vor allem die letztere sind jedoch für die hier betrachteten mechatronischen Systeme zu stark. Darüber hinaus ist der Ansatz nur dann anwendbar, wenn der Startgraph gegeben ist. Auch diese Annahme trifft in mechatronischen Systemen im Allgemeinen nicht zu.

## 6.2 Model Checking von Graphtransformationssystemen

Eine Möglichkeit, die Korrektheit eines Graphtransformationssystems nachzuweisen, stellt das Model Checking dar. Die Grundlage für das Model Checking von Graphtransformationssystemen wurde von Heckel et al. in [HEWC97] eingeführt. Aufbauend auf dieser Idee können Graphtransformationssysteme auf zwei Arten mittels Model Checking verifiziert werden. Zum einen können die Systeme auf die Eingabesprache eines existierenden Model Checkers abgebildet werden, so wie dies in CheckVML und OBGG erfolgt. Zum anderen ist es auch möglich, Model Checking Techniken direkt auf Graphtransformationssysteme anzuwenden. Ein solcher Model Checker für Graphtransformationssysteme ist Groove (siehe auch Abschnitt 5.2).

In [HEWC97] haben Heckel, Ehrig, Wolter und Corradini die Grundlage für das Model Checking von Graphtransformationssystemen eingeführt. Die von den Autoren betrachteten Re-

geln werden im Double Pushout Ansatz angewendet und dürfen keine negative Anwendungsbedingung besitzen. Die Graphen der betrachteten Systeme sind typisiert.

Ansätze, die auf dem Ansatz von Heckel et al. aufbauen und in Werkzeugen realisiert wurden, sind der CheckVML-Ansatz von Varrò et al. (Check Visual Modeling Languages, [SV03, Var04], der Groove-Ansatz von Rensink (GRaph-based Oject-Oriented VErification[2], siehe Abschnitt 5.2 und [Ren03, Ren04]) sowie der OBGG-Ansatz von Dotti et al. (Object-Based Graph Grammars, [DFRS03]). Wobei sowohl bei Varrò als auch bei Rensink der Single Pushout Ansatz zur Regelanwendung verwendet wird. Der OBGG-Ansatz verwendet dagegen den klassischen Double Pushout Ansatz ohne negative Anwendungsbedingungen. Zusätzlich werden die Regeln in diesem Ansatz so eingeschränkt, dass Objekte nicht gelöscht werden dürfen.

Die Idee der Ansätze besteht darin, ein gegebenes Graphtransformationssystem auf ein Transitionssystem (z.B. Kripkestruktur oder Graphtransitionssystem, siehe Abschnitt 5.2) abzubilden. Jeder Zustand eines solchen Transitionssystems entspricht dann einem Graphen. Der Startzustand entspricht dem Initialgraphen des Graphtransformationssystems. Kann eine Regel auf einen bestimmten Graphen angewendet werden, so existiert im Transitionssystem für diese Regelanwendung eine Transition. Der Ausgangszustand der Transition repräsentiert den Anwendungsgraphen und der Zielzustand den aus der Regelanwendung resultierenden Graphen.

Heckel et al. verwenden zur Beschreibung der Eigenschaften, die durch die Verifikation nachgewiesen werden sollen, graphische Konsistenzbedingungen (siehe Abschnitt 6.1) bzw. temporallogische Formeln. Eine temporallogische Formel für Graphtransformationssysteme setzt sich aus den üblichen All- und Existenzquantoren sowie den Pfadquantoren temporallogischer Formeln zusammen (siehe [CGP02] Kapitel 3). Die atomaren Eigenschaften solcher Formeln werden durch graphische Konsistenzbedingungen beschrieben. In CheckVML und OBGG werden die Eigenschaften mittels temporallogischer Formeln beschrieben. Groove verwendet zur Spezifikation der Eigenschaften geforderte und verbotene Regeln (siehe Abschnitt 5.2).

Wurde ein Graphtransformationssystem in ein Transitionssystem überführt, so kann Model Checking [CGP02] dazu verwendet werden, um nachzuweisen, dass das System korrekt ist im Bezug auf eine Menge von Eigenschaften. Die Realisierung des Model Checkings kann auf verschiedene Arten erfolgen. In CheckVML wird ein Graphtransformationssystem zunächst auf einen abstrakten Zustandsautomaten (engl. abstract state machine) abgebildet. Ein solcher abstrakter Zustandsautomat kann dann auf die Eingabesprache eines existierenden Model Checkers abgebildet werden, wie zum Beispiel auf PROMELA die Eingabesprache von SPIN [Hol03]. Auch Dotti et al. verwenden zur Verifikation SPIN. Rensink hat dagegen mit Groove einen speziellen Model Checker für Graphtransformationssysteme entwickelt.

Die Abbildung von Graphtransformationssystemen auf Transitionssysteme ermöglicht die Verwendung von Model Checkern zur Verifikation, d.h. sie kann dazu verwendet werden, einen Nachweis operationaler Invarianten zu führen. Durch diese Technik können jedoch nur Systeme mit endlichem Zustandsraum betrachtet werden. Um zu erreichen, dass das zu verifizierende System einen endlichen Zustandsraum hat, verwendet CheckVML ein beschränktes Model Checking. Dazu muss a priori festgelegt werden wie viele Objekte von einem Typ zur Laufzeit erzeugt werden dürfen. Nach [RSV04] ist CheckVML deshalb für Systeme mit vielen dyna-

[2] http://www.cs.utwente.nl/∼groove

misch erzeugten Objekten nicht so gut geeignet wie Groove. Diese Tatsache war ein Grund für die Anbindung von Groove an Fujaba.

Aber auch wenn ein System einen endlichen Zustandsraum besitzt, kann das so genannte Zustandsraumexplosionsproblem (engl. state space explosion problem) die Verifikation verhindern. Das bedeutet, wenn das System aus zu vielen Zuständen besteht, ist das Model Checking nicht mehr möglich. Zwar unterstützen CheckVML und Groove verschiedene Optimierungstechniken, es ist jedoch trotzdem nicht möglich, Systeme mit mehr als 700.000 Zuständen mittels CheckVML und 500.000 Zuständen mittels Groove zu verifizieren [RSV04].

Um überhaupt ein Model Checking durchführen zu können, wird von allen vorgestellten Ansätzen/Werkzeugen ein Initialgraph benötigt.

Da für mechatronische Systeme nicht garantiert werden kann, dass diese einen endlichen Zustandsraum besitzen und der Initialgraph zum Verifikationszeitpunkt noch nicht bekannt sein muss, sind Ansätze zum Model Checking von Graphtransformationssystemen in dieser Domäne nur begrenzt einsetzbar. Gegen die Verwendung des OBGG-Ansatzes spricht zudem die starke Einschränkung der Regeln.

#### Kompositionale Verifikation

Um Graphtransformationssysteme mit großem, aber endlichem Zustandsraum verifizieren zu können, schlägt Heckel in [Hec98] und Heckel, Lajios und Menge in [HLM05] eine kompositionale Verifikation vor. Dazu wird das Gesamtsystem in kleinere Teilsysteme, die so genannten Sichten, unterteilt. Eine Sicht besteht aus einem Teil des Typgraphen, der angibt, welche Typen in der Sicht betrachtet werden. Jeder Sicht wird ein Anfangszustand zugewiesen, der dem ursprünglichen Anfangszustand entspricht, aber auf die in der Sicht zulässigen Typen eingeschränkt ist. Ebenso entsprechen die Regeln der Sicht den Regeln des Gesamtsystems, werden aber auf die Typen der Sicht eingeschränkt. Wird eine der Regeln angewendet, so erfolgt die Anwendung in allen Sichten synchron unter dem gleichen Auftreten der linken Regelseite. Auf diese Weise wird garantiert, dass Eigenschaften, die für eine Sicht nachgewiesen wurden, auch im Gesamtsystem gelten.

Damit die Dekomposition Auswirkungen auf die Verifikation hat, muss sie so gewählt werden, dass die Anwendung einer Regel nur in möglichst wenigen Sichten Auswirkungen hat. Die Dekomposition muss manuell vom Entwickler durchgeführt werden. Ist die Dekomposition erfolgt, so kann die Verifikation erfolgen. Dabei wird die in [HEWC97] beschriebene Idee verwendet, um jede der Sichten unabhängig von allen anderen Sichten zu verifizieren.

Mittels eines solchen kompositionalen Vorgehens lassen sich auch Systeme mit größerem Zustandsraum verifizieren. Allerdings muss die Dekomposition in Sichten manuell durch den Entwickler erfolgen, was fehleranfällig ist. Bei der Verifikation der Story Patterns ist eine solche manuelle Dekomposition in Teilsysteme, die voneinander unabhängig verifiziert werden können, nicht notwendig. Sie ergibt sich aus der Verwendung der Kulturhierarchien. Wurden die Sichten gut gewählt, ermöglicht der Ansatz zwar die Verifikation von Graphtransformationssystemen mit einem großen Zustandsraum, die mit einem nicht-kompositionalen Vorgehen nicht verifiziert werden können, jedoch gilt auch hier, dass der Zustandsraum endlich und ein Initialgraph gegeben sein muss.

## 6.3 Analyse mittels Petrinetztechniken

Im voran gegangenen Abschnitt wurde Model Checking dazu verwendet, Eigenschaften für ein Graphtransformationssystem nachzuweisen. Baldan, Corradini und König [BCK01, BK02, BCK, BCK04] sowie Padberg und Enders [PE02] verwenden dagegen Petrinetztechniken zur Analyse von Graphtransformationssystemen.

Da die Anwendung beider Ansätze starken Restriktionen unterliegen, die in mechatronischen Systemen im Allgemeinen nicht erfüllt werden und es für beide Ansätze keine Werkzeugunterstützung gibt, sollen die beiden Ansätze hier nur kurz skizziert werden. In beiden Ansätzen erfolgt die Regelanwendung im Double Pushout Ansatz, wobei die Regeln keine negativen Anwendungsbedingungen besitzen dürfen. Zudem benötigen beide Ansätze einen Initialgraphen, um die Analyse durchführen zu können.

### 6.3.1 Approximierte Entfaltung

Das Ziel von Baldan, Corradini und König [BCK01, BK02, BCK, BCK04] besteht darin, Graphtransformationssysteme mittels existierender Techniken zu verifizieren. Dazu werden die Graphtransformationssysteme, auch solche mit unendlichem Zustandsraum, durch eine approximierte Entfaltung (engl. apporixmated unfolding) auf eine endliche Struktur, die so genannten Petrigraphen, abgebildet. Ein Petrigraph besteht aus einem Petrinetz und einem Graphen. Die Transitionen des Petrinetzes entsprechen den Regelanwendungen des ursprünglichen Graphtransformationssystems. Jeder Graph, der durch das Graphtransformationssystem erzeugt werden kann, entspricht einer erreichbaren Markierung des Petrinetzes, sodass zur Verifikation des Systems existierende Petrinetztechniken verwendet werden können.

Durch die approximierte Entfaltung entsteht ein Petrigraph, der aus einem Graphen und einem Petrinetz besteht. Jeder Graph, der durch das Graphtransformationssystem erzeugt werden kann, kann durch einen Graphhomomorphismus auf den Graphen des Petrigraphen abgebildet werden. Außerdem gilt, dass jeder durch das Graphtransformationssystem erzeugbare Graph einer erreichbaren Markierung im Petrinetzes des Petrigraphen entspricht. Deshalb gilt, dass jede Eigenschaft, die im Graphtransformationssystem erfüllt ist, auch im Petrigraphen erfüllt ist. Bei der Verifikation auf dem Graphen können Erreichbarkeitsanalysen durchgeführt und Verklemmungen erkannt werden. Die Analyse des Petrinetzes ermöglicht zudem den Nachweis von Lebendigkeitseigenschaften und Transitionsinvarianten.

Dieser Ansatz ist jedoch für die Analyse von mechatronischen Systeme nicht geeignet, da er starken Restriktionen unterliegt. Zu den bereits zu Beginn dieses Abschnitts genannten Einschränkungen kommt, dass durch die Regeln nur Kanten gelöscht werden dürfen. Knoten dürfen dagegen nur neu erzeugt werden. Diese Restriktion wird in einem mechatronischen System im Allgemeinen nicht erfüllt.

### 6.3.2 Regelinvarianten in Graphtransformationssystemen

Das Ziel von Padberg und Enders [PE02] besteht darin, Petrinetzanalysetechniken auf Graphtransformationssysteme zu übertragen.

Die Autoren übertragen die Begriffe Markierung (engl. marking), Schalten von Transitionen (engl. firing), Erreichbarkeit von Markierungen (engl reachable), Lebendigkeit des Netzes (engl. liveness), Beschränkung (engl. bounded) und vorallem Transitionsinvarianten.

Die ersten fünf Begriffe können direkt auf Graphtransformationssysteme übertragen werden, z.B. entspricht ein erreichbarer Graph im Graphtransformationssystem einer erreichbaren Markierung im Petrinetz. Da Graphtransformationssysteme mächtiger sind als Petrinetze, ist eine direkte Übertragung der Transitionsinvarianten auf Regelinvarianten nicht möglich. Eine Transitionsinvariante besagt, dass nach dem Schalten einer beliebigen Sequenz von Transitionen die ursprüngliche Markierung wieder erreicht wird. Dementsprechend besagt eine Regelinvariante, dass die Anwendung einer Sequenz von Regeln wieder im ursprünglichen Graphen endet.

Eine Sequenz von Regeln ist eine Regelinvariante, wenn alle Elemente, die durch die Anwendungen der Regeln gelöscht werden, durch die Anwendungen auch wieder erzeugt werden. Um festzustellen, ob dies der Fall ist, werden die Regeln der Sequenz zu einer Regel mit multiplen Pushout zusammengefasst. Gilt dann, dass es einen Isomorphismus gibt, der die linke Seite des multiplen Pushouts $L$ auf die rechte Seite $R$ abbildet, so stellt die Sequenz der Regeln einen Regelinvariante dar.

Abgesehen davon, dass der Ansatz zur Analyse einen Startgraphen benötigt und die Regeln keine negativen Anwendungsbedingungen enthalten dürfen, ist es nicht möglich, beschriftete bzw. typisierte Graphen zu verwenden. Das Hauptproblem dieses Ansatzes besteht jedoch darin, dass jede mögliche Regelsequenz betrachtet werden muss. Da in einer solchen Sequenz jede Regel beliebig oft auftreten kann, gibt es unendlich viele Möglichkeiten eine solche Sequenz zu bilden, obwohl die Zahl der Regeln endlich ist.

## 6.4 Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten vorgestellt. Es wurden verschiedene Vorgehensweise erläutert, die dazu verwendet werden können, die Korrektheit von Graphtransformationssystemen zu garantieren.

Der erste dazu vorgestellte Ansatz modifiziert die Regeln eines Graphtransformationssystems so, dass nur korrekte Zustände durch die Regelanwendung erzeugt werden können. Allerdings erfolgt eine solche Modifikation auch dann, wenn die Regel schon vorher korrekt war, wodurch die negativen Anwendungsbedingungen der Regeln unnötig vergrößert werden. Der darauf aufbauende Ansatz, der eine Regel nur dann verändert, wenn deren Anwendung zu einem Fehler führen kann, benötigt einen Startgraphen und schränkt die Ausdrucksstärke der Transformationsregeln stark ein.

Die übrigen Ansätze verifizieren, ob ein gegebenes Graphtransformationssystem korrekt ist. Diese Ansätze benötigen alle einen Startgraphen, um die Verifikation durchführen zu können. Nur der Ansatz zur approximierten Entfaltung von Baldan, Corradini und König ist zur Verifikation von Graphtransformationssystemen mit unendlichen Zustandsräumen geeignet, unterliegt aber anderen Einschränkungen, die eine Anwendung des Ansatzes für die Verifikation der Software mechatronischer Systeme verhindern.

# Kapitel 7

# Zusammenfassung und Ausblick

In den Kapiteln 2 und 3 wurde ein Ansatz zur kompositionalen Modellierung und Verifikation der Software mechatronischer Systeme vorgestellt. In Kapitel 5 wurde gezeigt, wie die Modellierung und Verifikation in der Fujaba Real-Time Tool Suite prototypisch umgesetzt worden ist. In diesem Kapitel soll der gesamte Ansatz noch einmal zusammengefasst werden und ein Ausblick auf zukünftige Arbeiten gegeben werden.

## 7.1 Zusammenfassung

Ein mechatronisches System kann als Operator-Controller-Modul aufgefasst werden, dessen Informationsverarbeitung in die Bereiche Controller, reflektorischer Operator und kognitiver Operator unterteilt ist. Die Software, die zur Informationsverarbeitung im reflektorischen Operator eingesetzt wird, ist für die Steuerung des Controllers sowie für die Koordination mit anderen OCMs verantwortlich. Da die Software des reflektorischen Operators sicherheitskritisch ist, bestand das Ziel dieser Arbeit darin, einen Ansatz zur kompositionalen Modellierung und Verifikation der Software zu entwickeln.

Der vorgestellte Ansatz baut auf den existierenden Konzepten der MECHATRONIC UML auf und erweitert diese.

In der MECHATRONIC UML wird die Softwarearchitektur eines mechatronischen Systems mittels Komponentendiagrammen beschrieben. Die Interaktion, die zwischen zwei oder mehr Komponenten stattfinden soll, wird durch Koordinationsmuster spezifiziert.

Verhalten, sowohl komponenteninternes Verhalten als auch das Kommunikationsverhalten zwischen verschiedenen Komponenten, wird durch Real-Time Statecharts modelliert.

Die Verifikation der Koordinationsmuster und Komponenten erfolgt durch komopsitionales Model Checking, d.h. jedes Koordinationsmuster und jede Komponente kann unabhängig von den anderen Koordinationsmustern und Komponenten verifiziert werden. Im Gegensatz zu den meisten anderen kompositionalen Ansätzen ist es nicht notwendig, zunächst das gesamte Modell zu erstellen und dann in kleinere verifizierbare Teilmodelle zu unterteilen. Die Dekomposition des Modells ergibt sich aus der Modellierung.

Nach der erfolgreichen Verifikation der Koordinationsmuster und Komponenten gilt: Interagieren mehrere Komponenteninstanzen ausschließlich über Instanzen der Koordinationsmuster, so ist diese Interaktion sicher. Allerdings sind mechatronische Systeme sehr dynamisch. Daraus folgt, dass bei der Instanzierung einer Komponente nicht bekannt ist, mit welchen anderen Komponenten sie zur Laufzeit interagieren muss. Da in mechatronischen Systemen zudem nur begrenzte Speicher- und Rechenkapazitäten vorhanden sind, ist es nicht möglich, zwischen jedem Paar von Komponenteninstanzen alle möglichen Muster zu instanzieren. Stattdessen ist es notwendig, zur Laufzeit die benötigten Koordinationsmuster zu instanzieren und nicht mehr benötigte Instanzen eines Koordinationsmusters wieder zu löschen.

Eine Methode, die ein Koordinationsmuster instanziert, kann als Story Pattern modelliert werden. Ein solches Story Pattern wird dann durch das komponenteninterne Real-Time Statechart als entry()-, do()- oder exit()-Methode in einem Zustand aufgerufen oder als Seiteneffekt beim Schalten einer Transition ausgeführt.

Bei der Verifikation der Komponente wird davon ausgegangen, dass die aufgerufenen Methoden korrekt sind, sie werden durch das Model Checking nicht überprüft. Um die Korrektheit der Methoden nachzuweisen, wurde in dieser Arbeit ein neuer Ansatz vorgestellt.

In diesem Ansatz wird ein Systemzustand als Graph aufgefasst, dessen Knoten den im Systemzustand vorkommenden Objekten entsprechen und dessen Kanten die Links des Systemzustands darstellen. Ein Story Pattern, das Objekte und Links erzeugt oder löscht, stellt dann eine spezielle Form einer Graphtransformationsregel dar. Sicherheitseigenschaften, die bei der Anwendung einer solchen Regel erhalten bleiben müssen, werden durch Graphmuster spezifiziert. Eigenschaften, die kritische Situationen oder Unfälle modellieren und niemals auftreten dürfen, werden als verbotene Graphmuster beschrieben. Auch diese Graphmuster können durch Story Patterns dargestellt werden.

Die meisten der existierenden Ansätze zur Verifikation von Graphtransformationssystemen benötigen zum einen einen initialen Graphen, also einen Anfangszustand des Systems. Zum anderen muss gelten, dass ausgehend vom Initialgraphen nur endlich viele Graphen durch die Anwendung der Regeln erreichbar sind. Da diese Einschränkungen in einem mechatronischen System zumeist jedoch nicht erfüllt sind, wurde in dieser Arbeit ein Ansatz vorgestellt, der diesen Einschränkungen nicht unterliegt. Statt nachzuweisen, dass alle erreichbaren Graphen korrekt sind, zeigt der Ansatz, dass eine Regelanwendung auf einen korrekten Graphen immer wieder in einem korrekten Graphen resultiert. Die Erreichbarkeit eines Graphen wird dabei außer Acht gelassen. Das heißt, der Ansatz zeigt, dass die Abwesenheit der verbotenen Graphmuster eine induktive Invariante des Systems ist. Der Nachweis wird dabei rückwärts geführt, d.h. für einen inkorrekten Graphen wird geprüft, ob dieser das Resultat einer Regelanwendung auf einen korrekten Graphen ist.

