Gottfried Wilhelm
Leibniz Universität Hannover
Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

MSpec – Eine Technik zur textuellen Modellierung und Simulation multimodaler szenariobasierter Spezifikationen

Bachelorarbeit

im Studiengang Informatik

von

Florian Wolfgang Hagen König

Prüfer: Prof. Dr. Joel Greenyer
Zweitprüfer: Prof. Dr. Kurt Schneider
Betreuer: Prof. Dr. Joel Greenyer

Hannover, 09.09.2014

Erklärung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 09.09.2014

Florian Wolfgang Hagen König

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Problemstellung 1
1.2. Lösungsansatz 2
1.3. Struktur der Arbeit 2

2. Grundlagen **5**

2.1. Verteilte reaktive Systeme 5
2.2. Modal Sequence Diagram (MSD) 6
2.2.1. Syntax 7
2.2.2. Semantik 15
2.3. Play-Out 20
2.4. Domain-specific Language (DSL) 21
2.5. Xtext 21
2.6. Eclipse Modeling Framework 21
2.6.1. Ecore-Model 22
2.6.2. Generator-Model 22

3. Die Modellierungssprache MSpec **23**

3.1. Syntax 23
3.1.1. Import Ecore-Model 24
3.1.2. Specification 24
3.1.3. UseCase 25
3.1.4. Collaboration 25
3.1.5. Scenario 27
3.2. Semantik 33
3.2.1. Ablauf eines MSpec-Scenarios 34
3.2.2. Der Message-Pointer 34
3.2.3. Unifikation zweier Nachrichten 35

4. Übersetzung in ausführbaren Code **37**

4.1. Konzepte 37
4.1.1. Ermittlung von Daten eines Scenario 37

4.1.2. Das Scenario als Zustandsautomat 37
4.2. Übersetzung in Java 41
4.2.1. Übersetzung in ein Ecore-Model 41
4.2.2. Ecore-Model-Code generieren 42
4.2.3. Fertigstellung des Model-Codes 42

5. Implementierung **47**
5.1. Der MWE2 Workflow 47
5.2. Scoping 48
5.3. Code-Validation 50
5.4. Quickfix-Provider 52
5.5. Label-Provider 53

6. Verwandte Arbeiten **55**
6.1. ScenarioTools 55
6.2. Can Programming Be Liberated, Period? 55
6.3. Behavioral Programming 56
6.4. WebSequenceDiagrams.com 56

7. Zusammenfassung und Ausblick **59**
7.1. Zusammenfassung 59
7.2. Ausblick 60

A. Xtext-Grammatik von MSpec **61**

Literaturverzeichnis **65**

1. Einleitung

In der heutigen Softwareentwicklung geht es meist nicht mehr nur um einzelne, eigenständige Programme, sondern um verteilte reaktive Systeme, die aus vielen Komponenten bestehen. Besonders in den Bereichen Verkehr, Logistik, Produktion, Kommunikation und Medizin sind viele sicherheitskritische Computersysteme miteinander verbunden und müssen präzise zusammenarbeiten. Diese Systeme so zu planen, dass sie hinterher auch korrekt funktionieren, ist die große Herausforderung im Entwurf. Die Probleme schleichen sich aber oft schon in den Anforderungen ein. Im szenariobasierten Entwurf werden diese Anforderungen als einzelne Szenarien dargestellt. Für diese Darstellung gibt es Konzepte wie Live Sequence Charts (LSCs) und die hier betrachtete Modifikation Modal Sequence Diagrams (MSD). MSDs bieten neben einer intuitiven Verwendung insbesondere die Möglichkeit präzise zu spezifizieren, wie die einzelnen Systemkomponenten in Reaktion auf Umweltereignisse reagieren müssen. Hierbei wird spezifiziert, was genau bei bestimmten Umweltereignissen passieren kann, muss oder nicht passieren darf. Oft werden aber an dieser Stelle Fehler gemacht, die erst später im Endprodukt entdeckt werden. Die Anforderungen können sehr vielseitig sein und Wechselwirkungen untereinander entwickeln, wodurch sich die Möglichkeit ergibt, dass sich Anforderungen gegenseitig behindern oder widersprechen. Bei sehr vielen Anforderungen und damit sehr vielen Szenarien können Entwickler leicht den Überblick verlieren und solche Wechselwirkungen übersehen. Diese Fehler können gravierende Konsequenzen nach sich ziehen und sie zu korrigieren kann viel Zeit und Geld kosten.

1.1. Problemstellung

Es ist also wichtig, Fehler in den Anforderungen zu finden, bevor sie umgesetzt werden. Um dies zu bewerkstelligen, sind automatisierte Analysemethoden nötig, mit denen diese Szenarien analysiert und simuliert werden können. Es gibt bereits die Werkzeuge ScenarioTools (siehe [GBPLM13]) und die Play-Engine (siehe [HM03]), die eine solche Simulation von Anforderungen ermöglichen, indem sie grafische Editoren bieten, um Modal Sequence Diagrams zu erstellen, und zudem ein Play-Out dieser ermöglichen. Die grafischen Darstellungen der

Szenarien dieser Editoren sind zwar auf den ersten Blick übersichtlich und leicht zu lesen, jedoch verlieren sich diese Eigenschaften bei zunehmender Komplexität der Szenarien. Die grafischen Editoren dieser Werkzeuge sind zudem leider noch nicht ausgereift und weisen Schwächen in der Bedienbarkeit und Erweiterbarkeit auf. Des Weiteren ist die Notwendigkeit spezieller Editoren für Szenarien in der Praxis oft nicht akzeptiert.

1.2. Lösungsansatz

Es muss also eine benutzerfreundlichere Lösung her: Da grafische Darstellungen für komplexe Probleme zu kompliziert werden, bietet sich eine Lösung auf Textebene an. Es soll nun ermöglicht werden, diese Szenarien textuell darzustellen und so die Anforderungen übersichtlich und ohne die Notwendigkeit eines grafischen Editors aufzustellen. Der Schlüssel dafür ist, eine Sprache speziell für Szenarien, die ebenso auf dem Prinzip der Modal Sequence Diagrams basiert, zu entwickeln. Mithilfe dieser Sprache soll dann Code generiert werden können, durch den Play-Out ermöglicht wird. Dies soll das Aufstellen und die Darstellung komplexer Szenarien erleichtern und eine frühe Simulation und Analyse der Anforderungen ermöglichen.
Im Rahmen dieser Arbeit wird eine Modellierungsprache mit Namen MSpec zur textuellen Darstellung von Szenarien in Anlehnung an Modal Sequence Diagrams präsentiert. Dazu werden Syntax und Semantik der Sprache erläutert und anhand von Beispielen gezeigt, wie eine Spezifikation in MSpec aussehen könnte. Diese Beispiele werden zudem mit bedeutungsgleichen Szenarien in Form von MSDs verglichen. Ebenfalls werden Konzepte für die Codeübersetzung für das Play-Out und die Implementierung der Sprache beschrieben.

1.3. Struktur der Arbeit

Diese Arbeit ist wie folgt strukturiert.
In Kapitel 1 wird die Motivation, das Ziel und die Struktur der Arbeit beschrieben.
Kapitel 2 erläutert grundlegende Konzepte und Begrifflichkeiten, die für das Verständnis dieser Arbeit notwendig sind.
In Kapitel 3 werden die Syntax und die Semantik der Modellierungssprache MSpec vorgestellt.
Kapitel 4 befasst sich mit der Übersetzung der Sprache in ausführbaren Java-Code, der für das Play-Out genutzt werden kann.

In Kapitel 5 wird die Implementierung der Sprache und des Editors beschrieben.
Kapitel 6 befasst sich mit verwandten Arbeiten. Hier werden Arbeiten und Artikel vorgestellt, die sich mit den gleichen oder mit ähnlichen Themen befassen, wie diese Arbeit.
In Kapitel 7 werden die Erfolge und Entwicklungen der Arbeit noch einmal zusammengefasst und es wird ein Fazit gezogen.

2. Grundlagen

In diesem Kapitel werden Grundlagen erläutert, die für das Verständnis dieser Arbeit nötig sind.

2.1. Verteilte reaktive Systeme

Reaktive Systeme tragen ihren Namen aus dem Grund, dass sie ständig auf verschiedene Ereignisse oder Signale entsprechend bestimmter Bedingungen reagieren müssen. Diese kontroll- oder ereignisgesteuerten Systeme ermöglichen einen dauerhaften Datenaustausch mit ihrer Umwelt, um zB. Daten aus Sensoren zu erhalten, diese zu verarbeiten und dann an ein anderes System weiterzugeben. Die in einem solchen System ablaufenden Reaktionen sind oft sehr komplex und darum schwer nachzuvollziehen, was eine Analyse bzw. Spezifikation und Implementierung zu einer schweren Aufgabe macht. So muss der Spezifikateur des Systems genau angeben, welches Ereignis wann auftreten darf bzw. muss und wie genau das System darauf zu reagieren hat. Ein solches Ereignis wird *Message-Event* genannt. Zudem muss ein reaktives System meist in Echtzeit arbeiten, um beispielsweise auf Notfälle oder zeitkritische Benutzereingaben schnell reagieren zu können. Ein reaktives System muss allerdings nicht unbedingt nur mit anderen Systemen interagieren, sondern oft besteht es selbst aus weiteren Komponenten, die untereinander kommunizieren und so eine Reaktivität untereinander entwickeln.
Reaktive Systeme sind heutzutage nicht mehr wegzudenken, denn nahezu jedes alltägliche Computersystem ist ein reaktives System. Sei es eine Steuerungssoftware eines Logistikzentrums, eines Kraftwerks oder einer Fabrik, die auf Sensoren für Kapazität, Temperatur oder Fehlerzustände reagieren muss, oder einfache Benutzeranwendungen wie Texteditoren oder Computerspiele, die auf Benutzereingaben reagieren müssen. Weitere Einsatzbereiche der Konzepte von reaktiven Systemen und dazugehörige Formalismen werden von Harel et al. [HFKH07] erläutert. Abbildung 2.1 zeigt eine schematische Darstellung eines Logistik-Systems, bestehend aus den Komponenten „Terminal", „LogisticSystem", „Robot 1" und „Robot 2". Dieses Schema wird im weiteren Verlauf wieder verwendet werden. Über das Terminal lässt sich das LogisticSystem be-

fehligen, welches zwei Roboter für den Transport von Gegenständen kontrolliert. Der Pfeil symbolisiert ein Message-Event vom Terminal zum LogisticSystem.

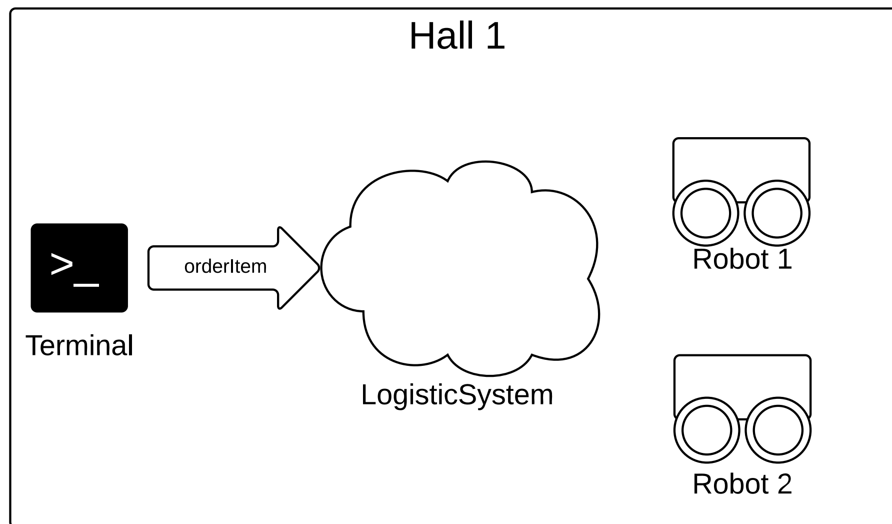

Abbildung 2.1.: Eine schematische Darstellung eines beispielhaften Logistik-Systems, bestehend aus den Komponenten „Terminal“, „LogisticSystem“, „Robot 1“ und „Robot 2“.

2.2. Modal Sequence Diagram (MSD)

Modal Sequence Diagrams sind eine von David Harel und Shahar Maoz (siehe [HM08]) entwickelte Erweiterung der UML 2.0 *Sequence Diagrams* (siehe UML 2.0 Superstructure Specification [UML05]) um Eigenschaften der *Live Sequence Charts* (siehe [DH01]) wie der Unterscheidung zwischen *existenziellen* und *universellen* MSDs und der multimodalen Natur der LSCs. In ihrem Artikel beschreiben Harel und Maoz ihre Anpassung der Operatoren *assert* und *negate*, die im UML 2.0 Standard neu hinzukamen, um benötigtes und verbotenes Verhalten spezifizieren zu können. Diese werden nun nicht mehr als Operatoren, sondern als reine Modalitäten betrachtet. MSDs sollen so mehr Aussagekraft und eine genauere Semantik in die UML 2.0 Sequence Diagrams bringen und

sie zudem zur formalen Verifizierung, Synthese und zur szenariobasierten Ausführung befähigen. In dieser Arbeit werden nur universelle MSDs betrachtet, darum werden auch nur diese im Weiteren beschrieben. Die folgenden Abschnitte erklären die Syntax und die Semantik der Modal Sequence Diagrams.

2.2.1. Syntax

Die Syntax eines MSDs ist von rein grafischer Natur. Das Diagramm hat, wie in Abbildung 2.2 einen durchgängigen Rahmen, der anzeigt, dass es sich um ein universelles MSD handelt. Darin befindet sich in der oberen linken Ecke ein Feld für den Namen des Diagramms. Innerhalb des Diagramms kann dann eine Nachrichtensequenz zwischen Objekten dargestellt werden. Die Objekte, um die es dabei geht, sind aus einem vorher definierten Objektsystem zu wählen. Abbildung 2.3 zeigt ein solches Objektsystem für das zuvor beschriebene Logistik-System. Die Kreise symbolisieren Objekte mit der Eigenschaft *name*:*Class*. Dabei steht *name* für den Namen und *Class* für die Klasse, die dieses Objekt instanziiert. Diese Objekte können in einer Software Instanzen von Klassen bzw. im realen Raum die tatsächlich existierenden Objekte, wie beispielsweise einen Lager-Roboter, symbolisieren. Zusammen ergeben diese Objekte ein System, in dem sie untereinander Nachrichten austauschen können. Diese Nachrichten heißen Message-Events. Dieses Objektsystem beschreibt also das System aus dem Abschnitt über reaktive Systeme, das aus einem Terminal für die Benutzereingabe, dem LogisticSystem zur Verwaltung des Lagers und zwei Robotern für den Transport von Gegenständen besteht. Das Terminal sendet dem LogisticSystem gerade das Message-Event „orderItem“. Um dieses Verhalten in MSDs zu modellieren, benötigen wir als erstes die Repräsentanten der Objekte, die *Lifelines*.

