Universität Paderborn

Fachbereich 17 – Mathematik/Informatik
Arbeitsgruppe Softwaretechnik
Warburger Str. 100
33098 Paderborn

Abbildungen von UML Spezifikationen auf Abstract State Machines

Diplomarbeit
für den integrierten Studiengang Informatik

von

Charalampos Chrysikopoulos
Vogeliusweg 17.204
33100 Paderborn

vorgelegt bei
Prof. Dr. Wilhelm Schäfer

und
Prof. Dr. Franz Josef Rammig

Paderborn, April 2002

Eidesstattliche Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen worden ist. Alle Ausführungen, die wörtlich oder sinngemäß übernommen worden sind, sind als solche gekennzeichnet.

Paderborn, den 07.04.2002

_____

Charalampos Chrysikopoulos

Danksagung

Als erstes möchte ich mich bei meinen Eltern bedanken, die mir dieses Studium ermöglicht haben. Ich danke auch meinem Betreuer Ulrich Nickel für seine wertvollen Ratschläge sowie Herrn Prof. Dr. Wilhelm Schäfer für die Erstkorrektur und Herrn Prof. Dr. Franz Josef Rammig für die Zweitkorrektur der Arbeit. Außerdem danke ich allen Mitarbeitern der AG-Schäfer für die gute Zusammenarbeit und die angenehme Atmosphäre. Schließlich möchte ich mich bei A. Grahl für seine mühsame Korrektur bedanken.

Inhalt

1. Einleitung

1.1 Motivation

In den heutigen eingebetteten Systemen wird der Einsatz von Hardware, der die Funktionalität des Systems realisiert, durch verteilte und komplexe Softwaresysteme ersetzt. Aus diesem Grund werden Modellierungsarten und Softwareentwicklungstechniken gebraucht, um stabile und flexible verteilte Softwaresysteme zu produzieren. Dabei ist die Integration der Softwareentwicklung in den gesamten Systementwicklungsprozess und die Analysetechniken erwünscht. Das ISILEIT Projekt ('Integrative Spezifikation von verteilten Leitsystemen der flexibel automatisierten Fertigung') hat als Ziel, eine durchgängige Methodik für den integrierten Entwurf, die Analyse und die Validierung verteilter Fertigungsleitsysteme zu erarbeiten. Das Case-Tool Fujaba (‚From UML to Java and Back Again'), das im Rahmen von ISILEIT entwickelt wurde, unterstützt die Anwendung dieser Methodik. In Fujaba [17] werden sowohl SDL Diagramme wie auch UML Klassendiagramme, Kollaborations-diagramme, Statecharts und Aktivitätsdiagramme benutzt, um eine ausführbare Spezifikation des Systems zu erzeugen. Die statische und operationale Semantik der einzelnen Konstrukte wurde durch Graphgrammatiken formalisiert. Fujaba stellt also eine automatische Codegenerierung zur Verfügung, durch die die Validierung des Systems erst möglich wird.

Ein weiteres Ziel ist es, eine formale Verifikation des Systems mittels Model-Checking zu schaffen. Dadurch wird das Testen aller möglichen Verhaltensarten des Systems möglich.

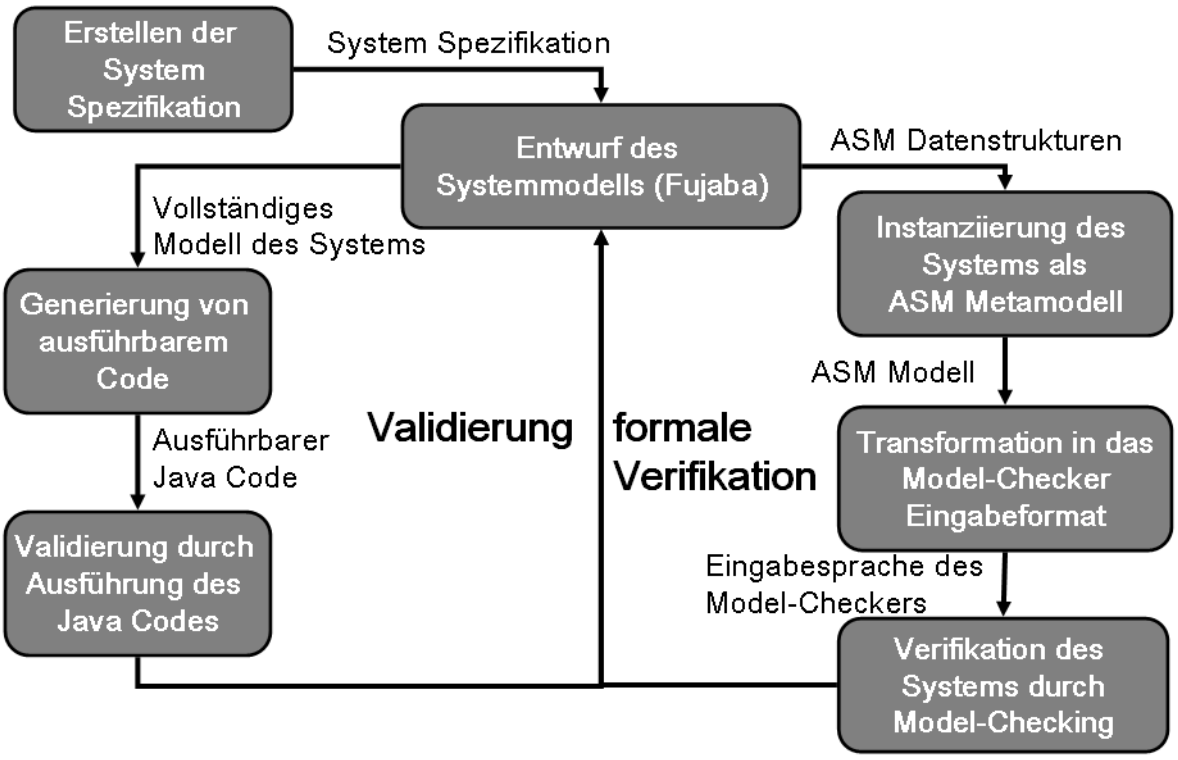

Abbildung 1.1: Entwicklungsprozess eines Systems

In Abbildung 1.1 wird der Entwicklungsprozess eines Systems in ISILEIT graphisch dargestellt. Um die Spezifikation des Systems zu erstellen, wird zunächst nach einer Machbarkeitsstudie die Sicht der Ingenieure über das System, die durch eine semiformale Sprache definiert wird, genutzt. Nachdem die Spezifikation erstellt worden ist, wird diese als Eingabe in die Phase des Entwurfs geleitet.

In der Entwurfsphase werden die statischen und dynamischen Aspekte des Systems durch geeignete Spezifikationssprachen beschrieben. Die Kommunikation im System wird durch SDL Diagramme definiert, während die Modellierung komplexer Objektstrukturen durch UML Diagramme definiert wird. Für diesen Prozess wird Fujaba eingesetzt. Mit Hilfe dieses Tools ist es schließlich möglich, die statischen und dynamischen Aspekte des Systems vollständig zu modellieren. Aus dem vollständigen Modell des Systems ist es weiterhin möglich, automatisch ausführbaren Java Code zu generieren. Dieser kann dann ausgeführt werden. Mit Hilfe von Debugging Tools, wie z. B. DOBS, welches in Fujaba integriert ist, wird es möglich, das System zu visualisieren und sein Verhalten zu betrachten. Somit findet die Validierung des Systems mittels Simulation statt. Schließlich führen unerwünschte Verhaltensweisen des Systems durch die Validierung des ausführbaren Codes zur weiteren Entwicklung des Modells. Die Simulation des Systems ist zwar hilfreich, doch sie setzt voraus, dass sich das Modell des Systems in fortgeschrittenen Phasen befindet und viele Details mitmodelliert worden sind. Ein zweiter Nachteil ist, dass durch das Betrachten des Verhaltens des Systems nicht alle Fehler zu erkennen sind. Aus diesen Gründen ist es wünschenswert, eine formale Spezifikation des Systemmodells anzubieten, die schon in den früheren Phasen des Entwurfs einsetzbar ist.