Für eine Graphtransformationsregel gilt, dass sie nur Auswirkungen auf Elemente des Anwendungsgraphen haben kann, auf die die Elemente der Regel abgebildet werden. Zudem kann die Anwendung einer Regel nur auf zwei Arten zu einer Verletzung einer Eigenschaft führen, d.h. einen Graphen erzeugen, der ein verbotenes Graphmuster enthält. Zum einen, wenn sie ein Element erzeugt, sodass die Anwendungsbedingung des verbotenen Graphmusters erfüllt ist, die negative Anwendungsbedingung jedoch nicht. Zum anderen, wenn sie ein Element der negativen Anwendungsbedingung löscht.

Diese Fakten können beim Nachweis induktiver Invarianten ausgenutzt werden. Statt vollständige Graphen zu betrachten, werden Graphmuster betrachtet. Ein solches Graphmuster besteht aus der rechten Seite einer Regel sowie aus einem verbotenen Graphmuster. Es wird als Ergebnisgraphmuster bezeichnet und repräsentiert die Menge aller Graphen, auf die die rechte Regelseite und das verbotene Graphmuster abgebildet werden können. Da das Ergebnisgraphmuster ein verbotenes Graphmuster enthält, beschreibt es eine Menge von inkorrekten Graphen. Zudem enthält das Ergebnisgraphmuster die rechte Regelseite, sodass eine Rückwärtsanwendung der entsprechenden Regel auf das Ergebnisgraphmuster möglich ist.

Wird die Regel rückwärts auf das Ergebnisgraphmuster angewendet, so resultiert das Startgraphmuster. Enthält dieses Startgraphmuster keines der verbotenen Graphmuster, so gilt, dass jeder korrekte Graph, auf den das Startgraphmuster abgebildet werden kann, durch Anwendung der betrachteten Regel in einen Graphen überführt werden kann, der das Ergebnisgraphmuster enthält und somit inkorrekt ist.

Das Verfahren kann auch für Systeme mit unendlich großem Zustandsraum angewendet werden. Darüber hinaus wird zur Verifikation kein Initialgraph benötigt. In einer Aufwandsabschätzung wurde gezeigt, dass der Ansatz linear in der Anzahl der Regeln und verbotenen Graphmuster, aber exponentiell in der Größe der Graphen der Regel und verbotenen Graphmuster ist. Das Verfahren wurde als Plugin in der Fujaba Real-Time Tool Suite prototypisch realisiert. In einer Evaluierung konnte gezeigt werden, dass das Verfahren trotz seiner Komplexität anwendbar ist, wenn die Graphen der Regeln und der verbotenen Graphmuster nicht zu groß sind.

Ist die Verifikation der Koordinationsmuster und Komponenten sowie der Methoden, die in den Komponenten intern aufgerufen werden, abgeschlossen, so ist auch der gesamte Verifikationsprozess abgeschlossen. Eine Verifikation des gesamten Modells ist nicht notwendig und meistens auf Grund der Größe des Modells auch nicht möglich. Die vorgestellten Modellierungs- und Verifikationskonzepte garantieren, dass das gesamte Modell korrekt ist, wenn es syntaktisch korrekt aus den verifizierten Koordinationsmustern und Komponenten zusammengesetzt wird.

Damit auch der Code, der später im mechatronischen System wirklich ausgeführt wird, die verifizierten Sicherheitseigenschaften erfüllt, muss er automatisch generiert werden. Da zur Modellierung nur Diagrammarten verwendet werden, für die auch eine formale Semantik definiert wurde, ist eine solche automatische Codegenerierung möglich.

## 7.2 Ausblick

Im Fachgebiet Softwaretechnik laufen zurzeit verschiedene Arbeiten, in denen die MECHATRONIC UML erweitert wird. Dabei betrachten die Arbeiten vor allem Erweiterungen um hybride Aspekte, d.h. die Modellierung und Verifikation von kontinuierlichen und diskreten Werten (siehe dazu auch [Bur05]).

In diesem Abschnitt soll genauer auf die Erweiterungsmöglichkeiten des Ansatzes zum automatischen Nachweis induktiver Invarianten eingegangen werden.

Dieser Ansatz wurde dazu entwickelt, um die Methoden, die durch Story Patterns beschrieben und in einem mechatronischen System ausgeführt werden, verifizieren zu können. Die

Ausführung einer solchen Methode, d.h. die Anwendung eines Story Patterns auf eine Instanzsituation führt dazu, dass neue Objekte und Links erzeugt oder existierende Objekte und Links gelöscht werden. Ein wichtiges Konzept der objektorientierten Modellierung, das zwar durch die Story Patterns, nicht aber durch den vorgestellten Verifikationsansatz, unterstützt wird, ist die Vererbung. Um auch Story Patterns verifizieren zu können, in denen Vererbung verwendet wurde, ist eine Erweiterung des Ansatzes notwendig.

Neben der Vererbung unterstützen Story Patterns auch die Modifikation von Attributwerten eines existierenden Objekts. Eine Erweiterung des Verifikationsansatzes um Attribute und deren Modifikation ist deshalb angebracht.

In den Klassendiagrammen, die dazu verwendet werden, um die Typen der Objekte und ihrer möglichen Verbindungen zu spezifizieren, können Kardinalitäten spezifiziert werden. Diese geben für ein Objekt an, wieviele andere Objekte von einem bestimmten Typ adjazent zu ihm sein können. Um diese Kardinalitäten bei der Verifikation berücksichtigen zu können, müssen zusätzliche verbotene Graphmuster angegeben werden. Dies ist zwar prinzipiell möglich, hat sich jedoch bei der Evaluierung des Ansatzes in der Fujaba Real-Time Tool Suite als eine sehr lästige Arbeit herausgestellt. Deshalb sollte der Ansatz so erweitert werden, dass die Kardinalitäten bei der Verifikation berücksichtigt werden ohne das explizit zusätzliche verbotene Graphmuster spezifiziert werden müssen.

Die Story Patterns werden dazu verwendet, um Methoden eines mechatronischen Systems zu spezifizieren. Zurzeit ist es unter bestimmten Umständen möglich, mit dem Ansatz von Seibel [Sei05] festzustellen, wie lange diese Ausführung maximal dauert. Eine explizite Modellierung und Verifikation von Zeit und Zeiteigenschaften wird momentan jedoch nicht von den Story Patterns unterstützt und kann auch mit dem Ansatz aus Kapitel 3 noch nicht verifiziert werden. Da Zeit in mechatronischen Systemen jedoch eine besondere Rolle spielt, sollte sie in die Modellierung und die Verifikation der Story Patterns aufgenommen werden.

Story Patterns stehen meist nicht allein, sondern beschreiben eine Aktivität eines Story Diagramms. Wird ein Story Diagramm auf eine Instanzsituation angewendet, so wird zunächst das Story Pattern der Startaktivität angewendet. Dann das Story Pattern der nächsten Aktivität und so weiter, bis das Story Pattern der Stopaktivität angewendet wurde. Eine Sicherheitseigenschaft ist eine induktive Invariante eines Systems, wenn ein Story Diagramm angewendet auf eine korrekte Instanzsituation wieder in einer korrekten Instanzsituation resultiert. Die Instanzsituationen nach dem Anwenden des Story Patterns der Startaktivität und vor der Anwendung der Stopaktivität müssen die Sicherheitseigenschaft nicht erfüllen. Der in Kapitel 3 vorgestellte Ansatz ist zur Verifikation solcher Story Diagramme im Allgemeinen aufgrund seiner Komplexität nicht anwendbar. Für sehr einfache Story Diagramme mit relativ kleinen Story Patterns sollte eine Anwendung des Ansatzes jedoch möglich sein.

Alle bisher vorgestellten Erweiterungsmöglichkeiten führen dazu, dass die Komplexität des Ansatzes weiter steigt. Um diese Komplexität in den Griff zu bekommen, reichen die Mittel, die bisher im Plugin zum automatischen Nachweis induktiver Invarianten eingesetzt werden, jedoch nicht aus. Eine Möglichkeit, die Verifikation trotz der erhöhten Komplexität durchführen zu können, besteht darin, das Verifikationsproblem auf ein Constraint-Solving-Problem abzubilden. Zum Lösen des Problems kann dann ein existierender Constraint-Solver (z.B. ILOG

Solver[1]) verwendet werden. Zudem wurde in [GSK+06] eine Idee skizziert, wie man das Problem symbolisch kodieren und dadurch den Verifikationsaufwand verringern kann. Dazu werden die Ergebnisgraphmuster in Mengen zusammengefasst. Die Bildung dieser Mengen sowie die Rückwärtsanwendung der Regeln und der Test, ob die resultierenden Startgraphmuster korrekt sind, erfolgt mittels des BDD-basierten Interpreters CroCoPat [BNL05].

Eine weitere Möglichkeit, der Komplexität entgegenzuwirken, besteht darin, die Kompositionalität der Modellierung bei der Verifikation stärker auszunutzen. So wurden in Abschnitt 2.3.2 die Kulturhierarchien eingeführt, um mechatronische Systeme kompositional modellieren und verifizieren zu können. Zurzeit werden die Informationen aus einer solchen Hierarchie nur insofern bei der Verifikation ausgenutzt, dass bei der Verifikation der Regeln einer Kultur nur die Regeln und verbotenen Graphmuster von übergeordneten Kulturen berücksichtigt werden. Untergeordnete Kulturen oder Kulturen, die in anderen Zweigen der Hierarchie angeordnet sind brauchen nicht berücksichtigt werden. Es ist jedoch vorstellbar, dass Verifikationsergebnisse aus höheren Hierarchien bei der Verifikation untergeordneter Kulturen ausgenutzt werden können. Beispielsweise haben die Verhaltensregeln einer höheren Hierarchieebene eine geringere Priorität als die einer untergeordneten Ebene. Die Instanzierungs- und Verhaltensregeln erfüllen die Sicherheitseigenschaften ihrer Hierarchieebene und aller höheren Ebenen. Da die Verhaltensregeln der höheren Ebenen nur anwendbar sind, wenn keine Regel der aktuell betrachteten Kultur anwendbar ist, sollten sie auch deren Sicherheitseigenschaften nicht verletzen können.

Neben der Erweiterung, die es ermöglicht, noch ausdruckstärkere Story Patterns zu verifizieren, besteht auch die Möglichkeit den Ansatz aus Kapitel 3 mit dem von Heckel und Wagner vorgestellten Ansatz zur Erzeugung konsistenter Graphtransformationsregeln zu verbinden. Die Idee besteht darin, den Ansatz von Heckel und Wagner an die Formalisierung aus Kapitel 3 anzupassen. Dann kann der Ansatz zum Nachweis von induktiven Invarianten dazu verwendet werden, um die Korrektheit einer Menge von Graphtransformationsregeln zu überprüfen. Wird bei dieser Überprüfung für eine Regel ein Gegenbeispiel generiert, das zeigt, dass die Regel inkorrekt ist, so kann der Ansatz von Heckel und Wagner dazu eingesetzt werden, um die Regel automatisch zu korrigieren.

Eine Erweiterung des Ansatzes aus Kapitel 3 wie zuvor beschrieben, ermöglicht auch seine Anwendung in anderen Kontexten. Beispielsweise ist es dann möglich, die von Tichy et al. [TSG04, TGSP05, TG05] vorgestellten Regeln zur Selbstadaption und zur Selbstreparatur zu verifizieren. Auf diese Weise könnte nicht nur die Sicherheit eines Systems erhöht werden, sondern auch seine Verlässlichkeit (engl. reliability). Außerdem ist es dann möglich, die Konsistenzbedingungen der von Gehrke in [Geh05] aufgestellten Regeln zur Modelltransformation zu verifizieren.

[1] www.ilog.com

# Literaturverzeichnis


[ACD90] ALUR, R.; COURCOUBETIS, C. ; DILL, D.: Model-Checking for Real-Time Systems. In: *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science*. Philadelphia, PA, 1990, S. 414–425

[AD90] ALUR, Rajeev; DILL, David L.: Automata for Modeling Real-Time Systems. In: *Proceedings of the 17th International Colloquium on Automata, Languages and Programming*. New York, NY, USA : Springer-Verlag New York, Inc., 1990, S. 322–335

[AD94] ALUR, Rajeev; DILL, David L.: A theory of timed automata. In: *Theoretical Computer Science* 126 (1994), Nr. 2, S. 183–235

[BCK] BALDAN, Paolo; CORRADINI, Andrea ; KÖNIG, Barbara: Static Analysis of Distributed Systems with Mobility Specified by Graph Grammars - A Case Study. In: *Proceedings of the 6th International Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, USA*

[BCK01] BALDAN, Paolo; CORRADINI, Andrea ; KÖNIG, Barbara: A Static Analysis Technique for Graph Transformation Systems. In: K.G.LARSEN (Hrsg.); M.NIELSEN (Hrsg.): *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001), Aalborg, Denmark* Bd. 2154, Springer-Verlag Heidelberg, 2001, S. 381–395

[BCK04] BALDAN, Paolo; CORRADINI, Andrea ; KÖNIG, Barbara: Verifying Finite-State Graph Grammars: An Unfolding-Based Approach. In: GARDNER, Philippa (Hrsg.); YOSHIDA, Nobuko (Hrsg.): *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004), London, UK* Bd. 3170, Springer-Verlag Heidelberg, 2004, S. 83–98

[Bec05] BECKER, Basil: *Automatische Überprüfung Induktiver Invarianten für Graphtransformationssysteme*. Paderborn, Deutschland, Universität Paderborn, Studienarbeit, 2005

[BG03] BURMESTER, Sven; GIESE, Holger: The Fujaba Real-Time Statechart PlugIn. In: GIESE, Holger (Hrsg.); ZÜNDORF, Albert (Hrsg.): *Proceedings of the 1st International Fujaba Days 2003, Kassel, Deutschland* Bd. tr-ri-04-247, Universität Paderborn, 2003, S. 1–8

[BGH+05] BURMESTER, Sven; GIESE, Holger; HIRSCH, Martin; SCHILLING, Daniela ; TICHY, Matthias: The Fujaba Real-Time Tool Suite: Model-Driven Development of Safety-Critical, Real-Time Systems. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA*, ACM Press, 2005, S. 670 – 671

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

[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.): *Proceedings of 1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004), Setubal, Portugal*, INSTICC Press, 2004, S. 222–229

[BGS05a] BECKER, Basil; GIESE, Holger ; SCHILLING, Daniela: A Plugin for Checking Inductive Invariants when Modeling with Class Diagrams and Story Patterns. In: *Proceedings of the 3rd International Fujaba Days 2005, Paderborn, Deutschland* Bd. tr-ri-05-259, Universität Paderborn, 2005, S. 45 – 48

[BGS05b] BURMESTER, Sven; GIESE, Holger ; SCHÄFER, Wilhelm: Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In: *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Deutschland*, Springer Verlag, 2005 (Lecture Notes in Computer Science (LNCS)), S. 1–15

[BGT05] BURMESTER, Sven; GIESE, Holger ; TICHY, Matthias: Model-Driven Development of Reconfigurable Mechatronic Systems with Mechatronic UML. In: ASSMANN, Uwe (Hrsg.); RENSINK, Arend (Hrsg.) ; AKSIT, Mehmet (Hrsg.): *Model Driven Architecture: Foundations and Applications* Bd. 3599, Springer-Verlag Heidelberg, 2005, S. 47 – 61

[BK02] BALDAN, Paolo; KÖNIG, Barbara: Approximating the Behaviour of Graph Transformation Systems. In: CORRADINI, A. (Hrsg.); EHRIG, H. (Hrsg.); KREOWSKI, H.-J. (Hrsg.) ; ROZENBERG, G. (Hrsg.): *Proceedings of the 1st International Conference on Graph Transformation (ICGT 2002), Barcelona, Spanien* Bd. 2505, Springer-Verlag Heidelberg, 2002, S. 14–29

[BNL05] BEYER, Dirk; NOACK, Andreas ; LEWERENTZ, Claus: Efficient Rational Calculation for Software Analysis. In: *IEEE Transactions on Software Engineering* 31 (2005), Nr. 2, S. 137–149

[BTG04] BURMESTER, Sven; TICHY, Matthias ; GIESE, Holger: Modeling Reconfigurable Mechatronic Systems with Mechatronic UML. In: ASSMANN, U. (Hrsg.): *Proceedings of Model Driven Architecture: Foundations and Applications (MDAFA 2004), Linköping, Sweden*, 2004, S. 155 – 169

[Bur02] BURMESTER, Sven: *Generierung von Java Real-Time Code für zeitbehaftete UML Modelle*, Universität Paderborn, Paderborn, Deutschland, Diplomarbeit, 2002

[Bur05] BURMESTER, Sven: *Model-Driven Engineering of Reconfigurable Mechatronic Systems (erscheint)*. Paderborn, Deutschland, Universität Paderborn, Dissertation, 2005

[CGP02] CLARK, Edmund M.; GRUMBERG, Orna ; PELED, Doron A.: *Model Checking*. 4. MIT Press, 2002

[Cha03] CHARPENTIER, Michel: Composing Invariants. In: ARAKI, Keijiro (Hrsg.); GNESI, Stefania (Hrsg.) ; MANDRIOLI, Dino (Hrsg.): *Proceedings of the 12th International Symposium of Formal Methods Europe (FME 2003), Pisa, Italien* Bd. 2805, Springer-Verlag Heidelberg, 2003, S. 401 – 421

[DFRS03] DOTTI, Fernando L.; FOSS, Luciana; RIBEIRO, Leila ; SANTOS, Osmar M.: Verification of Distributed Object-Oriented Systems. In: NAJM, Elie (Hrsg.); NESTMANN, Uwe (Hrsg.) ; STEVENS, Perdita (Hrsg.): *Proceedings of the 6th Internation Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003), Paris, Frankreich* Bd. 2884, Springer-Verlag Heidelberg, 2003, S. 261 – 275

[FNT98] FISCHER, Thorsten; NIERE, Jörg ; TORUNSKI, Lars: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling*. Paderborn, Deutschland, Univerisity of Paderborn, Diplomarbeit, 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.): *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT 1998), Paderborn, Deutschland* Bd. 1764, Springer Verlag, 1998, S. 296–309

[GB03] GIESE, Holger; BURMESTER, Sven: Real-Time Statechart Semantics / Universität Paderborn. Paderborn, Deutschland, 2003 (tr-ri-03-239). – Forschungsbericht

[GBK+03] GIESE, Holger; BURMESTER, Sven; KLEIN, Florian; SCHILLING, Daniela ; TICHY, Matthias: Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In: HENDERSON-SELLERS, B. (Hrsg.); DEBENHAM, J. (Hrsg.): *2nd International Workshop on Agent-Oriented Methodologies (OOPSLA 2003), Anaheim, USA*, Center for Object Technology Applications and Research (COTAR), University of Technology, Sydney, Australia, 2003, S. 21–32

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

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

[Geh05] GEHRKE, Matthias: *Entwurf mechatronischer Systeme auf Basis von Funktionshierarchien und Systemstrukturen (erscheint)*, Universität Paderborn, Dissertation, 2005

[Gei02] GEIGER, Leif: *Design Level Debugging mit Fujaba*, Technische Universität Braunschweig, Studienarbeit, 2002

[Gie03] GIESE, Holger: A Formal Calculus for the Compositional Pattern-Based Design of Correct Real-Time Systems / Universität Paderborn. Paderborn, Deutschland, 2003 (tr-ri-03-240). – Forschungsbericht

[GS04] GIESE, Holger; SCHILLING, Daniela: Towards the Automatic Verification of Inductive Invariants for Invinite State UML Models / Universität Paderborn. Paderborn, Deutschland, 2004 (tr-ri-04-252). – Forschungsbericht

[GSK+06] GIESE, Holger; SCHILLING, Daniela; KLEIN, Florian; BECKER, Basil ; BEYER, Dirk: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), Shanghai, China (angenommen)*, ACM Press, 2006

[GST+03] GIESE, Holger; SCHILLING, Daniela; TICHY, Matthias; BURMESTER, Sven; SCHÄFER, Wilhelm ; FLAKE, Stephan: Towards the Compositional Verification of Real-Time UML Designs / Universität Paderborn. Paderborn, Deutschland, 2003 (tr-ri-03-241). – Forschungsbericht. – 1–47 S