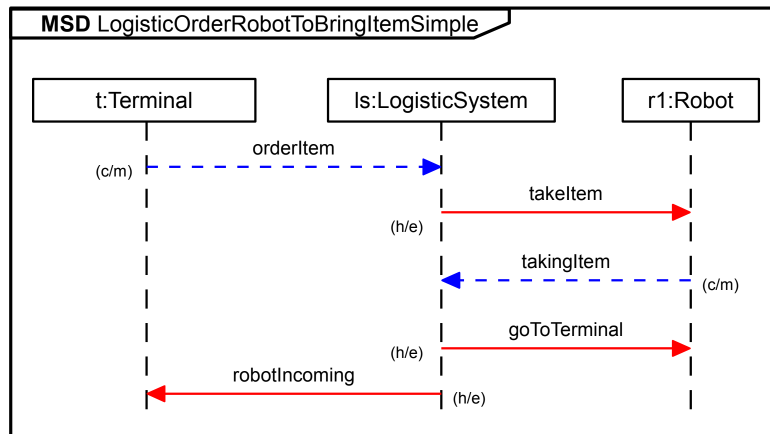

Abbildung 2.2.: Ein beispielhaftes MSD, welches in einem Logistik-System vorkommen kann

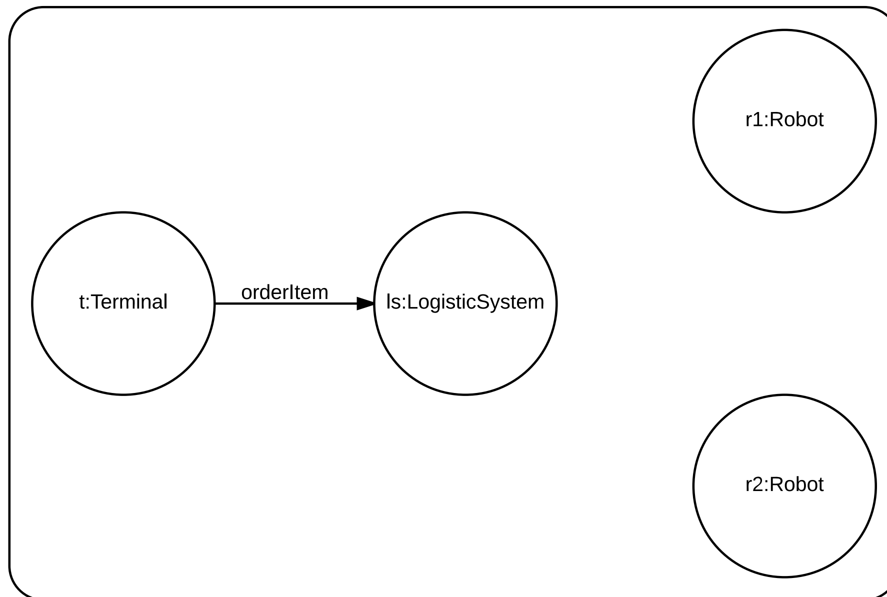

Abbildung 2.3.: Ein Beispiel eines Objektsystems zur Modellierung des Logistik-Systems

Lifelines

Die Lifeline repräsentiert ein Objekt, das aktiv an einem Szenario teilnimmt. Das Objekt wird durch die Lifeline als vertikal von oben nach unten verlaufende Linie dargestellt. Über dieser Lifeline befindet sich ein Rechteck mit dem Namen des Objekts und dessen Klasse, getrennt von einem Doppelpunkt. Ein MSD kann beliebig viele Lifelines beinhalten, jedoch gehört immer nur eine Lifeline zu einem Objekt des Objektsystems. Die Lifelines fungieren wie ein Zeitstrahl: Die Zeit verläuft dabei von oben nach unten. Zwischen den Lifelines können nun Nachrichten, die sogenannten *Diagram-Messages* eingezeichnet werden. Messages, die weiter oben zwischen Lifelines eingezeichnet sind, bezeichnen also Nachrichten, die vor denen auftreten müssen, die darunter eingezeichnet sind.

Diagram Message

Ein Message-Event in einem reaktiven System kann in einem MSD mit einer Diagram-Message beschrieben werden. Eine Diagram-Message wird durch eine Linie mit einem Pfeilende dargestellt. Die Linie beginnt bei der Lifeline des sendenden Objekts und endet mit der Pfeilspitze auf der Lifeline des empfangenen Objekts. Die Diagram-Message hat zudem zwei weitere Modalitäten: Die *temperature* und die *execution kind*. Die temperature einer Diagram-Message kann entweder *cold* oder *hot* sein (siehe Abb. 2.4). Den Temperaturen sind im Diagramm Farben zugeordnet: Blau steht für cold und Rot steht für hot. Die execution kind wird in der Beschaffenheit des Pfeils ausgedrückt: Eine durchgezogene Linie beschreibt eine Nachricht mit der Modalität executed, wohingegen eine gestrichelte Linie die Modalität monitored beschreibt. Wenn eine Nachricht die Modalität hot hat, darf keine andere Nachricht auftreten, die im MSD davor oder danach eingetragen ist. Ist die Modalität *cold* und eine andere Nachricht tritt auf, so wird das MSD an dieser Stelle abgebrochen. Die execution kind einer Nachricht kann entweder *executed* oder *monitored* sein. An einem Punkt im MSD, wo eine Nachricht erwartet wird, die executed ist, muss diese auch auftreten. Wird sie jedoch monitored erwartet, kann sie auftreten, muss es aber nicht.

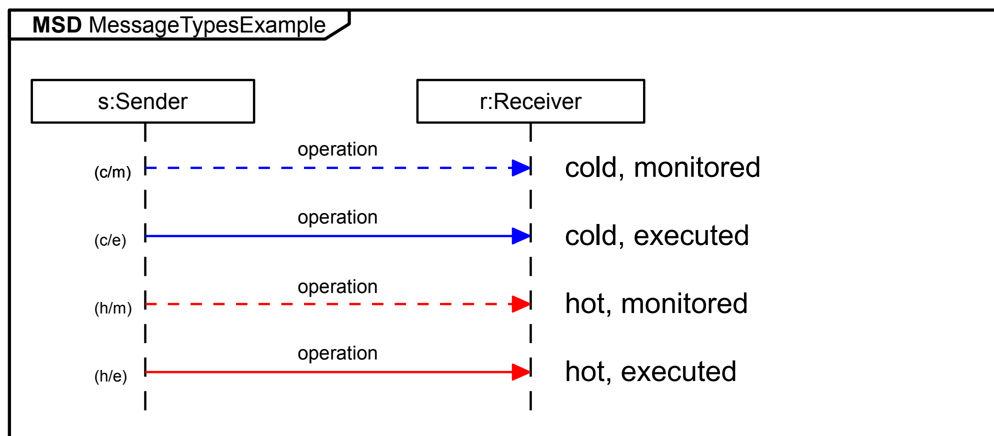

Abbildung 2.4.: Eine grafische Darstellung von Nachrichten mit verschiedenen Modalitäten. Rote Nachrichten sind hot (h), blaue Nachrichten sind cold (c). Gestrichelte Linien bedeuten, dass die Nachricht monitored (m) ist, durchgezogene Linien beschreiben eine Nachricht, die executed (e) ist.

Des Weiteren besteht in MSDs die Möglichkeit, Alternativen, Schleifen und

parallel gesendete Nachrichten in die Diagramme einzutragen. Diese und weitere Besonderheiten werden in den folgenden Abschnitten erläutert.

Alternative

Alternativen beschreiben mehrere mögliche Abläufe in einem Diagramm. Eine solche Alternative besteht in einem dafür gekennzeichneten Bereich im Diagramm (siehe Abb. 2.5). Der horizontale Strich in dem Alt-Block beschreibt eine Trennlinie, an der die zweite mögliche Abfolge beginnt. Es kann hier also entweder die Abfolge oberhalb oder unterhalb dieser Trennlinie passieren. Beide Möglichkeiten sind für das modellierte System akzeptabel. Es können hier allerdings nicht bloß zwei Möglichkeiten modelliert werden, sondern beliebig viele.

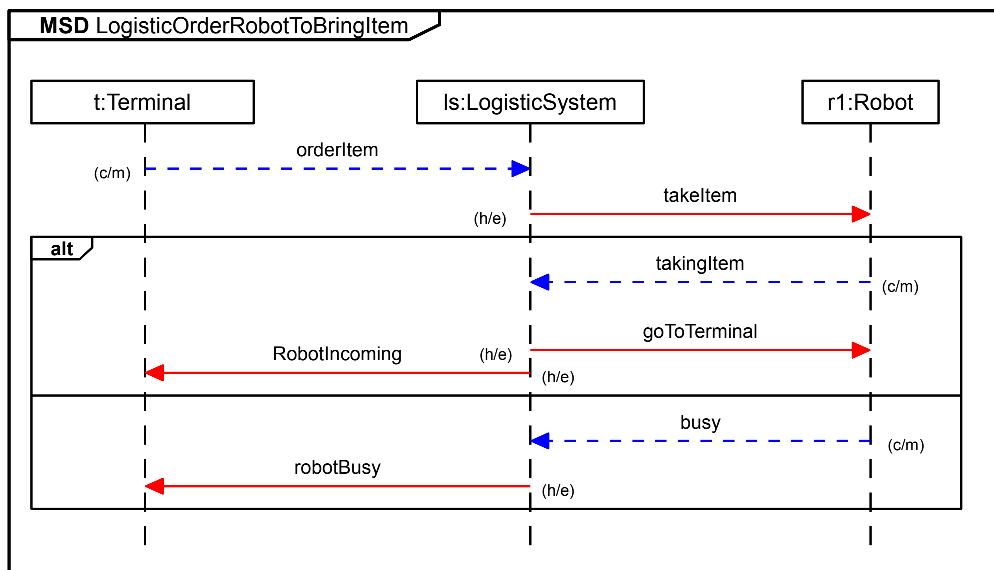

Abbildung 2.5.: Ein beispielhaftes Szenario als MSD mit einer Alternative

Schleife

Schleifen werden ähnlich wie Alternativen mit einem Bereich markiert (siehe Abb. 2.6). Der Nachrichtenverlauf in diesem Loop-Block kann sich beliebig oft wiederholen, bevor der Rest des Diagramms ausgeführt wird. Dies kann zum Beispiel genutzt werden um eine Reihe Benutzereingaben unbekannter Länge zu modellieren.

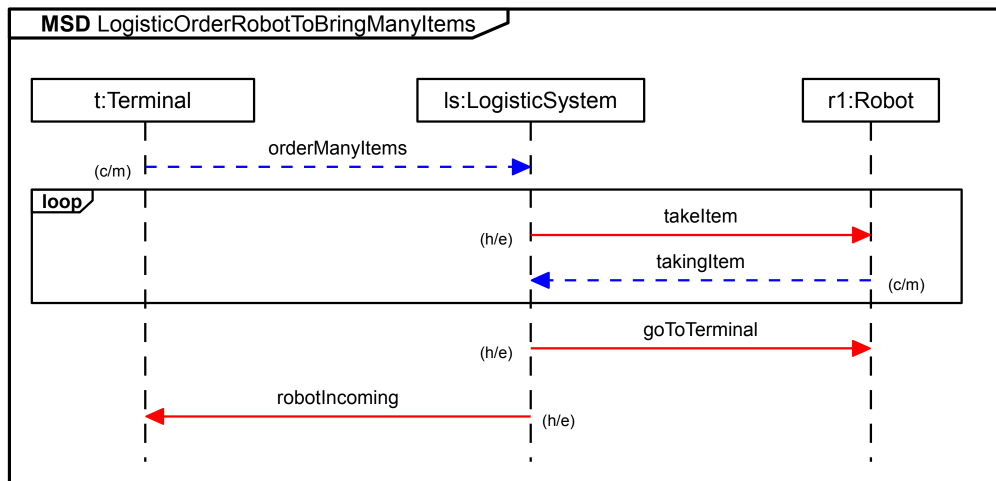

Abbildung 2.6.: Ein beispielhaftes Szenario als MSD mit einer Schleife

Parallele Abfolgen

Parallele Abfolgen sind etwas spezieller, wenn auch wieder darstellungsgleich mit der Alternative. Wenn zwei Nachrichten in einem Parallel-Block modelliert werden, bedeutet dies, dass beide gleichzeitig bzw. in beliebiger Reihenfolge auftreten können. Abbildung 2.7 zeigt ein MSD mit zwei Nachrichten, die parallelisiert wurden. Hier darf die Nachricht „robotIncoming“ entweder vor oder nach „communicate“ auftreten.

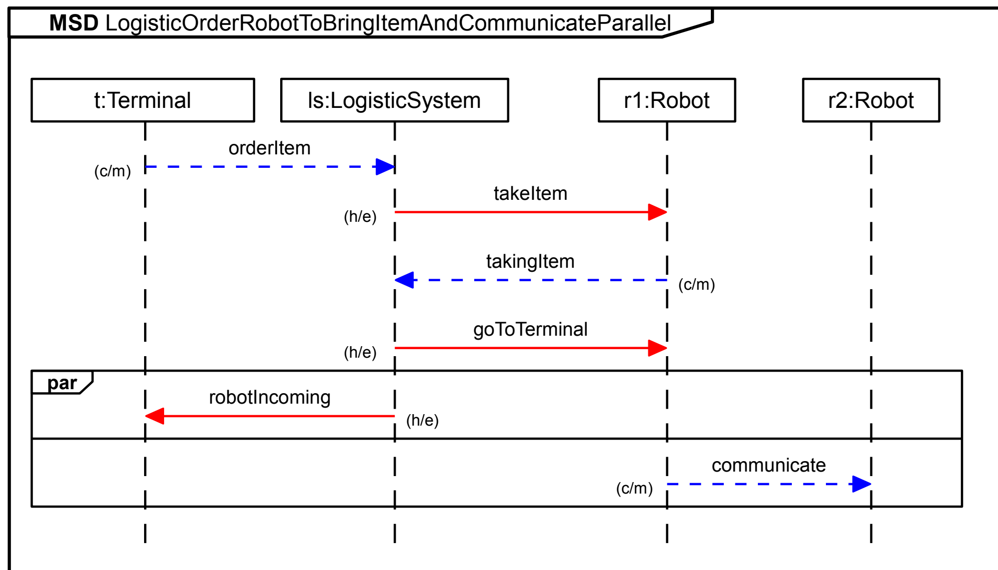

Abbildung 2.7.: Ein beispielhaftes Szenario als MSD mit paralleler Abfolge

Kondition

Des Weiteren gibt es nun die Möglichkeit, Konditionen in die Diagramme einzubinden. Diese Konditionen sind Ausdrücke, die sich auf vorher festgelegte Daten beziehen und zu wahr oder falsch ausgewertet werden können. Diese Ausdrücke sind Teil der Object Constraint Language (OCL) (siehe [OCL]), welche allein zur Auswertung von Ausdrücken verwendet werden kann, es können keine Änderungen irgendwelcher Art am Modell selbst vorgenommen werden. Zum Beispiel kann auf eine frühere Variable, die in einer Nachricht gesendet wurde, zugegriffen und deren Wert mit einem booleschen Operator ausgewertet werden. Das Beispiel in Abb. 2.8 zeigt ein MSD mit boolescher Kondition, in dem überprüft wird, ob ein Roboter noch Platz hat, um weitere Gegenstände aufzunehmen.

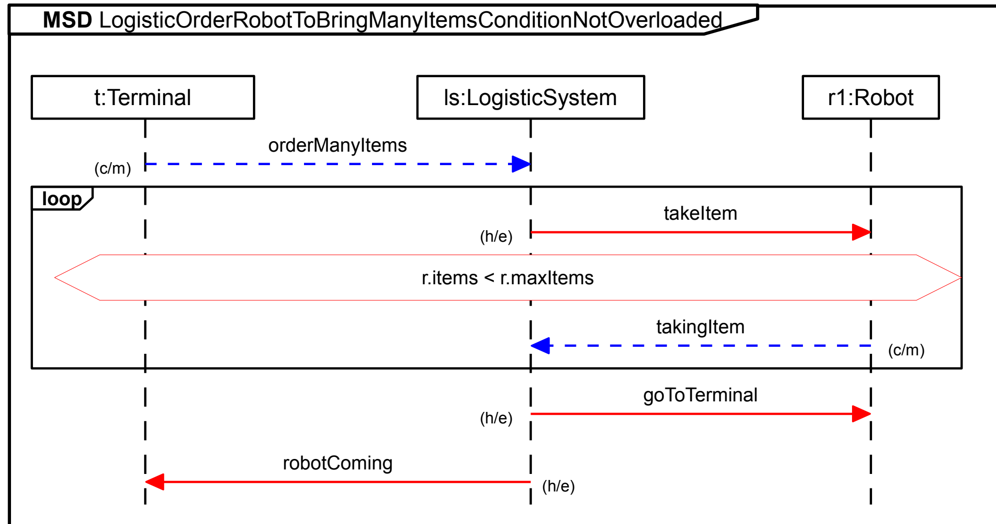

Abbildung 2.8.: Ein beispielhaftes Szenario als MSD mit boolescher Kondition

Es gibt außerdem einen Sonderfall der Kondition: Die Sync-Condition. Nachrichten können in besonderen Fällen eine beliebige Reihenfolge aufweisen, wenn die Nachrichten zwischen voneinander unabhängigen Lifelines gesendet werden (siehe Abb. 2.9). Die Nachrichten „robotIncoming“ und „communicate“ werden zwischen voneinander unabhängigen Lifelines gesendet und die Reihenfolge ist darum unklar. Also wird nun wie bei einem Parallel-Block während der Laufzeit entschieden, welche Nachricht als Erstes erscheint. Beide Möglichkeiten sind von dem System akzeptiert. Sollte ein solches Verhalten in dem Modell nicht erwünscht sein, so hilft die Sync-Condition (siehe Abb. 2.10). So kann sichergestellt werden, dass eine bestimmte Reihenfolge eingehalten wird.