Die formale Verifikation [12] soll mittels eines Model-Checkers geschehen. Das erfordert, dass das Modell der Entwurfsphase als Eingabe für einen Model-Checker transformiert wird. Für diesen Zwischenschritt wird eine andere formale Methode benutzt, die Abstract State Machines (ASM). ASM ist eine formale Spezifikationssprache, die auf Basis mathematischer Strukturen und eines Zustandsmodells die Abbildung der gesamten Semantik der verschiedenen Spezifikationssprachen auf eine gemeinsame Abstraktionsebene möglich macht. Um die Zustandsmenge der Berechnung des Model-Checkers klein zu halten, gibt es die Möglichkeit verschiedene Abstraktionsebenen der Spezifikation auszuwählen. In der Entwurfsphase wird also das Modell in Form von Initialwerten für vordefinierte ASM Datenstrukturen generiert. Diese werden als nächstes mit einem ASM Metamodell gebunden, das eine Abstraktionsebene repräsentiert und als Interpreter der Datenstrukturen zu verstehen ist. Die bewerteten ASM Strukturen mit dem ausgewählten ASM Metamodell bilden dann das ASM Modell des Systems. In der nächsten Phase wird der ASM Code in eine Low-level Sprache transformiert, die als Eingabe für den Model-Checker dient. Mit dieser Eingabe ist es schließlich möglich, das System mit einem Model-Checker zu überprüfen und aus den Ergebnissen Informationen über den weiteren Entwurf zu gewinnen. Der Vorteil der formalen Verifikation gegenüber der Validierung ist, dass alle möglichen Verhaltensweisen des Modells systematisch überprüft werden.

Ziel dieser Diplomarbeit ist es, die Abbildung einer Teilmenge der Spezifikationen der Entwurfsphase auf ASM zu definieren und einen Teil der niedrigsten Abstraktionsebene des ASM Modells zu spezifizieren. Dabei werden Klassendiagramme und Aktivitätsdiagramme sowie auch die Semantik der Story Patterns auf ASM abgebildet. Schließlich wird ein automatischer Generierungsmechanismus vorgestellt, der die oben genannten Spezifikationen in ASM Code generiert. Die Semantik der Statecharts wird in [13] beschrieben. Statecharts werden mittels Story Patterns formalisiert. Das bedeutet, dass die Semantik von Statecharts

durch die Konzepte dieser Arbeit auf ASM abgebildet werden können. Statecharts werden jedoch in dieser Arbeit nicht betrachtet.

1.2 Aufbau der Arbeit

In Kapitel 2 wird das Vorgehen der formalen Spezifikation näher betrachtet. Dabei wird grob das Bilden des ASM Modells mittels ASM Metamodellen beschrieben. In Kapitel 3 wird eine Einführung in ASM gegeben. Für die Zwecke dieser Diplomarbeit wird das ASM Tool AsmL vorgestellt. Anschließend werden die wichtigsten Strukturen und Ausdrücke mit Beispielen erläutert. Kapitel 4 beschäftigt sich mit der Abbildung der einzelnen Spezifikationen. Zuerst werden die statischen Eigenschaften des Systems betrachtet, die durch das Klassendiagramm modelliert werden. Dabei werden ASM Datenstrukturen beschrieben, die die Semantik des Klassendiagramms auf ASM definieren. Die Methoden des Klassendiagramms werden durch Story Patterns und Aktivitätsdiagramme modelliert. Story Patterns beschreiben das Verhalten der Methode, während Aktivitätsdiagramme die Reihenfolge der Ausführung des einzelnen Story Patterns bestimmen. Die Abbildung von Story Patterns geschieht ähnlich wie beim Klassendiagramm. Anschließend wird auch das Ausführen der Story Patterns in ASM beschrieben. Beim Ausführen eines Story Patterns erfolgt eine Teilgraphensuche; sie wird durch eine Regel realisiert. Diese Regel wird aufgrund ihre Komplexität in einem eigenen Kapitel betrachtet. Nachdem die Story Patterns in ASM abgebildet sind, werden als nächstes die Methoden betrachtet. Für jede Methode des Klassendiagramms wird eine ASM Funktion generiert. Der Rumpf der Funktion wird durch das entsprechende Aktivitätsdiagramm bestimmt. In Kapitel 5 wird die Integration in das Case-Tool Fujaba beschrieben. An einem Beispiel wird in Kapitel 6 die Generierung des ASM Codes demonstriert. Schließlich werden in Kapitel 7 nach einer Zusammenfassung noch drei weiterführende Punkte angesprochen: die Abbildung von Java Ausdrücken in ASM, die Unterstützung verschiedener Abstraktionsebenen und die Anbindung an einen Model-Checker.

2. Grundlagen

Für die Validierung wurde in Fujaba die statische und dynamische Semantik der Spezifikation durch Graphgrammatiken formalisiert. Diese haben den Vorteil, dass sie für Graphtransformationen sehr verbreitet sind und passend zu den objektorientierten Konzepten sind. Die Graphgrammatiken beschreiben die Struktur und das Verhalten des Systems, indem sie Klassen und Objekte auf Graphen und Objektinteraktionen auf Graphersetzungsregeln abbilden. Dies bildet eine robuste Basis für automatische Codegenerierung und Simulation des Systems.

Der Vorgang der formalen Verifikation des Systemmodells, der im Kapitel 1 beschrieben worden ist, erfordert als Zwischenschritt die Abbildung aller Spezifikationssprachen, die in der Phase des Entwurfs benutzt werden, auf einer gemeinsamen Spezifikationsebene, so dass diese als Eingabe für den Model-Checker transformiert werden kann. Für die Realisierung des Zwischenschrittes wurde die Spezifikationssprache Abstract State Machines (ASM) ausgewählt. Im ISILEIT Projekt gibt es bereits ein Konzept für die Abbildung von SDL auf ASM und eine Anbindung an den Model-Checker SMV [10]. Im Allgemeinen sind die heutigen Systemmodelle zu groß und die Rechnerleistung zu klein, um den praktischen Einsatz der formalen Verifikation mittels Model-Checking für das gesamte Modell zu realisieren. Aus diesem Grund wird die Möglichkeit zur Verfügung gestellt, das Systemmodell in verschiedenen Abstraktionsebenen zu beschreiben, um die Zustandsmenge bei der Ausführung des Model-Checkers möglichst klein zu halten. Dazu werden ASM Metamodelle benutzt. Ein ASM Metamodell ist als Interpreter der ASM Kodierung des Systemmodells zu verstehen. Durch die Wahl eines ASM Metamodells werden Einzelheiten des Systems abstrahieren, die für den aktuellen Zeitpunkt des Entwurfs unwichtig sind.

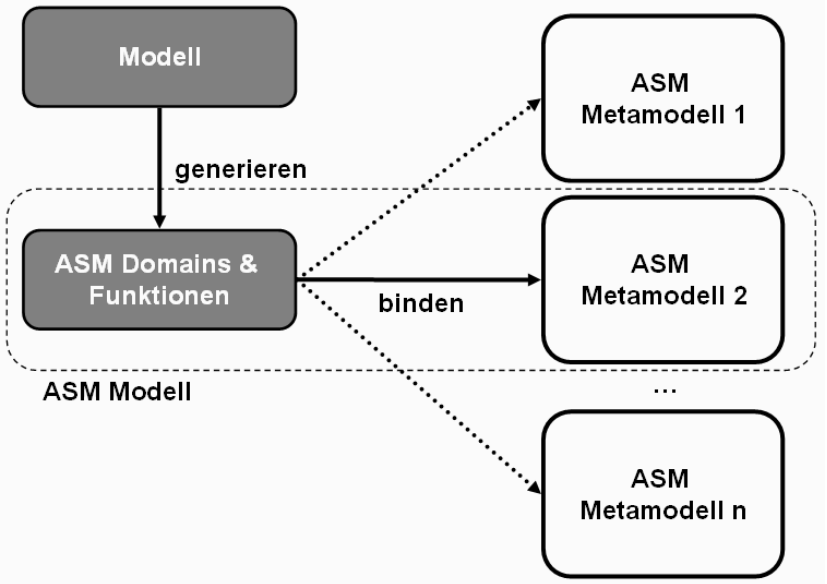

Abbildung 2.1: Erstellen des ASM Modells

In Abbildung 2.1 ist der Vorgang des Erstellens des ASM Modells graphisch dargestellt. Das Systemmodell wird in ASM Datenstrukturen automatisch kodiert. Der generierte ASM Code wird dann mit einem bestehenden ASM Metamodell verknüpft. Jedes ASM Metamodell