[GTB+03] GIESE, Holger; TICHY, Matthias; BURMESTER, Sven; SCHÄFER, Wilhelm ; FLAKE, Stephan: Towards the Compositional Verification of Real-Time UML Designs. In: *Proceedings of the European Software Engineering Conference (ESEC), Helsinki, Finland*, ACM Press, 2003, S. 38–47

[HE00] HECKEL, Reiko; ENGELS, Gregor: Graph Transformation and Visual Modeling Languages. In: *Bulletin of the European Association for Theoretical Computer Science (EATACS)* (2000), Nr. 71

[Hec98] HECKEL, Reiko: Compositional Verification of Reactive Systems Specified by Graph Transformation. In: ASTESIANO, Egidio (Hrsg.): *Proceedings of the 1st International Conference on Fundamental Approaches to Software Engineering (FASE 1998), Held as Part of the European Joint Conferences on the Theory and Practice of Software (ETAPS 1998), Lisabon, Protugal* Bd. 1382, Springer-Verlag Heidelberg, 1998, S. 138–153

[HEWC97] HECKEL, Reiko; EHRIG, Hartmut; WOLTER, Uwe ; CORRADINI, Andrea: Integrating the Specification Techniques of Graph Transformation and Temporal Logic. In: *Proceedings of the 22nd International Symposium on Mathematical Foundations of Computer Science (MFCS 1997), Bratislava, Slovakia* Bd. 1295, Springer-Verlag Heidelberg, 1997, S. 219 – 228

[HG03] HIRSCH, Martin; GIESE, Holger: Towards the Incremental Model Checking of Complex RealTime UML Models. In: GIESE, Holger (Hrsg.); ZÜNDORF, Albert (Hrsg.): *Proceedings of the 1st International Fujaba Days 2003, 13.-14.October, Kassel, Deutschland* Bd. tr-ri-04-247, Universität Paderborn, 2003

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

[Hir04] HIRSCH, Martin: *Effizientes Model Checking von UML-RT Modellen und Realtime Statecharts mit UPPAAL*, Universität Paderborn, Diplomarbeit, 2004

[HLM05] HECKEL, Reiko; LAJIOS, Georgios ; MENGE, Sebastian: Modulare Analyse Stochastischer Graphtransformationssysteme. In: *Software Engineering* Bd. 64, Gesellschaft für Informatik, 2005, S. 141–152

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

[Hol03] HOLZMAN, Gerard J.: *The SPIN Model Checker: Primer and Reference Manual*. Addision Wesley Professional, 2003

[HW95] HECKEL, Reiko; WAGNER, Annika: Ensuring consistency of conditional graph rewriting - a constructive approach. In: CORRADINI, Andrea (Hrsg.); MONTANARI, Ugo (Hrsg.): *Proceedings of Joint Computergraph/Semagraph Workshop on Graph Rewriting and Computation (SEGRAGRA 1995), Volterra, Italien* Bd. 2, Elsevier, 1995

[KG04] KLEIN, Florian; GIESE, Holger: Advanced separation of concerns for mechatronic multi-agent systems through dynamic communities. In: AL., Ricardo C. (Hrsg.): *Proceedings of the 3rd Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2004) held in Conjunction with the International Conference on Software Engineering (ICSE 2004), Edinburgh, Schottland*, IEE, 2004, S. 112–119

[KG05] KLEIN, Florian; GIESE, Holger: Separation of concerns for mechatronic multi-agent systems through dynamic communities. In: CHOREN, Ricardo (Hrsg.); GARCIA, Alessandro (Hrsg.); LUCENA, Carlos (Hrsg.) ; ROMANOVSKY, Alexander (Hrsg.): *Software Engineering for Multi-Agent Systems III: Research Issues and Practical Applications* Bd. 3390. Springer Verlag, 2005, S. 272–289

[Kin95] KINDLER, Ekkart: Invariants, composition, and substitution. In: *Acta Informatika* 32 (1995), S. 299 – 312

[KMPP02] KOCH, Manuel; MANCINI, Luigi V. ; PARISI-PRESICCE, Francesco: Decidability of Safety in Graph-Based Models for Access Control. In: GOLLMANN, D. (Hrsg.); KARJOTH, G. (Hrsg.) ; WAIDNER, M. (Hrsg.): *Proceedings of the 7th European Symposium on Research in Computer Security (ESORICS 2002), 14.-16.Oktober, Zürich, Schweiz* Bd. 2502, Springer-Verlag Heidelberg, 2002, S. 229 – 243

[KPP03] KOCH, Manuel; PARISI-PRESICCE, Francesco: UML Specificationd Access Control Policies and their Formal Verification / George Mason University. Fairfax, USA, 2003 (ISE-TR-03-10). – Forschungsbericht

[LPY97] LARSEN, K.; PETTERSSON, P. ; YI, W.: UPPAAL in a Nutshell. In: *Springer International Journal of Software Tools for Technology* 1 (1997), Nr. 1

[Mat01] MATZ, Michael: *Konzeption und Implementierung eines Verfahren zum Nachweis der Konsistenz in einer attributierten Graphgrammatik*, Technische Universität Berlin, Diplomarbeit, 2001

[NNWZ00] NICKEL, Ulrich A.; NIERE, Jörg; WADSACK, Jörg P. ; ZÜNDORF, Albert: Roundtrip Engineering with FUJABA. In: EBERT, J. (Hrsg.); KULLBACH, B. (Hrsg.) ; LEHNER, F. (Hrsg.): *Proceedings of the 2nd Workshop on Software-Reengineering (WSR00), Bad Honnef, Germany*, Fachberichte Informatik, Universität Koblenz-Landau, 2000

[NNZ00] NICKEL, Ulrich A.; NIERE, Jörg ; ZÜNDORF, Albert: Tool demonstration: The FUJABA environment. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE00), Limerick, Ireland*, ACM Press, 2000, S. 742–745

[OHG04] OBERSCHELP, Oliver; HESTERMEYER, Thorsten ; GIESE, Holger: Strukturierte Informationsverarbeitung für selbstoptimierende mechatronische Systeme. In: GAUSEMEIER, Jürgen (Hrsg.); WALLASCHECK, Jürgen (Hrsg.): *Proceedings of the 2nd Paderborner Workshop Intelligente Mechatronische Systeme, Paderborn, Deutschland* Bd. 145, 2004, S. 43–56

[PE02] PADBERG, Julia; ENDERS, Bettina E.: Rule Invariants in Graph Transformation Systems for Analyzing Safety-Critical Systems. In: CORRADINI, Andrea (Hrsg.); EHRIG, Hartmut (Hrsg.); KREOWSKI, Hans-Jörg (Hrsg.) ; ROZENBERG, Grzegorz (Hrsg.): *Proceedings of the First International Conference on Graph Transformation (ICGT 2002), 7.-12.October, Barcelona, Spanien* Bd. 2505, Springer-Verlag Heidelberg, 2002, S. 334 – 350

[Ren03] RENSINK, Arend: Towerds [sic] Model Checking Graph Grammars. In: LEUSCHEL, M. (Hrsg.); GRUNER, S. (Hrsg.) ; PRESTI, S. L. (Hrsg.): *Workshop on Automated Verification of Critical Systems (AVoCS 2003), Southampton, GB* Bd. DSSE–TR–2003–2, University of Southampton, 2003, S. 150–160

[Ren04] RENSINK, Arend: The GROOVE Simulator: A Tool for State Space Generation. In: PFALZ, John (Hrsg.); NAGL, Manfred (Hrsg.) ; BÖHLEN, Boris (Hrsg.): *Proceedings of the 2nd Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE 2004), Charlottesville, USA* Bd. 3062, Springer-Verlag Heidelberg, 2004, S. 479–485

[Roz97] ROZENBERG, Grzegorz: *Handbook of Graph Grammars and Computing by Graph Transformation*. Bd. 1. Foundations. World Science Publishing Co. Pte. Ltd., 1997

[RSV04] RENSINK, Arend; SCHMIDT, Ákos ; VARRÓ, Dániel: Model Checking Graph Transformations: A Comparison of Two Approaches. In: EHRIG, Hartmut (Hrsg.); ENGELS, Gregor (Hrsg.); PARISI-PRESICCE, Francesco (Hrsg.) ; ROZENBERG, Grzegorz (Hrsg.): *Proceedings of the 2nd International Conference on Graph Transformations (ICGT 2004), Rom, Italien* Bd. 3256, Springer-Verlag Heidelberg, 2004, S. 226 – 241

[Ruf01] RUF, Jürgen: RAVEN: Real-Time Analyzing and Verification Environment. In: *Journal on Universal Computer Science (J.UCS), Springer* 7 (2001), Nr. 1, S. 89 – 104

[Sei05] SEIBEL, Andreas: *Story Diagramme für eingebettete Echtzeitsysteme*, Universität Paderborn, Studienarbeit, 2005

[Ste05] STECKER, Alexander: *Model Checking von Real-Time Statecharts mit RAVEN*. Paderborn, Deutschland, Universität Paderborn, Studienarbeit, 2005

[Sto96] STOREY, Neil R.: *Safety Critical Computer Systems*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1996

[SV03] SCHMIDT, Ákos; VARRÓ, Dániel: CheckVML: A Tool for Model Checking Visual Modeling Languages. In: STEVENS, Perdita (Hrsg.); WHITTLE, Jon (Hrsg.) ; BOOCH, Grady (Hrsg.): *Proceedings of the 6th International Conference on the Unified Modeling Language (UML 2003), 20.-24.Oktober, San Francisco, USA* Bd. 2863, Springer-Verlag Heidelberg, 2003, S. 92 – 95

[Szy02] SZYPERSKI, Clemens: *Component Software - Beyond Object-Oriented Programming*. 2. Addison-Wesley, 2002

[TG05] TICHY, Matthias; GIESE, Holger: Extending Fault Tolerance Patterns by Visual Degradation Rules. In: *Proceedings of the Workshop on Visual Modeling for Software Intensive Systems (VMSIS 2005) at the the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2005), Dallas, USA*, 2005, S. 67 – 74

[TGSP05] TICHY, Matthias; GIESE, Holger; SCHILLING, Daniela ; PAULS, Wladimir: Computing Optimal Self-Repair Actions: Damage Minimization versus Repair Time. In: LEMOS, Rogério de (Hrsg.); ROMANOVSKY, Alexander (Hrsg.): *Proceedings of the Workshop on Architecting Dependable Systems (WADS 2005), held at the International Conference on Software Engineering (ICSE 2005),St. Louis, USA*, ACM Press, 2005

[TSG04] TICHY, Matthias; SCHILLING, Daniela ; GIESE, Holger: Design of Self-Managing Dependable Systems with UML and Fault Tolerance Patterns. In: *Proceedings of the Workshop on Self-Managed Systems (WOSS 2004) held at the ACM SIGSOFT Foundations in Software Engineering (FSE 2004), Newport Beach, USA*, 2004

[UML05] UML 2.0 Superstructure Specification / Object Management Group. 2005. – Forschungsbericht

[Var04] VARRÓ, Dániel: Automated Formal Verification of Visual Modeling Languages by Model Checking. In: *Software and System Modeling* 3 (2004), Nr. 2, S. 85–113

[Wen03] WENDEHALS, Lothar: *10 Steps to Build a Fujaba Plugin*. 2003. – http://www.se.eecs.uni-kassel.de/~fujabawiki/index.php/HowTos

[Zün01] ZÜNDORF, Albert: *Rigorous Object Oriented Software Development with Fujaba*. Paderborn, Deutschland, Universität Paderborn, Habilitation, 2001

# Anhang A

# Das Shuttle-Beispiel

In diesem Kapitel werden alle Regeln (Verhaltens- und Instanzierungsregeln) des Shuttle-Beispiels vorgestellt. Die Regeln wurden in Kapitel 2 in den drei Kulturen Movement-, ControlledMovement- und CoordinatedMovement-Kultur definiert. Für die Regeln der Movement-Kultur sowie der CoordinatedMovement-Kultur konnte gezeigt werden, dass sie bezüglich der Sicherheitseigenschaften korrekt sind. Die Regeln werden in Abschnitt A.1 erläutert. Zusätzlich wird in Abschnitt A.1.3 für jede Regel deren Priorität und die im Evaluierunsabschnitt (Abschnitt 5.4) eingeführte Größe angegeben.

Die Sicherheitseigeschaften, die in jeder der drei Kulturen gelten müssen, werden ebenfalls in diesem Kapitel (in Abschnitt A.2) erläutert. Für die Sicherheitseigenschaften wird auch die jeweilige Größe angegeben.

Der Ansatz aus Kapitel 3 und die entsprechende Implementierung in der Fujaba Real-Time Tool Suite (Abschnitt 5.3) berücksichtigen zurzeit nicht die im Klassendiagramm spezifizierten Kardinalitäten. Deshalb werden in Abschnitt A.2.3 verbotene Graphmuster beschrieben, die bei der Verifikation dazu verwendet wurden, die Kardinalitäten zu modellieren. Zu jedem dieser Graphmuster wird auch seine Größe genannt.

## A.1 Regeln

In diesem Abschnitt werden alle Regeln des Shuttle-Beispiels in Form von Story Pattern dargestellt und erläutert. Der Abschnitt ist in drei Unterabschnitte gegliedert. Der erste Abschnitt A.1.1 beschreibt die Regeln der Movment-Kultur. Diese enthält die Regeln member, noMember, moveNext, approachSwitch sowie moveSwitch. In Abschnitt A.1.2 werden die Regeln der ControlledMovement-Kultur vorgestellt. Diese Kultur enthält alle Regeln der ihr übergeordneten Movement-Kultur sowie die Regeln createPublication und deletePublication. Im dritten Abschnitt (Abschnitt A.1.3) werden die Regeln der CoordinatedMovement-Kultur beschrieben. Neben den Regeln der übergeordneten Movement- und ControlledMovment-Kulturen enthält die Kultur die Regeln createDC, moveDC und deleteDC.

### A.1.1 Die Regeln der Movement-Kultur

In diesem Abschnitt werden die beiden Instanzierungsregeln member und noMember sowie die drei Verhaltensregeln moveNext, approachSwitch und moveSwitch der Movement-Kultur als Story Pattern dargestellt und erläutert. Die Verhaltensregeln der Movement-Kultur beschreiben die Grundfunktionalität eines Shuttles, die Vorwärtsbewegung sowohl auf einer geraden Strecke als auch beim Annähern und Passieren einer Weiche.

Möchte ein Shuttle einen Track befahren, der zu einer anderen ControlledArea gehört als der Track, auf dem es sich zur Zeit befindet, so muss es zunächst in die Community der neuen ControlledArea aufgenommen werden. Dazu wird zwischen dem Shuttle und der ControlledArea ein neuer contains-Link erzeugt, falls es diesen noch nicht gibt. Die Zugehörigkeit eines Tracks zu einer ControlledArea wird nicht direkt durch Links definiert. Stattdessen enthält jede ControlledArea eine BaseStation. Alle Tracks, die von der BaseStation überwacht werden, sind automatisch Teil der ControlledArea. Deshalb ist in der Regel member in Abbildung A.1, die die Aufnahme eines Shuttles in einer ControlledArea beschreibt, auch das entsprechende BaseStation-Objekt angegeben.

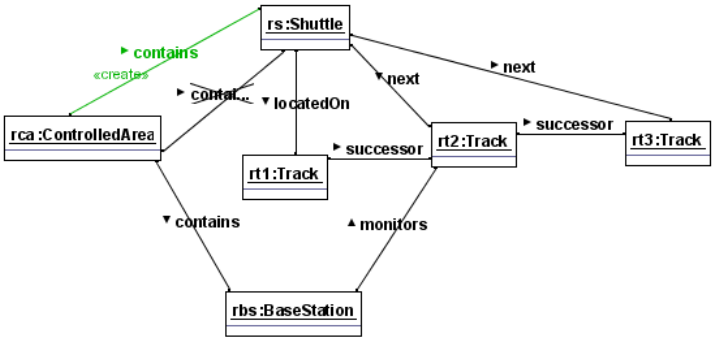



Abbildung A.1: Die Regel member

Während die member-Regel die Aufnahme eines Shuttles in eine ControlledArea beschreibt, beschreibt die Regel noMember, die in Abbildung A.13 als Story Pattern gegeben ist, das ein Shuttle nicht mehr zu einer ControlledArea gehört, sobald es ein Track befährt, dass nicht von der BaseStation der ControlledArea überwacht wird.

Die Regel moveNext (siehe Abbildung A.3) beschreibt die Fahrt eines Shuttles auf gerader Strecke, d.h. wenn keine Weiche in der Nähe ist. Dabei muss die Regel zusammen mit den beiden anderen Regeln die Sicherheitseigenschaften der Kultur, in diesem Fall collision und notMember (siehe Abschnitt A.2), einhalten. Das bedeutet, die Anwendung der Regel darf nicht dazu führen, dass sich zwei Shuttles auf einem Track befinden. Da die Regel nicht für die Weichenfahrt eines Shuttles anwendbar sein soll, darf der Track auf den das Shuttle rs1 als nächstes fahren möchte (dargestellt durch den next-Link) keine Weiche sein, d.h. sie darf nur rt1 : Track als Vorgänger haben. Dies wird durch das negative Objekt rt5 vom Typ Track dargestellt. Um eine Kollision von zwei Shuttles zu vermeiden, muss zudem gelten, dass sich auf den beiden Tracks rt2 und rt3, auf die das Shuttle rs1 als nächstes und übernächstes fahren möchte, kein weiteres Shuttle befinden; Dargestellt durch die beiden negativen Shuttle-Objekte rs2 und rs3.

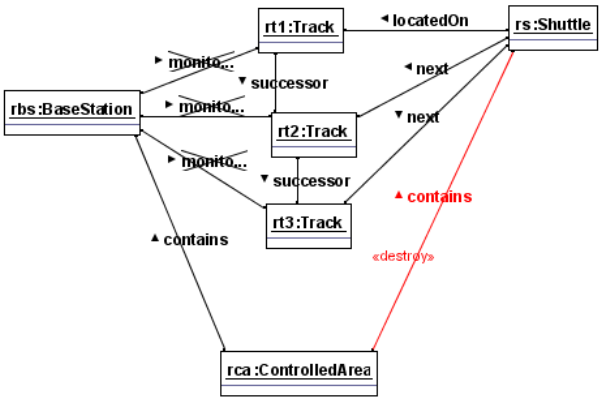



Abbildung A.2: Die Regel noMember

Sind diese Anforderungen erfüllt, so kann die Regel angewendet werden und $rs1$ weiterfahren. In diesem Fall wird der locatedOn-Link gelöscht und ein neuer zwischen dem Shuttle und rt2 erzeugt. Außerdem muss die Angabe, auf welchen Track das Shuttle als nächstes und übernächstes fahren möchte, aktualisiert werden. Dazu wird der next-Link zwischen dem Shuttle und rt2 gelöscht und ein neuer zum Track rt4 erzeugt.

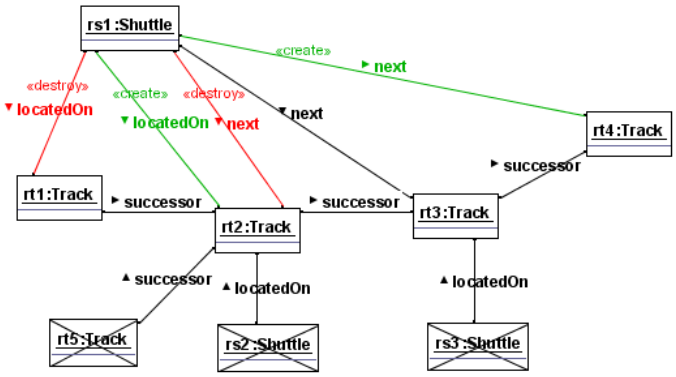



Abbildung A.3: Die Regel moveNext

Damit ein Shuttle sicher eine Weiche passieren kann ohne mit einem anderen Shuttle zu kollidieren, werden zwei weitere Regeln benötigt. Die erste, approachSwitch aus Abbildung A.4, beschreibt, wie sich ein Shuttle einer Weiche nähert. Die Weiche wird durch den Track rt3 dargestellt, der mit rt2 und rt5 zwei Vorgänger hat.

Durch seine beiden next-Links gibt das Shuttle zu verstehen, dass es diese Weiche passieren möchte. Damit dies jedoch sicher geschehen kann, darf sich auf dem nächsten Track rt2, aber auch auf der Weiche selber, kein weiteres Shuttle befinden. Bei einer Weiche können auch Shuttles vom parallelen Track kommen. Für eine sichere Weichendurchfahrt darf sich auf diesem parallelen Track ($rt5$) kein anderes Shuttle befinden.

Ist die Anwendungsbedingung der Regel erfüllt, so wird das Shuttle, wie in der moveNext Regel, auf den nächsten Track gesetzt und seine next-Links aktualisiert.