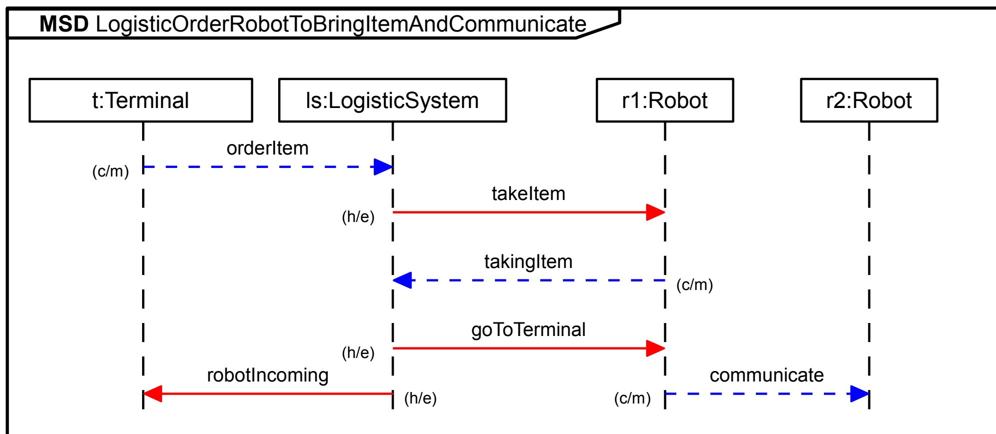

Abbildung 2.9.: Ein beispielhaftes Szenario als MSD mit zufälliger Nachrichtenreihenfolge

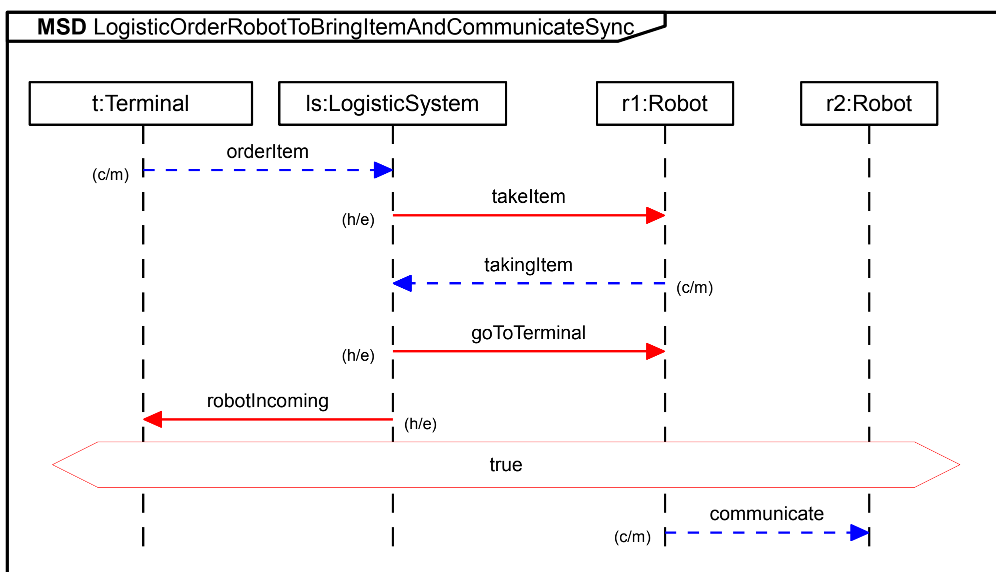

Abbildung 2.10.: Ein beispielhaftes Szenario als MSD mit einer Sync-Condition

2.2.2. Semantik

Dieser Abschnitt nimmt die Semantik der Modal Sequence Diagrams etwas präziser unter die Lupe und erklärt deren Funktionsweise im Detail.

Ablauf eines MSDs

Wie bereits im vorherigen Abschnitt erwähnt, erfüllen die Lifelines auch den Zweck eines Zeitstrahls, wodurch die Diagram-Messages einer gewissen Reihenfolge unterliegen. Die oberste Nachricht ist die erste, die das MSD erwartet. Sie ist die initialisierende Nachricht. Tritt diese Nachricht auf, wird eine neue Instanz des MSD erzeugt, ein *aktives MSD*. Diese neue Instanz ist dann in einem Zustand, in dem es die erste Nachricht überwunden hat und eine oder mehrere weitere Nachrichten erwartet. Diese erwarteten Nachrichten heißen dann *enabled*, wohingegen alle anderen Nachrichten *disabled* sind. Welche Nachrichten enabled sind, bestimmt der *Cut*, der Zustand des MSD.

Der Cut

Der Cut beschreibt den Zustand eines aktiven MSDs. Er wird als gedachte horizontale Linie in das Diagramm eingezeichnet und liegt zwischen zwei Messages. Der Cut bestimmt die Nachrichten, die für das aktive MSD enabled sind. Dies sind die Nachrichten direkt unter dem Cut. In Abbildung 2.11 sind mehrere Zustände eines MSD eingezeichnet:

- **inactive**: Dieser Cut zeigt den Zustand eines nicht initialisierten MSD. Es befindet sich noch im Startzustand.
- **1. (h/e)**: Dieser Cut beschreibt ein gerade initialisiertes MSD. Es ist aus dem Startzustand herausgetreten und wurde damit initialisiert. Nun ist die Nachricht „takeItem“ enabled. Da diese (h/e) ist, hat der Cut ebenfalls die Modalitäten (h/e).
- **2. (c/m)**: Das MSD ist einen weiteren Schritt vorangekommen und erwartet nun die Nachricht „takingItem“. Diese Nachricht ist nun enabled. Der Cut besitzt nun die Modalitäten cold und monitored.
- **3. (h/e)**: Das MSD hat nun auch die Nachricht „takingItem“ empfangen, wodurch es sich nun im Zustand befindet, in dem die Nachricht „goToTerminal“ enabled ist.
- **4. (h/e)**: Das MSD befindet sich im letzten Zustand und die Nachricht „robotIncoming“ ist enabled. Tritt diese Nachricht auf, ist das MSD am Ende angelangt und wird verworfen, da es keine weiteren Nachrichten erwartet.

Der Cut spezifiziert genau, welche Nachrichten im aktuellen Zustand auftreten dürfen bzw. müssen. Alle anderen Nachrichten würden das MSD verletzen.

Sagen wir nun: Es tritt ein Message-Event auf. Wie wird entschieden, ob die Nachricht auch die ist, die das MSD erwartet? Das auftretende Message-Event muss mit der enabled Diagram-Message des aktiven MSD unifizierbar sein.

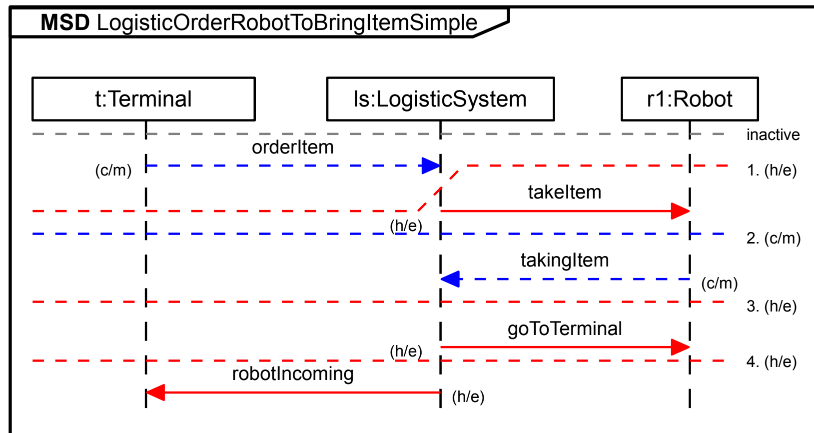

Abbildung 2.11.: Ein MSD mit eingezeichneten Zuständen (Cut)

Unifikation zweier Nachrichten

Um zu bestimmen, dass zwei Nachrichten identisch sind, müssen diese unifizierbar sein. Um dies verifizieren zu können, ist es nötig alle beteiligten Komponenten der Nachricht daraufhin zu prüfen, ob diese die selben Objekte referenzieren. Die sendende Lifeline der Diagram-Message muss also das selbe Objekt des selben Objektsystems referenzieren, wie der Sender des auftretenden Message-Events. Das gleiche gilt für den Empfänger. Als Letztes wird überprüft, ob der Inhalt des Message-Events, also die Operation, die selbe Operation des Empfängers der Diagram-Message referenziert. Sind all diese Referenzen identisch, sind die Nachrichten unifizierbar und das Message-Event löst die Diagram-Message des MSD aus. Tritt dagegen ein Message-Event auf, welches nicht mit der enabled Diagram-Message des aktiven MSDs unifizierbar ist, so ist das MSD verletzt: Es tritt eine *Violation* auf.

Wann ist ein MSD verletzt?

Ein MSD kann auf drei Arten verletzt werden. Eine der ersten zwei Verletzungen tritt auf, wenn ein anderes Ereignis auftritt, als vom MSD erwartet wird. Das Ereignis ist also an dem Cut des MSDs nicht vorgesehen:

- *Cold-Violation*: Wenn der Cut die Modalität cold trägt, also das erwartete Ereignis cold ist, tritt eine Cold-Violation auf. In diesem Fall wird das

MSD beendet und das Ereignis ist legitim. Es ist hier anzunehmen, dass dieses Systemverhalten in einem anderen MSD gehandhabt wird.

- *Safety-Violation*: Ist der Cut dagegen hot, weil das erwartete Ereignis hot ist, tritt eine Safety-Violation (Hot-Violation) auf. Ist dies der Fall, so ist ein möglicherweise schwerer Fehler aufgetreten. Das System verstößt also gegen die Spezifikation.

Darüber hinaus gibt es noch eine weitere Art der Verletzung:

- *Liveness-Violation*: Eine Liveness-Violation tritt dann auf, wenn benötigte Ereignisse oder Bedingungen nicht eintreffen, das System also nicht mehr voranschreitet. Eine solche Verletzung kann dadurch resultieren, dass ein MSD eine unendliche Folge von Nachrichten beschreibt. Wenn in dieser unendlichen Folge kein gültiges Ereignis mehr auftritt, ist die Liveness-Property des MSDs verletzt und die Liveness-Violation tritt auf.

Requirement versus Assumption

Ein MSD beschreibt also wie ein System sich zu verhalten hat. Allerdings kann es nicht nur Spielregeln für das System, sondern auch für die Umwelt aufstellen. Für diese Unterscheidung werden zwei Arten von MSDs betrachtet. Die *Requirement-MSDs* und die *Assumption-MSDs*, wie von Greenyer erläutert (siehe [Gre11]). Ein Requirement-MSD beschreibt ein Szenario, das vom System gehandhabt wird und durch ein Signal aus der Umgebung gestartet wird. Ein solches Szenario legt bestimmte Spielregeln fest, die für das gesamte System gelten müssen. Wird also ein solches MSD verletzt, ist das gleichbedeutend mit einem Fehler des Systems, also einem Problem in der Modellierung oder der Spezifikation des Systems. Ein Assumption-MSD legt Spielregeln für die Funktionsweise der Umgebung fest. Da die Umgebung aber nicht steuerbar ist, kann ihr Verhalten nur vermutet werden und es muss davon ausgegangen werden, dass diese Spielregeln auch verletzt werden können. Dieses MSD kann also verletzt werden, ohne dass der Fehler beim System liegen muss.

Vom Diagramm zum Automaten

Um den Ablauf eines MSD algorithmisch beschreiben zu können, kann es nach Harel und Maoz (siehe [HM08]) in einen Automaten übersetzt werden. So besteht der Automat für ein MSD aus einer Menge an Zuständen S und einer Menge an Transitionen T. Jeder Zustand beschreibt einen akzeptablen Cut im MSD und besitzt für jedes akzeptable Event eine Transition zu einem nächsten

Zustand. Dazu hat jeder Zustand eine weitere Transition t auf sich selbst für jedes Event im System Σ, das nicht im MSD spezifiziert ist M ($t \in \Sigma \setminus M$). Diejenigen Zustände, die keine Transitionen für Events mit der Modalität hot tragen, sind akzeptierende Zustände. Zudem gibt es noch zwei weitere Zustände: Einen akzeptierenden sc für die Cold-Violation und einen ablehnenden sh für die Safety-Violation. Jeder Zustand für einen cold Cut hat also eine Transition $t \in M \setminus mc$ zu dem akzeptierenden Zustand, wobei mc die Menge an cold Events ist. Analog hat jeder Zustand für einen hot Cut eine Transition $t \in M \setminus mh$ zum ablehnenden Zustand, wobei mh die Menge an hot Events ist. Der Startzustand ist der, der den inaktiven Cut repräsentiert. Er hat, zusätzlich zu seiner Transition zum nächsten Zustand, eine Transition auf sich selbst für alle möglichen Events im System Σ. Abbildung 2.12 zeigt einen Zustandsautomaten für das Beispiel MSD in Abbildung 2.2. Der Zustand *s0* ist hier der Startzustand, *s5* ist der Endzustand, an dem das MSD abgeschlossen ist. Alle Endzustände haben dazu noch eine Transition zu sich selbst für alle Events im System Σ.

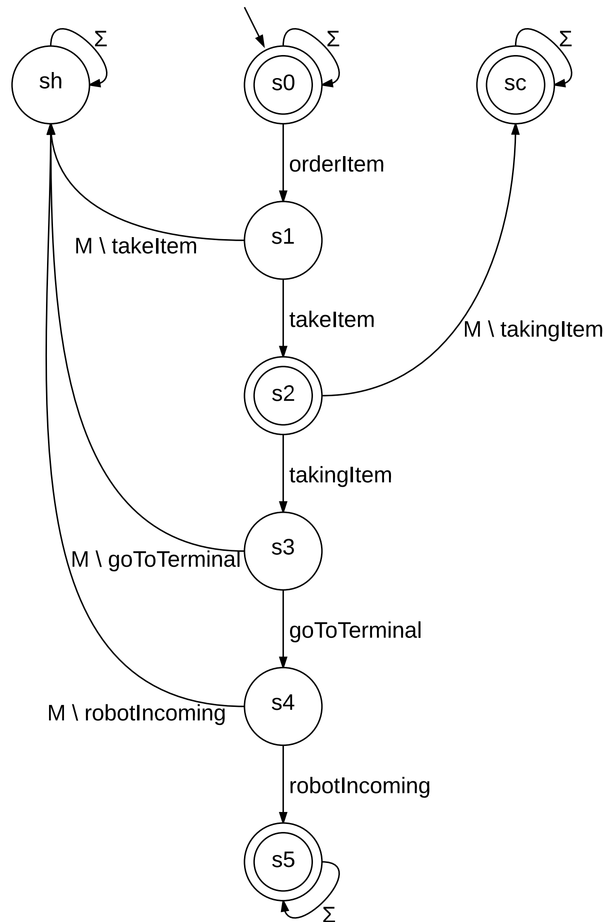

Abbildung 2.12.: Ein Zustandsautomat für das Beispiel MSD 2.2

2.3. Play-Out

Play-Out heißt ein Algorithmus, der von Harel und Marelly zuerst für Live Sequence Charts und später von Harel und Maoz (siehe [MH06]) auch für Modal Sequence Diagrams definiert wurde. Das Grundprinzip des Algorithmus sieht etwa wie folgt aus: Er erhält zuerst eine Menge an MSDs. Dann wartet er auf ein Event von der Umgebung (zB. Sensoren, nicht kontrollierbare Objekte). Tritt ein solches Event auf, werden aktive Kopien derjenigen MSDs erzeugt, die durch dieses Event initialisiert werden. In diesem Zustand wird der Algorithmus selbst

aktiv und wählt entweder nichtdeterministisch oder mit Hilfe von Benutzereingaben eine enabled Message eines aktiven MSDs, welche keine Safety-Violation verursacht und sendet diese. Dies tut der Algorithmus solange, bis:

- alle aktiven MSDs beendet sind oder
- es keine Nachricht mehr gibt, die gesendet werden kann, ohne eine Safety-Violation auszulösen. In diesem Fall terminiert der Algorithmus mit einem Fehler.

2.4. Domain-specific Language (DSL)

Eine domänenspezifische Sprache ist eine formale Sprache, die speziell für ein Problem oder Problemfeld entwickelt wurde. Eine solche Sprache ist problemspezifisch (domänenspezifisch) und ist eine Art Fachkraft für ihre Domäne. Das bedeutet, sie kann zur Lösung oder Darstellung bestimmter Probleme genutzt werden. Für andere Probleme ist sie allerdings nutzlos. Ein Beispiel für eine DSL ist die Sprache SQL. Das Gegenteil zu einer DSL ist eine universell einsetzbare Sprache wie Java oder C. Diese haben den Vorteil, für viele Problemstellungen auf diversen Fachgebieten einsetzbar zu sein, jedoch haben sie auch den Nachteil, sehr komplex und vergleichsweise schwer zu erlernen zu sein.