entspricht einer vordefinierten Abstraktionsebene. Die Auswahl des ASM Metamodells hängt davon ab, in welcher Phase des Entwurfs sich das Modell befindet und wie detailliert das Modell verifiziert werden soll. Das ASM Modell wird dann in eine Model-Checker Sprache oder in eine BDD Repräsentation transformiert und als Eingabe für den Model-Checker gegeben. Als nächstes können die Ergebnisse ausgewertet werden, und dann ist es möglich, Änderungen im Systemmodell vorzunehmen. Diese Methodik ermöglicht die Verkleinerung der Zustandsmenge, die mit dem Model-Checker berechnet wird, und ermöglicht die Verifizierung des Systems in einer beliebigen Abstraktionsebene. Wie bereits in Kapitel 1 erwähnt, wird in dieser Diplomarbeit eine Abbildung von Klassendiagrammen und Aktivitätsdiagrammen sowie auch die Semantik der Story Patterns auf ASM definiert. Diese Abbildung beschreibt einen Teil der niedrigsten Abstraktionsebene eines ASM Modells, d.h. das Systemmodell ist in jedem Implementierungsdetail auf ASM abgebildet.

2.1 Überblick

Abbildung 22 zeigt einen Überblick der ASM Spezifikation und beschreibt, aus welchen Diagrammarten jeder Teil der Spezifikation generiert wird. Auf der linken Seite werden die verschiedenen Diagramme dargestellt, die im Case-Tool Fujaba entwickelt worden sind und das Systemmodell beschreiben. Auf der rechten Seite der Abbildung sieht man den gesamten ASM Code, der das ASM Modell des Systems ist. Da die Abbildung von SDL Diagrammen und Statecharts in dieser Arbeit nicht betrachtet wird, existieren für sie auf im ASM Modell keine Einträge. Wie schon erwähnt, bildet dieses Modell ein Teil der niedrigsten Abstraktionsebene der ASM Spezifikation.

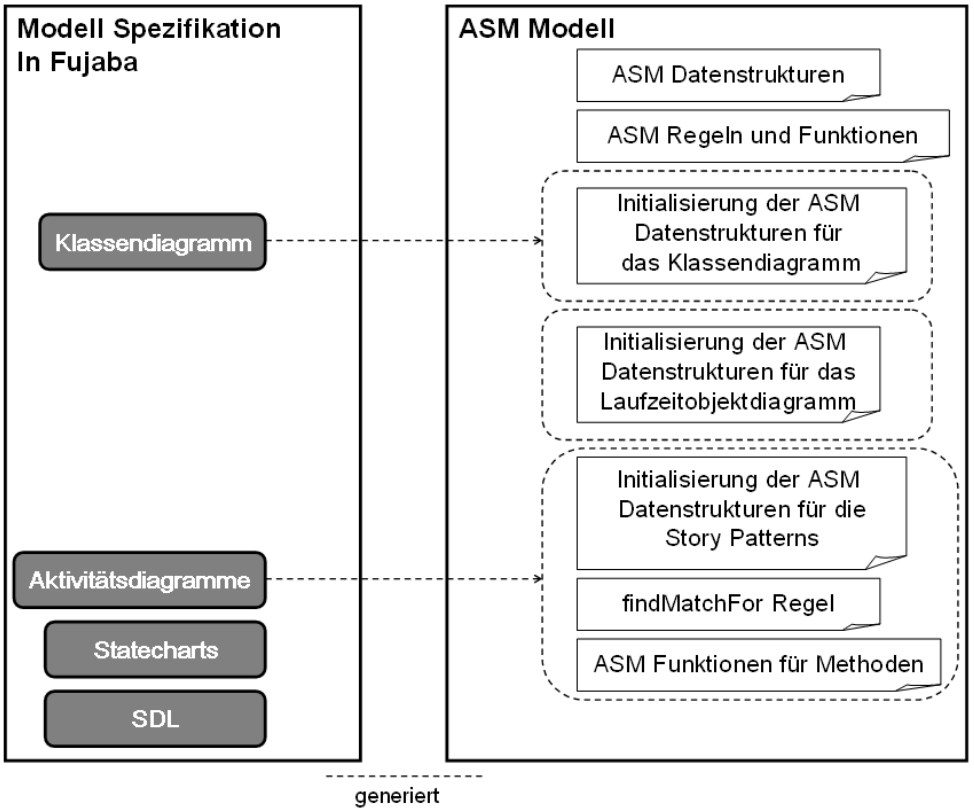

Abbildung 2.2: Überblick der Abbildung der Modellspezifikation auf ASM

Die ASM Datenstrukturen sind statisch und werden nicht generiert. Sie beschreiben die Struktur des Klassendiagramms, der Laufzeitobjektstruktur und der Story Patterns. Für diese Strukturen werden die initialen Werte generiert. Weiterhin existieren andere Strukturen, die der internen Funktionalität des Modells dienen. Außer den Datenstrukturen existieren Regeln und Funktionen, die für die Zustandsübergänge des ASM Modells zuständig sind. Ein großer Teil dieser Regeln beschreibt die Semantik der Story Patterns.

Aus den Klassendiagrammen, die die statischen Eigenschaften des Modells beschreiben, werden die Initialwerte der ASM Datenstrukturen generiert, die das Klassendiagramm beschreiben. Nach der Initialisierung der Datenstrukturen des Klassendiagramms folgt die Initialisierung der Laufzeitobjektstruktur. Die Methoden, die durch Aktivitätsdiagramme und Story Patterns modelliert werden, realisieren das Verhalten der Klassen. Das Verhalten der Methoden wird durch den Einsatz von Story Patterns beschrieben, dessen operationale Semantik durch Graphersetzungsregeln formalisiert sind. Ein Aktivitätsdiagramm, welches die Story Patterns der Methode enthält, beschreibt den Kontrollfluss der Methode. Zunächst werden die Initialwerte für die ASM Datenstrukturen generiert, die die Story Patterns beschreiben. Für die Ausführung der Story Patterns wird die Regel *FindMatchFor* generiert. Schließlich wird für jede Methode des Klassendiagramms eine entsprechende ASM Funktion generiert, deren Rumpf durch das entsprechende Aktivitätsdiagramm bestimmt wird.

Die Abbildung von Statecharts, sowie auch von SDL auf ASM wird hier nicht näher betrachtet. In Kapitel 4 werden die oben beschriebenen Abbildungen im Detail erläutert. Dabei wird die Abbildung 2.2 kapitelweise aufgebaut, um eine bessere Verständigung zu ermöglichen. Im folgenden Kapitel wird eine Einführung in ASM und in die AsmL Sprache gegeben, die von dem Tool AsmL benutzt wird, um ausführbare ASM Spezifikationen zu definieren.

3. Einführung in ASM und AsmL

Dieses Kapitel ist eine Einführung in die Sprache ASM. Es wird erklärt, was ASM ist, und es werden einige wichtige Begriffe erläutert. Eine detaillierte Einführung in ASM ist in [1] und [2] zu finden. Weiterhin wird die AsmL Sprache des Werkzeugs AsmL, welches für die Zwecke dieser Arbeit benutzt worden ist, näher betrachtet.

3.1 ASM (Abstract State Machines)

Die ASM wurden von Yuri Gurevich entwickelt, um die Lücke zwischen Spezifikationsmethoden und -modellen zu schließen. Die Turingthese besagt, dass jeder Algorithmus durch eine Turingmaschine simuliert werden kann. Sie ist eine korrekte Annäherung an dieses Problem. Eine Turingmaschine kann die Semantik eines Algorithmus beschreiben, es ist aber praktisch unrealistisch, einen Algorithmus durch eine Turingmaschine zu beschreiben, da sie mit Bits arbeitet. Es existiert also eine große Lücke zwischen der Abstraktionsebene eines Algorithmus und einer Turingmaschine.