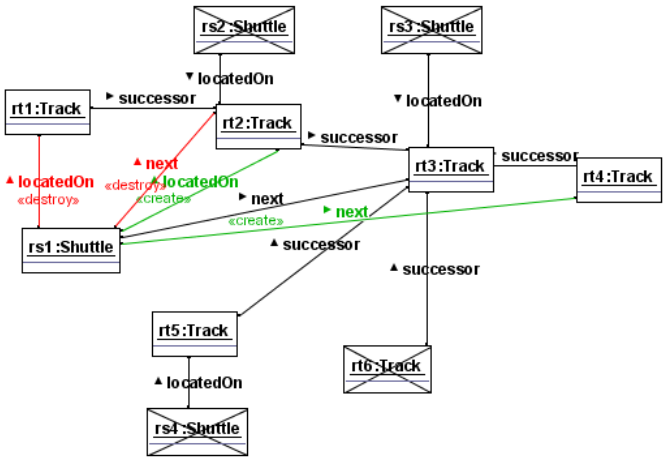



Abbildung A.4: Die Regel approachSwitch

Wurde die Regel approachSwitch angewendet, so erreicht das Shuttle den Track, der direkt vor der Weiche ist. Um die Weiche passieren zu können, wird die Regel moveSwitch (siehe Abbildung A.5) angewendet. In dieser Regel stellt das Track-Objekt rt2 die Weiche dar. Das Shuttle darf auf die Weiche fahren, wenn sich weder auf der Weiche noch auf dem Track rt1, der parall zu dem ist, auf dem sich das Shuttle befindet, ein anderes Shuttle fährt. Außerdem darf sich auch kein anderes Shuttle auf dem Track befinden, auf den das Shuttle als übernächstes fahren möchte (rt4).

Kann die Regel angewendet werden, so fährt das Shuttle auf die Weiche und aktualisiert dabei auch seine next-Links.

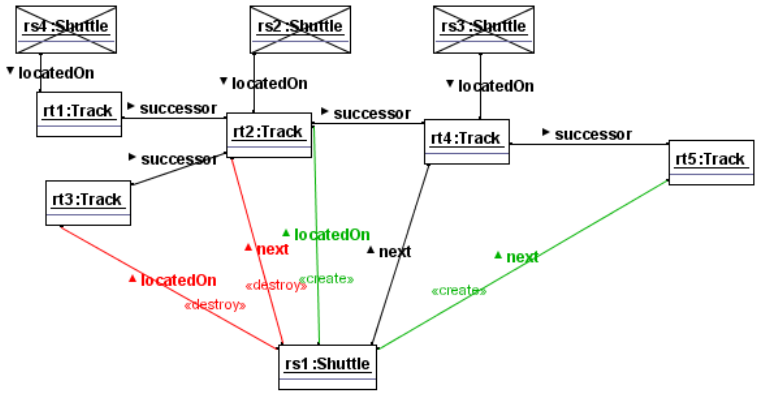



Abbildung A.5: Die Regel moveSwitch

### A.1.2 Die Regeln der ControlledMovement-Kultur

Von der ControlledMovement-Kultur werden alle Verhaltensregeln und Sicherheitseigenschaften übernommen, die in der übergeordneten Movement-Kultur definiert sind. Zu diesen fünf Regeln

fügt die ControlledMovement-Kultur zwei Instanzierungsregeln hinzu, die Regel createPublication sowie die Regel deletePublication. Diese beiden Regeln sind dafür verantwortlich eine Instanz einen Publication-Koordinationsmusters zwischen einem Shuttle und einer BaseStation zu erzeugen bzw. zu löschen. Die Instanzierung bzw. das Löschen einer solchen Instanz entspricht einer Anmeldung bzw. Abmeldung eines Shuttles bei der entsprechenden BaseStation (siehe dazu auch Abschnitt 2.3.2).

Bevor ein Shuttle in eine neue ControlledArea einfahren darf, muss es sich bei deren BaseStation anmelden. Ist das Shuttle angemeldet, so sendet die BaseStation Informationen über alle anderen bei ihr gemeldeten Shuttles. Auf diese Weise erfahren die Shuttle, welche anderen Shuttles sich in ihrer Nähe befinden und können entsprechend für eine sichere und kollisionsfreie Fahrt sorgen. Das Anmelden bei einer BaseStation erfolgt mittels der createPublication-Regel, die in Abbildung A.6 dargestellt ist.

Wenn ein Shuttle als nächstes auf einen Track fahren möchte, der von einer BaseStation überwacht wird, mit dem das Shuttle noch kein Publication-Koordinationsmuster ausführt, so ist das Shuttle noch nicht bei der BaseStation angemeldet. Dargestellt wird das durch ein negatives Publication-Objekt, mit Links zum Shuttle und zur BaseStation. In diesem Fall wird zwischen dem Shuttle und der BaseStation ein neues Publication-Objekt erzeugt und das Shuttle somit bei der BaseStation angemeldet.

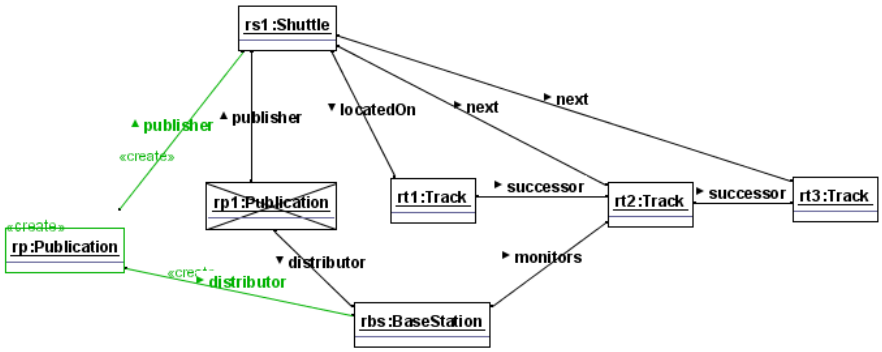



Abbildung A.6: Die Regel createPublication

Verlässt das Shuttle eine ControlledArea, so kann es sich bei der entsprechenden BaseStation wieder abmelden. Diese Abmeldung erfolgt mit der in Abbildung A.7 dargestellten Regel deletePublication.

Von einer BaseStation kann sich ein Shuttle abmelden, wenn diese weder den Track überwacht, auf dem sich das Shuttle zur Zeit befindet, noch einen Track, auf den das Shuttle als nächstes oder übernächstes fahren möchte. Dargestellt wird dies durch die negativen monitors-Links, die die BaseStation zu den Tracks hat, zu denen das Shuttle einen locatedOn-Link oder einen next-Link besitzt.

Wird die Anwendungsbedingung der Regel erfüll, so wird die entsprechende Instanz des Publication-Koordinationsmusters gelöscht.

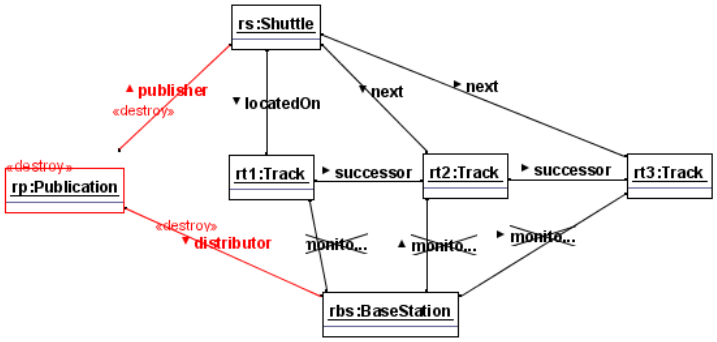



Abbildung A.7: Die Regel deletePublication

### A.1.3 Die Regeln der CoordinatedMovement-Kultur

Die CoordinatedMovement-Kultur übernimmt zum einen die fünf Regeln der Movement-Kultur und zum anderen die beiden Regeln der ControlledMovement-Kultur. Zu diesen sieben Regeln fügt sie eine weitere Verhaltensregeln und drei weitere Instanzierungregeln hinzu.

In der CoordinatedMovement-Kultur wird das DistanceCoordiantion-Koordinationsmuster eingeführt, damit zwei Shuttles einen Konvoi bilden können. Die Konvoibildung erfolgt indem der Abstand zwischen zwei Shuttles reduziert wird. Die Minimierung des Abstandes der beiden Shuttles wird dadurch modelliert, dass die beiden Shuttles nun auf benachbarten Tracks fahren dürfen.

Die Instanzierung des DistanceCoordination-Musters zwischen zwei Shuttles erfolgt durch die in Abbildung A.8 gezeigt createDC-Regel.

Diese Regel ist anwendbar, wenn zwei Shuttles hintereinanderherfahren und zwischen den Tracks, auf denen sich die beiden Shuttle befinden, nur ein weiterer Track ist. Die Regel darf nicht angewendet werden, wenn das vordere Shuttle das Koordinationsmuster schon ausführt und dabei die frontRole übernommen hat oder das hintere Shuttle das Muster ausführt und dabei bereits die rearRole übernommen hat. In diesem Fall existiert entweder bereits ein DistanceCoordiantion-Muster zwischen den beiden Shuttles oder zwischen den beiden Shuttles befindet sich noch ein drittes Shuttle.

Kann die Regel angewendet werden, so wird eine Instanz des DistanceCoordination-Musters erzeugt, zu dem das hintere Shuttle einen rear-Link besitzt und das vordere einen front-Link.

Die Regel createDC erzeugt eine Instanz des DistanceCoordination-Musters, wenn sich die Shuttles auf einer normalen Strecke befinden. Um auch die rechtzeitige Instanzierung des Musters an einer Weiche garantieren zu können, wird eine weitere Regel, die Regel createDCSwitch, die in Abbildung A.9 dargestellt ist, benötigt.

In dieser Regel stellt der Track rt3 die Weiche dar. Das Shuttle rs2 befindet sich näher an der Weiche und hat deshalb das Vorfahrtsrecht vor dem Shuttle rs1. Das bedeutet, dass rs2 im DistanceCoordination-Muster die Rolle frontRole übernimmt, während rs1 die rearRole übernimmt. Das entsprechende Muster wird jedoch nur instanziert, wenn $rs2$ noch in keiner Instanz des DistanceCoordination-Musters als front agiert und rs1 in noch keiner Instanz von DistanceCoordination als rear. Gilt diese Bedingung nicht, so bedeutet dies entweder, dass bereits eine

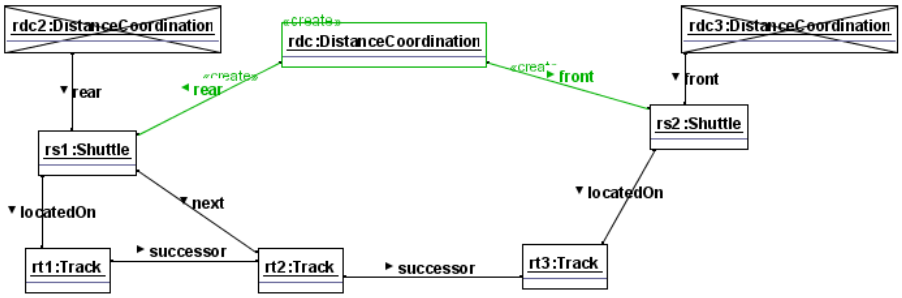



Abbildung A.8: Die Regel createDC

Instanz des Musters zwischen den beiden Shuttles existiert. Es kann aber auch sein, dass rs1 bereits im Konvoi fährt und ein anderes Shuttle vor sich hat, in diesem Fall muss sich das vordere Shuttle mit rs2 koordinieren. Eine dritte Möglichkeit besteht darin, dass rs2 schon in einem Konvoi fährt und ihm in diesem Konvoi ein anderes Shuttle folgt. In diesem Fall muss rs1 solange auf rt1 warten, bis sich das letzte Shuttle des Konvois auf dem Track rt4 befindet. Dann kann es mit diesem Shuttle das Muster ausführen und seine Fahrt fortsetzen.

Ist die Anwendungsbedingung der Regel erfüllt, so wird eine Instanz des DistanceCoordination-Musters erzeugt. Das Shuttle rs1 bekommt dann einen rear-Link zu dieser Musterinstanz und rs2 einen front-Link.

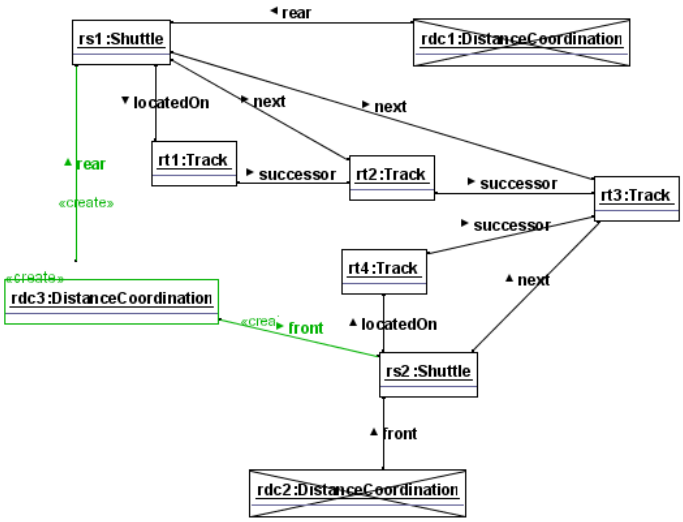



Abbildung A.9: Die Regel createDCSwitch

Nachdem mittels createDC oder createDCSwitch eine Instanz des DistanceCoordination-Musters erzeugt wurde, können die beiden Shuttles einen Konvoi bilden. Dazu wird die Regel moveDC (siehe Abbildung A.10) angewendet. Die Bildung des Konvois erfolgt, indem der Abstand zwischen den beiden Shuttles verringert wird, sodass die beiden Shuttles sich auf direkt aufeinander folgenden Tracks befinden. Ist zwischen den beiden Shuttles ein Track frei, so wird das hintere der beiden Shuttles durch moveDCauf diesen Track gesetzt.

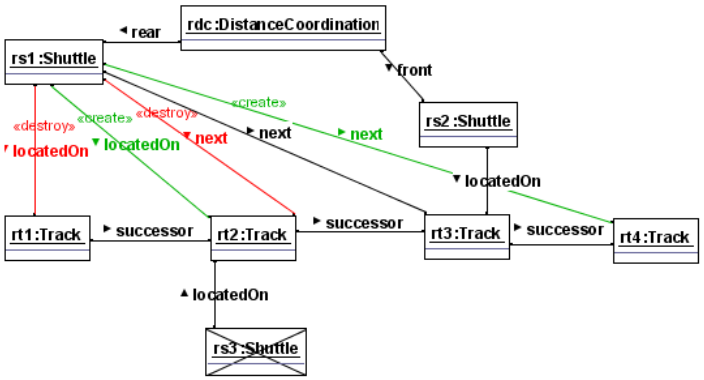



Abbildung A.10: Die Regel moveDC

Trennen sich die Wege der beiden Shuttles oder ist das vordere Shuttle schneller als das hintere, so wird der Konvoi aufgelöst. In diesem Fall muss auch die Instanz des DistanceCoordination-Musters gelöscht werden. Dies ist deshalb erforderlich, da ein Shuttle höchstens über zwei Instanzen des DistanceCoordination-Musters mit anderen Shuttles kommunizieren kann; Einmal übernimmt es dabei die rearRole und einmal die frontRole. Würde eine nicht mehr benötigte Musterinstanz nicht gelöscht, so würde dies dem Shuttle die Möglichkeit nehmen, einen Konvoi mit einem anderen Shuttle zu bilden.

Die Regel deleteDC in Abbildung A.11 ist für das Löschen einer Instanz des DistanceCoordination-Musters zuständig. Die Regel löscht eine solche Instanz, wenn sich das vordere Shuttle nicht mehr auf einem Track befindet, auf den das hintere Shuttle als nächstes oder übernächstes fahren möchte. Dies wird dargestellt durch die beiden negativen locatedOn-Links des vorderen Shuttles.

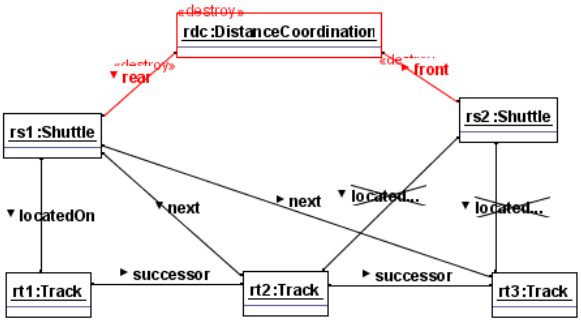



Abbildung A.11: Die Regel deleteDC

### Prioritäten und Größen der Regeln

Nachdem in den vorangegangenen Abschnitten alle Regeln des Shuttle-Beispiels erläutert wurden, werden in diesem Abschnitt ihre Prioritäten aufgelistet. Zudem wird für jede Regel deren Größe angegeben. Dabei entspricht die Größe einer Regel der Anzahl Knoten (Objekte in Story

Patterns) und Kanten (Links in Story Patterns), die bei der Regelanwendung nicht gelöscht werden (siehe dazu auch Abschnitt 5.4) bzw. negativ sind. Die Prioritäten und Größen der Regeln werden in Tabelle A.1 aufgeführt.

| Regel | Priorität | Größe | |
|---|---|---|---|
| | | Objekte | Links |
| member | 10 | 6 | 9 |
| noMember | 11 | 6 | 9 |
| moveNext | 0 | 8 | 9 |
| approachSwitch | 1 | 10 | 11 |
| moveSwitch | 2 | 9 | 10 |
| createPublication | 8 | 7 | 10 |
| deletePublication | 9 | 5 | 8 |
| createDC | 5 | 8 | 9 |
| moveDC | 3 | 8 | 10 |
| deleteDC | 7 | 8 | 10 |

Tabelle A.1: Charakteristiken der Regeln im Shuttle-System

## A.2 Verbotene Graphmuster

Im voran gegangenen Abschnitt wurden die Verhaltens- und Instazierungsregeln des Shuttle-Beispiels erläutert. Im folgenden Abschnitt A.2.1 sollen nun die Sicherheitseigenschaften erläutert werden, die diese Regeln einhalten müssen. Diese Sicherheitseigenschaften werden durch verbotenen Graphmuster bzw. verbotene Story Patterns modelliert. Neben den fünf Sicherheitseigenschaften werden in Abschnitt A.2.3 weitere acht verbotene Story Patterns gezeigt. Diese werden benötigt, da der Ansatz aus Kapitel 3 und dessen Umsetzung aus Abschnitt 5.3 die im Klassendiagramm gegebenen Kardinalitäten nicht berücksichtigt.

### A.2.1 Sicherheitseigenschaften

In den drei Kulturen wurden insgesamt fünf Sicherheitseigenschaften spezifiziert. Davon stellt das verbotene Graphmuster collision der Movement-Kultur einen Unfall dar. Die übrigen vier verbotenen Graphmuster stellen kritische Situationen dar.

#### Die verbotenen Graphmuster der Movement-Kultur

Die Movement-Kultur enthält die beiden verbotenen Graphmuster collision und notMember.

Das verbotenen Graphmuster collision aus Abbildung A.12 stellt einen Unfall dar, den es auf jeden Fall zu verhindern gilt. Da die Tracks so kurz sind, dass nur jeweils ein Shuttle darauf passt, ist ein Unfall eingetreten, wenn sich zwei Shuttles auf einem Track befinden. Das bedeutet, die

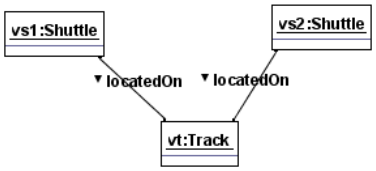



Abbildung A.12: Der Unfall collision

Sicherheitseigenschaft ist verletzt, sobald es zwei Shuttle-Objekte gibt, die einen locatedOn-Link zum selben Track-Objekt haben.

Fährt ein Shuttle in eine neue ControlledArea ein, so wird es durch die Anwendung Mitglied der entsprechenden Movement-Community. Dadurch werden dem Shuttle dann die Rollen und Regeln der Community zugewiesen. Eine kritische Situation ist eingetreten, wenn ein Shuttle in eine neue ControlledArea einfährt ohne in die entsprechende Community aufgenommen zu werden. Modelliert wird diese kritische Situation durch das verbotenen Story Pattern notMember in Abbildung A.13. Dieses verbotene Story Pattern besagt, dass eine kritische Situation eingetreten ist, wenn sich ein Shuttle auf einem Track befindet, dass von der BaseStation einer ControlledArea überwacht wird, jedoch zwischen dem Shuttle und der ControlledArea kein contains-Link existiert.