2.5. Xtext

Xtext (siehe [Xte]) ist ein Plugin für die Eclipse IDE und bietet Werkzeuge zum schnellen Erstellen von domänenspezifischen Sprachen (DSL). Es bietet einen Editor zur Strukturierung von DSLs und generiert dazu passende und individuell anpassbare Parser, Linkmanager, Compiler und Interpreter. Diese so erzeugten DSLs können dann direkt in Eclipse eingebunden und verwendet werden. Es kann je nach Bedarf angepasst werden, wie sich die Sprache bei Benutzereingaben verhält, wie mächtig die Autokorrektur ist, welche Sichtbarkeit zwischen Objekten herrscht und welcher Code daraus generiert werden soll.

2.6. Eclipse Modeling Framework

Das Eclipse Modeling Framework (EMF) (siehe [EMF]) bietet umfangreiche Funktionen zur Modellierung von Systemen und Software. Insbesondere bietet das EMF auch Codegeneratoren, um automatisch aus Modellen ausführbaren

Java-Code zu generieren. Für diese Arbeit besonders wichtig sind die Teilbereiche Ecore und GenModel. Die Implementierung der im laufe dieser Arbeit vorgestellten DSL wird auf Modelldefinitionen in Form von Ecore-Model Dateien zugreifen und daraus mit Hilfe von einem GenModel Code generieren.

2.6.1. Ecore-Model

Ecore bietet die Möglichkeit strukturierte Modelle zu erzeugen. Diese Modelle enthalten Paket-, sowie Klassen-, Attribut-, Operations- und Referenz-Definitionen. Abbildung 2.13 zeigt das Ecore-Model eines Logistik-Systems. Dies ist das selbe System, welches schon zuvor beschrieben wurde. Dieses Ecore-Model beinhaltet die drei Klassen des Systems (LogisticSystem, Terminal, Robot) und deren Operationen.

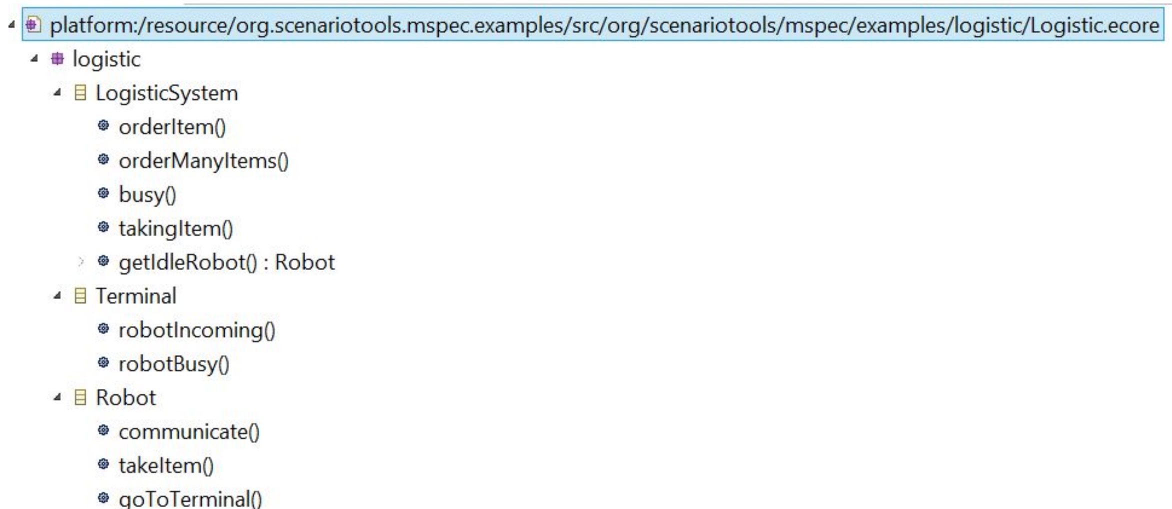

Abbildung 2.13.: Ein beispielhaftes Ecore-Model eines Logistik-Systems

2.6.2. Generator-Model

Das GenModel kann an ein Ecore-Model gebunden werden und ermöglicht es daraufhin aus diesem Ecore-Model passende Java-Dateien zu generieren. Es kann Model-, Edit-, Editor- und Test-Code generiert werden. Für die Codegenerierung der in dieser Arbeit implementierten DSL ist jedoch nur der Model-Code nötig. Das GenModel erzeugt dabei drei Pakete: *model*, *model.impl*, *model.util*. Diese beinhalten Interfaces, Implementierungen und Factory-Klassen.

3. Die Modellierungssprache MSpec

Xspec ist eine domänenspezifische Sprache (DSL) zur Modellierung und Simulation multimodaler szenariobasierter Spezifikation. Mit ihrer Hilfe ist es möglich, Objektsysteme und Interaktionen zwischen Objekten in Szenarien textuell darzustellen. Ein Dokument der Sprache MSpec hat die Dateiendung *.mspec*. Im folgenden Abschnitt wird die Syntax der Sprache allgemein und anhand von Beispielen beschrieben. Im Anschluss wird im Abschnitt 3.2 die Semantik der Sprache erläutert.

3.1. Syntax

Die Syntax der Sprache MSpec kann anhand des Codebeispiels 3.1 wie folgt erklärt werden: Als Erstes werden eine oder mehrere *Import*-Kommandos erwartet. Darauf folgt die *Specification*, in der die eigentlichen Modellierungsblöcke, die *UseCases*, liegen. In einem UseCase sind eine *Collaboration* und beliebig viele *Scenarios* vorhanden. Innerhalb dieser Scenarios kann nun der Nachrichtenaustausch zwischen den Objekten modelliert werden. Auf die einzelnen Elemente der Sprache wird im Folgenden weiter eingegangen.

```
import "Logistic.ecore"

specification LogisticSystem {

 usecase LogisticSystemOfHall1 {

  collaboration Hall1 {
   obj env Terminal t
   obj sys LogisticSystem ls
   obj sys Robot r1
   obj sys Robot r2
  }

  scenario r LogisticOrderRobotToBringItemSimple {
   msg (c,m) t -> ls.orderItem()
```

```
    msg (h,e) ls -> r1.takeItem()
    msg (c,m) r1 -> ls.takingItem()
    msg (h,e) ls -> r1.goToTerminal()
    msg (h,e) ls -> t.robotIncoming()
   }

   scenario r LogisticOrderRobotToBringManyItems {
    ...
   }

 }
}
```

Listing 3.1: Eine beispielhafte Spezifikation in MSpec

3.1.1. Import Ecore-Model

Durch einen Import ist es möglich eine Ecore-Datei zu importieren. Dem Schlüsselwort `import` wird der Pfad zu der zu importierenden Datei in Form eines Strings, also in Anführungsstrichen, nachgestellt. Wenn eine Datei importiert wurde, stehen sämtliche Klassen und Pakete innerhalb der Datei in der Collaboration zur Verfügung. Es können auch mehrere Dateien importiert werden, indem mehrere `import` Befehle nacheinander angegeben werden (siehe Codebeispiel 3.3).

```
Import ::= "import" String
```

Listing 3.2: Die Syntax eines MSpec-Imports

```
import "logistic.ecore"
import "some_other_ecore.ecore"
```

Listing 3.3: Beispiel zweier Import-Befehle

Weitere Informationen zur Import-Funktion und dem Referenzieren von Objekten in Xtext sind in diesem Blog [imp] zu finden.

3.1.2. Specification

Das Schlüsselwort `specification` initialisiert ein Specification-Element und schafft damit den Rahmen des MSpec-Dokuments. Eine Specification benötigt einen Namen in Form einer ID und öffnet einen Block, der eine beliebige Anzahl an UseCases beinhaltet. ID ist ein eindeutiger Bezeichner.

```
Specification ::= "specification" ID "{" {UseCase}* "}"
```

Listing 3.4: Die Syntax einer MSpec-Specification

3.1.3. UseCase

Ein UseCase wird mit dem Schlüsselwort `usecase` initialisiert und benötigt einen Namen. Er ist vom Use-Case in UML (siehe OMG UML-Spezifikation [UML]) inspiriert und beinhaltet Definitionen der beteiligten Objekte im modellierten System, sowie deren geplante Interaktionen. In diesem Block können zwei Strukturen erstellt werden: Das Collaboration-Element und eine beliebige Anzahl des Scenario-Elements. Dem UseCase muss eine Collaboration zugewiesen werden, auf das die Scenarios zugreifen können. Dazu können beliebig viele Abfolgen von Systemereignissen als Scenario modelliert werden.

```
UseCase ::= "usecase" ID "{" Collaboration {Scenario}* "}"
```

Listing 3.5: Die Syntax eines MSpec-UseCase

```
usecase LogisticSystem {
 collaboration Hall_1 {
  ...
 }
 scenario r Scn1{
  ...
 }
 scenario r Scn2{
 ...
 }
}
```

Listing 3.6: Beispiel eines MSpec-UseCase

Das Codebeispiel 3.6 zeigt schematisch den Aufbau eines MSpec-UseCase. Hier heißt der UseCase „LogisticSystem“ und beinhaltet eine Collaboration „Hall_1“ und die Scenarios „Scn1“ und „Scn2“.

3.1.4. Collaboration

Eine Collaboration in MSpec ähnelt ihrer Vorlage, der UML Collaboration (siehe OMG UML-Spezifikation [UML]) und wird mit dem Schlüsselwort `collaboration` initialisiert. Die Collaboration benötigt dann noch einen Namen. Innerhalb dieses Blocks können Objekte definiert werden, die für die nachfolgenden Scenarios

relevant sind. Allerdings stehen diese Objekte hier gleichermaßen für die realen Objekte im System, wie auch für die Rollen, die sie in den Scenarios spielen. Die Objekte in diesem Block können demnach direkt als Rollen in Scenarios verwendet werden. Nichtsdestotrotz können diese Rollen innerhalb einzelner Scenarios zusätzlich an andere Objekte gebunden werden (siehe dazu Abschnitt 3.1.5).
Ein Objekt kann mit den Schlüsselwörtern `obj sys` oder `obj env` erzeugt werden. Ersteres erzeugt ein steuerbares Systemobjekt und letzteres ein nicht steuerbares Umgebungsobjekt. Zum besseren Verständnis: Ein Systemobjekt ist ein Teil des Systems, welches modelliert werden soll. Es ist vollkommen anpassbar auf die Umgebung und muss so modelliert werden, dass die gewünschte Funktionalität im Rahmen bestimmter Umstände gegeben ist. Ein Umgebungsobjekt hingegen ist ein Teil der Umgebung und kann weder angepasst noch gesteuert werden. Die Umgebung ist das Umfeld, in dem das System funktionieren soll. Hierzu zählen zB. auch Sensoren des Systems, die Daten oder Signale liefern, mit denen das System arbeiten soll. Danach folgt die Klassenreferenz auf das vorher importierte Ecore-Model (siehe Abschnitt 3.1.1). Die Autovervollständigung wird hier direkt assistieren und die importierten Objekte vorschlagen. Zuletzt benötigt das neue Objekt einen einzigartigen Namen um es später identifizieren zu können. Das folgende Codebeispiel 3.8 zeigt eine Collaboration in MSpec, welche für die weiteren Beispiele in diesem Kapitel verwendet wird.

```
Collaboration ::= "collaboration" ID "{" {Object}* "}"
Object ::= ("obj sys" | "obj env") ID ID
```

Listing 3.7: Die Syntax einer MSpec-Collaboration

```
collaboration Hall_1 {
 obj env Terminal t           // User-Terminal
 obj sys LogisticSystem ls    // Control system
 obj sys Robot r1             // Transport robot 1
 obj sys Robot r2             // Transport robot 2
}
```

Listing 3.8: Beispiel einer Collaboration in MSpec

Das Codebeispiel 3.8 zeigt eine Collaboration in MSpec, welche den Namen „Hall_1“ trägt. Sie beinhaltet vier Objekte: Ein Environment-Objekt „t“ des Typs „Terminal“, ein System-Objekt „ls“ des Typs „LogisticSystem“ und zwei System-Objekte „r1“ und „r2“ des Typs „Robot“. Die Typen stammen aus dem vorher importierten Ecore-Model „logistic.ecore“.

3.1.5. Scenario

Das Scenario ist das Herzstück der Sprache, hier werden die Systemabläufe modelliert. Ein Scenario beschreibt die akzeptablen Abläufe innerhalb eines Systems, indem es einen Nachrichtenaustausch modelliert. Seinem Vorbild, dem hier betrachteten Modal Sequence Diagram, entsprechend, gibt es auch in MSpec Message-Events, Alternativen, Schleifen, parallele Abläufe und Konditionen. Diese Operationen werden in MSpec unter dem Begriff Scenario-Feature zusammengefasst. Das Codebeispiel 3.10 zeigt ein beispielhaftes MSpec-Scenario. Die Bedeutung und Verwendungsweise der Elemente des MSpec Scenario werden im weiteren Verlauf genauer erläutert.
Das MSpec-Scenario wird durch das Schlüsselwort `scenario` initialisiert. Darauf muss ein `r` für Requirement oder ein `a` für Assumption folgen. Dieser Parameter bestimmt, ob das Scenario ein Requirement- oder ein Assumption-MSD darstellt.
Innerhalb des Scenario-Blocks können zuerst Rolebindings angewendet werden, mit denen Rollen der Collaboration modifiziert werden können. Danach können Scenario-Features angegeben werden.

```
Scenario ::= "scenario" type ID "{" {RoleBinding}* {
   ScenarioFeature}* "}"
type ::= ("r" | "a")
RoleBinding ::= "bind" ID "to" String
ScenarioFeature ::= (Message | Loop | Alternative | Parallel |
   Condition | Sync)
Message ::= "msg" "(" ("h" | "c") "," ("m" | "e") ")" ID "->" ID
    "." ID "(" {String}* ")"
Loop ::= "loop" "{" {ScenarioFeature}* "}"
Alternative ::= "alt" "{" {Case}* "}"
Case ::= "case" "{" {ScenarioFeature}* "}"
Parallel ::= "par" "{" {Case}* "}"
Condition ::= "cond" "(" ("h" | "c") ")" String
Sync ::= "sync" ID {"," ID}+
```

Listing 3.9: Die Syntax eines MSpec-Scenario

```
scenario r LogisticOrderRobotToBringItemSimple{
 // Some RoleBindings
 ...
 // Some Message-Events
 msg (c,m) t -> ls.orderItem()
 msg (h,e) ls -> r1.takeItem()
 msg (c,m) r1 -> ls.takingItem()
 msg (h,e) ls -> r1.goToTerminal()
 msg (h,e) ls -> t.robotIncoming()
```

```
}
```

Listing 3.10: Beispiel eines MSpec-Scenario

Rolebinding

Rolebinding bedeutet, dass einem Objekt eine spezifische Zugehörigkeit gegeben wird. Deklarierte Objekte in der Collaboration sind als dem System zugehörige Komponenten und damit auch als eben diese Rollen in den Scenarios definiert. Soll aber in einem Scenario die Rollenzugehörigkeit angepasst werden, so kann ein Rolebinding definiert werden. Mit dem Schlüsselwort `bind` wird ein solches Rolebinding initialisiert. Dem Schlüsselwort wird dann die zu modifizierende Rolle nachgestellt. Das Schlüsselwort `to` dient als Trennwort, welches zur neuen Rollenbindung führt. Es kann nun ein in der Collaboration deklariertes Objekt und dessen Operation gewählt werden, an die die Rolle für dieses Scenario gebunden sein soll. Solange eine Rolle nicht durch dieses Kommando neu gebunden wird, bleibt ihre Bindung zum unabhängigen Objekt bestehen. Eine Rollenbindung könnte zum Beispiel so aussehen:

```
scenario r RoleBinding{
 bind r1 to ls.getIdleRobot
 ...
}
```

Listing 3.11: Codebeispiel zum Rolebinding

Die Rolle „r1“ (Robot) wird für dieses Scenario an die Ausgabe der Operation getIdleRobot des Objekts „ls“ (LogisticSystem) gebunden. Dies funktioniert allerdings nur dann, wenn die Operation auch den gleichen Typ zurückgibt, der für die Rolle nötig ist.