Mit ASM ist es möglich, einen beliebigen Algorithmus zu simulieren, ohne dass dieser in einer niedrigeren Abstraktionsebene beschrieben wird; er bleibt in seiner natürlichen Abstraktionsebene. Das besagt auch die ASM These: "Für jeden Algorithmus *A* in einer bestimmten Abstraktionsebene gibt es eine ASM Maschine *B*, die schrittweise *A* simuliert". Diese These ist für den sequentiellen Fall bewiesen [3], ein Beweis für den verteilten Fall steht noch aus.

Ein Problem ist, die Korrektheit des Algorithmus mathematisch zu beweisen. Um das zu erreichen, wird ein Modell in einer hohen Abstraktionsebene konstruiert und seine Korrektheit durch Beobachtung und Experimentieren bewiesen. Danach ist es möglich, dieses Modell durch schrittweise Verfeinerung formaler zu spezifizieren. Um die Korrektheit zu überprüfen existieren zahlreiche Werkzeuge[1], die ASM Codes ausführen. ASM benutzt mathematische Strukturen, um die Zustände der Berechnung eines Algorithmus zu beschreiben. Dadurch wird an Präzision gewonnen. Die Syntax der Sprache ist einfach, so dass sie auch für Anfänger im Programmieren verständlich ist. Der ASM Code ist ausführbar und das hat den Vorteil, dass die Spezifikation direkt kompiliert und auf Korrektheit kontrolliert werden kann. Ein weiterer Vorteil von ASM ist die Anpassbarkeit. ASM erlaubt die Beschreibung eines Systems in verschiedenen Abstraktionsebenen. Durch mehrere Abstraktionsebenen ist es möglich, verschiedene Merkmale des Systems zu betrachten, indem für einzelne Analysen unwichtige Einzelheiten nicht erscheinen. Jede niedrigere Abstraktionsebene muss nach Gurevich, wie oben schon erwähnt, auf Korrektheit bewiesen werden. Dabei muss die Korrektheit der vorherigen Abstraktionsebene beachtet werden.

[1] In [28] ist eine Sammlung von ASM Werkzeugen zu finden.

3.1.1 Berechnung

ASM definiert ein zustandsbasiertes Berechnungsmodell. Der Ablauf eines Algorithmus in ASM wird durch Zustände und Transitionen beschrieben. Die Berechnung des Algorithmus ist eine endliche oder unendliche Folge von Zuständen S_i, beginnend mit einem Initialen Zustand S_0. In jedem Zustand ist eine endliche Anzahl von Aktualisierungen erlaubt, und die Menge aller Aktualisierungen bildet eine Transitionsregel t_i. Die Aktualisierungen haben keine Auswirkung zu dem Zustand (double buffering), solange die nicht aktiviert werden. Das Ausführen einer Transitionsregel, die zu einem klar definierten Zeitpunkt erfolgt, aktiviert die Aktualisierungen und hat als Ergebnis den Wechsel eines Zustands S_i zu einem neuen Zustand S_{i+1}, der durch die Menge der Aktualisierungen der Transitionsregel t_i gebildet wird.

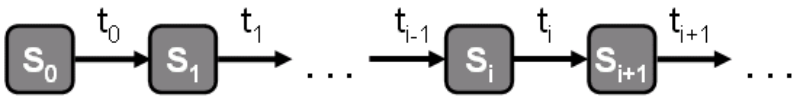

Abbildung 3.1: Eine Berechnung

In 3.2.6 wird das Prinzip der zustandsbasierten Berechnung näher betrachtet. Ein detailliertes Beispiel wird in 3.2.10 gegeben.

3.1.2 Statische Algebra

Die Berechnung eines Algorithmus in ASM wird durch eine Folge von Zuständen beschrieben. Ein Zustand kann durch eine mathematische Struktur repräsentiert werden, die *statische Algebra* genannt wird. In Abbildung 3.1 sind S_0 bis S_{i+1} Zustände der Berechnung eines Algorithmus. Es handelt sich dabei um statische Algebras. Eine *Signatur s* ist eine Menge von Funktionsnamen. Eine statische Algebra einer Signatur *s* ist eine nichtleere Menge *S* von Interpretationen der Funktionsnamen in *s*. Eine *Funktion* von *r* Parametern ist eine Abbildung von S_r nach *S*. Eine Funktion ohne Parameter ist ein *ausgezeichnetes Element*. Die Elemente `true` und `false` sind solche ausgezeichneten Elemente und sind in jeder statischen Algebra enthalten.

Ein Beispiel: Die Signatur *s* besteht aus den Funktionsnamen `0` und `1`, die keine Parameter haben, und aus den Funktionsnamen `+` und `*`. Eine statische Algebra über diese Signatur könnte die Menge aller natürlichen Zahlen sein. Die Interpretation von `+` und `*` ist die Addition und Multiplikation natürlicher Zahlen.

In statischen Algebras sind keine partiellen Funktionen erlaubt. Für diesen Zweck existiert ein ausgezeichnetes Element `undef`, mit dem eine partielle Funktion definiert werden kann. Die Funktion `divide(a, b)` über die Menge aller natürlicher Zahlen kann nur definiert werden, wenn `b` ungleich `0` ist. Es gilt also `divide(a,0) =` **`undef`**.

Ein *Universum* ist eine Funktion *U* ohne Parameter mit Werten in `{true, false}`. In einem Universum *U* gilt, dass für jedes Element e_1 in *U* $U(e_1)$=**`true`** und für jedes e_2, das nicht in *U* ist, $U(e_2)$=**`false`**. In der Menge der natürlichen Zahlen z. B. wäre `U(23)=`**`true`** und `U(2/3)=`**`false`**.

Relationen sind spezielle Funktionen über einem Universum. Eine Relation *R* über einem Universum *U* ist nicht definiert, falls mindestens ein Argument kein Element von *U* ist. Zum Beispiel gilt für die Relation < über dem Universum der natürlichen Zahlen: `<(2,3)=`**`true`** und `<(3,2)=`**`false`**.

3.1.3 Transitionsregeln

Transitionsregeln sind zuständig für die Transformation einer statischen Algebra in eine andere. In Abbildung 3.1 sind alle Pfeile Transitionsregeln. Dabei wird die Algebra verändert; es findet also eine Evolution der Algebra statt. Aus diesem Grund wurde früher die ASM "evolving algebra" genannt. Eine statische Algebra beschreibt also eine Momentaufnahme der Berechnung.

Eine Transitionsregel ist eine Menge von Aktualisierungen. Die einfachste Form einer Aktualisierung ist die Änderung einer Funktion auf einem Wert ihres Definitionsbereichs. Eine Aktualisierung *U*, die nur unter einer Bedingung *B* anwendbar ist, also eine bewachte Aktualisierung, hat die Form

`if` *B* **`then`** *U* **`endif`**

wobei *B* ein Boolescher Ausdruck ist.

Alle Aktualisierungen einer statischen Algebra werden parallel ausgeführt. Das hat zur Folge, dass inkonsistente Zustände erzeugt werden können, z.B. `U(e)=1` und `U(e)=2`. In diesem Fall wird die Berechnung abgebrochen.

3.1.4 Statische, dynamische und externe Funktionen

Die Funktionen einer statischen Algebra unterteilen sich in statische, dynamische und externe Funktionen. Statische Funktionen sind Funktionen, die nicht aktualisiert werden können, sie bleiben also während der Berechnung des Algorithmus unveränderbar. Dynamische Funktionen sind solche, die während der Ausführung der ASM aktualisiert werden können. Sie repräsentieren daher den internen Zustand des Systems. Statische Funktionen werden benutzt, um auf Daten zuzugreifen, dynamische Funktionen, um Daten zu aktualisieren. Externe Methoden können, wie die statischen, nicht aktualisiert werden, sie können aber in jedem Zustand anders sein. Eine externe Funktion gibt, vergleichbar einem Orakel, einen zufälligen Wert und wird nicht von ASM kontrolliert. Solche Funktionen werden benutzt, um eine Schnittstelle mit dem Benutzer oder mit dem Betriebssystem zu modellieren.