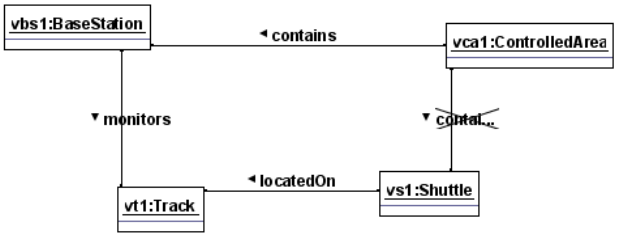



Abbildung A.13: Die kritische Situation notMember

### Die verbotenen Graphmuster der ControlledMovement-Kultur

In der ControlledMovement-Kultur gelten die Sicherheitseigenschaften der übergeordneten Movement-Kultur: collision und notMember. Darüber hinaus ist in der ControlledMovement-Kultur eine weitere Sicherheitseigenschaft in Form eines Story Patterns definiert.

Das verbotene Story Pattern noPublication, dargestellt in Abbildung A.14, stellt eine kritische Situation dar. Diese ist eingetreten, wenn sich ein Shuttle auf einem Track befindet ohne mit der BaseStation, die diesen Track überwacht, das Publication-Muster auszuführen. Dies ist deshalb kritisch, da das Shuttle dann keine Informationen über die Positionen der anderen Shuttles der ControlledArea, zu der die BaseStation gehört, erhält und auch die anderen Shuttles nichts von dem neuen Shuttle erfahren.

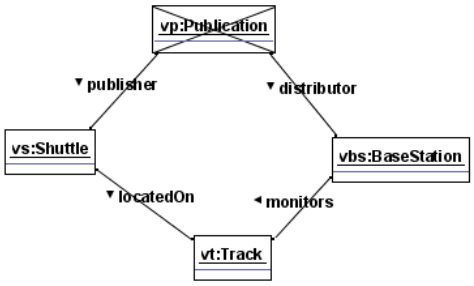



Abbildung A.14: Die kritische Situation noPublication

### A.2.2 Die verbotenen Graphmuster der CoordinatedMovement-Kultur

Die Regeln der CoordinatedMovement-Kultur müssen alle Sicherheitseigenschaften erfüllen, die in den übergeordneten Kulturen, Movement und ControlledMovement, spezifiziert sind. Zusätzlich zu diesen drei Sicherheitseigenschaften gelten in der CoordinatedMovement-Kultur auch noch die beiden Eigenschaften impendingCollision und impendingCollisionSwitch. Beide Sicherheitseigenschaften sind in Form von verbotenen Story Patterns spezifiziert.

Das verbotene Story Pattern impendingCollision beschreibt eine kritische Situation. Diese ist eingetreten, wenn zwei Shuttles hintereinanderherfahren, das vordere Shuttle sich auf dem Track befindet, auf den das hintere als nächtes fahren möchte, die beiden Shuttles das DistanceCoordination-Muster jedoch nicht miteinander ausführen. DistanceCoordination-Muster miteinander auszuführen. In diesem Fall droht eine Kollision, da die beiden Shuttles mit sehr geringem Abstand hintereinanderherfahren, das vordere Shuttle jedoch keine Rücksicht auf das hintere nimmt. Würde das vordere Shuttle plötzlich stark bremsen, so bliebe dem hinteren Shuttle keine Zeit, um zu reagieren.

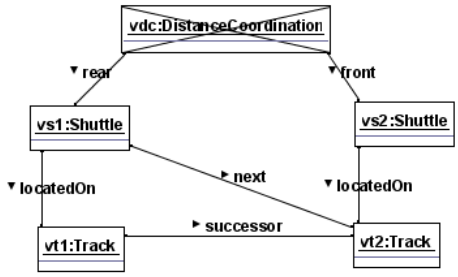



Abbildung A.15: Die kritische Situation impendingCollision

Eine kritische Situation kann auch an einer Weiche auftreten. Dies ist genau dann der Fall, wenn sich zwei Shuttles auf den parallelen Tracks direkt vor einer Weiche befinden, jedoch nicht das DistanceCoordination-Muster miteinander ausführen. Einerseits würde in diesem Fall die Vorwärtsbewegung eines Shuttles dazu führen, dass sich die beiden Shuttles auf zwei aufeinander folgenden Tracks befinden, ohne jedoch das DistanceCoordination-Muster auszuführen, dies würde dann der kritischen Situation impendingCollision entsprechen. Zum anderen wäre in diesem Fall aber auch nicht geregelt, welches Shuttle die Weiche zuerst passieren darf Somit

könnten beide Shuttles weiterfahren und würden sich dann auf demselben Track befinden, was eine Kollsion der beiden Shuttles bedeutet. Diese kritische Situation, impendingCollisionSwitch, wird in Abbildung A.16 dargestellt.

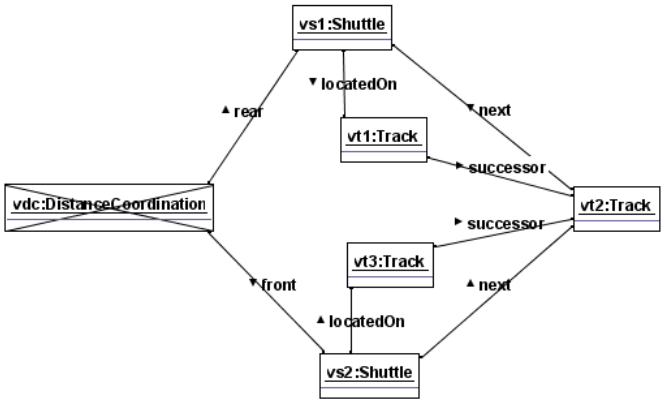



Abbildung A.16: Die kritische Situation impendingCollisionSwitch

## A.2.3 Kardinalitäten

Da der Ansatz aus Kapitel 3 und dessen Umsetzung aus Abschnitt 5.3 die im Klassendiagramm gegebenen Kardinalitäten nicht berücksichtigt, müssen diese durch zusätzlich verbotene Graphmuster spezifiziert werden. Wird dies nicht gemacht, so erhält man bei der Verifikation Gegenbeispiele, die das Klassendiagramm verletzen. Die in diesem Beispiel verwendeten verbotenen Graphmuster sollen nachfolgend kurz vorgestellt werden. Das Klassendiagramm des Shuttle-Beispiels ist noch einmal in Abbildung A.17 dargestellt.

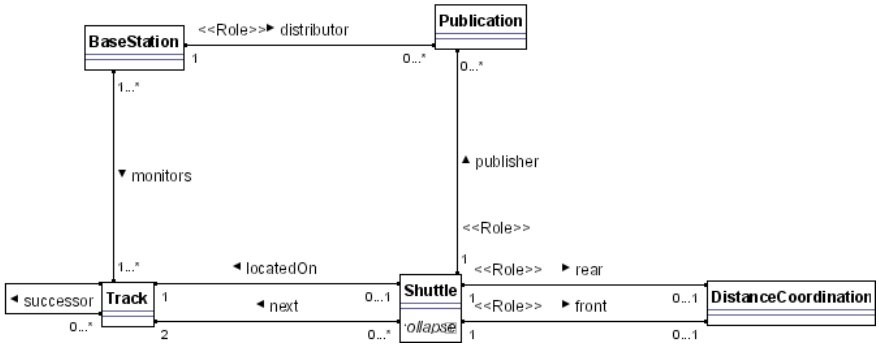



Abbildung A.17: Die erweiterte Ontologie

Die Kardinalitäten des Klassendiagramms fordern, dass ein Shuttle sich genau auf einem Track befindet. Deshalb verbietet das Story Pattern singleLocatedOn in Abbildung A.18, dass ein Shuttle meherere locatedOn-Links zum selben Track besitzt. Dementsprechend verbietet das

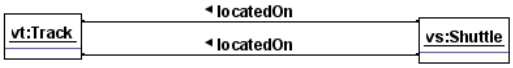



Abbildung A.18: singleLocatedOn

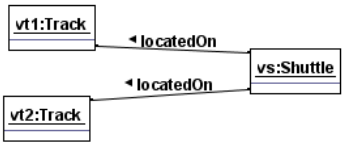



Abbildung A.19: unumbigousLocatedOn

Story Pattern unambigousLocatedOn, Abbildung A.19, dass ein Shuttle locatedOn-Links zu unterschiedlichen Tracks hat.

Während ein locatedOn-Link angibt, auf welchem Track sich ein Shuttle befindet, werden die next-Links dazu verwendet, um anzugeben auf welchen Track das Shuttle als nächstes fahren möchte. Deshalb ist es nicht erlaubt, dass ein next-Link auf den selben Track zeigt wie der locatedOn-Link des Shuttles. Dies wird durch das verbotene Story Pattern locatedOnNext aus Abbildung A.20 dargestellt.

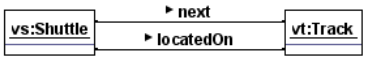



Abbildung A.20: locatedOnNext

Um eine sichere Fahrt eines Shuttles garantieren zu können, muss es immer zwei next-Links besitzen, die anzeigen, auf welche Tracks das Shuttle als nächstes und übernächstes fahren möchte. Die beiden next-Links dürfen jedoch nicht zum selben Track-Objekt zeigen. Diese Eigenschaft wird durch das verbotene Story Pattern singleNext in Abbildung A.21 spezifiziert.

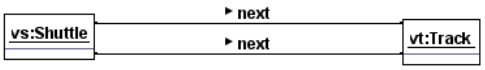



Abbildung A.21: singleNext

Das Klassendiagramm aus Abbildung A.17 fordert, dass ein Shuttle genau zwei next-Links hat. Deshalb verbietet das Story Pattern tooManyNext aus Abbildung A.22, dass ein Shuttle mehr als zwei next-Links hat.

Während die bisher in diesem Abschnitt betrachteten verbotenen Story Patterns Aussagen über ein Shuttle und seine Links gemacht haben, schränken die folgenden verbotenen Story Patterns das Schienensystem ein.

Die verbotenen Story Pattern singleSuccessor, Abbildung A.23, und twoWayTracks, Abbildung A.24, beschreiben, dass zwischen zwei Tracks nur genau ein successor-Link sein darf. Dabei verbietet singleSuccessor, dass sich zwischen zwei Tracks mehrere successor-Links in

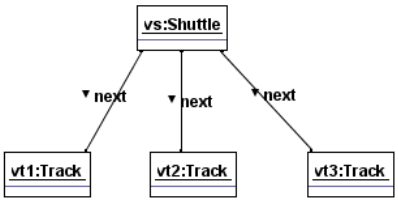



Abbildung A.22: tooManyNext

dieselbe Richtung existieren. Das Story Pattern twoWayTracks verbietet, dass zwischen zwei Tracks jeweils ein successor-Link in jede Richtung existiert.

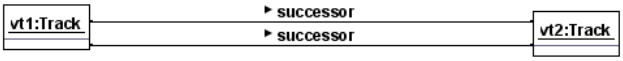



Abbildung A.23: singleSuccessor

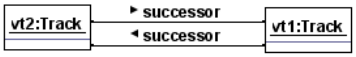



Abbildung A.24: twoWayTracks

Im Klassendiagramm in Abbildung A.17 ist angegeben, dass ein Track höchstens zwei Vorgänger oder Nachfolger haben darf. Das verbotene Story Pattern tooManyPredecesors in Abbildung A.25 zeigt, dass ein Track mehr als zwei Vorgänger hat. Dem gegenüber verbietet das Story Pattern tooManySuccessors in Abbildung A.26, dass ein Track mehr als zwei Nachfolger hat.

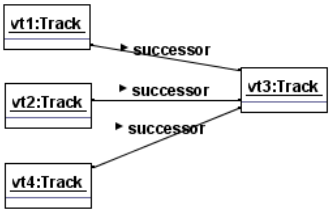



Abbildung A.25: tooManyPredecesors

## A.2.4 Größen der verbotenen Graphmuster

Nachdem in den voran gegangenen Abschnitten die verbotenen Graphmuster erläutert wurden, werden in der Tabelle A.2 die Größen der verbotenen Graphmuster aufgelistet.

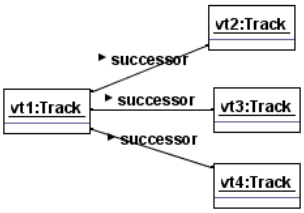



Abbildung A.26: tooManySuccessors

| Verbotene Graphmuster | Größe | |
|---|---|---|
| | Objekte | Links |
| collision | 3 | 2 |
| notMember | 4 | 4 |
| noPublication | 4 | 4 |
| impendingCollision | 5 | 6 |
| impendingCollisionSwitch | 6 | 8 |
| singleLocatedOn | 2 | 2 |
| unumbigousLocatedOn | 3 | 2 |
| locatedOnNext | 2 | 2 |
| singleNext | 2 | 2 |
| tooManyNext | 4 | 3 |
| singleSuccessor | 2 | 2 |
| twoWayTracks | 2 | 2 |
| tooManyPredesesors | 4 | 3 |
| tooManySuccessors | 4 | 3 |

Tabelle A.2: Charakteristiken der verbotenen Graphmuster im Shuttle-Beispiel

# Anhang B

# Theoretische Ergebnisse und Beweisskizzen

Für die in Kapitel 3 aufgeführten Lemmata und Theoreme sollen in diesen Kapitel deren Beweise skizziert werden.

## B.1 Graphmuster

**Lemma 1.** *(Implikation von Teilgraphmustern)*
*Für zwei Graphmuster* $\mathsf{p}$ *und* $\mathsf{p}'$ *gilt:*

$$(\mathsf{p} \subseteq \mathsf{p}') \Rightarrow (\forall \mathsf{G} \in \mathcal{G}[\mathsf{p}'] : \mathsf{G} \models \mathsf{p}) \quad \textit{und} \tag{B.1}$$

$$(\exists \mathsf{G} \in \mathcal{G}[\mathsf{p}'] : \mathsf{G} \not\models \mathsf{p}) \Rightarrow (\mathsf{p} \not\subseteq \mathsf{p}'). \tag{B.2}$$

***Beweis:*** *Implikation B.1: Um zu zeigen, dass die Implikation stimmt, muss zum einen gezeigt werden, dass es einen Isomorphismus gibt, der* $\mathsf{P}$ *auf* $\mathsf{G}$ *abbildet und zum anderen, dass es keinen Isomorphismus gibt, er ein* $\hat{\mathsf{P}} \in \hat{\mathcal{P}}$ *auf* $\mathsf{G}$ *abbildet.*

*Per Definition gilt* $(\mathsf{p} \subseteq \mathsf{p}') \Rightarrow \exists \mathsf{iso} \in \mathcal{ISO} : \mathsf{P} \preceq_{\mathsf{iso}} \mathsf{P}'$ *und* $\forall \mathsf{G} \in \mathcal{G}[\mathsf{p}'], \exists \mathsf{iso}' \in \mathcal{ISO} : \mathsf{G} \Rightarrow \mathsf{P}' \preceq_{\mathsf{iso}'} \mathsf{G}$. *Daraus folgt* $\exists \mathsf{iso}'' \in \mathcal{ISO} : \mathsf{iso}'' := \mathsf{iso}' \circ \mathsf{iso}$, *für diesen Isomorphismus* $\mathsf{iso}''$ *gilt* $\mathsf{P} \preceq_{\mathsf{iso}''} \mathsf{G}$. *Womit die erste Forderung erfüllt ist.*

*Nach Definition 20 gilt* $\forall \hat{\mathsf{P}} \in \hat{\mathcal{P}}, \exists \hat{\mathsf{P}}' \in \hat{\mathcal{P}}, \mathsf{iso}^{\mathsf{iv}} \in \mathcal{ISO} : \mathsf{iso} \mid_{\mathsf{P}} = \mathsf{iso} \wedge (\hat{\mathsf{P}}' \setminus (\mathsf{P}' \setminus \mathsf{iso}(\mathsf{P}))) \preceq_{\mathsf{iso}^{\mathsf{iv}}} \hat{\mathsf{P}}$. *Aus* $\exists \mathsf{iso}' \in \mathcal{ISO} : \mathsf{P}' \preceq_{\mathsf{iso}'} \mathsf{G}$ *und* $\forall \hat{\mathsf{P}}' \in \hat{\mathcal{P}}', \nexists \mathsf{iso}''' \in \mathcal{ISO} : \mathsf{iso}''' \mid_{\mathsf{P}'} = \mathsf{iso}' \wedge \hat{\mathsf{P}}' \preceq_{\mathsf{iso}'''} \mathsf{G}$ *folgt, dass* $\forall \hat{\mathsf{P}}' \in \hat{\mathcal{P}}', \nexists \mathsf{iso}^{\mathsf{v}} \in \mathcal{ISO} : (\hat{\mathsf{P}}' \setminus (\mathsf{P}' \setminus \mathsf{iso}(\mathsf{P}))) \preceq_{\mathsf{iso}^{\mathsf{v}}} \mathsf{G}$ *und damit gilt auch* $\forall \hat{\mathsf{P}} \in \hat{\mathcal{P}}, \nexists \mathsf{iso}^{\mathsf{vi}} \in \mathcal{ISO} : \mathsf{iso}^{\mathsf{vi}} \mid_{\mathsf{P}} = \mathsf{iso}'' \wedge \hat{\mathsf{P}} \preceq_{\mathsf{iso}^{\mathsf{vi}}} \mathsf{G}$. *Damit ist auch die zweite Forderung erfüllt und es gilt* $\mathsf{G} \models \mathsf{p}$.

*Implikation B.2: folgt direkt aus Implikation B.1.* □

## B.2 Das Systemmodell

**Lemma 2.** *Für zwei Graphen* $G_1, G_2 \in \mathcal{G}$*, die typkonform zu* $G_\Omega$ *sind, gilt, dass auch ihre Vereinigung* $G_1 \cup G_2$ *typkonform zu* $G_\Omega$ *ist.*
***Beweis:*** *Beide Graphen sind typkonform zu* $G_\Omega$*, das bedeutet, dass ihre Beschriftungsfunktionen das gleiche Alphabet* $G_\Omega$ *verwenden. Somit kann eine Vereinigung der beiden Graphen gebildet werden. Da bei der Vereinigung keine neuen Elemente erzeugt werden, sondern ein neuer Graph aus den Elementen von* $G_1$ *und* $G_2$ *gebildet wird, ist auch die Vereinigung typkonform zu* $G_\Omega$*.* □

**Lemma 3.** *Für zwei Graphen* $G_1, G_2 \in \mathcal{G}$*, die typkonform zu* $G_\Omega$ *sind, gilt, dass auch der Schnitt* $G_1 \cap G_2$ *und die Subtraktion* $G_1 \setminus G_2$ *typkonform sind.*
***Beweis:*** *Beide Graphen sind typkonform zu* $G_\Omega$*, somit ist das Bilden des Schnitts und der Subtraktion möglich. Beim Schnitt und der Subtraktion werden keine neuen Elemente erzeugt, sondern Elemente aus einem Graphen entfernt, der typkonform zu* $G_\Omega$ *ist, somit ist auch der Schnitt und die Subtraktion typkonform zu* $G_\Omega$*.* □

**Lemma 4.** *Für zwei Graphen* $G_1, G_2 \in \mathcal{G}$ *und einen totalen Graphhomomorphismus* $m \in \mathcal{M} : G_1 \mapsto G_2$ *gilt, dass* $G_1$ *genau dann typkonform zu* $G_\Omega$ *ist, wenn* $G_2$ *typkonform zu* $G_\Omega$ *ist.*
***Beweis:*** *Die Definition 9 verlangt für einen totalen Graphhomomorphismus, dass er die Knoten- und Kantenbeschriftungen erhält. Existiert ein* $m \in \mathcal{M} : G_1 \mapsto G_2$ *so gilt, dass entweder beide Graphen* $G_1$ *und* $G_2$ *typkonform zum Typgraphen* $G_\Omega$ *sind oder beide Graphen nicht typkonform sind.* □

**Theorem 1.** *Für jeden typkonformen Graphen* $G_1 \in \mathcal{G}$ *und jede typkonforme Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *gilt: falls ein* $G_2 \in \mathcal{G}$ *existiert, mit* $G_1 \models\!\Rightarrow_r G_2$*, so ist auch* $G_2$ *typkonform.*
***Beweis:*** *Die Regel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *ist auf den Graphen* $G_1$ *anwendbar, wenn* $\exists iso \in \mathcal{ISO} : L \preceq_{iso} G$ *und* $\forall \hat{L} \in \hat{\mathcal{L}}, \nexists iso' : iso' \mid_L = iso \wedge \hat{P} \preceq_{iso'} G$ *gilt. Nach Lemma 4 gilt, dass Homomorphismen die Typkonformität erhalten. Kann die Regel unter dem Auftreten* $o$ *mit* $o : o \mid_L = iso$ *angewendet werden, so werden aus* $G_1$ *durch Subtraktion alle Elemente entfernt auf die die Elemente aus* $L \setminus R$ *abgebildet werden können, d.h.* $G_1' := G_1 \setminus o(L \setminus R)$*. Da die Subtraktion nach Lemma 3 die Typkonformität erhält, ist der resultierende Graph* $G_1'$ *typkonform zu* $G_\Omega$*.* $G_1'$ *wird nun um alle Elemente aus* $R \setminus L$ *erweitert, d.h.* $G_2 := G_1' \cup o(R \setminus L)$*. Da die Vereinigung nach Lemma 2 die Typkonformität erhält, ist* $G_2$ *typkonform zu* $G_\Omega$*.* □