Message-Events

Das Message-Event entspricht der Diagram-Message eines Modal Sequence Diagrams und wird mit dem Schlüsselwort `msg` initialisiert. Darauf folgen in runden Klammern die Modalitäten, mit einem Komma getrennt. Für cold steht das Schlüsselwort `c`, für hot das Schlüsselwort `h`. Nach dem Komma folgt ein `m` für monitored oder ein `e` für executed. Anschließend wird der Sender aus den in der Collaboration eingetragenen Objekten ausgewählt und diesem ein Pfeil (->) nachgestellt. Dieser Pfeil zeigt symbolisch auf den Empfänger der Nachricht, der hinter dem Pfeil steht und ebenfalls aus den Objekten der Collaboration ausgesucht wird. Die aufgerufene Operation wird aus den Operationen des mit dem Objekt verknüpften Ecore-Models ausgesucht und mit Hilfe eines Punktes

mit dem Empfänger verbunden. Zuletzt wird der Ausdruck mit Klammern für die Parameter beendet.

```
msg (c,m) s -> r.operation() // cold, monitored
msg (c,e) s -> r.operation() // cold, executed
msg (h,m) s -> r.operation() // hot, monitored
msg (h,e) s -> r.operation() // hot, executed
```

Listing 3.12: Codebeispiel zu den verschiedenen Message-Events in MSpec

Das Codebeispiel 3.12 zeigt die unterschiedlichen Message-Events in MSpec. Diese Darstellung entspricht der Darstellung der Nachrichtentypen in MSDs im Grundlagenkapitel (siehe Abb. 2.4).

Alternative

Die Alternative beschreibt den Sachverhalt, dass mehrere Abläufe gültig sind. Ist dies der Fall, kann ein Alt-Block mit dem Schlüsselwort `alt` geöffnet werden. Innerhalb dieses Blocks können beliebig viele alternativen Abläufe modelliert werden. Dafür wird mit dem Schlüsselwort `case` ein neuer Fall hinzugefügt. In jedem dieser Case-Blöcke kann dann ein Ablauf modelliert werden, wie im Scenario-Block. Es können außerdem weitere Elemente verschachtelt werden. So kann in einem Case-Block zum Beispiel ein weiterer Alt-Block, Par-Block oder ein Loop-Block stehen. Innerhalb dieses Blocks können also alle Funktionen des MSpec-Scenario, mit Ausnahme der Rolebindings, verwendet werden.

```
scenario r LogisticOrderRobotToBringItem {
 msg (c,m) t -> ls.orderItem()
 msg (h,e) ls -> r1.takeItem()
 alt {
  case {
   msg (c,m) r1 -> ls.takingItem()
   msg (h,e) ls -> r1.goToTerminal()
   msg (h,e) ls -> t.robotIncoming()
  }
  case {
   msg (c,m) r1 -> ls.busy()
   msg (h,e) ls -> t.robotBusy()
  }
 }
}
```

Listing 3.13: Codebeispiel zur Alternative in MSpec

Dieses Beispiel entspricht dem MSD LogisticOrderRobotToBringItem 2.5 aus dem Grundlagenkapitel zur Alternative in MSDs (siehe Abschnitt 2.2.1). Dieses

Scenario wird durch die Nachricht `t -> ls.orderItem` initialisiert und erwartet danach die Nachricht `ls -> r1.takeItem`. Nach dieser können entweder die Nachrichten der ersten oder die der zweiten Case-Anweisung auftreten. Es sind also die Nachrichten `r1 -> ls.takingItem` und `r1 -> ls.busy` aktiv.

Loop

Ein Loop-Block kann mit dem Schlüsselwort `loop` geöffnet werden und ermöglicht es, einen Fall zu modellieren, in dem eine oder mehrere Nachrichten mehrfach hintereinander auftauchen können. Dies kann zum Beispiel bei einer Benutzereingabe oder bei einer Warteschlange nützlich sein. Innerhalb dieses Blocks können beliebig viele Nachrichten, sowie Alt-, Par- oder weitere Loop-Blöcke erzeugt werden. Das nachfolgende Beispiel zeigt ein MSpec-Scenario, das einen Loop beinhaltet. Es entspricht außerdem dem MSD LogisticOrderRobotToBringManyItems 2.6 aus dem Grundlagenkapitel zu Schleifen in MSDs (siehe Abschnitt 2.2.1).

```
scenario r LogisticOrderRobotToBringManyItems {
 msg (c,m) t -> ls.orderManyItems()
 loop {
  msg (h,e) ls -> r1.takeItem()
  msg (c,m) r1 -> ls.takingItem()
 }
 msg (h,e) ls -> r1.goToTerminal()
 msg (h,e) ls -> t.robotIncoming()
}
```

Listing 3.14: Codebeispiel zum Loop in MSpec

Dieses Scenario wird durch die Nachricht `t -> ls.orderManyItems` initialisiert und geht dann direkt in den Loop hinein. Nun können die Nachrichten innerhalb des Loops beliebig oft nacheinander auftreten, solange bis die Nachricht `ls -> r1.goToTerminal` auftritt.

Parallel

Wenn es nötig sein sollte, dass mehrere Nachrichten in beliebiger Reihenfolge ausgeführt werden können, hilft das Parallel-Kommando. Mit dem Schlüsselwort `par` kann ein Parallel-Block geöffnet werden. Innerhalb dieses Blocks können, wie bei der Alternative, mehrere Case-Anweisungen mit dem Schlüsselwort `case` geöffnet werden. Jeder dieser Case-Blöcke enthält dann Nachrichten, die parallel ablaufen können. Das bedeutet, dass zwar innerhalb eines Case-Blocks die Nachrichten eine feste Reihenfolge haben, jedoch können die Nachrichten der verschiedenen Case-Blöcke in beliebiger Reihenfolge auftreten. In

den Case-Blöcken können außerdem weitere Scenario-Features verwendet werden. Im nachfolgenden Beispiel wird eine solche parallele Abfolge dargestellt. Dieses Beispiel ist äquivalent zum Beispiel der parallelen Abfolge in MSDs im Grundlagenkapitel (siehe Abschnitt 2.2.1 bzw. Abb. 2.7).

```
scenario r LogisticOrderRobotToBringItemAndCommunicateParallel {
        msg (c,m) t -> ls.orderItem()
        msg (h,e) ls -> r1.takeItem()
        msg (c,m) r1 -> ls.takingItem()
        msg (h,e) ls -> r1.goToTerminal()
        par {
                case {
                        msg (h,e) ls -> t.robotIncoming()
                }
                case {
                        msg (c,m) r1 -> r2.communicate()
                }
        }
}
```

Listing 3.15: Codebeispiel zur parallelen Abfolge in MSpec

Das Codebeispiel 3.15 zeigt ein Scenario in MSpec, welches durch die Nachricht `t -> ls.orderItem` initialisiert wird und einen parallelen Nachrichtenverlauf beinhaltet. So können nach der Nachricht `ls -> r1.goToTerminal` diese zwei Reihenfolgen auftreten:

```
ls -> t.robotIncoming
r1 -> r2.communicate
```

```
r1 -> r2.communicate
ls -> t.robotIncoming
```

An diesem Punkt sind also die Nachrichten `ls -> t.robotIncoming` und `r1 -> r2.communicate` aktiviert.

Condition

Eine Condition ist ein boolescher Ausdruck innerhalb eines Scenarios bzw. eines MSDs. Er wird zur Laufzeit zu wahr oder falsch ausgewertet und bietet so eine spezielle Art, das System zu modellieren. So kann zum Beispiel eine vorher festgelegte Variable später auf Gültigkeit überprüft werden. Eine Condition kann entweder hot oder cold sein. Im nachfolgenden Beispiel wird das zuvor gezeigte Schleifen-Szenario so modifiziert, dass in der Schleife geprüft wird, ob der Roboter überhaupt noch Platz hat, um weitere Gegenstände aufzunehmen.

Dieses Beispiel ist gleichbedeutend mit dem aus dem Grundlagenkapitel zur booleschen Kondition (siehe Abschnitt 2.2.1 bzw. Abb. 2.8).

```
scenario r
   LogisticOrderRobotToBringManyItemsConditionNotOverloaded {
 msg (c,m) t -> ls.orderManyItems()
 loop {
  msg (h,e) ls -> r1.takeItem()
  cond (c) "r1.getItems() < r1.getMaxItems()"
  msg (c,m) r1 -> ls.takingItem()
 }
 msg (h,e) ls -> r1.goToTerminal()
 msg (h,e) ls -> t.robotIncoming()
}
```

Listing 3.16: Codebeispiel zur booleschen Kondition in MSpec

Dieses Codebeispiel ist identisch zu dem Beispiel zum Loop, mit der Ausnahme, dass eine Condition `"r1.getItems()< r1.getMaxItems()"` hinzugefügt wurde. Diese besagt, dass das Scenario an der Stelle nur dann fortfahren darf, wenn die getragenen Gegenstände des Roboters nicht seine Maximalkapazität überschreiten, ansonsten tritt eine Cold-Violation auf.

Sync

Die Sync-Condition ist eine spezielle Art der Condition und wird verwendet, um bestimmte Lifelines zu synchronisieren. Das kann nötig sein, wenn Nachrichten modelliert werden, die voneinander unabhängige Sender und Empfänger haben. Dann kann der Fall eintreten, der im Grundlagenkapitel beschrieben wurde (siehe 2.9), in dem es nicht ersichtlich ist, welche Nachricht zuerst auftritt. Diesen Fall darf es in MSpec nicht geben, weil das Scenario dann für das Play-Out nicht eindeutig genug spezifiziert ist. Außerdem soll eine MSpec-Spezifikation das Verhalten eines Systems möglichst genau modellieren können, weshalb eine solche Ungenauigkeit nicht auftreten darf. Für diesen Fall gibt es extra zwei Konzepte in MSpec, um hier die Genauigkeit zu maximieren: Es kann entweder ein Parallel-Block, wie bereits vorher beschreiben, oder eine Sync-Condition benutzt werden. Soll also die unbestimmte Abfolge modelliert werden, ist ein Parallel-Block zu verwenden, soll aber eine bestimmte Reihenfolge festgelegt werden, müssen die Lifelines synchronisiert werden. Eine Nachricht innerhalb eines Blocks (Loop, Alt, Par) kann allerdings keine Asynchronität mit einer Nachricht außerhalb des Blocks erfahren, da diese Blöcke automatisch eine Synchronisation über alle Lifelines erzwingen. Das nachfolgende Beispiel zeigt eine Möglichkeit, das Beispiel aus dem Grundlagenkapitel zur Synchronisation (siehe Abb. 2.10) in MSpec darzustellen.

```
scenario r LogisticOrderRobotToBringItemAndCommunicateSync {
 msg (c,m) t -> ls.orderItem()
 msg (h,e) ls -> r1.takeItem()
 msg (c,m) r1 -> ls.takingItem()
 msg (h,e) ls -> r1.goToTerminal()
 msg (h,e) ls -> t.robotIncoming()
 sync ls, r1
 msg (c,m) r1 -> r2.communicate()
}
```

Listing 3.17: Codebeispiel zur Sync-Condition in MSpec

Dieses Scenario wird wieder durch die Nachricht `t -> ls.orderItem` initialisiert und erwartet die Nachrichten `ls -> r1.takeItem`, `r1 -> ls.takingItem` und `ls -> r1.goToTerminal`. Danach würden zwei Nachrichten folgen, die in MSDs eine unbestimmte Reihenfolge hätten. Hier wurde entschieden, dass die Nachrichten eine bestimmte Reihenfolge haben sollen, also wurde die Sync-Condition `sync ls, r1` zwischen den Nachrichten eingefügt.

Kommentare

Die Funktion, Kommentare in den Code zu schreiben, wird bereits automatisch von Xtext bereitgestellt und funktioniert genauso wie in Java. Um eine einzelne Kommentar-Zeile zu erzeugen, werden dieser Zeile zwei Schrägstriche `//` vorangestellt. Um einen ganzen Kommentar-Block zu erzeugen, wird ein Schrägstrich, gefolgt von einem *-Zeichen eingegeben. Die Autovervollständigung ergänzt automatisch das schließende Zeichen `*/`.
Ein Beispiel:

```
// Kommentar-Zeile

/*
 * Kommentar-Block
 */
```

Listing 3.18: Codebeispiel zu Kommentaren in MSpec

3.2. Semantik

Dieser Abschnitt beschreibt die Semantik der MSpec-Elemente noch einmal präziser, indem es diese mit den Konzepten der Modal Sequence Diagrams in Verbindung bringt und Äquivalenzen definiert.

3.2.1. Ablauf eines MSpec-Scenarios

Der Ablauf eines MSpec-Scenarios gestaltet sich aufgrund der Natur der textuellen Darstellung und der Voraussetzung der Äquivalenz mit MSDs so, dass ein Scenario von oben nach unten abläuft. Das bedeutet, dass die Messages in genau der Reihenfolge auftreten müssen, wie sie im Dokument angegeben sind, außer die Reihenfolge wurde wie zuvor beschrieben parallelisiert. Ein Scenario wird immer durch die an erster Stelle stehende Nachricht initialisiert. Dies ist die *initializing Message*. Wird ein Scenario initialisiert, wird eine neue Instanz dieses Scenario generiert und ist nun als *active Scenario* vorhanden. Ein active Scenario bezeichnet eine laufende Instanz eines MSpec-Scenarios. Es hat einen Zustand, der durch die aktiven, also die erwarteten Nachrichten definiert ist.

3.2.2. Der Message-Pointer

Der Zustand eines active Scenarios wird anhand des *Message-Pointers* bestimmt. Der Message-Pointer ist äquivalent zum Cut in MSDs und zeigt auf die Messages, die gerade aktiv sind, auf die das Scenario also wartet, um voranzuschreiten. Da der Cut von MSDs in sequenzieller Textform schwer vorzustellen ist, wird er durch eine Reihe von Pointern ersetzt. Ein simples Scenario, welches nur eine rein sequenzielle Abfolge (ohne Loops, Alternativen oder Parallelen) beschreibt, benötigt nur einen Pointer, der auf die als Nächstes erwartete Message zeigt. In dem Fall, der in Codeausschnitt 3.19 dargestellt ist, benötigt das Scenario zwei Message Pointer, einen für jede alternative Nachricht. Loops und Parallelen sind äquivalent dazu. Eine Message, auf die ein Pointer zeigt, heißt dann active Message.

```
scenario r LogisticOrderRobotToBringItem {
 msg (c,m) t -> ls.orderItem()
 msg (h,e) ls -> r1.takeItem() // <- Alter Message-Pointer
 // Nachricht ls -> r1.takeItem trifft ein.
 alt {
  case {
   msg (c,m) r1 -> ls.takingItem() // <- Neuer Message-Pointer
   msg (h,e) ls -> r1.goToTerminal()
   msg (h,e) ls -> t.robotIncoming()
  }
  case {
   msg (c,m) r1 -> ls.busy() // <- Neuer Message-Pointer
   msg (h,e) ls -> t.robotBusy()
  }
 }
}
```

Listing 3.19: Codeausschnitt mit Alternative in MSpec

3.2.3. Unifikation zweier Nachrichten

Die Unifikation zweier Nachrichten funktioniert hier analog zu der Unifikation in MSDs. Zwei Nachrichten sind genau dann unifizierbar, wenn ihre Sender und Empfänger, sowie der Operationsaufruf den selben Objekten zugeordnet werden können. Beispielweise sind die Nachrichten im Codebeispiel 3.20 unifizierbar, wohingegen die Nachrichten in Codebeispiel 3.21 nicht unifizierbar sind.

```
msg (h,e) ls -> r1.takeItem()
msg (h,e) ls -> r1.takeItem()
```

Listing 3.20: Codebeispiel für zwei unifizierbare Nachrichten.

```
msg (h,e) ls -> r1.takeItem()
msg (h,e) ls -> r2.takeItem() // r2 != r1
```

Listing 3.21: Codebeispiel für zwei nicht unifizierbare Nachrichten.

Wenn eine Nachricht im System auftritt, müssen, die Unifikation betreffend, zwei Dinge geschehen:

- Die aufgetretene Nachricht muss mit jeder initializing Message jedes MSpec-Scenarios verglichen werden. Ist sie mit einer davon unifizierbar, wird eine neue Instanz des zugehörigen Scenarios generiert und als neues active Scenario initialisiert. Der Message-Pointer dieses active Scenarios wird dann auf die Nachricht gesetzt, die als nächstes erwartet wird.

- Die auftretende Nachricht muss mit jeder active Message der Message-Pointer von jedem active Scenario verglichen werden. Wenn sie mit einer dieser unifizierbar ist, so muss das dazugehörige active Scenario voranschreiten. Das bedeutet, dass der Message-Pointer auf die nächste(n) Message(s) gesetzt wird. Sollte die aufgetretene Nachricht die letzte im active Scenario sein, so ist dieses abgeschlossen und wird verworfen. Sollte die Nachricht nicht mit einer der active Messages eines active Scenarios unifizierbar sein, so kommt es zu einer Verletzung des Scenarios. Die möglichen Verletzungen sind äquivalent zu denen der MSDs (Safety- / Cold-Violation).