3.1.5 Weitere Eigenschaften von ASM

Nichtdeterminismus

Ein Algorithmus in ASM kann *nichtdeterministische* Entscheidungen treffen. So eine Entscheidung wird mit dem Konstrukt `choose` realisiert.

```
choose u in U
  …
endchoose
```

Sei *U* ein Universum, dann wird mit obigem Code nichtdeterministisch ein Element *u* aus *U* ausgewählt. Das Element *u* wird dann bis zu dem Term *endchoose* lokal sichtbar sein.

Rekursion

Die Funktionen in ASM können rekursiv aufgerufen werden.

Verteilte ASM

Verteilte Systeme können in ASM durch die Benutzung von Multiagenten modelliert werden. Die Idee ist, dass das System aus mehreren Agenten besteht, die Elemente aus einer endlichen Menge *AGENT* sind. Jeder Agent führt sein eigenes Programm aus und kann sich selber durch das ausgezeichnete Element `self` identifizieren. Agenten können in Gruppen geteilt werden und Agenten derselbe Gruppe können miteinander kommunizieren. Weiterhin kann ein Agent während der Berechnung des Algorithmus aktiv oder nicht aktiv sein. Eine sequentielle Berechnung ist in diesem Fall eine Folge von Zuständen S_0, S_1, …, S_{n-1}, S_n, wobei S_0 der initiale Zustand ist und S_{n+1} durch die Ausführung eines Agents aus dem Zustand S_n berechnet wird. Wird für die Berechnung eines neues Zustandes eine Menge von Agenten ausgeführt, so handelt es sich um eine quasi-sequentielle Berechnung.

3.1.6 ASM Werkzeuge

Für die Zwecke dieser Diplomarbeit wurden zwei ASM Werkzeuge betrachtet. ASM Workbench [10] basiert auf der Spezifikationssprache ASM-SL (ASM based Specification Language), die eine eingeschränkte Dialekt von ASM ist. Ein wichtiger Vorteil von ASM-SL ist eine zukünftige Schnittstelle für Model-Checking (ASM2SMV)[2]. Für die Fehlersuche existiert eine graphische Oberfläche mit einer Visualisierungskomponente.

Das zweite Werkzeug, das für die Zwecke dieser Diplomarbeit benutzt wurde, ist die AsmL (Abstract state machine Language), die von Microsoft entwickelt wird. AsmL ist eine relativ neue Spezifikationssprache, die syntaktisch und semantisch verwandt mit ASM-SL ist, was die Wiederverwendung der Schnittstelle zum SMV Model-Checker erlaubt, und wird von Yuri Gurevich (dem Entwickler von ASM) betreut. Durch die Wahl von AsmL werden aktuelle technologische Entwicklungen und Trends gefolgt. Die Version des Compilers, mit dem diese Arbeit entstanden ist, ist 1.5v1h. Ein Nachteil dieser Version von AsmL ist, dass es nicht möglich ist, eigene Domänen, die wiederum für die Definition neuer Domänen benutzt werden, zu definieren. Dies soll aber in der zukünftigen Version unterstützt werden. Außerdem soll die zukünftige Version von AsmL auch eine graphische Oberfläche bieten, die die Visualisierung der Berechnung übernehmen soll. Da AsmL ein viel versprechendes Projekt ist, wurde es für die Zwecke dieser Diplomarbeit ausgewählt.

[2] Die Anbindung an den Modellchecker SMV wird in Kapitel 6 grob beschrieben.

3.2 AsmL (Abstract state machine Language)

In diesem Kapitel wird eine kurze Einführung in die Sprache AsmL gegeben, die auf die Bedürfnisse dieser Diplomarbeit begrenzt ist. Weitere Informationen über die AsmL finden sich in [9]. Um eine bessere Verständigung zu schaffen, wird in diesem Teilkapitel das Beispiel eines Graphen betrachtet. Der Graph besteht aus Knoten und Kanten. Jeder Knoten und jede Kante besitzen einen Namen, eine Kante verbindet zwei Knoten.

3.2.1 Typen

Vordefinierte Typen sind `undef`, `Boolean`, `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Char`, `String`. Wie auch in ASM ist `undef` ein ausgezeichnetes Element, das nicht definiert ist.

Vordefinierte Typen	
`Boolean`	**`Long`**
`Byte`	**`Float`**
`Short`	**`Double`**
`Integer`	**`Char`**
`String`	

Tabelle 3.1: Vordefinierte Typen in AsmL

3.2.2 Behältertypen

Um eine endliche Menge von Elementen eines bestimmten Typs zu definieren, wird `Set of T` oder `Set[T]` benutzt, wobei `T` der gewünschte Typ ist. Um eine endliche geordnete Menge von Elementen eines Typs `T` zu deklarieren, wird `Seq of T` oder `Seq[T]` benutzt. Um eine Abbildung von einer endlichen Menge von Elementen des Typs `S` auf einer endlichen Menge von Elementen des Typs `T` zu beschreiben, wird `Map of S to T` oder `S->T` benutzt. Schließlich wird mit $T_1*T_2*T_3*\ldots*T_n$ ein n-Tupel von Elementen $(t_1, t_2, t_3, \ldots, t_n)$ definiert, wobei t_1 vom Typ T_1 ist, t_2 vom Typ T_2 usw.

Vordefinierte Strukturen	
`Set of` `T` **`Set`**`[T]`	Endliche Menge von Elementen des Typs T
`Seq of` `T` **`Seq`**`[T]`	Endliche geordnete Menge von Elementen des Typs T
`Map of` `S` **`to`** `T` `S -> T`	Abbildung (Map) von einer endlichen Menge von Elementen vom Typ S auf einer endlichen Menge von Elementen vom Typ T
$T_1*T_2*T_3*\ldots*T_n$	n-Tupel von Elementen $(t_1,t_2, t_3,\ldots,t_n)$, wobei t_1 vom Typ T_1 ist, t_2 vom Typ T_2 usw.

Tabelle 3.2: Vordefinierte Strukturen in AsmL

3.2.3 Strukturen

AsmL bietet eine Vielzahl von Strukturen. Im Rahmen dieser Diplomarbeit werden Strukturen in Form von Klassen benutzt, jedoch kein Gebrauch der Objektorientierung von AsmL gemacht. Der Grund dafür ist, dass ASM Objektorientierung nicht unterstützen, und Ziel ist es, ASM Code zu generieren, der sehr nah an der ursprünglichen ASM Spezifikation ist und kompatibel mit der vorhandenen SDL Spezifikation. Der Grund für die Benutzung von Klassen als Strukturen ist, dass AsmL noch keine Domains unterstützt. D.h. es ist nicht möglich, verschachtelte Strukturen zu bilden. Die Unterstützung von Domains ist ein zukünftiges Ziel im AsmL Projekt und es ist denkbar, dass der hier beschriebene generierte AsmL Code dadurch verfeinert werden kann.

Elemente, Mengen und Abbildungen werden also als Strukturen in Form von Klassen definiert. Der folgende AsmL Code definiert ein Element des Typs `String`. Es wird eine Klasse *Node* deklariert, die ein Attribut *name* vom Typ `String` enthält.

```
01 class Node
02   name v as String    // Attribut name vom Typ String
```

Um ein neues Element dieser Struktur zu definieren, wird der entsprechende Konstruktor benutzt. Der Standardkonstruktor hat keine Parameter und erzeugt ein Element mit Attributwert gleich `undef`. Es ist möglich, ein Element mit dem Konstruktor zu initialisieren, indem in der gleichen Reihenfolge, wie sie in der Klassendeklaration beschrieben worden sind, Werte für die Attribute als Parameter gegeben werden.

```
01 let einKnoten1 as Node = new Node()          // Konstante
02 var einKnoten2 as Node = new Node("track1")  // Variable

03 einKnoten1.name := "track2"
```