**Lemma 5.** *Wird eine typkonforme Regel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *wie in Definition 27 erweitert und die erweiterte Regel auf einen typkonformen Graphen* $G$ *im DPO*$^{iso}$ *angewendet, so erfüllt die Regelanwendung die Lose-Kanten-Bedingung.*
***Beweis:*** *Sei* $[L_{re}, \hat{\mathcal{L}}_{re}] \rightarrow_{re} R_{re}$ *eine erweiterte Regel,* $G_1 \in \mathcal{G}$ *ein Anwendungsgraph und* $G_2 \in \mathcal{G}$ *ein Ergebnisgraph, wobei* $G_1$ *und* $G_2$ *typkonform zu* $G_\Omega$ *sind und es gilt* $G_1 \models\!\Rightarrow_{re} G_2$*.*

*Angenommen, die Anwendung von* $r$ *unter Auftreten* $o$ *auf den Graphen* $G_1$ *erzeugt eine lose Kante, d.h.* $\exists e \in E_{G_1}, \nexists e' \in E_{L_{re}} : o(e') = e \wedge ((src(e') \in (N_{L_{re}} \setminus N_{R_{re}})) \vee (tgt(e') \in (N_{L_{re}} \setminus N_{R_{re}})))$*. Das bedeutet, es müssen zwei Fälle betrachtet werden. Erstens der Fall, dass bei einer Kante der Startknoten fehlt und zum zweiten der Fall, dass bei einer Kante der Zielknoten fehlt.*

*Fall 1: es gibt eine Kante, bei der der Startknoten fehlt, d.h.* $\exists \mathsf{n} \in (\mathsf{N_L} \setminus \mathsf{N_R}) : \mathsf{o(n)} = \mathsf{src(e)}$. *Da* $\mathsf{r_{re}}$ *anwendbar ist bedeutet das, dass es keinen Graphen* $\hat{\mathsf{L}}_{\mathsf{re}} \in \hat{\mathcal{L}}_{\mathsf{re}}$ *gibt, der die Anwendung verbietet, d.h.* $\nexists \hat{\mathsf{L}}_{\mathsf{re}} \in \hat{\mathcal{L}}_{\mathsf{re}} : \mathsf{N}_{\hat{\mathsf{L}}_{\mathsf{re}}} := \mathsf{N}_{\mathsf{L}_{\mathsf{re}}} \cup \{\mathsf{n}\}$, $\mathsf{E}_{\hat{\mathsf{L}}_{\mathsf{re}}} := \mathsf{E}_{\mathsf{L}_{\mathsf{re}}} \cup \{\mathsf{e}\}$, $\mathsf{src}_{\hat{\mathsf{L}}_{\mathsf{re}}} := \begin{cases} src_{L_{re}}(x) & : \; x \in E_{L_{re}} \\ n & : \; x == e \end{cases}$, $\mathsf{tgt}_{\hat{\mathsf{L}}_{\mathsf{re}}} := \begin{cases} tgt_{L_{re}}(x) & : \; x \in E_{L_{re}} \\ n & : \; x == e \end{cases}$, $\mathsf{l}_{\mathsf{N}_{\hat{\mathsf{L}}_{\mathsf{re}}}}(\mathsf{x}) := \begin{cases} l_{N_{L_{re}}}(x) & : \; x \in N_{L_{re}} \\ t_n & : \; x == n \end{cases}$ *und* $\mathsf{l}_{\mathsf{E}_{\hat{\mathsf{L}}_{\mathsf{re}}}}(\mathsf{x}) := \begin{cases} l_{E_{L_{re}}}(x) & : \; x \in E_{L_{re}} \\ t_e & : \; x == e \end{cases}$. *Da für jede korrekt typisierte Kante, die inzident zu einem zu löschenden Knoten ist, ein Graph zur negativen Anwendungsbedingung hinzugefügt wird, gilt auch, dass für jede korrekt typisierte Kante* $\mathsf{e''}$ *mit* $\mathsf{o(n)} = \mathsf{src(e'')}$ *ein Graph in die negative Anwendungsbedingung aufgenommen wird. Da es jedoch keinen Graphen in der negativen Anwendungsbedingung gibt, der die Regelanwendung verbietet, sodass der Knoten* $\mathsf{o(n)}$ *gelöscht wird und aus der Kanten* $\mathsf{o(e)}$ *eine lose Kante wird, folgt daraus, dass* $\mathsf{e}$ *nicht korrekt typisiert ist. Diese ist jedoch ein Widerspruch zur Annahme, da* $\mathsf{G_1}$ *typkonform zu* $\mathsf{G_\Omega}$ *ist.*

*Fall 2: erfolgt analog zu Fall 1.* □

# B.3 Erweiterte Graphtransformationen

**Theorem 2.** *(Rückwärtsanwendung einer Graphtransformationsregel im DPO*$^{\mathsf{iso}}$*)*

*Für jede Graphtransformationsregel* $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$, *ihr Inverses* $[\mathsf{L}^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{\mathsf{r}^{-1}} \mathsf{R}^{-1}$ *und ein Auftreten* $\mathsf{o}$ *von* $\mathsf{r}$ *in* $\mathsf{G_1}$ *gilt im DPO*$^{\mathsf{iso}}$*:*

$\mathsf{G_1} \models\!\Rightarrow_{(\mathsf{r},\mathsf{o})} \mathsf{G_2} \Leftrightarrow \mathsf{G_2} \models\!\Rightarrow_{(\mathsf{r}^{-1},\mathsf{o}^{-1})} \mathsf{G_1}$.

***Beweis:*** *„* $\Rightarrow$ *“ : Da* $\mathsf{G_2}$ *durch die Anwendung von* $\mathsf{r}$ *unter dem Auftreten* $\mathsf{o}$ *auf den Graphen* $\mathsf{G_1}$ *entstanden ist, muss es einen Isomorphismus* $\mathsf{iso} \in \mathcal{ISO}$ *geben, für den gilt* $\mathsf{iso} = \mathsf{o}^{-1} \mid_{\mathsf{L}^{-1}} \wedge \mathsf{L}^{-1} \preceq_{\mathsf{iso}} \mathsf{G_2}$. *Somit ist* $\mathsf{r}^{-1}$ *anwendbar wenn gilt* $\forall \hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}^{-1}, \nexists \mathsf{iso'} \in \mathcal{ISO} : \mathsf{iso'} \mid_{\mathsf{L}^{-1}} = \mathsf{iso} \wedge \hat{\mathsf{L}} \preceq_{\mathsf{iso'}} \mathsf{G_2}$.

*Die Menge* $\hat{\mathcal{L}}^{-1}$ *kann in zwei Teilmengen unterteilt werden: Die erste Menge* $\hat{\mathcal{L}}_1^{-1}$ *enthält alle Graphen* $\hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}^{-1}$, *die durch die Anwendung von* extendNAC *erzeugt wurden, um eine Anwendung von* $\mathsf{r}^{-1}$ *zu verhindern, wenn andernfalls lose Kanten entstehen. Die zweite Menge* $\hat{\mathcal{L}}_2^{-1} \in \hat{\mathcal{L}}^{-1}$ *enthält alle Graphen* $\hat{\mathsf{L}}^{-1}$, *die durch die Konvertierung der negativen Anwendungsbedingung* $\hat{\mathcal{L}}$ *entstanden sind.*

*Für die Graphen der Menge* $\hat{\mathcal{L}}_1^{-1}$ *gilt, dass ihre Knoten entweder auch in* $\mathsf{L}^{-1}$ *enthalten sind oder adjazent zu Knoten sind, die bei der Anwendung von* $\mathsf{r}^{-1}$ *gelöscht und damit durch die Anwendung von* $\mathsf{r}$ *erzeugt werden. Da Knoten, die durch die Anwendung von* $\mathsf{r}$ *erzeugt wurden, nur adjazent zu Knoten sein können, die zu* $\mathsf{R}$ *und somit auch zu* $\mathsf{L}^{-1}$ *gehören, gibt es keinen Isomorphismus* $\mathsf{iso'} \in \mathcal{ISO}$ *der einen Graphen aus* $\hat{\mathcal{L}}^{-1}$ *auf* $\mathsf{G_2}$ *abbildet und für den gilt* $\mathsf{iso'} \mid_{\mathsf{L}^{-1}} = \mathsf{iso}$. *Somit wird die Regelanwendung durch die Teilmenge* $\hat{\mathcal{L}}_1^{-1}$ *nicht verhindert.*

*Für die Menge* $\hat{\mathcal{L}}_2^{-1}$ *gilt, dass ihre Graphen durch die Konvertierung der negativen Anwendungsbedingung* $\hat{\mathcal{L}}$ *von* $\mathsf{r}$ *entstanden sind, d.h.* $\forall \hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}_2^{-1}, \exists \hat{\mathsf{L}} \in \hat{\mathcal{L}} : \hat{\mathsf{L}} \models\!\Rightarrow_{\mathsf{L} \rightarrow_{(\mathsf{r'},\mathsf{o})} \mathsf{R}} \hat{\mathsf{L}}^{-1}$. *Da die Graphen aus* $\hat{\mathcal{L}}_2^{-1}$ *durch die Anwendung der gleichen Regel entstanden sind, wie der Graph* $\mathsf{G_2}$, *gilt, dass es nur dann einen Isomorphismus* $\mathsf{iso_1} \in \mathcal{ISO}$ *geben kann, der einen Graphen* $\hat{\mathsf{L}}^{-1} \in \hat{\mathcal{L}}_2^{-1}$ *auf* $\mathsf{G_2}$ *abbildet, wenn es einen Isomorphismus* $\mathsf{iso_2} \in \mathcal{ISO}$ *und einen Graphen* $\hat{\mathsf{L}} \in \hat{\mathcal{L}}$

*gibt, sodass gilt* $\hat{L} \preceq_{iso_2} G_1$ *und* $\hat{L} \models\!\Rightarrow_{L \to_{(r',o)} R} \hat{L}^{-1}$. *Dies ist jedoch ein Widerspruch, da* r *auf* $G_1$ *anwendbar ist. Somit wird die Anwendung von* $r^{-1}$ *auf* $G_2$ *unter dem Auftreten* $o^{-1}$ *auch durch die Menge* $\hat{\mathcal{L}}_2^{-1}$ *nicht verhindert.*

*„* $\Leftarrow$ *“ : erfolgt analog zu „* $\Rightarrow$ *“.* $\square$

**Theorem 3.** *Für eine Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \to_r R$ *und zwei Graphmuster* $p := [P, hat\mathcal{P}], p' := [P', \hat{\mathcal{P}}']$ *gilt im DPO*$^{iso}$*:*

$$(p \not\models\!\Rightarrow_r p')$$
$$\Leftrightarrow$$
$$(\forall G_1, G_2 \in \mathcal{G}, iso \in \mathcal{ISO} : P \cup P' \to G_1 \cup G_2 : (G_1 \models p \wedge G_1 \models\!\Rightarrow_{r,iso|_{L \cup R}} G_2) \Rightarrow (G_2 \models p')).$$

***Beweis:*** *„* $\Rightarrow$ *“ : Annahme* $(G_1 \models p \wedge G \models\!\Rightarrow_{(r,iso|_{R \cup L})} G_2) \Rightarrow (G_2 \models p')$ *gilt nicht. Dann muss einer der beiden folgenden Fälle gelten: (1)* $\nexists iso' \in \mathcal{ISO} : iso' \mid_{P'} = iso \wedge P' \preceq_{iso'} G_2$ *oder es gilt (2)* $\exists \hat{P} \in \hat{\mathcal{P}}, iso' \in \mathcal{ISO} : iso' \mid_{P'} = iso \mid_{P'} \wedge \hat{P}' \preceq_{iso'} G_2$.

*Fall (1): falls kein solcher Isomorphismus* iso′ *existiert, muss gelten* $\exists n \in N_{P'}, e \in E_{P'} : iso'(n) = undefiniert \vee iso'(e) = undefiniert$. *Da* iso *alle Elemente aus* P *auf* $G_1$ *abbildet und* $p \not\models\!\Rightarrow_r p'$ *gilt muss* n *(oder* e*) entweder (a) nur in* P′ *aber nicht in* P *enthalten sein und somit durch die Anwendung von* r *erzeugt werden oder (b)* n *(oder* e*) ist sowohl Teil von* P *als auch von* P′ *wird aber durch die Anwendung von* r *gelöscht, d.h.* $n \in (N_L \setminus N_R)$ *(bzw.* $e \in (E_L \setminus E_R)$*).*

*(a) da* $p \not\models\!\Rightarrow_{r,o} p'$ *gilt, muss* r *den Knoten* n *(bzw. die Kante*e*) erzeugen, d.h.* $n \in (N_R \setminus N_L)$ *(bzw.* $e \in (E_R \setminus E_L)$*). Dann wird* n *(bzw.* e*) jedoch auch bei der Anwendung von* r *unter dem Auftreten* o *auf* $G_1$ *erzeugt und ist somit auch in* $G_2$ *enthalten.*

*(b)* n *(bzw.* e*) gehört sowohl zu* P *als auch zu* P′*, dann muss die Anwendung von* r *unter* o *das Element löschen, in diesem Fall wird das Element aber auch gelöscht, wenn* r *unter* o *auf* p *angewendet wird und somit würde nicht* $p \not\models\!\Rightarrow_{(r,o)} p'$ *gelten.*

*Fall 2: es gilt* $\exists \hat{P}' \in \hat{\mathcal{P}}', iso' \in \mathcal{ISO} : iso' \mid_{P'} = iso \mid_{P'} \wedge \hat{P}' \preceq_{iso'} G_2$. *Da* $p \not\models\!\Rightarrow_r p'$ *und* $G_1 \models p$ *gilt, muss* $\hat{P}'$ *durch die Anwendung von* r *erzeugt worden sein. Die Menge* $\hat{\mathcal{P}}'$ *setzt sich aus zwei Teilmengen zusammen. (a) Die erste Menge,* $\hat{\mathcal{P}}'_1$*, enthält alle* $\hat{P}'$*, die durch* extendNAC *erzeugt werden. (b) Die zweite Menge,* $\hat{\mathcal{P}}'_2$*, enthält alle* $\hat{P}'$*, für die gilt* $\exists \hat{P} \in \hat{\mathcal{P}} : \hat{P} \models\!\Rightarrow_{L \to_{r'} R} \hat{P}'$.

*(a)* extendNAC *prüft für jeden durch* r *neu erzeugten Knoten im Typgraphen, Knoten von welchem Typ adjazent zum neuen Knoten sein können. Für jeden dieser Knoten wird ein Graph in die negative Anwendungsbedingung eingefügt. Somit beschreiben die* $\hat{P}' \in \hat{\mathcal{P}}'_1$*, dass ein neu erzeugter Knoten nur adjazent zu Knoten aus* R *sein kann. Daraus folgt, dass* $\forall \hat{P} \in \hat{\mathcal{P}}'_1, \nexists iso' \in \mathcal{ISO} : \hat{P} \preceq_{iso'} G_2$.

*(b)* $\forall \hat{P} \in \hat{\mathcal{P}}'_2, \exists \hat{P} \in \hat{\mathcal{P}} : \hat{P} \models\!\Rightarrow_{L \to_{r'} R} \hat{P}'$*, daraus folgt, dass* $\hat{P}'$ *durch die Anwendung von* r *unter dem Auftreten* o *nur entstehen kann, wenn* $\hat{P}$ *durch einen Isomorphismus aus* $G_1$ *abgebildet werden kann, dann würde aber* $G_1 \not\models p$ *gelten, was eine Widerspruch zur Voraussetzung ist.*

*Aus Fall 1 und Fall 2 folgt „* $\Rightarrow$ *“.*

*„* $\Leftarrow$ *“ : zuerst wird gezeigt, dass* $P \models\!\Rightarrow_{(r,o)} P'$ *gilt. Die rechte Seite der Implikation gilt für alle Graphen* $G_1$ *und* $G_2$*, für die gilt* $G \models p \wedge G_1 \models\!\Rightarrow_{(r,o)} G_2$*. Somit können* $G_1$ *und* $G_2$ *beliebig*

*gewählt werden, also auch* $G_1 = P$ *und* $G_2 = P'$*. Daraus folgt dann* $P = G_1 \models\!\Rightarrow_r G_2 = P'$*, dies entspricht* $P \models\!\Rightarrow_r P'$*.*

*Angenommen der rechte Teil der Implikation gilt, der linke aber nicht, d.h.* $\neg(p \not\models\!\Rightarrow_{(r,o)} p') \Rightarrow (p \not\models\!\Rightarrow_{(r,o)} p'')$*. Da* $P \models\!\Rightarrow_{(r,o)} P'$ *gilt, muss auch* $P'' \approx P'$ *gelten. Nach „*$\Rightarrow$*“ gilt* $(p \not\models\!\Rightarrow_{(r,o)} p'') \Rightarrow (\forall G_1, G_2 \in \mathcal{G},\ iso \in \mathcal{ISO} : P \cup P'' \mapsto G_1 \cup G_2 : (G_1 \models p \wedge G \models\!\Rightarrow_{(r,o)} G_2) \Rightarrow (G_2 \models p''))$*. Wegen* $P'' \approx P'$ *und* $\neg(p \not\models\!\Rightarrow_{(r,o)} p')$ *muss dann jedoch gelten* $(\exists \hat{P}' \in \mathcal{P}', iso' \in \mathcal{ISO} : \hat{P}' \preceq_{iso'} G_2) \Rightarrow (G_2 \not\models p')$*, was aber ein Widerspruch zur Voraussetzung ist.* □

**Theorem 4.** *Für eine Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$*, ihr Inverses* $[L^{-1}, \hat{\mathcal{L}}^{-1}] \rightarrow_{r^{-1}} R^{-1}$*, zwei Graphmuster* $p := [P, \hat{\mathcal{P}}]$ *und* $p' := [P', \hat{\mathcal{P}}']$ *und ein Auftreten* $o : L \cup R \mapsto P \cup P'$ *gilt im DPO*$^{iso}$*:*

$$(p \not\models\!\Rightarrow_{(r,o)} p') \Leftrightarrow (p' \not\models\!\Rightarrow_{(r^{-1},o^{-1})} p).$$

***Beweis:*** *Folgt direkt aus den beiden Theoremen 2 und 3.* □

# B.4 Systemeigenschaften und Invarianten

**Lemma 6.** *Eine Eigenschaft* $\phi$ *ist eine induktive Invariante eines Systems* $S := (G_\Omega, \mathcal{G}^i_S, \mathcal{R}_S)$*, falls* $\mathcal{T}[\phi, \neg\phi] = \emptyset$*.*

***Beweis:*** *„*$\Rightarrow$*“ : Da* $\phi$ *eine induktive Invariante ist, gilt* $\forall G_1, G_2 \in \mathcal{G}[G_\Omega],\ r \in \mathcal{R},\ o \in \mathcal{ISO} : (G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G \models \phi) \Rightarrow (G_2 \models \phi)$*. Damit gilt dann auch* $\mathcal{T}[\phi, \neg\phi] = \emptyset$*.*

*„*$\Leftarrow$*“ : Aus* $\mathcal{T}[\phi, \neg\phi]$ *folgt* $\nexists r \in \mathcal{R},\ G_1, G_2 \in \mathcal{G}[G_\Omega],\ o \in \mathcal{ISO} : ((G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G \models \phi) \wedge G_2 \not\models \phi)$*. Deshalb gilt* $\forall r \in \mathcal{R},\ G_1, G_2 \in \mathcal{G}[G_\Omega],\ o \in \mathcal{ISO} : (\neg(G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G \models \phi) \vee \neg(G \not\models \phi))$*. Das ist äquivalent zu* $\forall r \in \mathcal{R}, G_1, G_2 \in \mathcal{G}[G_\Omega], o \in \mathcal{ISO} : (G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G_1 \models \phi) \Rightarrow (G_2 \models \phi)$ *und somit ist* $\phi$ *eine induktive Invariante.* □