4. Übersetzung in ausführbaren Code

Xtext bietet die Möglichkeit, relativ einfach einen Codegenerator passend zu der erstellten DSL zu implementieren. Dabei wird dieser automatisch bei jedem Speichervorgang des MSpec-Dokuments ausgeführt, um die generierten Dateien aktuell zu halten. Diese Funktion wurde für MSpec jedoch abgeschaltet, da der Codegenerator recht komplex wurde und das häufige Speichern des Dokuments dadurch zu lange dauerte. Stattdessen gibt es nun den Eintrag „Generate Code“ im Kontextmenü des *Package Explorers*. Wird auf diesen geklickt, startet der Codegenerator, wie beim Speichern vorgesehen. Im Folgenden werden nun zuerst einige Konzepte für die Codeübersetzung vorgestellt und anschließend auf die momentane Implementierung eingegangen.

4.1. Konzepte

Dieser Abschnitt erklärt einige Konzepte, die bei der Übersetzung von MSpec in ausführbaren Code hilfreich sind.

4.1.1. Ermittlung von Daten eines Scenario

Das Scenario besteht aus einer Baumstruktur. Es enthält eine Feature-Liste, in der alle Scenario-Features des Scenarios enthalten sind. Ist ein Feature ein Loop-, Alternativ- oder Parallel-Block, so beinhalten diese bzw. deren zugehörigen Case-Klassen ebenfalls Scenario-Features. So ist es also möglich, alle Scenario-Features durch Navigieren durch diesen Baum zu ermitteln.

4.1.2. Das Scenario als Zustandsautomat

Um die Zustände für die Codegenerierung zu erhalten, kann das Scenario auf die gleiche Weise wie ein MSD in einen Zustandsautomaten übersetzt werden. Um alle möglichen Zustände eines Scenarios zu erhalten, kann erneut die Baumstruktur der Scenarios ausgenutzt werden. So wird eine strikte Abfolge von

Messages zu einer Zustandsreihe, die immer auf den nächsten Zustand zeigt (siehe Abbildung 4.1).
Die Zustände eines Scenarios beschreiben die Menge aller Messages, die aktiv sind. Diese hängen vom bisherigen Verlauf des Scenarios ab. Je nachdem an welchem Punkt im Scenario sich der Message-Pointer aufhält, sind andere Nachrichten aktiviert. Um aus dieser Position einen Zustand zu machen, können Zähler für jede Lifeline ergänzt werden. Diese Zähler sind bei der Initialisierung auf 0 zu setzen und werden bei jedem Auftauchen einer Nachricht auf der entsprechenden Lifeline um 1 erhöht.
In den Folgenden Abbildungen von Zustandsautomaten gibt es immer drei Objekte und die Zustände sind demnach durch drei Zähler definiert. Der Zustand XYZ beschreibt also die Zähler state_t=X, state_ls=Y und state_r1=Z. Hinweis: Der Einfachheit der Darstellung halber, wurden der akzeptierende Zustand der Cold-Violation und der ablehnende Zustand der Safety-Violation, sowie die dazugehörigen Transitionen in den folgenden Grafiken nicht eingetragen.

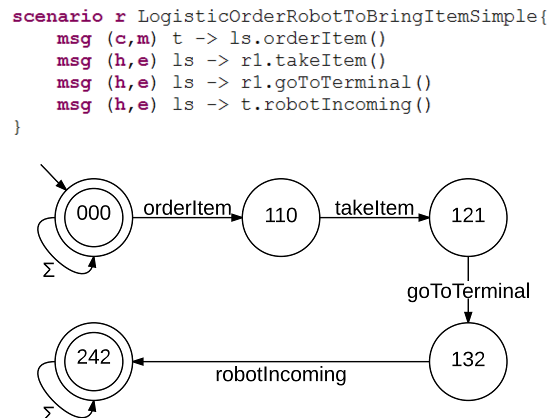

Abbildung 4.1.: Übersetzung einer Nachrichtensequenz in einen Zustandsautomaten

Loop

Kommen dagegen Loops, Alternativen, oder parallele Abfolgen in der Sequenz vor, entstehen Verzweigungen im Zustandsautomaten. Beim Loop gabelt sich der Automat jeweils ein mal am Anfang und am Ende des Loops auf. Am Anfang, um den Loop null mal auszuführen und am Ende, um ihn N mal

auszuführen, wobei $N>0$ gilt. Abbildung 4.2 zeigt eine solche Übersetzung. Die drei Zähler innerhalb der Zustände beschreiben dabei wieder die Occurrence-Points der Nachrichten an den Lifelines „t“, „ls“, „r1“. Die Besonderheit beim Loop ist die, dass die Zustände 110 und 132 identisch sind, da der Loop hier wieder von vorne anfängt.

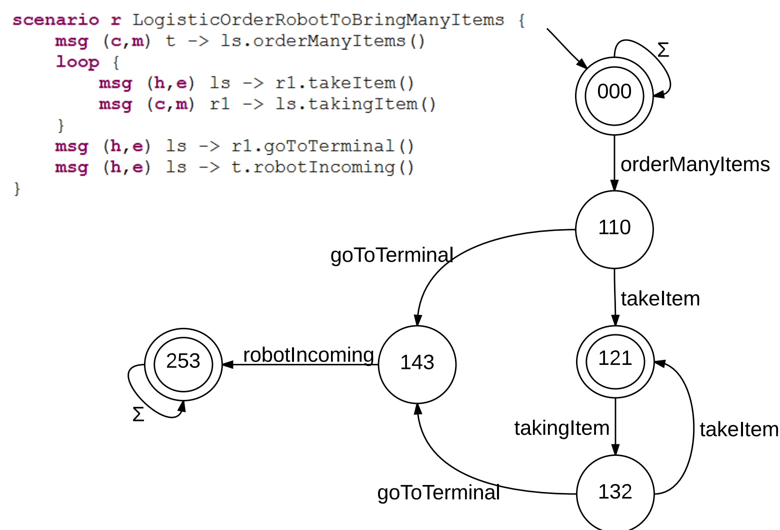

Abbildung 4.2.: Übersetzung eines Loops in einen Zustandsautomaten

Alternative

Allgemein kann ein Zustand als eine Menge an akzeptablen Nachrichten beschrieben werden. Im Fall der Alternative gabelt sich der Zustandsautomat in N Zweige auf, wobei N die Anzahl an Alternativen ist. Abbildung 4.3 zeigt ein Beispiel mit $N=2$.

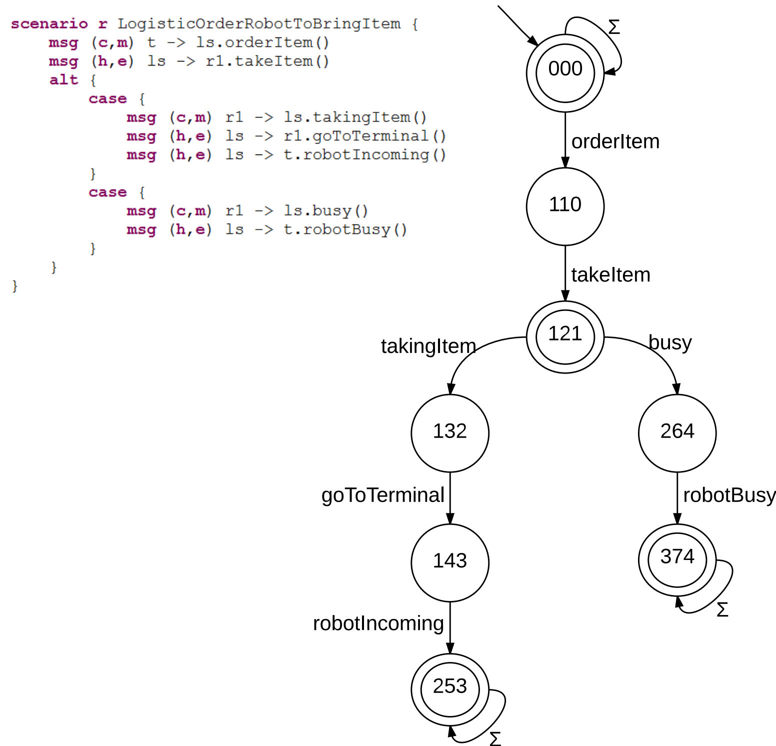

Abbildung 4.3.: Übersetzung einer Alternative in einen Zustandsautomaten

Parallel

Beim Parallel-Block kann der Zustandsautomat sehr schnell sehr groß werden. Hier kann jeder Zustand bis zu N Transitionen zu bis zu N Folgezuständen haben, wobei N die Anzahl der Parallelfälle ist. Abbildung 4.4 zeigt diese Übersetzung mit N=2. Hinweis: Dieses Scenario hat vier teilnehmende Objekte.

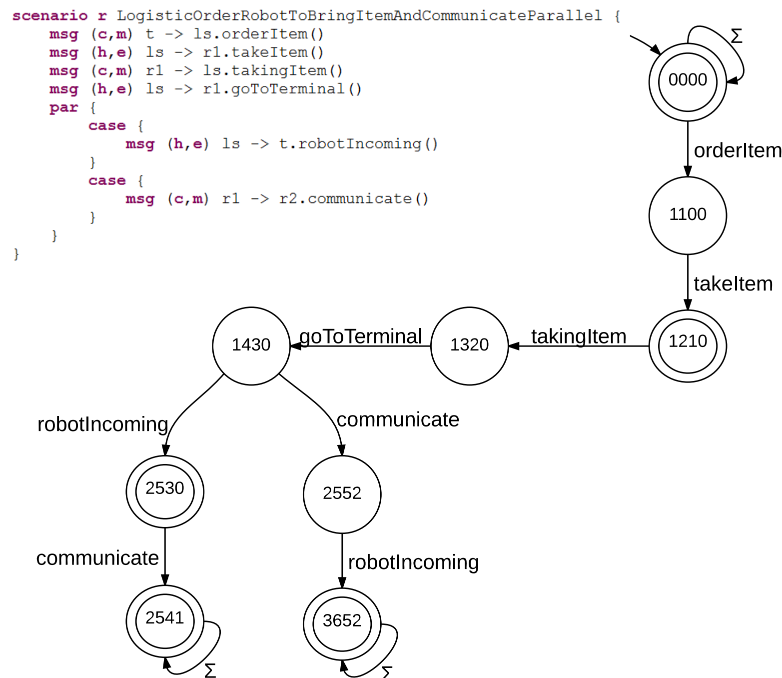

Abbildung 4.4.: Übersetzung einer parallelen Abfolge in einen Zustandsautomaten

4.2. Übersetzung in Java

Parallel zu dieser Arbeit wird eine Java-Implementierung zur Ausführung von MSDs entwickelt. Diese Implementierung soll ein MSD, wie es für diese Arbeit betrachtet wird, in eine für einen Play-Out Algorithmus ausführbare Form bringen. Daher ist es passend, die Codegenerierung so zu entwerfen, dass eine MSpec-Spezifikation in diese Form gebracht werden kann. Dafür führt der „MSpecGenerator“ eine Klasse „SpecificationGenerator“ aus, der die Codegenerierung in drei Schritten ausführt.

4.2.1. Übersetzung in ein Ecore-Model

Der erste Schritt ist es, die Spezifikation in ein Ecore-Model zu übersetzen. Dafür wird ein EPackage „SpecificationPackage“ angelegt, in dem sich eine Klasse

„*<SpecificationName>*Specification“ befindet, wobei *<SpecificationName>* gegen den Namen der MSpec-Spezifikation ersetzt wird. Diese Klasse instanziiert das Objektsystem, sowie die Scenarios und die Messages. Sie enthält zudem auch Informationen zu den initialisierenden Nachrichten.
Zudem wird für jedes MSpec-Scenario eine weitere Klasse in diesem EPackage erstellt. Diese Klassen implementieren das Interface *ActiveScenario*, welches später für die Ausführung nötig ist. In diesen Klassen werden EReferences auf alle im jeweiligen Scenario auftretenden Objekte erzeugt, die den Namen „obj_*<ObjectName>*“ tragen, wobei *<ObjectName>* gegen den Namen des Objects ersetzt wird. Der EType der EReference ist dann die Klassenreferenz des Objects. Außerdem werden EReferences für jede Messge im jeweiligen Scenario in diesen Klassen erzeugt. Diese EReferences tragen den Namen „msg_*<MessageName>*“, wobei *<MessageName>* gegen den Namen des Operators ausgetauscht wird. Der EType der EReference ist ModalMessageEvent, welches wiederum für die spätere Ausführung nötig ist. Zuletzt wird für jedes Object, das an den Scenarios teilnimmt, also eine Nachricht sendet bzw. empfängt, eine Zustandsvariable hinzugefügt. Diese heißt „state_*<ObjectName>*“ und hat den EType EInt.

4.2.2. Ecore-Model-Code generieren

Im nächsten Schritt wird der Model Code des erzeugten Ecore-Models generiert. Dafür wird ein GenModel erzeugt, welches als Model Directory den Unterordner „mspec-gen“ des Projektes erhält, sodass der generierte Model-Code in diesem Ordner landet. Dem GenModel müssen dann alle benötigten GenPackages übergeben werden. Nun kann es mit Eingabe des erzeugten Ecore-Models und aller Ecore-Models, die an der Spezifikation beteiligt sind (alle Imports), initialisiert werden. Nun wird der Model-Code durch Aufruf des GenModel-Generators generiert.

4.2.3. Fertigstellung des Model-Codes

Im letzten Schritt müssen die generierten Klassen für die Scenarios vervollständigt werden, indem die Methoden für das Interface ActiveScenario implementiert werden. Dazu ist es nötig, spezifische Daten aus dem MSpec-Dokument zu extrahieren. Neben dem Objektsystem, für das alle beteiligten Objekte eines Scenarios, sowie alle verwendeten Nachrichten benötigt werden, muss ermittelt werden, wann welche Nachricht auftreten darf. Dafür ist es notwendig, die Zustandsautomaten, die in Abschnitt 4.1.2 vorgestellt wurden, algorithmisch zu erfassen. Der generierte Code für den Zustandsautomaten in Abbildung 4.1

könnte folgendermaßen aussehen:

```
if (state_t==0 && state_ls==0 && state_r1==0 && isUnifiable(
   occurredMessage,orderItem)){
 state_t++;
 state_ls++;
}
if (state_t==1 && state_ls==1 && state_r1==0 && isUnifiable(
   occurredMessage,takeItem)){
 state_ls++;
 state_r1++;
}
if (state_t==1 && state_ls==2 && state_r1==1 && isUnifiable(
   occurredMessage,goToTerminal)){
 state_ls++;
 state_r1++;
}
if (state_t==1 && state_ls==3 && state_r1==2 && isUnifiable(
   occurredMessage,robotIncoming)){
 state_t++;
 state_ls++;
 setFinished(true);
}
```

Listing 4.1: Codebeispiel für einen Zustandsautomaten in Java

Die Übersetzung des Scenarios in einen solchen Javacode ist durch folgenden Pseudocode leicht zu realisieren:

```
Scenario scenario
List allObjects = getAllParticipatingObjects(scenario)
for each object in allObjects do{
 state(object)=0
}
List allMessages = getAllMessages(scenario)
for each message in allMessages do{
 generateIfStatement(allIfStatements, message)
 state(message.getSender())++;
 state(message.getReceiver())++;
 generateNextStateCode(states)
}
```

Listing 4.2: Pseudocode für die Generierung des Java-Zustandsautomaten

Loop

Einen Loop in einem Scenario zu übersetzen, ist schon etwas schwieriger. Hier muss das Zurückspringen in den Zustand vor dem Loop berücksichtigt werden,

sowie das Überspringen des Loops von diesem Zustand. Der Pseudocode 4.3 zeigt eine passende Realisierung.

```
Scenario scenario
List allObjects = getAllParticipatingObjects(scenario)
for each object im allObjects do{
 state(object)=0
}
for each object im allObjects do{
 loopState(object)=0
}
List allMessages = getAllMessages(scenario)
for each message in allMessages do{
 generateIfStatement(allIfStatements, message)
 if( isInLoop(message) ) {
  if( isFirstMessageInLoop(message) ) {
   loopState(allObjects) = 0
  }
  if( isLastMessageInLoop(message) ) {
   // return to start
   state(allObjects) - loopState(allObjects)
  }
  loopState(message.getSender())++;
  loopState(message.getReceiver())++;
 }
 if( isMessageAfterLoop(message) ) {
  state(allObjects) + loopState(allObjects)
 }
 state(message.getSender())++;
 state(message.getReceiver())++;
 generateNextStateCode(states)
}
```