In Zeile 1 wird ein neues Element *einKnoten1* mit dem Standardkonstruktor erzeugt. In Zeile 2 wird ein anderes Element *einKnoten2* erzeugt und mit dem Wert *„track1"* initialisiert. Jedes Objekt in AsmL hat eine Identifikationsnummer, im Unterschied zu den Domains, da jedes Element unabhängig von seinem Wert unterschiedlich ist. Aus diesem Grund entsteht beim Vergleichen von Elementen ein zusätzlicher Aufwand, es wird aber an Präzision gewonnen. In Zeile 3 wird das Attribut *name* des Elementes *einKnoten1* durch die Punkt Notation mit *„track2"* initialisiert. Der `let` Ausdruck in Zeile 1 deklariert eine Konstante, d.h. der Name *einKnoten1* kann keinen anderen Wert bekommen. Der `var` Ausdruck in Zeile 2 deklariert eine Variable mit dem Namen *einKnoten2*. Wäre in der Struktur *Node* das Attribut *name* als `var` deklariert worden, dann wäre es möglich, den Wert des Attributs des Objekts *einKnoten1* zu ändern, obwohl der Name des Objekts seinen Wert nicht ändern lässt. Die Struktur *Nodes* definiert eine Menge von Elementen des Typs *Node*.

```
01 class Nodes
02   var elements as Set[Node]    // Attribut als Variable deklariert

03 let setOfNodes = new Nodes( {einKnoten, einKnoten2} )

04 einKnoten3 = new Node(„Shuttle")
05 setOfNodes.elements(einKnoten3):=true
06 einKnoten4 = new Node(„Station")
07 setofNodes.elements() := setofNodes.elements() union {einknoten3}
```

Das Attribut *elements*, das als Variable in der Struktur *Nodes* deklariert ist, hat den Typ einer endlichen Menge mit Elementen des Typs *Node*. Es ist zu beachten, dass nicht die Struktur *Nodes* die Menge ist, sondern das Attribut *elements* in der Struktur. Trotzdem wird in dieser Arbeit so eine Struktur als Menge bezeichnet.

In Zeile 3 wird ein neues Element des Typs *Nodes* deklariert und durch den Konstruktor mit dem Wert *{einKnoten, einKnoten2}* initialisiert, der eine Menge mit zwei Elementen beschreibt. In Zeile 5 wird das neu erzeugte Element *einKnoten3* in die Menge hinzugefügt, indem eine Hilfsfunktion von AsmL für Mengen benutzt wird. Jede Menge besitzt eine Funktion, die den gleichen Namen wie die Menge hat und als Parameter ein Element der Menge bekommt. Gehört das Element in die Menge, so gibt die Funktion den Wert `true` zurück, sonst `false`. Durch diese Funktion ist es möglich, Elemente in eine Menge hinzuzufügen, indem der Wert dieser Funktion für das entsprechende Element auf `true` gesetzt wird.

Eine andere Möglichkeit, ein Element oder mehrere in eine Menge hinzuzufügen, ist die Vereinigung zweier Mengen. In Zeile 7 wird ein weiteres Element des Typs *Node* in die Menge *setOfNodes.elements* hinzugefügt, indem eine Vereinigung mit der Menge *{einknoten3}* erzeugt wird ist.

```
01 class Edge
02   name as String

03 class Edges
04   elements as Edge -> Node * Node
```

In Zeile 1 wird die Struktur *Edge* deklariert, die gleich der Struktur *Node* ist. In Zeile 4 wird die Struktur *Edges* deklariert. Das Attribut *elements* der Struktur *Edges* ist eine Abbildung (`Map`). Definitionsbereich von *elements* ist eine Menge von Elementen des Typs *Edge*, Bildbereich ist eine Menge von Elementen des Typs *Node* * *Node*.

```
01 oneEdge1 = new Edge(„trackHasShuttle“)
02 oneEdge2 = new Edge(“nextTrack”)
03 oneEdge3 = new Edge(“trackHasStation”)

04 setOfEdges = new Edges({oneEdge1 |-> (einKnoten1, einKnoten3),
                           oneEdge2 |-> (einKnoten2, einKnoten4),
                           oneEdge3 |-> (einKnoten1, einKnoten2)})
```

In den Zeilen 1 bis 3 werden drei Elemente des Typs *Edge* deklariert und initialisiert. In Zeile 4 wird das Element *setOfEdges* deklariert und mit drei Elementen initialisiert. Ein Element einer Abbildung wird durch die Notation ‚`|->`’ beschrieben. Das erste Element der Abbildung *setOfEdges.elements* ist die Abbildung des Elements *oneEdge1* auf dem Tupel *(einKnoten1, einKnoten3)*. Um auf den Definitionsbereich der Abbildung zuzugreifen, wird die Funktion **`dom`**`(M as Map)` benutzt, die als Parameter die Abbildung bekommt und als Rückgabewert den Definitionsbereich der Abbildung gibt.

```
01 defBereich as Set[Edge] = dom(setOfEdges.elements)
02 bildBereich as Set[Node * Node] = ran(setOfEdges.elements)
```

In Zeile 1 wird eine Menge von Elementen des Typs *Edge* deklariert und mit dem Definitionsbereich der Abbildung *setOfEdges.elements* initialisiert. Entsprechend wird auch in Zeile 2 eine Menge *bildBereich* von Elementen des Typs *Node* * *Node* deklariert, die mit der Menge der Bilder der Abbildung *setOfEdges.elements* initialisiert wird.

Es besteht auch die Möglichkeit, Mengen oder Abbildungen durch iterierte Ausdrücke zu definieren.

```
01 let A = {1, 2, 3, 4, 5}
02 let B = {1..10}    // ist gleich {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
03 let C = { i | i in B where i<6}   // ist gleich {1, 2, 3, 4, 5}
```

In den Zeilen 1 und 2 werden zwei Mengen mit Elementen vom Typ `Integer` definiert. *A* enthält die Elemente 1 bis 5 und *B* die Elemente 1 bis 10. In Zeile 3 wird eine Menge *C* definiert, die auch Elemente vom Typ `Integer` enthält. Die Initialisierung der Menge *C* geschieht durch einen iterierten Ausdruck, der besagt, dass in der Menge *C* alle Elemente der Menge *B* enthalten sind, die kleiner als 6 sind. Es gilt also, dass die Mengen *A* und *C* gleich sind.

In der gleichen Weise ist es möglich, eine Abbildung zu initialisieren.

```
01 let A = {1..10}
02 let B = {i |-> i+1 | i in A where i<5}  // {1|->2, 2|->3,3|->4}
03 let C = {2|->3, 3|->4}
04 writeln(B(2))
```

In Zeile 2 wird eine Abbildung *B* definiert, deren Definitionsbereich aus allen Elementen aus *A* besteht, die kleiner 5 sind. Jedes Element *i* des Definitionsbereiches von *B* hat als Bild das Element *i+1*. Die Menge *C* ist schließlich gleich der Menge *B*. In Zeile 4 wird die Zahl 3 ausgegeben.

3.2.4 Funktionen und Regeln

Eine Funktion besteht aus einem Namen, einer Liste von Eingabeparameternamen und deren Typ und einem Ausgabeparametertyp. Funktionen haben zwei Zwecke, nämlich die Abfrage von Daten in einem aktuellen Zustand der Maschine oder die Aktualisierung der Daten im aktuellen Zustand. Im zweiten Fall wird die Funktion Regel genannt.

```
01 getName(n as Node) as String =
02   return n.name
```

Die Funktion *getName* bekommt als Parameter einen Knoten und gibt den Namen des Knotens zurück. Mit dem Ausdruck `return` wird der Rückgabewert der Funktion definiert. Der Typ des Rückgabewertes muss mit dem Rückgabetyp der Deklaration der Funktion übereinstimmen. In diesem Fall dient die Funktion *getName* dazu, eine Abfrage von Daten durchzuführen.

```
01 renameNode(n as Node, newName as String) =
02   n.name := newName
```

Die Funktion *renameNode* bekommt als Parameter einen Knoten und einen `String` Wert. Als *newName* wird der neue Name des Knotens *n* bezeichnet. Die Funktion *renameNode* benennt den Knoten um. Dabei wird der Wert des Attributes *name* des Knotens *n* aktualisiert. Aus diesem Grund ist *renameNode* eine Regel.

```
01 skip
```

`Skip` ist ein Ausdruck für die Beschreibung der leeren Aktion. Es ist wie eine Funktion ohne Parameter. Ein Beispiel ist im folgenden Paragraphen zu finden.