**Lemma 7.** *Für eine Sicherheitseigenschaft der Form* $\bigwedge_{j \in J}(\neg p_j)$ *und ein System* $S := (\mathcal{G}[G_\Omega], \mathcal{G}^i_S, \mathcal{R}_S)$ *ist die Menge* $\mathcal{T}[\phi, \neg\phi]$ *genau dann nicht leer und somit das System inkorrekt, wenn gilt:*

$$\exists G_\phi, G_{\neg\phi} \in \mathcal{G}[G_\Omega], r \in \mathcal{R}_S, o \in \mathcal{ISO} : \\ (G_\phi \models\!\Rightarrow_{(r,o)} G_{\neg\phi}) \wedge (\exists i \in J : G_{\neg\phi} \models p_i) \wedge (\forall j \in J : G_\phi \not\models p_j). \quad \text{(B.3)}$$

***Beweis:*** *Folgt direkt aus der Definition von* $\mathcal{T}[\phi, \neg\phi]$ *und der Struktur der Sicherheitseigenschaften.* □

## B.5 Nachweis von induktiven Invarianten

**Lemma 8.** *Für eine Graphtransformationsregel* $[L, \hat{\mathcal{L}}] \rightarrow_r R$ *und ein Graphmuster* $p := [P, \hat{\mathcal{P}}]$ *gilt im DPO*$^{iso}$*:*

$$\forall G, G' \in \mathcal{G}, o \in \mathcal{ISO} : G \models\!\Rightarrow_{(r,o)} G' \wedge G \not\models_o p \wedge G' \models_{iso} p$$
$$\Rightarrow$$
$$((R \setminus L) \cap o(P) \neq \emptyset) \vee (\exists iso \in \mathcal{ISO}, \hat{P} \in \hat{\mathcal{P}} : iso \mid_P = o \mid_P \wedge L \cap R \cap \hat{P} \neq \emptyset). \quad \text{(B.4)}$$

***Beweis:*** *Annahme die Implikation gilt nicht, d.h. es gilt (1) dass* $(R \setminus L) \cap iso(P) = \emptyset$ *und (2)* $\forall iso' \in \mathcal{ISO}, \hat{P} \in \mathcal{P} : iso \mid_P = iso \wedge (L \cap R \cap iso'(P)) = \emptyset$*.*

*(1) es gilt* $(R \setminus L) \cap iso(P) = \emptyset \Rightarrow \nexists n \in N_P, \; e \in E_P : \; (iso(n) \in (N_{G_1} \setminus N_{G_2})) \vee (iso(e) \in (E_{G_1} \setminus E_{G_2})) \Rightarrow P \preceq_{iso} G_1$*.*

*(2) es gilt* $\forall iso' \in \mathcal{ISO}, \; \hat{P} \in \hat{\mathcal{P}} : \; iso' \mid_P = iso \; \wedge \; L \cap R \cap iso'(\hat{P}) = \emptyset \; \Rightarrow \; \nexists n \in N_{\hat{P}} : iso'(n) \in (L \cap R)$*. Nach (1) gilt* $P \preceq_{iso} G_1$*. Somit muss die Anwendung von* $r$ *einen Knoten aus* $(N_{\hat{P}} \setminus N_P)$ *oder eine Kante aus* $(E_{\hat{P}} \setminus E_P)$ *entfernen. Für alle Knoten aus* $(N_{\hat{P}} \setminus N_P)$ *gilt, wegen der Verwendung des DPO*$^{iso}$*, dass es einen Pfad zu einem Knoten aus* $P$ *gibt. Das bedeutet, dass ein Knoten aus* $N_{\hat{P}} \setminus N_P$ *entfernt oder in diese Menge eingefügt werden kann, wenn ein adjazenter Knoten* $n'$ *explizit durch die Regel erhalten bleibt, d.h. es muss gelten* $n' \in (N_L \cap N_E \cap iso(N_{\hat{P}}))$*. Ebenso kann eine Kante aus* $(E_{\hat{P}} \setminus E_P)$ *nur gelöscht werden, wenn ein inzidenter Knoten* $n'$ *explizit erhalten bleibt. Da die Menge* $(N_L \cap N_E \cap iso(N_{\hat{P}}))$ *leer ist, muss somit gelten, dass* $\forall \hat{P} \in \mathcal{P} : \hat{P} \not\preceq_{iso} G_1$*.*

*Beide Punkte zusammen ergeben einen Widerspruch zur Annahmen.* □

**Lemma 9.** *Für ein System* $S := (G_\Omega, \mathcal{G}^i, \mathcal{R}_S)$ *und eine Sicherheitseigenschaft* $\phi := \bigwedge_{j \in J}(\neg p_j)$ *gilt im DPO*$^{iso}$ $\forall i \in J, r \in \mathcal{R}$*:*

$$(\forall egm \in \mathcal{EGM}_{\approx}(r, p_i), sgm, o \in \mathcal{ISO} : (egm \not\models\!\Rightarrow_{(r^{-1},o^{-1})} sgm) \wedge (\exists j \in J : p_j \subseteq sgm))$$
$$\Rightarrow$$
$$(\forall G_1, G_2 \in \mathcal{G}[G_\Omega] : (G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G_2 \models p_i) \Rightarrow (\exists j \in J : G_1 \models p_j)).$$

***Beweis:*** *Angenommen, die Implikation gilt nicht. Dann gilt* $\neg(\forall G_1, G_2 \in \mathcal{G}[G_\Omega], \; o \in \mathcal{ISO} : (G_1 \models\!\Rightarrow_{(r,o)} G_2 \; \wedge \; G_2 \models p_i) \; \Rightarrow \; (\exists j \in J : G_1 \models p_j))$*, d.h.* $\exists G_1, G_2 \in \mathcal{G}[G_\Omega] : (G_1 \models\!\Rightarrow_{(r,o)} G_2 \wedge G_2 \models P_i)$ *und es gilt* $\forall j \in J : G_1 \not\models p_j$*. Da* $\forall j \in J : G_1 \not\models p_j$ *gilt, folgt daraus, dass die Anwendung von* $r$ *unter dem Auftreten* $o$ *auf* $G_1$ $p_i$ *erzeugt haben muss. Nach Lemma 8 muss* $G_2$ *ein Element enthalten, auf das* $o$ *ein Element der rechten Regelseite von* $r$ *abbildet und auf das zusätzlich ein Element aus* $p_i$ *abgebildet werden kann. Da die Menge* $\mathcal{TGP}_{\approx}(r, p_i)$ *alle möglichen Ergebnisgraphmuster für* $r$ *und* $p_i$ *enthält, muss gelten* $\exists egm \in \mathcal{TGP}_{\approx}(r, p_i) : G_2 \models egm$*. Aus Theorem 2 folgt dann, dass die zu* $r$ *inverse Regel* $r^{-1}$ *auf* $G_2$ *angewendet werden kann. Diese Regelanwendung resultiert in einem Graphen* $G_1$ *für den gilt* $G \models sgm$*.* $G_1 \not\models \phi$ *impliziert nach Bedingung B.1 aus Lemma 1, dass es keinen Zeugen für ein verbotenes Graphmuster in* $G_1$ *gibt. Somit muss gelten* $\forall j \in J : p_j \not\subseteq sgm$*. Dies stellt jedoch ein Widerspruch zur Voraussetzung dar.* □

**Theorem 5.** *Für ein System* $\mathsf{S} := (\mathsf{G}_\Omega, \mathcal{G}_\mathsf{S}^\mathsf{i}, \mathcal{R}_\mathsf{S})$ *ist die Sicherheitseigenschaft* $\phi := \bigwedge_{\mathsf{j}\in\mathsf{J}}(\neg\mathsf{p}_\mathsf{j})$ *genau dann eine induktive Invariante, wenn gilt:*

$$\begin{aligned}&\forall \mathsf{i} \in \mathsf{J}, \mathsf{r} \in \mathcal{R}_\mathsf{S}, \mathsf{egm} \in \mathcal{EGM}_\approx(\mathsf{r}, \mathsf{p}_\mathsf{i}), \mathsf{sgm}, \mathsf{o} \in \mathcal{ISO} :\\ &\quad (\mathsf{egm} \not\Longrightarrow_{(\mathsf{r}^{-1},\mathsf{o}^{-1})} \mathsf{sgm}) \Rightarrow (\exists \mathsf{j} \in \mathsf{J} : \mathsf{p}_\mathsf{j} \subseteq \mathsf{sgm}).\end{aligned} \tag{B.5}$$

***Beweis:*** *Nach Lemma 9 gilt, dass* $\forall \mathsf{j} \in \mathsf{J},\ \mathsf{r} \in \mathcal{R}_\mathsf{S},\ \mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}[\mathsf{G}_\Omega],\ \mathsf{o} \in \mathcal{ISO} : (\mathsf{G}_1 \models\!\!\Longrightarrow_{(\mathsf{r},\mathsf{o})} \mathsf{G}_2 \wedge \mathsf{G}_2 \models \mathsf{p}_\mathsf{i}) \Rightarrow \exists \mathsf{j} \in \mathsf{J} : \mathsf{G} \models \mathsf{p}_\mathsf{j}$*. Dies entspricht, da* $\phi := \bigwedge_{\mathsf{j}\in\mathsf{J}}(\neg\mathsf{p}_\mathsf{j})$*,* $\forall \mathsf{G}_1, \mathsf{G}_2 \in \mathcal{G}[\mathsf{G}_\Omega],\ \mathsf{r} \in \mathcal{R}_\mathsf{S},\ \mathsf{o} \in \mathcal{ISO} : (\mathsf{G}_1 \models\!\!\Longrightarrow_{(\mathsf{r},\mathsf{o})} \mathsf{G}_2 \wedge \mathsf{G}_2 \not\models \phi) \Rightarrow \mathsf{G} \not\models \phi$*. Nach Lemma 7 muss dann* $\mathcal{T}[\phi, \neg\phi] = \emptyset$ *gelten. Lemma 6 garantiert dann, dass* $\phi$ *eine induktive Invariante von* $\mathsf{S}$ *ist.* $\square$

# Anhang C

# Algorithmen

In diesem Kapitel werden Algorithmen in Pseudocode-Notation beschrieben, die die theoretischen Ergebnisse aus Kapitel 3 darstellen.

## C.1 Graphtransformationen

Dieser Abschnitt beschreibt die Algorithmen, die benötigt werden, um Graphtransformationsregeln anzuwenden. Neben dem Algorithmus, der die eigentliche Regelanwendung beschreibt (Abschnitt C.1.2), werden auch Algorithmen vorgestellt, die die Regeln so erweitern, dass sie im $DPO^{iso}$ Ansatz korrekt angewendet werden (Abschnitt C.1.1).

### C.1.1 Negative Anwendungsbedingungen

Damit eine Graphtransformationsregel im $DPO^{iso}$ Ansatz angewendet wird, muss garantiert werden, dass sowohl die Identifikationsbedingung als auch die Lose-Kanten-Bedingung bei jeder Regelanwendung erfüllt werden. Die Erfüllung der Identifikationsbedingung wird dadurch erreicht, dass das Auftreten einer Regel in einem Anwendungsgraphen durch einen Isomorphismus gegeben sein muss. Damit die Lose-Kanten-Bedingung immer erfüllt wird, muss die linke Regelseite so erweitert werden, dass die Regel nur dann anwendbar ist, wenn bei der Anwendung keine losen Kanten entstehen. Der Algorithmus, der diese Erweiterung beschreibt, ist im folgenden Abschnitt gegeben.

Durch die Erweiterung der linken Regelseite um zusätzliche Graphen in der negativen Anwendungsbedingung, können redundante Graphen entstehen. Dies ist immer dann der Fall, wenn es zwei Graphen gibt, von denen der eine ein Teilgraph des anderen ist. In diesem Fall kann der größere der beiden Graphen aus der negativen Anwendungsbedingung entfernt werden. Immer wenn dieser größere Graph eine Regelanwendung verhindert, kann der entsprechende Match auch dazu verwendet werden, um den kleineren Graphen auf den Anwendungsgraphen abzubilden. Das bedeutet, dass die Regelanwendung auch durch den kleineren Graphen verboten wird. Der Algorithmus zum Entfernen redundanter Graphen aus der negativen Anwendungsbedingung wird nach dem Algorithmus zum Erweitern der linken Regelseite beschrieben.

### Der Algorithmus extendNAC

Abbildung C.1 zeigt einen Algorithmus in Pseudocode-Notation, der die Erweiterung der negativen Anwendungsbedingung vornimmt. Als Eingabe erhält der Algorithmus eine Regel $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_{\mathsf{r}} \mathsf{R}$ sowie den dazu gehörigen Typgraph $\mathsf{G}_\Omega$. Die äußere For-Schleife betrachtet alle Knoten, die gelöscht werden sollen. In den Zeilen 5 bis 18 werden für jeden dieser Knoten alle inzidenten Kanten betrachtet, die den zu löschenden Knoten als Startknoten haben. Ist die Kante kein Teil der linken Regelseite, so wird ein Graph erzeugt, der neben der linken Regelseite diese Kante enthält. Ist der Zielknoten der Kante noch nicht im Graphen enthalten, so wird auch er in den Graphen aufgenommen. In Zeile 16 wird der Graph zur negativen Anwendungsbedingung hinzugefügt. In den Zeilen 19 bis 33 wird der Fall betrachtet, dass der zu löschende Knoten Zielknoten einer Kante ist, die nicht durch die Regel gelöscht wird.

```
01: Set<Graph> extendNAC(Set<Graph>(L, L̂, R), Graph G_Ω)
02: begin
03:   Set<Graph> addNAC = L̂
04:   forall n ∈ N_L − N_R do
05:     forall (l_N(n), t_e, n') ∈ correctTypedEdges(G_Ω) do
06:       // edge with source in L but not in R
07:       if (e ∉ E_L) then
08:         E' := E_L ∪ {e}
09:         if (n' ∉ N_L) then
10:           N' := N_L ∪ {n'}
11:        fi
12:         src' := λ x.if (x == e) then n else src(x) fi
13:         tgt' := λ x.if (x == e) then n' else tgt(x) fi
14:         l'_N := λ x.if (x == n') then t_n' else l_N(x) fi
15:         l'_E := λ x.if (x == e) then t_e else l_E(x) fi
16:         addNAC.add(N', E', src', tgt', l'_N, l'_E)
17:       fi
18:     end
19:     forall (n', t_e, l_N(n)) ∈ correctTypedEdges(G_Ω) do
20:       // edge with target in L but not in R
21:       if (e ∉ E_L) then
22:         E' := E_L ∪ {e}
23:         if (n' ∉ N_L) then
24:           N' := N_L ∪ {n'}
25:         fi
26:         src' := λ x.if (x == e) then n' else src(x) fi
27:         tgt' := λ x.if (x == e) then n else tgt(x) fi
28:         l'_N := λ x.if (x == n') then t_n' else l_N(x) fi
29:         l'_E := λ x.if (x == e) then t_e else l_E(x) fi
30:         addNAC.add(N', E', src', tgt', l'_N, l'_E)
31:       fi
32:     end
33:   end
34:   return addNAC
35: end
```

Abbildung C.1: Algorithmus extendNAC, der die negative Anwendungsbedingung der linken Regelseite erweitert, sodass die Regelanwendung die Lose-Kanten-Bedingung erfüllt

### Der Algorithmus minimizeNAC

Der Algorithmus zur Minimierung von negativen Anwendungsbedingungen ist in Abbildung C.2 dargestellt. Dieser Algorithmus prüft in Zeile 4 bis 10 für jeden Graphen $\hat{L}$ der negativen Anwendungsbedingung, ob es einen anderen Graphen $\hat{L}'$ gibt, der Teilgraph von $\hat{L}$ ist. Kann kein solcher Graph gefunden werden, wird $\hat{L}$ in die minimale negative Anwendungsbedingung aufgenommen, Zeile 11. Als Rückgabe liefert der Algorithmus in Zeile 13 eine Menge von Graphen, die die minimale negative Anwendungsbedingung beschreiben.

```
01: Set <Graph> minimizeNAC(<Graph> L̂) begin
02:   Set <Graph> HL := ∅
03:   forall L̂ ∈ L̂ do
04:     Boolean flag := true;
05:     forall L̂' ∈ L̂ − {L̂} do
06:       if (∃ iso : iso|_L) = id_L ∧ L̂' ⪯_iso L̂) then
07:         flag := false
08:       fi
09:     end
10:     if (flag = true) then HL := HL ∪ {L̂} fi
11:   end
12:   return HL
13: end
```

Abbildung C.2: Algorithmus minimizeNAC zur Minimierung von negativen Anwendungsbedingungen

## C.1.2 Anwendung von Graphtransformationsregeln

Nachdem eine Graphtransformationsregel um zusätzliche Graphen in der negativen Anwendungsbedingung erweitert wurde, kann ihre Anwendung erfolgen. Der Algorithmus der eine Regelanwendung im DPO$^{iso}$ Ansatz vornimmt, wenn diese erweitert wurden, ist im folgenden Abschnitt gegeben.

### Der Algorithmus apply

Der Algorithmus, der die Anwendung einer Regel r auf einen Graphen G beschreibt ist in Abbildung C.3 gegeben, wobei iso das Auftreten von r in G beschreibt. In Zeile 02 wird überprüft, ob der übergebene Isomorphismus die linke Regelseite auf einen Teilgraphen des Anwendungsgraphen abbildet. Ist dies nicht der Fall, so kann die Regel nicht angewendet werden. Andernfalls wird ein neuer Isomorphismus gewählt, der den übergebenen Isomorphismus so erweitert, dass er auch die Elemente der rechten Regelseite abbildet. Mittels diesem neuen Isomorphismus erfolgt dann (Zeile 05) die eigentliche Regelanwendung. Dazu werden alle Elemente aus dem Anwendungsgraphen entfernt, auf den die Elemente der linken aber nicht der rechten Regelseite durch den Isomorphismus abgebildet werden. Elemente, die in der rechten, aber nicht in der linken Regelseite enthalten sind, werden durch den Isomorphismus in den Anwendungsgraphen eingefügt. Der resultierende Graph wird dann (Zeile 09) zurückgeliefert.

```
01: Graph apply(Graph G, Rule L→_r R, Isomorphism iso)
02:   if (L ≺_iso G) then
03:     choose Isomorphism iso': R ↦ G with iso'|_L = iso ∧ iso'(R \ L) ∩ G = ∅
04:     //apply rule
05:     TG := (G \ iso'(L \ R)) ∪ iso'(R)
06:     //all other nodes and edges remain unchanged
07:     return TG
08:   else
09:     return G
10:   fi
11: end
```

Abbildung C.3: Algorithmus apply, der die Anwendung einer Graphtransformationsregel unter dem Single Pushout Approach beschreibt

# C.2 Erweiterte Graphtransformationen

Mit dem im voran gegangenen Abschnitt eingeführten Algorithmus ist es möglich, eine Regel in Vorwärtsrichtung auf einen Anwendungsgraphen anzuwenden. Wurde die Regel zuvor um zusätzliche Graphen in der negativen Anwendungsbedingung erweitert, so erfüllt diese Regelanwendungen die Identifikations- und die Lose-Kanten-Bedingung des Eingeschränkten Single Pushout Ansatz. Eine Regelanwendung im Single Pushout Ansatz kann auch in Rückwärtsrichtung erfolgen. Außerdem ist es nicht nur möglich, eine Regel auf einen Anwendungsgraphen, sondern auch auf ein Graphmuster anzuwenden. Die benötigten Algorithmen zur Rückwärtsanwendung von Graphtransformationsregeln werden im folgenden Abschnitt C.2.1 erklärt. Der Algorithmus zur Anwendung einer Graphtransformationsregel auf ein Graphmuster ist Inhalt von Abschnitt C.2.2.

## C.2.1 Rückwärtsanwendung von Graphtransformationsregeln

Nachdem beschrieben wurde, wie eine Regel im DPO$^{iso}$ Ansatz korrekt in Vorwärtsrichtung angewendet wird, soll in diesem Abschnitt die Rückwärtsanwendung betrachtet werden. Auch bei der Rückwärtsanwendung müssen die Identifikations- und die Lose-Kanten-Bedingung des DPO$^{iso}$ Ansatz erfüllt werden.

Der Algorithmus reverse verändert eine Graphtransformationsregel so, dass sie in Rückwärtsrichtung angewendet werden kann und dabei die Bedingungen des DPO$^{iso}$ Ansatz einhält. Dazu ist es zunächst notwendig, die negative Anwendungsbedingung anzupassen. Dies erfolgt mittels des Algorithmus convertNAC, der in Abbildung C.4 gegeben ist. Die eigentlich Anwendung der Regel erfolgt dann mit dem bereits beschrieben Algorithmus zur Regelanwendung apply aus Abbildung C.3.

### Der Algorithmus convertNAC

Damit eine Regel korrekt in Rückwärtsrichtung angewendet werden kann, muss ihre negative Anwendungsbedingung angepasst werden. Dies ist aus verschiedenen Gründen erforderlich. Als

erstes wird bei einer Rückwärtsanwendung aus der ursprünglichen linken Regelseite die neue rechte Regelseite und diese darf keine negative Anwendungsbedingung besitzen. Zweitens, auch bei einer Rückwärtsanwendung müssen die Identifikations- und die Lose-Kanten-Bedingung des DPO$^{iso}$ Ansatzes eingehalten werden.