Listing 4.3: Pseudocode für die Generierung des Java-Zustandsautomaten mit einem Loop

Alternative

Eine Alternative gabelt sich ähnlich wie der Loop auf, nur dass es statt zwei nun N Wege gibt. Der Pseudocode 4.4 zeigt eine mögliche Realisierung. Es werden immer beim Sprung von einem Fall zum nächsten die Anzahl der Zustandserhöhungen in dem vorherigen Fall auf die aktuellen States draufgerechnet.

```
Scenario scenario
List allObjects = getAllParticipatingObjects(scenario)
for each object im allObjects do{
 state(object)=0
```

```
}
for each object im allObjects do{
 altState(object) = 0
}
List allMessages = getAllMessages(scenario)
for each message in allMessages do{
 generateIfStatement(allIfStatements, message)
 if( isInAlt(message) ) {
  if( isLastMessageInAlt(message) ) {
   altState(message.getSender())++;
   altState(message.getReceiver())++;
   state(allObjects) - altState(allObjects)
  }
  elseif( isFirstMessageinAlt(message) ) {
   state(allObjects) + altState(allObjects)
   altState(message.getSender())++;
   altState(message.getReceiver())++;
  }
  else {
   altState(message.getSender())++;
   altState(message.getReceiver())++;
  }
 }
 state(message.getSender())++;
 state(message.getReceiver())++;
 generateNextStateCode(states)
}
```

Listing 4.4: Pseudocode für die Generierung des Java-Zustandsautomaten mit Alternativen

5. Implementierung

Die Modellierungssprache MSpec ist mit Hilfe des Eclipse Plugins Xtext implementiert wurden. Die Xtext-Grammatik ist im Anhang (siehe A) zu finden. Diese Grammatik wird durch einen MWE2 Workflow kompiliert, wodurch auch je nach Anpassung des Workflows mehrere Xtend-(oder je nach Einstellung des Workflows auch Java-) Dateien zur Anpassung des Editors für die MSpec-Dokumente von dem Plugin zur Verfügung gestellt werden. Die folgenden Abschnitte erläutern, wie diese Funktionen angepasst wurden, um die Sprache funktionsfähig und den Editor möglichst benutzerfreundlich zu machen. Generell ist die Struktur der Sprache wie ein Baum aufgebaut. Das bedeutet für das MSpec Dokument, dass das oberste Objekt die Specification ist, die eine Anzahl an UseCases enthält, die wiederum weitere Objekte enthalten. Um von einem Objekt auf ein anderes verweisen zu können, ist es also nötig, durch das Dokument wie durch einen Baum zu navigieren. Die Bezeichnung für die Elemente der Sprache in diesem Abschnitt gleicht der Bezeichnung in der Implementierung in Xtext. So steht zum Beispiel die Bezeichnung *XScenario* für die Implementierung des MSpec-Scenarios und *XMessage* steht für das MSpec-Message-Event.

5.1. Der MWE2 Workflow

Die Modeling Workflow Engine 2 ist ein deklarativer Codegenerator. Eine ihrer Anwendungsweisen ist es, einen Workflow zu erstellen, der für die Codegenerierung einer Xtext-Grammatik in Java-Klassen genutzt werden kann. Im Workflow können Variablen und Komponenten eingestellt werden, die je nach gewünschter Funktionalität nicht nur den Xtext Quellcode von MSpec in passende Java-Klassen kompilieren, sondern auch Dateien zur Anpassung benötigter Editor-Komponenten erstellen. Für die Modellierungssprache MSpec werden hier viele Klassen für die Oberfläche des Editors benötigt, aber auch einige für die Funktionalität der Sprache selbst und für die Codegenerierung, derer MSpec im Stande sein soll. Außerdem werden im Workflow benötigte Abhängigkeiten wie Ecore deklariert und importiert. Xtext generiert für eine neue DSL selbstständig einen Workflow zur Codegenerierung, der bereits die meisten Features

korrekt beinhaltet. Es mussten nur ein paar geringfügige Änderungen am Workflow vorgenommen werden, um erstens Ecore in die Sprache einbinden zu können und zweitens die Import-Funktion von Xtext verwendbar zu machen.

5.2. Scoping

Mit Hilfe des Scope-Providers ist es möglich genau zu bestimmen, wann welches Objekt in der Autovervollständigung auftaucht und zu welchen Objekten diese Referenzen zuzuweisen sind. Ein Scope beschreibt einen Pool von Objektreferenzen, die in einem bestimmten Feld im Dokument gültig sind. Ein Feld kann hier etwa eine Objektreferenz auf ein Objekt in der Collaboration sein. In diesem Fall muss der Scope-Provider so angepasst werden, dass der Editor weiß, wo diese Objekte definiert sind, damit die Autovervollständigung diese anzeigen und die Codevalidierung die Objekte finden kann. Der Scope-Provider wurde hier komplett in Xtend implementiert. Xtend ist eine Programmiersprache, die relativ ähnlich zu Java ist und direkt in Java umgewandelt werden kann. Eine Regel im Scope-Provider wird durch eine Funktion implementiert, die den Namen „scope_[Name des Objekts]_[Name des Feldes]“ trägt, wobei [Name des Objekts] und [Name des Feldes] Platzhalter für die Bezeichner des Objekts sind, für dessen Feld das Scope manipuliert werden soll. Diese Funktion erhält als Eingabeparameter das Objekt, das vom Editor gerade bearbeitet wird. Nun ist es möglich, das Standard-Scope dieses Objektes abzufragen und zu verändern, oder ein neues, leeres Scope zu erstellen, oder das Scope auf eine Liste von Objekten eines anderen Objekts zu setzen. Der Rückgabewert der Funktion ist dann das neue Scope, das angewendet werden soll. Glücklicherweise ist der automatisch generierte Scope-Provider schon fast ausreichend für die gewünschte Funktionalität. Im Nachfolgenden werden Fälle gezeigt, in denen es dennoch nötig war, den Scope-Provider anzupassen.

XMessage

Die XMessage ist das Objekt in MSpec, welches für die Message-Events steht. Damit dieses Objekt richtig funktioniert, mussten drei Anpassungen gemacht werden. Es mussten einerseits für den Sender und den Empfänger die Referenzen auf die Objekte der Collaboration und andererseits die Referenzen auf die EOperations des Ecore-Models des Empfängers ermittelt werden. Es mussten also die drei Funktionen „scope_XMessage_sender“, „scope_XMessage_receiver“ und „scope_XMessage_call“ angelegt werden. Da durch die Funktionalität des Scope-Providers erstmal nur das betroffene Objekt (die XMessage) bekannt war,

war es nötig, von diesem Objekt aus den übergeordneten Container zu ermitteln. Sofern dieser ein XScenario ist, kann der XUseCase als übergeordneter Container ermittelt werden, worin die gesuchte XCollaboration eingetragen ist. Ist der übergeordnete Container der XMessage allerdings kein XScenario, sondern ein XScenarioAlternative, XScenarioLoop oder XScenarioParallel, muss ein weiteres Mal einen Container nach oben navigiert werden. Für den Fall, dass mehrere XScenarioFeatures verschachtelt sind, war eine Schleife nötig, die so oft eine Ebene nach oben navigiert, bis das XScenario erreicht ist. Von da aus ist es kein Problem mehr, bis zur XCollaboration zu navigieren. Um auf die EOperations des Empfängers zugreifen zu können, ist dieses Navigieren durch den Baum allerdings nicht nötig, da direkt auf die Referenz des Empfängers zugegriffen werden kann. Von da aus kann einfach die Referenz des XObjects auf die EClass verwendet werden, um ein Scope auf die EOperations des Empfängers zu erhalten. Außerdem wurde diese Funktion noch erweitert, um zudem die EOperations aller SuperTypes der EClass zu ermitteln, wodurch auch vererbte EOperations im Scope zu finden sind.

XRoleBinding

Für die XRoleBinding müssen, ähnlich wie bei der XMessage, die Referenzen auf die EOperations des Objekts ermittelt werden, an dessen Operation die Rolle von nun an gebunden sein soll. Hier wurde eine Funktion „scope_XRoleBinding_op “ implementiert, die genauso wie bei der XMessage hier einfach auf die EClass Referenz des XObjects zugreift und mit Hilfe einer Schleife die EOperations aller SuperTypes hinzufügt und als neues Scope festlegt.

XObject

Das XObject betreffend gab es nur eine nötige Anpassung. Hier war es nämlich möglich, für die Objektreferenzen auf die importierten Ecore-Models nicht nur deren Qualifiedname (siehe [FQN]) anzugeben, sondern es konnte ebenfalls deren Namespace URI angegeben werden, was allerdings zu Fehlern bei der Validierung führte. Um dieses Problem zu beheben, wurde eine Funktion implementiert, die über alle Objekte im aktuellen Scope iteriert und diejenigen, die über ihre Namespace URI bestimmt sind, daraus löscht.

5.3. Code-Validation

Die Code-Validation überprüft, ob der eingegebene Code auch korrekt ist. Von Haus aus werden bereits Syntaxfehler und nicht im Scope liegende Eingaben als Fehler markiert. Ist es jedoch nötig, weitere Fehler oder Warnungen zu spezifizieren, so ist es möglich im Validator eigene Validation-Rules zu erstellen. Diese Regeln werden dann überprüft, wenn ein neues Objekt hinzugefügt wird oder der Code neu eingelesen wird. Beispielsweise kann überprüft werden, ob die Namen bestimmter Objekte mit einem Groß- oder Kleinbuchstaben anfangen. Ist dieser Check erfolgreich, passiert nichts, detektiert er allerdings einen Fehler, so kann ein Fehler (error) oder eine Warnung (warning) ausgeben werden, was dann zu einer dementsprechenden Anzeige im Editor führt. Der Fehler bzw. die Warnung wird dann an einer bestimmten Stelle der Code-Zeile angezeigt. Im Folgenden wird gezeigt, welche Fehler und Warnungen ergänzt wurden.

Warnung: XUseCase soll mit einem Großbuchstaben anfangen

Eine einfache Regel: Ein XUseCase soll mit einem Großbuchstaben anfangen. Die Funktion überprüft den ersten Buchstaben des Namens des betroffenen XUseCase und wenn dieser kein Großbuchstabe ist, wird eine Warnung mit dem Text „Name should start with a capital.“ angezeigt.

Warnung: XScenario soll mit einem Großbuchstaben anfangen

Diese Regel ist analog zu der vorherigen, mit dem Unterschied, dass hier der Name eines XScenario überprüft wird.

Fehler: Der Name eines XUseCase muss eindeutig sein

Wichtig für die Ausführung der MSpec Spezifikation ist es, dass die Use-Cases einen eindeutigen Namen haben. Dies wird beim Erstellen bzw. Bearbeiten eines XUseCase durch eine Funktion sichergestellt, die zu der übergeordneten XSpecification navigiert und von dort aus über alle XUseCases iteriert und die Namen auf Gleichheit prüft. Sollte der neue Name schon existieren, wird ein Fehler mit dem Text „The name of a Use-Case has to be unique.“ ausgegeben.

Fehler: Der Name eines XScenario muss eindeutig sein

Ähnlich wie beim XUseCase, muss auch der Name eines XScenario eindeutig sein, zumindest innerhalb des selben Use-Case. Über alle Use-Cases darf der selbe Name allerdings mehrfach vorkommen. Die Funktion wird exakt so implementiert, wie die der Namensüberprüfung eines XUseCase. Der übergeordnete Container des XScenario ist ein XUseCase, welcher eine Liste aller relevanten XScenarios enthält. Über diese Liste wird iteriert und dabei auf Namensgleichheit geprüft. Sollte der Name schon vergeben sein, wird ein Fehler mit dem Text „The name of a Scenario must be unique.“ ausgegeben.

Fehler: Zu viele oder zu wenig Argumente einer XMessage

Eine XMessage referenziert eine EOperation, welche auch Argumente haben kann. Sollte sie eine bestimmte Anzahl an Argumenten haben, so ist es auch nötig, diese beim Aufruf zu übergeben. Ist dies nicht der Fall, werden also zu wenige oder zu viele Argumente übergeben, so wird eine Fehlermeldung mit dem Text „Too many Arguments!“ bzw. „Too few Arguments!“ angezeigt. Um dies zu überprüfen wird die Größe des Liste der übergebenen Argumente mit der Größe der Liste der EParameters der EOperation verglichen.

Fehler: Eine Rollenbindung muss gültig sein

Eine XRoleBinding ist nur dann gültig, wenn der Rückgabewert der zu bindende Operation identisch mit der Klasse ist, die für die Rolle vorgesehen ist. Hierfür muss der EType der EOperation mit der EClass der XObject-Referenz verglichen werden. Sind diese identisch, ist alles korrekt, sind sie jedoch unterschiedlich oder ist der Rückgabewert der EOperation Void, so wird eine entsprechende Fehlermeldung angezeigt. Ihr Text lautet dann „Invalid role binding: Types do not match!“, gefolgt von den inkompatiblen Typen.

Fehler: Asynchrone Nachrichten dürfen nicht auftreten

In MSpec müssen alle Nachrichtenabläufe klar definiert werden. Eine Unklarheit wie in der Beispielabbildung 2.9 darf hier nicht auftreten. Darum achtet diese Regel darauf, ob die Sender und Empfänger zweier hintereinander stehender Nachrichten disjunkt sind. Dafür wird für die Nachricht der übergeordnete Container betrachtet und über dessen Index die vorherige Nachricht ermittelt. Daraufhin wird überprüft, ob Sender und Empfänger der beiden Nachrichten keine gleichen Objekte referenzieren. Ist dies der Fall, wird eine Fehlermeldung

mit dem Text „Asynchronous Messages detected! Sender and Receiver of this Message are not unifiable with those of a previous Message-Event.“ angezeigt. Da Block-Elemente, wie Loops, Alternativen und Parallelen, bereits gemäß ihrer Definition eine Synchronisation auslösen, ist hier keine Detektion nötig.

5.4. Quickfix-Provider

Der Quickfix-Provider von Xtext ermöglicht es dem Editor, bei bestimmten Fehlern im Code Hilfestellungen zur Korrektur zu geben. Ähnlich wie bei dem Scope-Provider und der Code-Validation können hier auch eigene Regeln, sogenannte Fixes, festgelegt werden. Diese Regeln beziehen sich auf vorher in der Code-Validation eingestellte Fehler und Warnungen. Der Quickfix-Provider kann ein mächtiges Werkzeug zur Auflösung von Fehlern sein, wenn er richtig einstellt wurde. Es ist möglich, auf das gesamte Xtext-Dokument zuzugreifen und diverse Änderungen im Code zu machen. Seine Anwendungen reichen von simplen Stringanpassungen, wie zB. Groß- und Kleinschreibung, bis hin zu kompletter Code Konversion, wie zB. das Einfügen von Code-Blöcken um anderen Code herum. Auf diese Fixes kann über das Kontextmenü des Editors, das beim Herüberfliegen mit der Maus erscheint, zugegriffen werden. Im Folgenden werden Anpassungen des Quickfix-Providers erläutert, die für die Realisierung von MSpec nötig waren. Der Quickfix-Provider für MSpec ist in Java geschrieben.

Fix: Großschreiben des Namens eines XUseCase

In der Code-Validation wurde festgelegt, dass ein XUseCase immer mit einem Großbuchstaben anfangen soll, andernfalls taucht eine Warnung auf. Dieser Fix gibt dem Benutzer die Option einen kleingeschriebenen XUseCase Namen so umzuwandeln, dass er mit einem Großbuchstaben anfängt. Dafür erscheint im Kontextmenü die Option „Convert to upper case.“, die durch einen Klick den ersten Buchstaben gegen einen entsprechenden Großbuchstaben ersetzt und so die Warnung auflöst.

Fix: Großschreiben des Namens eines XScenario

Dieser Fix ist analog zu dem vorherigen, mit dem Unterschied, dass hier der Name eines XScenario angepasst wird. Der entsprechende Eintrag im Kontextmenü heißt hier ebenfalls „Convert to upper case.“.