3.2.5 Bedingungsausdrücke

Um eine Bedingung auszudrücken wird folgende Form benutzt:

```
01 if a then w1 else w2
```

Der Ausdruck *a* muss vom Typ `Boolean` sein und wird Wächter genannt. Der Wert des Bedingungsausdrucks ist w_1, falls *a* `true` ist, sonst ist der Wert w_2. Falls keine `else` Klausel deklariert wird und *a* gleich `false` ist, wird der Wert des Bedingungsausdrucks als Vorgabe `skip` sein. Falls mehrere Fälle existieren, wird folgende Form benutzt:

```
01 if a1 then w1
02 ifelse a2 then w2
03 …
04 ifelse an-1 then wn-1
05 else an
```

Dieser Ausdruck ist äquivalent mit

```
01 if a1 then w1 else
02 (if a2 then w2 else
03 (…
04 (if an-1 then wn-1
05 else wn)…))
```

Im folgenden Beispiel wird überprüft, ob das Element *einKnoten4*, das eine Station ist, in der Menge der Knoten existiert.

```
01 if einknoten4 in setOfNodes.elements then …
```

Eine andere Art Bedingungen zu überprüfen, ist der `match`-Ausdruck.

```
01 match w with
02   pattern1: w1
03   …
04   patternn-1: wn-1
05   _: wn
```

Falls der Wert *w* passend zu $pattern_1$ ist, wird der Ausdruck den Wert w_1 bekommen, andernfalls werden die weiteren Patterns nach einer Übereinstimmung untersucht. Falls keines der Patterns mit dem Wert *w* übereinstimmt, ist es möglich, mit der Notation '_' einen Wert w_n als Standardwert zu definieren.

Diese Form ist äquivalent mit folgender Form:

```
01 if pattern1 matches v then
02   let pattern1 = v
03   w1
04 else (if …
05       else (if patternn matches v then
07               let patter n-1 = v
```

```
08                $w_{n-1}$
09            else $w_n$
```

3.2.6 ASM Maschinen

AsmL bietet einen Ausdruck an, der das Deklarieren einer neuen ASM Maschine als Unterprogramm definiert. Eine ASM Maschine ist genau ein ASM Programm, dessen Berechnung als eine Folge von Zuständen zu verstehen ist. Wenn ein Unterprogramm mit seiner Berechnung fertig ist, wird die Menge seiner Aktualisierungen zu der Menge der Aktualisierungen des Hauptprogramms hinzuaddiert.

Sei also S_0 der Anfangszustand des Programms (Abbildung 3.2). Im Zustand S_0 werden Mengen deklariert und initialisiert. Die Mengen, die als Variablen deklariert werden, sind Kandidaten für Aktualisierungen. Durch das Ausführen von Regeln auf diese Mengen ist es möglich die Daten zu verändern. Nach dem ersten Transitionsschritt werden die Mengen aktualisiert, und das Programm befindet sich in einem neuen Zustand S_1, in dem erneut Regeln aufgerufen werden können, um die Daten zu verändern, bis ein Zustand S_n erreicht wird, der der Endzustand des Programms ist.

Abbildung 3.2: Eine Berechnung

Um den Zustand S_0 zu beschreiben, wird eine ASM Maschine definiert. Um die Transitionen zu beschreiben, wird der Term `step` benutzt.

```
01 machine
02   // Zustand S0                    }
03   // Deklarationen                 }  S0
04   // Aktualisierungen              }
05 step
06   // Zustand S1                    }
07   // Deklarationen                 }  S1
08   // Aktualisierungen              }
09 step
10 …
11 step
12   // Zustand Sn                    }
13   // Aktualisierungen              }  Sn
```

In den Zustanden S_0 bis S_{n-1} ist es möglich neue Mengen zu deklarieren, die aktualisiert werden. Im Zustand S_n hat so etwas keinen Sinn, da keine weiteren Zustände existieren, die eine Aktualisierung beschreiben würden.

Im nächsten Fall wird im Zustand S_2 ein Unterprogramm in Form einer Regel aufgerufen (Abbildung 3.3). Diese Regel besteht aus drei Zuständen. Die Menge der Aktualisierungen, die im Unterprogramm definiert werden, werden im Zustand S_2 übernommen.

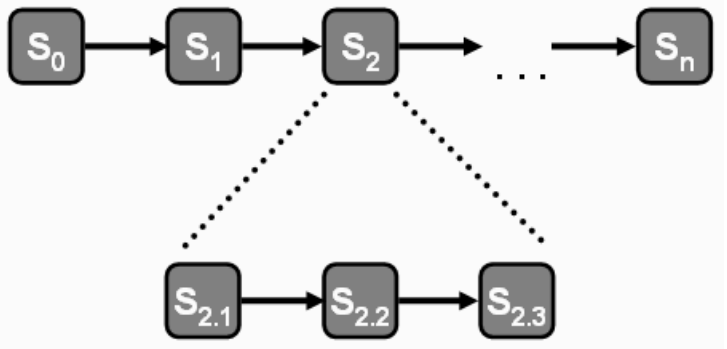

Abbildung 3.3: Die Berechnung einer Algorithmus mit einem Unterprogramm

Sei also im obigen Beispiel der Zustand S_2 wie folgt definiert:

```
01 …
02 step
03   // Zustand S2
04   // neue Mengen deklarieren
05   eineRegel(number) // Regel aufrufen
06 step
07 …
```

In Zeile 5 wird eine Regel aufgerufen, die auf alle global deklarierten Mengen Änderungen vornehmen darf. Ist es erwünscht, Änderungen der lokal deklarierten Mengen vom Zustand S_2 vorzunehmen, so müssen sie als Parameter zu der Regel gegeben werden. In diesem Beispiel bekommt die Regel *eineRegel* ein `Integer` als Parameter.

```
01 eineRegel(p as Integer) as String =
02   machine
03     // Zustand S2.1
04     einString as String // neue Variable deklarieren
05     p := … // Aktualisierung von p
06   step
07     // Zustand S2.2
08     einString := IntToStr(p) // Aktualisierung von einString
10   step
11     // Zustand S2.3
12     return einString
```

Im Zustand $S_{2.1}$ dieser Regel wird eine Aktualisierung in *p* vorgenommen. Um den neuen Wert von *p* in *einString* mittels einer Konvertierung zuzuweisen, ist es notwendig, in einen neuen Zustand $S_{2.2}$ zu gelangen, damit die Aktualisierung von *p* stattfindet. Das gleiche gilt auch für *einString*. Nach der Zuweisung in Zeile 8 ist im Zustand $S_{2.2}$ der neue Wert von *einString* nicht bekannt. Dieser wird erst im Zustand $S_{2.3}$ bekannt und durch den `return` Term im Zustand S_2 zurückgegeben. Die Aktualisierung von *p* in Zeile 5 wird auch für die Variable *number* gelten, die im Zustand S_2 als Parameter zu der Regel gegeben worden ist. Die Variable *p* ist ein lokales Binden des Namens *p* mit dem Wert der Variable *number*. Ein Beispiel einer ASM Maschine wird im Paragraph Iterationen (3.2.7) gegeben.

Falls eine ASM Maschine in einer Regel definiert wird, die rekursiv aufgerufen wird, dann wird für jeden Rekursionsschritt eine neue ASM Maschine erzeugt. Alle Aktualisierungen einer Transitionsregel müssen konsistent sein. Ist das nicht der Fall, so wird von AsmL eine `CollisionException` ausgelöst.

3.2.7 Iterationen

AsmL bietet verschiedene Arten von Iterationen. In dieser Arbeit werden die `foreach`- und die `while`-Schleife benutzt. Die `foreach`-Schleife wird eingesetzt, um sequenziell alle Elemente einer Menge oder alle Elemente einer Menge, die eine bestimmte Bedingung erfüllen, zu durchlaufen. Folgende Form beschreibt die Syntax der `foreach`-Schleife.

```
01 foreach binder [where B] do
02   block-list
```

In Zeile 1 werden nach dem `foreach`-Ausdruck ein oder mehrere *binder* deklariert. Ein *binder* ist ein Ausdruck, der in jeder Iteration der Schleife Elemente einer Menge (oder allgemein Werte) an eine Variable bindet. Nachdem der *binder* deklariert ist, besteht die Möglichkeit, dieses Binden unter einer bestimmten Bedingung B gelten zu lassen. Eine Bedingung ist ein boolescher Ausdruck. Mit dem `do`-Ausdruck wird das Ende des `foreach`-Ausdrucks definiert. Ab Zeile 2 folgt ein Block von Befehlen, die in jeder Iteration ausgeführt werden.