Um die negative Anwendungsbedingung konvertieren zu können, müssen zunächst alle Graphen aus der negativen Anwendungsbedingung entfernt werden, die eine Regelanwendung verhindern, wenn andernfalls lose Kanten entstehen würden (Zeilen 04 bis 06). Die verbleibenden Graphen der negativen Anwendung werden dann (Zeilen 07 bis 12) transformiert, indem die Regel (ohne negative Anwendungsbedingung) darauf angewendet wird. Durch das Entfernen aller Graphen, die eine Regelanwendung verhindern, sollten andernfalls lose Kanten entstehen, wird gewährleistet, dass die Regel nur auf solche Graphen angewendet wird, bei denen keine losen Kanten zurückbleiben. Nachdem diese Graphen modifiziert wurden, müssen nun die Graphen hinzugefügt werden, die eine Regelanwendung verhindern, wenn sonst lose Kanten entstehen. Dazu wird die Funktion extendNAC (Zeile 13) aufgerufen, wobei jedoch die linke und die rechte Regelseite vertauscht wurden.

```
01: Set<Graph> convertNAC(GraphRule [L, L̂]→r R)
02: begin
03:   Set <Graph> HL' := ∅
04:   Set <Graph> enac := extendNAC([L, L̂]→r R, G_Ω)
05:   //delete all graphs of the NAC which relate to a dangling edge
06:   Set <Graph> HL := {L̂ ∈ L̂ | ∄ L̂' ∈ enac: L̂' ≤ L̂}
07:   while HL ≠ ∅ do
08:     choose L̂ in HL
09:     L̂' := apply(L̂, L→r R)
10:     HL := HL \ {L̂}
11:     HL' := HL' ∪ {L̂'}
12:   end
13:   return HL' ∪ extendNAC(R→r-1 L, G_Ω)
14: end
```

Abbildung C.4: Algorithmus convertNAC zum Konvertieren von negativen Anwendungsbedingungen

### Der Algorithmus reverse

Der Algorithmus reverse Abbildung C.5 beschreibt die Transformation einer Graphtransformationsregel, sodass ihre Anwendung mittels des Algorithmus apply aus Abbildung C.3 die Bedingungen des DPO$^{iso}$ Ansatzes erfüllt.

In einem ersten Schritt (Zeile 03) wird dazu die negative Anwendungsbedingung mit Hilfe des Algorithmus convertNAC aus Abbildung C.4 konvertiert. Anschließend wird die resultierende negative Anwendungsbedingung mit dem Algorithmus minimizeNAC aus Abbildung C.2 minimiert, d.h. redundante Graphen werden entfernt (Zeile 04). Als letztes wird die Aufgabe der Graphen L und R vertauscht (Zeile 05). Das bedeutet, aus der ursprünglichen Nachbedingung, der rechten Regelseite R, wird die neue Anwendungsbedingung und aus der ursprünglichen Anwendungsbedingung L die neue Nachbedingung.

```
01: GraphRule reverse(GraphRule [L, L̂]→rR, Graph GΩ)
02: begin
03:   Set<Graph> nac := convertNAC( [L, L̂]→rR, GΩ)
04:   nac := minimizeNAC([R, nac]→r⁻¹L)
05:   return [R, nac]→r⁻¹L
06: end
```

Abbildung C.5: Algorithmus reverse zur Invertierung von Graphtransformationsregeln

### C.2.2 Anwendung von Regeln auf Graphmuster

Eine Graphtransformationsregel kann auch dazu genutzt werden, um ein Graphmuster zu verändern. Dabei muss jedoch berücksichtigt werden, dass das Graphmuster eine negative Anwendungsbedingung besitzen kann. Diese negative Anwendungsbedingung muss durch die Regelanwendung ebenfalls transformiert werden. Der Algorithmus, der eine Regel auf ein Graphmuster anwendet, wird im Folgenden beschrieben.

#### Der Algorithmus applyRuleToPattern

Der Algorithmus, der die Anwendung einer Regel r auf ein Graphmuster p beschreibt, ist in Abbildung C.6 gegeben. Die if-Anweisung in Zeile 02 prüft, ob die linke Regelseite ein Teilgraphmuster des Graphmusters $[\mathsf{P}, \hat{\mathcal{P}}]$ ist. Ist dies der Fall, wird in den Zeilen 03 bis 16 das Auftreten o der linken und rechten Regelseite bestimmt. Dieses Auftreten bildet Elemente der linken Regelseite mittels des Isomorphismus aus der if-Anweisung auf Elemente der Graphmusters ab. Für Elemente, die nur Teil der rechten Regelseite sind, wird die Identitätsfunktion verwendet. Das Auftreten wird dann dazu verwendet, um alle Elemente, die zur linken, aber nicht zur rechten Regelseite gehören aus dem Muster zu entfernen (Zeile 19). Elemente, die zur rechten aber nicht zur linken Regelseite gehören, werden mit Hilfe von o in das Muster eingefügt, Zeile 22. Ist dies erfolgt, so wird die negative Anwendungsbedingung des Musters angepasst. In der for-Schleife in den Zeilen 27 bis 30 werden die Graphen der negativen Anwendungsbedingung $\hat{\mathcal{P}}$ angepasst, indem die Regel r auf jeden der Graphen angewendet wird. Da $[\mathsf{L}, \hat{\mathcal{L}}]$ ein Teilgraphmuster von $[\mathsf{P}, \hat{\mathcal{P}}]$, ist eine Anwendung der Regel auf jeden Graphen aus $\hat{\mathcal{P}}$ immer möglich. Da die Regel r auf ein Graphmuster angewendet wird, wird auch die konvertierte negative Anwendungsbedingung von r in die negative Anwendungsbedingung von p aufgenommen. Dies ist in Zeile 32 beschrieben. Als Ergebnis liefert der Algorithmus ein Graphmuster $\mathsf{p}' := [\mathsf{P}'\hat{\mathcal{P}}']$, wobei $\mathsf{P}'$ durch die Anwendung der Regel auf P erzeugt wird und $\hat{\mathcal{P}}'$ sich aus den konvertierten negativen Anwendungsbedingungen von p und r zusammensetzt.

## C.3 Überprüfung induktiver Invarianten

Nachdem die Algorithmen zur Regelanwendung in Vorwärts- und Rückwärtsrichtung betrachtet wurden und ein Algorithmus zur Anwendung einer Regel auf Graphmuster vorgestellt wurde, können nun die Algorithmen eingeführt werden, die den Nachweis induktiver Invarianten durchführen.

```
01: GraphPattern applyRuleToPattern(GraphPattern [P, P̂],
                                    Rule r→[L, mathcalL]R, isomorphism iso, Graph G_Ω) begin
02:   if ((L ⪯iso G) ∧ (∄ L̂ ∈ L̂, P ∈ P̂, iso' :
                                    iso'^-1 | _P = iso ∧ (P̂ \ (P' \ iso(P))) ⪯iso' L̂) then
03:     //build the occurence o := ⟨ o_n, o_e⟩ by extending iso
04:     o_n := λ x. if (x ∈ N_L) then
05:                                   iso_n
06:                                 else
07:                                   if (x ∈ N_R \ N_L) then
08:                                     id
09:                                   fi
10:                                 fi
11:     o_e := λ x. if (x ∈ E_L) then
12:                                   iso_e
13:                                 else
14:                                   if (x ∈ E_R \ E_L) then
15:                                     id
16:                                   fi
17:                                 fi
18:     //build new Graph P' with P ⟹r P'
19:     //first remove all elements that belong to L but not to R
20:     P' := P \ o(L \ R)
21:     //add elements that belong to R but not to L
22:     P' := P' ∪ (R \ L)
23:     //all other elements remain unchanged
24:     //P̂' is build of adjusted P̂ and L̂
25:     //P̂ has to be converted by applying r to all graphs in P̂
26:     Set<Graph> P̂' := ∅
27:     for (P̂ ∈ P̂) do
28:       Graph P̂' := apply(P̂, r'→L R, o)
29:       P̂' := P̂' ∪ P̂'
30:     end
31:     //add converted L̂ to P̂'
32:     P̂' := P̂' ∪ convertNAC(r→[L, L]R, G_Ω)
33:     return [P', P̂']
34:   else
35:     return [P, P̂]
36:   fi
37: end
```

Abbildung C.6: Algorithmus applyRuleToPattern, der die Anwendung einer Graphtransformationsregel auf ein Graphmuster beschreibt

Dazu wird zunächst die Menge aller Ergebnisgraphmuster mittels des Algorithmus buildTGP gebildet (siehe Algorithmus in Abbildung C.7). Dabei können jedoch redundante Ergebnisgraphmuster entstehen, d.h. die vom Algorithmus zurück gelieferte Menge enthält Paare von Ergebnisgraphmustern, bei denen das eine Graphmuster ein Teilgraphmuster des zweiten ist. In diesem Fall, reicht es das Teilgraphmuster zu behalten, das andere Graphmuster kann aus der Menge entfernt werden Dies erfolgt mittels des Algorithmus reduceTGP in Abbildung C.8.

Der Algorithmus, der den Nachweis der induktiven Invarianten führt und diese beiden Algorithmen aufruft, wurde bereits in Abschnitt 3.7.2 erläutert.

## C.3.1 Bildung von Ergebnisgraphmustern

Der wichtigste Algorithmus beim Nachweis induktiver Invarianten ist der Algorithmus buildTGP, der die Ergebnisgraphmuster erzeugt. Dieser Algorithmus wird als nächstes vorgestellt. Die Menge der zurückgelieferten Ergebnisgraphmuster, kann redundante Ergebnisgraphmuster enthalten. Das bedeutet, die Menge enthält Paare von Graphmustern bei denen das eine ein Teilgraphmuster des anderen ist. In diesem Fall reicht es aus, das Teilgraphmuster zu behalten, das andere Graphmuster kann aus der Menge entfernt werden. Diese Reduktion der Menge der Ergebnisgraphmuster kann mit dem Algorithmus reduceTGP aus Abbildung C.8 erfolgen.

### Der Algorithmus buildTGP

Der Algorithmus buildTGP erzeugt für eine Graphtransformationsregel $[\mathsf{L}, \hat{\mathcal{L}}] \rightarrow_r \mathsf{R}$ und ein verbotenes Graphmuster $[\mathsf{P}, \hat{\mathcal{P}}]$ alle möglichen Ergebnisgraphmuster.

Die Menge der Ergebnisgraphmuster setzt sich nach Definition 35 aus zwei Teilmengen zusammen. Die erste beschreibt den Fall, dass Elemente, die durch die Regel erzeugt werden die Anwendungsbedingung des verbotenen Graphmusters vervollständigt. In diesem Fall gibt es mindestens ein Element, das Teil der rechten, aber nicht der linken Regelseite ist und das durch einen Isomorphismus auf ein Element der Anwendungsbedingung des verbotenen Graphmuster abgebildet werden kann. Die zweite Teilmenge besteht aus den Ergebnisgraphmustern, bei denen die Regel mindestens ein Element entfernt, das Teil eines Graphen der negativen Anwendungsbedingung des verbotenen Graphmusters ist. In diesem Fall gibt es mindestens einen Knoten, der bei der Regelanwendung erhalten bleibt und somit Teil der linken und rechten Regelseite ist. Zusätzlich muss dieses Element durch einen Isomorphismus auf ein Element des verbotenen Graphmusters abbildet.

Die Bildung der ersten Teilmenge wird in den Zeilen 08 bis 37 beschrieben. In der Zeile 08 wird die Menge aller Teilgraphen der linken Regelseite gebildet. Dabei müssen diese Teilgraphen jedoch mindestens ein Element enthalten, das nicht auch zur rechten Regelseite gehört. Anschließend (Zeile 09) wird die Menge aller Teilgraphen der Anwendungsbedingung des verbotenen Graphmusters gebildet. Für jedes Paar, bestehend aus einem Teilgraphen der linken Regelseite und einem Teilgraphen der Anwendungsbedingung des verbotenen Graphmusters, wird bestimmt, ob es einen Teilgraphisomorphismus gibt, der den Teilgraphen des verbotenen Graphmusters auf den Teilgraphen der linken Regelseite abbildet (Zeile 13). Ist dies der Fall (Zeile 14), so wird dieser Isomorphismus verwendet, um die Anwendungsbedingung des Ergebnisgraphmusters aus der linken Regelseite und der Anwendungsbedingung des verbotenen Graphmusters zu bilden. Anschließend wird die Menge aller Isomorphismen bestimmt, die den gefundenen Isomorphismus erweitern (Zeile 17). Diese Isomorphismen werden dazu verwendet, um die negativen Anwendungsbedingungen des Ergebnisgraphmusters zu bilden. Dazu werden zum einen alle Graphen der negativen Anwendungsbedingung der linken Regelseite um den Anwendungsgraphen des verbotenen Graphmusters erweitert (Zeile 16 bis 21). Und zum anderen werden alle Graphen der negativen Anwendungsbedingung des verbotenen Graphmusters um die rechte Regelseite erweitert (Zeile 24 bis 30). In Zeile 31 wird die Menge der Graphen der negativen Anwendungsbedingung minimiert, bevor in Zeile 33 das Ergebnisgraphmuster erzeugt und in Zeile 34 zur Menge der Ergebnisgraphmuster hinzugefügt.

Die Bildung der zweiten Teilmenge ist Inhalt der Zeilen 38 bis 58. In Zeile 41 wird zunächst die Menge $\Delta\mathsf{LS}^{-1}$ neu gebildet. Dabei besteht $\mathsf{LS}^{-1}$ aus allen Teilgraphen der linken Regelseite, bei denen es mindestens ein Element gibt, das auch zur rechten Regelseite gehört. Dann wird die Menge aller Teilgraphen aller Graphen der negativen Anwendungsbedingung des verbotenen Graphmusters gebildet (Zeile 41). Für jedes Paar, bestehend aus einem Graphen der linken Regelseite und des verbotenen Graphmusters, wird bestimmt, ob es einen Isomorphismus gibt, der den Teilgraphen des verbotenen Graphmuster auf den Teilgraphen der linken Regelseite abbildet (Zeile 44). Ist dies der Fall, so wird dieser Isomorphismus als erstes dazu verwendet, um die Anwendungsbedingung des Ergebnisgraphmusters zu bilden (Zeile 45). Als nächstes wird die Menge der Isomorphismen bestimmt, die den Isomorphismus erweitern. Mittels dieser Menge der Isomorphismen werden dann die Graphen der negativen Anwendungsbedingungen gebildet, indem die Anwendungsbedingung der linken Regelseite um die Elemente der Graphen der negativen Anwendungsbedingung des verbotenen Graphmusters erweitert wird (Zeilen 46 bis 51). Diese Menge wird in Zeile 52 minimiert, in Zeile 53 wird das Ergebnisgraphmuster gebildet und in Zeile 54 zur Menge der Ergebnisgraphmuster hinzugefügt.

Die Menge aller Ergebnisgraphmuster wird dann zurückgeliefert (Zeile 59).

### Der Algorithmus reduceTGP

Der zuvor vorgestellte Algorithmus buildTGP bestimmt für eine Regel und einem verbotenen Graphmuster alle möglichen Ergebnisgraphmuster. Dabei entstehen jedoch redundante Graphmuster, d.h. es gibt Paare von Graphmustern, bei denen das eine Muster ein Teilgraphmuster des zweiten ist. Dies ist zwar kein Fehler, hat jedoch Auswirkungen auf den Verifikationsalgorithmus, da dieser für jedes Ergebnisgraphmuster überprüfen muss, ob es durch die Anwendung der Regel auf ein korrektes Startgraphmuster erzeugt worden sein kann. Deshalb wird nun ein Algorithmus vorgestellt, mit dem die Anzahl der Ergebnisgraphmuster minimiert werden kann.

Der Algorithmus prüft für jedes Paar der Ergebnisgraphmuster aus der übergebenen Menge, ob das Ergebnisgraphmuster tgp ein Teilgraphmuster von tgp′ ist (Zeilen 05 bis 18). Ist dies der Fall wird tgp′ aus der Menge der Ergebnisgraphmuster entfernt, Zeile 12. Die minimierte Menge wird dann zurück gegeben (Zeile 19).

```
01: Set<GraphPattern> buildTGP(Rule [L^-1, L̂^-1]→_{r-1} R^-1,  GraphPattern[P, P̂])
02: begin:
03:   Set<GraphPattern> TGP := ∅
04:   Graph TGP
05:   Graph TĜP
06:   GraphPattern tgp

07:   //(L^-1 \ R^-1) ∩ P ≠ ∅
08:   Set<Graph> LS^-1 := {L' | (L' ≤ L^-1) ∧ ((L^-1 \ R^-1) ≠ ∅)}
09:   Set<Graph> PS := {P' | P' ≤ P}

10:   forall (L^-1 ∈ LS^-1) do
11:     forall (P' ∈ PS) do
12:       if (∃ iso : L^-1 =_iso P') then
13:         TGP = L^-1 ∪ iso(P)
14:         //calculate the NAC
15:         //extend all L̂^-1 by P
16:         IS𝒪' := {iso' | iso'|_P' = iso}
17:         forall (iso' ∈ IS𝒪') do
18:           forall (L̂^-1 ∈ L̂^-1) do
19:             TĜP := L̂^-1 ∪ iso'(P)
20:             T𝒢𝒫̂ := T𝒢𝒫̂ ∪ TĜP
21:           end
22:         end
23:         //extend all P̂ by L^-1
24:         forall (iso' ∈ IS𝒪') do
25:           forall (P̂ ∈ 𝒫̂) do
26:             TĜP := L^-1 ∪ P̂
27:             T𝒢𝒫̂ := T𝒢𝒫̂ ∪ TĜP
28:           end
29:         end
30:         T𝒢𝒫̂ := minimizeNAC(T𝒢𝒫̂)
31:         tgp := [TGP, T𝒢𝒫̂]
32:         T𝒢𝒫 := T𝒢𝒫 ∪ tgp
33:         T𝒢𝒫̂ := ∅
34:       fi
35:     end
36:   end

37:   //L^-1 ∩ R^-1 ∩ P̂ ≠ ∅
38:   TĜP := ∅
39:   LS^-1 := {L^-1 | L-1 ≤ (L^-1 ∩ R^-1)}
40:   P̂S := {P̂' | (P̂' ≤ P̂)}
41:   forall (L' ∈ LS^-1) do
42:     forall (P̂' ∈ P̂S) do
43:       if (∃ iso : L^-1 =_iso P̂') then
44:         TGP := L^-1 ∪ iso(P)
45:         //calculate the NAC
46:         IS𝒪' := {iso' | iso'|_P̂' = iso}
47:         forall (iso' ∈ IŜ𝒪') do
48:           forall (P̂ ∈ 𝒫̂) do
49:             TĜP := L^-1 ∪ iso'(P̂)
50:             T𝒢𝒫̂ := T𝒢𝒫̂ ∪ TĜP
51:           end
51:         end
52:         T𝒢𝒫̂ := minimizeNAC(T𝒢𝒫̂)
53:         tgp := [TGP, T𝒢𝒫̂]
54:         T𝒢𝒫 := T𝒢𝒫 ∪ tgp
55:         T𝒢𝒫̂ := ∅
56:       fi
57:     end
58:   end

59:   return T𝒢𝒫
60: end
```

Abbildung C.7: Algorithmus buildTGP zur Bildung der Ergebnisgraphmuster

```
01: Set<GraphPattern> reduceTGP(GraphPattern 𝒯𝒢𝒫)
02: begin:
03:   Set<GraphPattern> reducedSet := 𝒯𝒢𝒫
04:   Set<GraphPattern> tmpSet

05:   forall (tgp := [TGP, 𝒯Ĝ𝒫] ∈ 𝒯𝒢𝒫) do
06:     tmpSet := 𝒯𝒢𝒫 \ tgp
07:     forall (tgp' := [TGP', 𝒯Ĝ𝒫'] ∈ tmpSet) do
08:       if (∃ iso : TGP ⪯_iso TGP') then
09:         forall (TĜP ∈ 𝒯Ĝ𝒫) do
10:           if (∄ iso' : iso' |_TGP = iso ∧ TĜP ⪯_iso' TGP') then
11:             if (∃ TĜP' ∈ 𝒯Ĝ𝒫' ∧ ∃ iso'' :
                      iso''^-1 |_TGP = iso ∧ (TĜP' \ (TGP' \ iso(TGP))) ⪯_iso'' TĜP) then
12:               reducedSet := reducedSet \ tgp'
13:             fi
14:           fi
15:         end
16:       fi
17:     end
18:   end
19:   return reducedSet
20: end
```

Abbildung C.8: Algorithmus reduceTGP zur Reduzierung der Menge der Ergebnisgraphmuster

# Index