Fix: Einfügen einer Sync-Condition zwischen asynchronen Nachrichten

Wie bereits in Kapitel 3 beschrieben, darf es keine asynchronen Nachrichten in MSpec geben (siehe Abschnitt 3.1.5). Um dies zu detektieren wurde in der Code-Validation eine entsprechende Fehlermeldung eingetragen. Um den Benutzer mit dieser Fehlermeldung nicht allein stehen zu lassen, bietet der Editor zwei mögliche Fixes an. Der erste hat den Schriftzug „Add a Sync Condition.“ und fügt eine passende Sync-Condition ein, die die betroffenen Lifelines synchronisiert, bevor die zweite Nachricht erscheinen kann. So wird eine feste Reihenfolge der Nachrichten erzwungen. Die Ermittlung der betroffenen Lifelines wird über die selben Funktionen realisiert, die auch in der Code-Validation verwendet wurden. Aus diesem Grund sind diese Funktionen auch als statisch markiert. Die eingefügte Sync-Condition wird alle Lifelines synchronisieren, die an dem Senden und Empfangen der betroffenen Nachrichten beteiligt sind.

Fix: Einfügen eines Parallel-Blocks um asynchrone Nachrichten

Der zweite Fix bietet die Möglichkeit die Reihenfolge als ungewiss bestehen zu lassen. Gemäß seines Schriftzuges „Surround with parallel cases.“ umgibt er die beteiligten Nachrichten mit einem Parallel-Block, in dem beide Nachrichten in unterschiedlichen Cases stehen.

5.5. Label-Provider

Der Xtext Label-Provider ermöglicht es, Objekte der Sprache mit einem passenden Namen zu versehen. Auf diese Weise können Objekte zB. in der „Outline“ Ansicht in Eclipse ansehnlicher dargestellt werden.

XMessage

Für die XMessage wurde der Label-Provider so angepasst, dass diese als „Sender -> Empfänger.Operation()“ dargestellt werden, anstatt als „unnamed“. Dies würde nämlich ansonsten der Fall sein, da das Objekt XMessage keinen eigenen Namen besitzt.

XScenarioLoop, XScenarioAlternative, XScenarioParallel

Die Block-Objekte von MSpec besitzen ebenfalls keinen eigenen Namen, darum mussten auch diese angepasst werden:

- XScenarioLoop: Der Loop wird nun einfach mit „Loop“ gekennzeichnet.
- XScenarioAlternative: Die Alternative heißt jetzt „Alt N“, wobei N für die Anzahl an Alternativen steht. Dies ist die Anzahl der XScenarioAlternativeCase Objekte des XscenarioAlternative Objekts. Die XScenarioAlternativeCase Objekte heißen dann „case n„mit n als Index des Objekts.
- XScenarioParallel: Die Namen der XScenarioParallel Objekte und deren XScenarioParallelCase Objekte wurden analog zu denen der Alternative angepasst.

6. Verwandte Arbeiten

Im Folgenden werden vier weitere Arbeiten kurz vorgestellt, die einen ähnlichen Ansatz aufweise wie diese Arbeit, oder durch andere Eigenschaften interessant sind.

6.1. ScenarioTools

ScenarioTools ist ein grafischer Ansatz für das selbe Problem, welches auch diese Arbeit adressiert. Mit Hilfe dieses Eclipse Plugins ist es möglich MSDs auf grafischem Wege zu erzeugen und damit eine Spezifikation zu erstellen. Diese grafischen MSDs können dann simuliert und mit Hilfe von einem Play-Out Algorithmus in einer Debug-Umgebung analysiert werden. Diese Debug-Umgebung zeigt dem Benutzer alle möglichen Message-Events und lässt ihn eines auswählen, welches zu keiner Safety-Violation führt. Auch wenn der Editor dieses Werkzeugs leider noch nicht ganz ausgereift ist und Fehler aufweist, so ist es dennoch eine sehr nennenswerte Arbeit, besonders da dort schon die Simulation der MSDs gelungen ist. Das Ziel von ScenarioTools ist das selbe, wie das von MSpec: Spezifikation, automatisierte Analyse und Simulation. Zudem ist die hier gezeigte Implementierung von MSpec ein Teil von ScenarioTools und verwendet für die Übersetzung in Java Code Funktionen davon.

6.2. Can Programming Be Liberated, Period?

In seinem Artikel „Can Programming Be Liberated, Period?“ [Har08] schreibt David Harel, wie er sich eine Zukunft ohne die Notwendigkeit von Programmiersprachen vorstellt. Er beschreibt eine Art der Programmierung durch natürliche und spielerische Mittel. Der Autor beschreibt seine Sicht auf die Entwicklung der Computerprogrammierung der letzten Jahrzehnte und postuliert weitere Entwicklungen in eine Richtung, die sich möglicherweise komplett von der technischen Ebene entfernt. So soll das Programmieren von Computern in Zukunft durch einfaches Beschreiben von Verhaltensweisen und -regeln geschehen. Es wird beschrieben, wie Verhaltensweisen in Computern „eingespielt“ werden und

so der Computer lernt zu tun, was der Benutzer will. Dies spielt sehr in die Richtung des *behavioral programming*, das auch ein Gedanke im Hintergrund dieser Arbeit ist. Denn gelänge es, eine Methode zu entwickeln, mit Hilfe der Computer schon durch die Aufstellung der Spezifikation „programmiert“ werden können, würde dies herkömmliche Programmierweisen obsolet machen.

6.3. Behavioral Programming

Zum Thema behavioral programming passt auch der gleichnamige Artikel von Harel, Marron und Weiss (siehe [HMW12]), in dem die Autoren die selben Schwierigkeiten beschreiben, aufgrund derer diese Arbeit entstanden ist. Es werden Problematiken bei der Aufstellung und Umsetzung von Anforderungen erklärt und bekannte Lösungsansätze wie LSCs benannt. Die Autoren beschreiben Prinzipien des behavioral programming und veranschaulichen eine Methode *behavioral applications* in Java zu implementieren. Es wird das Prinzip der *b-threads* erläutert. Diese b-threads funktionieren sehr ähnlich, wie das Modell der reaktiven Systeme, wie es in dieser Arbeit betrachtet wird. So können mehrere b-threads parallel zu einander laufen, bis sie sogenannte *synchronization points* erreichen. An so einem Punkt erwartet ein b-thread eine bestimmte Menge von Events, bittet um Benachrichtigung für eine andere Menge von Events und verbietet eine weitere Menge von Events. Diese synchronization points haben eine deutliche Ähnlichkeit zu den in dieser Arbeit beschriebenen Zustände eines MSpec-Scenarios. Ein solcher Zustand hat nämlich auch eine Menge von Events, die erwartet werden und eine weitere Menge von Events, die verboten sind. Allgemein muss ein MSpec-Scenario während einer Simulation über jede auftretende Nachricht informiert werden. Diese b-threads sind daher eine durchaus sinnvolle Alternative für die Codeübersetzung von MSpec in Java-Code.

6.4. WebSequenceDiagrams.com

Für die Webseite „websequencediagrams.com“ [Web] wurde ein ähnlicher Ansatz entwickelt, wie in dieser Arbeit, wenn auch mit anderem Ziel und Umfang. Die Webseite bietet die Möglichkeit, Szenarien textuell nach einem ähnlichen Schema, wie bei MSpec, einzugeben. Dies ist allerdings nicht zur Spezifizierung von reaktiven Systemen gedacht, sondern zur Umwandlung von Text in ein Diagramm. Aus dem eingegebenen Text wird nämlich ein Sequenzdiagramm generiert, welches das eingegebene Verhalten widerspiegelt. Dieses Diagramm kann in verschiedenen Designs dargestellt und heruntergeladen werden. Der Editor,

in dem das Diagramm erstellt wird, ist relativ einfach und übersichtlich gehalten und bietet eine direkte Vorschau von dem, was der Benutzer eingibt. Eine solche Text-zu-Diagramm-Übersetzung ist eventuell für eine spätere Erweiterung der während dieser Arbeit entstandenen Implementierung interessant.

7. Zusammenfassung und Ausblick

Diese Arbeit wurde verfasst, um eine Möglichkeit zu entwickeln, eine simulierbare, szenariobasierte Spezifikation zu verfassen, ohne von der Notwendigkeit grafischer Editoren abhängig zu sein. Dieses Kapitel fasst noch einmal zusammen, was in dieser Arbeit erklärt und erreicht wurde.

7.1. Zusammenfassung

Das größte Problem der grafischen szenariobasierten Spezifikation ist die schnell wachsende Komplexität der Szenarien und damit der Grafiken. Heutige Computersysteme bestehen aus vielen Einzelkomponenten, die als reaktives System zusammenarbeiten und ein sehr komplexes Verhalten entwickeln können. Ein solches System zu spezifizieren kann mit rein grafischen Mitteln schnell zu unübersichtlich werden. Der in dieser Arbeit vorgestellte Lösungsansatz dieses Problems ist es, die Spezifikation auf eine Textebene zu beschränken und damit dieser grafischen Komplexität Einhalt zu gebieten, ohne die Möglichkeiten der Analyse und Simulation zu verlieren, die bereits von den grafischen MSDs geboten werden. Dieser Lösungsansatz ist besonders deshalb gut, da Text oftmals einfacher zu lesen ist als eine Grafik und zudem beliebig lang werden kann, ohne seine Lesbarkeit zu verlieren. Durch die entwickelte Modellierungssprache MSpec ist dies für die in dieser Arbeit betrachtete Variante der Modal Sequence Diagrams gelungen. Es wurden Äquivalenzen zwischen MSDs und MSpec aufgezeigt und erläutert, wie sie eins zu eins übersetzt werden können. Eine Simulation der MSpec-Spezifikation ist auf die gleiche Weise möglich, wie die der MSDs. Es wurde eine Implementierung in Xtext vorgestellt, die einen Editor aufweist, der mit Autovervollständigung, Code-Validierung und passender Fehlerkorrektur aufwartet. Zudem wurden Konzepte zur Codeübersetzung vorgestellt.
Durch die geglückte Umsetzung der Prinzipien der Modal Sequence Diagrams in eine textuelle Sprache ergibt sich die Möglichkeit, eine komplizierte Spezifikation übersichtlich und ohne die Notwendigkeit grafischer Editoren aufzustellen. Die Sprache MSpec ist das Ergebnis dieser Arbeit und stellt eine erweiterbare

und schnell erlernbare Technik dar, eine solche szenariobasierte Spezifikation mit Hilfe eines benutzerfreundlichen Editors aufzustellen. MSpec bietet damit eine Basis für zukünftige Sprachen zur Modellierung und Simulation multimodaler szenariobasierter Spezifikation.

7.2. Ausblick

Das Werkzeug MSpec steht nun sowohl als Konzept, als auch als Implementierung in Form eines Plugins für Eclipse zur Verfügung, welches von Haus aus einen relativ benutzerfreundlichen Editor mit sich bringt. Es wäre denkbar, in Zukunft eine erweiterte Oberfläche mit besserer Code-Ergänzung und einer Debug-Ansicht zu erhalten. Es können noch einige Verbesserungen des Editors in Richtung Fehlerdetektion und Fehlerkorrektur gemacht werden und für die Codegenerierung sind weitere Konzepte zu implementieren.
Die in dieser Arbeit entstandene Implementierung und die vorgestellten Konzepte genügen der gestellten Zielsetzung und bleiben offen für die weitere Entwicklung, besonders die Übersetzung in Code und die Simulation von Anforderungen betreffend.

A. Xtext-Grammatik von MSpec

Der Code A.1 zeigt die Xtext-Grammatik von MSpec. Das gesamte Projekt ist zu finden im Repository auf `https://bitbucket.org/jgreenyer/scenariotools/branch/mspec` in der Revision 6 vom 09.09.2014.

```
grammar org.scenariotools.mspec.MSpec with org.eclipse.xtext.
   common.Terminals

generate mSpec "http://www.scenariotools.org/mspec/MSpec"

import "http://www.eclipse.org/emf/2002/Ecore" as ecore

XSpecification:
 imports+=XImport*
 'specification' name=ID '{'
  usecases+=XUseCase*
 '}'
;

XImport: 'import' importURI=STRING;

Fqn:ID('.'ID)*;

XUseCase:
 'usecase' name=ID '{'
  collaboration=XCollaboration
  scenarios+=XScenario*
 '}'
;

XCollaboration:
 'collaboration' name=ID '{'
  objs+=XObject*
 '}'
;

XObject:
 type=('obj sys' | 'obj env') classref=[ecore::EClass | Fqn]
    name=ID
;
```

```
XScenario:
 'scenario' type=('r' | 'a') name=ID '{'
  rolebindings+=XRoleBinding*
  features+=XScenarioFeature*
 '}'
;

XRoleBinding:
 'bind' binding=[XObject] 'to' bindto=[XObject]'.'op=[ecore::
    EOperation]
;

XScenarioFeature:
 XMessage | XScenarioLoop | XScenarioAlternative |
    XScenarioParallel | XCondition
;

XMessage:
 'msg' option=XMessageOption sender=[XObject] '->' receiver=[
    XObject] '.' call=[ecore::EOperation] '(' (args+=STRING
    (',' args+=STRING)*)? ')'
;

XMessageOption:
 {XMessageOption}
 '(' (temp=('h' | 'c') ',' exec=('m' | 'e'))? ')'
;

XScenarioLoop:
 {XScenarioLoop}
 'loop' '{'
  features+=XScenarioFeature+
 '}'
;

XScenarioAlternative:
 {XScenarioAlternative}
 'alt' '{'
  cases+=XScenarioAlternativeCase+
 '}'
;

XScenarioAlternativeCase:
 {XScenarioAlternativeCase}
 'case' '{'
  features+=XScenarioFeature+
 '}'
```

```
;

XScenarioParallel:
 {XScenarioParallel}
 'par' '{'
  cases+=XScenarioParallelCase+
 '}'
;

XScenarioParallelCase:
 {XScenarioParallelCase}
 'case' '{'
  features+=XScenarioFeature+
 '}'
;

XCondition:
 XBooleanCondition | XConditionSync
;

XBooleanCondition:
 'cond' string=STRING
;

XConditionSync:
 'sync' roles+=[XObject] (',' roles+=[XObject])+
;
```

Listing A.1: Die Xtext-Grammatik von MSpec

Literaturverzeichnis

[DH01] Werner Damm and David Harel. Lscs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1):45–80, 2001.

[EMF] Eclipse Modeling Framework Project (EMF). `https://www.eclipse.org/modeling/emf/`. letzer Zugriff: August 2014.

[FQN] Fully qualified name - Wikipedia, the free encyclopedia. `http://en.wikipedia.org/wiki/Fully_qualified_name`. letzer Zugriff: August 2014.

[GBPLM13] Joel Greenyer, Christian Brenner, and Valerio Panzica La Manna. The scenariotools play-out of modal sequence diagram specifications with environment assumptions. *Electronic Communications of the EASST*, 58, 2013.

[Gre11] Joel Greenyer. *Scenario-based Design of Mechatronic Systems.* PhD thesis, University of Paderborn, Paderborn, October 2011 2011.

[Har08] David Harel. Can programming be liberated, period? *Computer*, 41(1):28–37, 2008.

[HFKH07] David Harel, Yishai Feldman, and M Krieger-Hauwede. *Algorithmik: die Kunst des Rechnens.* Springer-Verlag New York, Inc., 2007.

[HM03] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine.* Come, Let's Play: Scenario-based Programming Using LSCs and the Play-engine. Springer, 2003.

[HM08] David Harel and Shahar Maoz. Assert and negate revisited: Modal semantics for uml sequence diagrams. *Software & Systems Modeling*, 7(2):237–252, 2008.

[HMW12] David Harel, Assaf Marron, and Gera Weiss. Behavioral programming. *Communications of the ACM*, 55(7):90–100, 2012.

[imp] blog of itemis l.e. » Blog Archive » Xtext Cross References and Scoping – an Overview (Part 1). `http://blogs.itemis.de/leipzig/archives/776`. letzer Zugriff: August 2014.

[MH06] Shahar Maoz and David Harel. From multi-modal scenarios to code: compiling lscs into aspectj. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 219–230. ACM, 2006.

[OCL] OMG - Object Constraint Language (OCL). `http://www.omg.org/spec/OCL/`. letzer Zugriff: August 2014.

[UML] OMG - Unified Modeling Language (UML), V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`. letzer Zugriff: August 2014.

[UML05] UML: Unified modeling language superstructure specification v2.0, July 2005. OMG document `formal/2005-07-04`.

[Web] WebSequenceDiagrams.com - Draw and Edit Sequence Diagrams in seconds. `https://www.websequencediagrams.com`. letzer Zugriff: August 2014.

[Xte] Xtext - Language Development Made Easy! `https://www.eclipse.org/Xtext/`. letzer Zugriff: August 2014.