Im folgenden Beispiel wird eine `foreach`-Schleife benutzt, um einen Zähler zu inkrementieren. Das Ergebnis wird in jeder Iteration angezeigt.

```
01 foreach n in setOfNodes.elements where n.name = "Station" do
02   writeln("Node Station found.")
```

In Zeile 1 wird eine `foreach`-Schleife deklariert, die über alle Elemente der Menge *setOfNodes.elements* iteriert, die den Namen *Station* haben. In Zeile 2 wird das Ergebnis jeder Iteration angezeigt. Die Variable *n* wird in jeder Iteration mit einem neuen Element der Menge *setOfNodes.elements* gebunden. Im gesamten Block der Schleife sowie auch in der Bedingung der Schleife ist *n* lokal definiert. Jedes Element wird nur einmal betrachtet.

Eine `while`-Schleife wird so lange iteriert, bis eine bestimmte Bedingung nicht mehr erfüllt wird.

```
01 while B do
02   block-list
```

In Zeile 1 wird nach dem `while`-Ausdruck eine Bedingung B deklariert, die ein boolescher Ausdruck ist. Solange diese Bedingung wahr ist, findet eine weitere Iteration der Schleife statt. Sollte die Bedingung nicht mehr gelten, so wird die Schleife verlassen. Eine Iteration der Schleife bedeutet, dass die Befehle ausgeführt werden, die sich ab Zeile 2 im Block befinden.

Im folgenden Beispiel wird eine `while`-Schleife benutzt, um einen Zähler bis auf einen bestimmten Wert zu inkrementieren. In jeder Iteration der Schleife wird der Wert des Zählers angezeigt.

```
01 var i=1
02 while i<11 do
03   i:=i+1
04   writeln(i)
```

In Zeile 1 wird ein Zähler *i* deklariert und mit dem Wert 1 initialisiert. In Zeile 2 wird eine `while`-Schleife deklariert, die solange iteriert, bis *i* einen Wert hat, der kleiner als 11 ist. In

den Zeilen 3 und 4 erscheint der Block, der in jeder Iteration der Schleife aufgerufen wird. Als erstes wird der Wert von *i* inkrementiert und in der nächsten Zeile ausgegeben. Es ist zu beachten, dass in Zeile 3 der Wert von *i* aktualisiert wird, die Aktualisierung von *i* jedoch erst in der nächsten Iteration stattfindet. So wird in der ersten Iteration der Wert 1 und nicht der Wert 2 ausgegeben.

3.2.8 Aufzählungen

Eine weitere Struktur in AsmL ist die Aufzählung (enumeration). Eine Aufzählung besteht aus einem Namen und Elementen und hat folgende Form:

```
01 enum enumName
02   element1
03   …
04   elementn
```

In Zeile 1 definiert der Term `enum` eine Aufzählung mit dem Namen *enumName.* Im folgenden Block werden dann die Elemente der Aufzählung deklariert.

```
01 enum Color
02   white
03   black
04
05 run()
06   var n=black
07   match n with
08     black : writeln(“node is black”)
09     white : writeln(“node is white”)
```

In den Zeilen 1 bis 3 wird eine Aufzählung mit dem Namen *Color* und den Elementen *white* und *black* definiert. Ab Zeile 5 wird die Funktion *run* deklariert und implementiert. In Zeile 6 wird *n* (repräsentiert einen Knoten in einem Graphen) mit dem Wert *black* gebunden; daher bekommt *n* den Typ *Color*. In den Zeilen 7 bis 9 steht ein `match`-Ausdruck. In diesem wird der Wert von *n* überprüft. Ist dieser *black*, so wird ausgegeben, dass der Knoten schwarz ist, wenn nicht, dass er weiß ist.

3.2.9 Patterns

Mit Patterns in AsmL ist es möglich, einen Wert anhand seiner lexikalischen Form, die durch seinen Konstruktor definiert wird, in seine Komponenten zu zerlegen. Wie ein Pattern benutzt wird, wird im folgenden Beispiel erläutert.

```
01 let EdgesSet = setOfEdges.elements
02 foreach e in dom(EdgesSet) do
03   let (s, t) = EdgesSet(e)
04   writeln(“Edge ” + e.name + “ connects “ + s.name + “ with “ t.name)
```

Obiges Beispiel gibt für jede Kante ihren Anfangs- und Endknoten aus. In Zeile 2 wird eine `foreach`-Schleife deklariert, die über alle Elemente des Definitionsbereichs der Abbildung *EdgeSet* iteriert, also über alle Kantennamen. In Zeile 3 wird das Pattern `(s, t)` benutzt, um das Bild des Wertes *e* durch die Abbildung zu bekommen. Ein Bild der Abbildung *EdgesSet*

hat den Typ *Node * Node*. Die lexikalische Form dieses Typs wird zu dem Tupel `(s, t)` übertragen. So bekommt *s* den Wert des ersten Elements und *t* den Wert des zweiten.

3.2.10 Ein Beispiel in AsmL

Nachdem die wichtigsten Elemente der Sprache AsmL erklärt worden sind, folgt ein leicht modifiziertes Beispiel aus [7], das einen Sortieralgorithmus spezifiziert. Ziel des Beispiels ist es, eine globale Sicht und eine bessere Einordnung der Elemente, die in den letzten Paragraphen erklärt worden sind, zu erreichen. Das folgende Beispiel besteht aus der Funktion *run*, die die ausführbare Spezifikation startet, ähnlich wie die *main* Methode in den Programmiersprachen Java oder C. Zusätzlich wird die Funktion *indices* benutzt, die als Rückgabe die Menge aller Indizes einer Folge berechnet.

```
01 run() =
02   machine                                        // ASM Maschine
03     var A = [3, 10, 5, 7, 1]
04   step
05     until fixpoint do                            // Schleife
06       choose i in indices(A), j in indices(A) // nichtdeterministische
                                                    // Wahl
07         where i < j and A(i) > A(j)              // mit Bedingung
08         do A(i) := A(j)                          // Aktualisierungen
09            A(j) := A(i)
10   step
11   writeln(A)                                      // Ausgabe
```

In Zeile 2 wird eine ASM Maschine deklariert. In Zeile 3 wird eine Folge von Elementen vom Typ `Integer` initialisiert. Diese Folge bildet die Daten eines Zustandes des Algorithmus. Mit dem Term `step` in den Zeilen 4 und 10 werden die Transitionsregeln definiert. In Zeile 5 wird eine neue Art von Schleife benutzt. Die Schleife `until-fixpoint` iteriert solange, bis die Daten eines Zustandes auch im nächsten Zustand dieselben bleiben. Ist das der Fall, so wird die Schleife abgebrochen. In Zeile 6 werden zwei Indizes, *i* und *j*, aus der Menge der Indizes von *A* nichtdeterministisch ausgewählt, unter der Bedingung, dass falls *i* kleiner als *j* ist, der Wert der Folge *A* mit dem Index *i* größer als der Wert der Folge *A* mit dem Index *j* ist. Existieren *i* und *j*, so werden ihre Werte getauscht. Das Tauschen der Werte zweier Elemente der Folge *A*, die unsortiert sind, findet solange statt, bis die Folge sortiert ist.

Eine mögliche Berechnung des obigen Algorithmus wird in Abbildung 3.4 dargestellt. Durch die nichtdeterministische Auswahl in Zeile 6 ist es möglich, bei jeder Ausführung des Algorithmus verschiedene Paare von unsortierten Elementen zu bekommen. Der Zustand S_0 beschreibt den Anfangszustand vom `step` der Zeile 4. Der Zustand S_1 ist der erste Zustand, in dem Aktualisierungen stattfinden. In diesem Fall sind i=1 und j=4. Nach der ersten Iteration der Schleife finden die Aktualisierungen durch die Transitionsregel t_1 statt. So gelangt der Algorithmus in den Zustand S_2, dessen Folge geänderte Werte enthält. In der gleichen Art und Weise finden auch durch die Transitionsregel t_2 weitere Aktualisierungen der Folge *A* statt. Der Algorithmus befindet sich nun im Zustand S_3. Jetzt ist die Folge sortiert, es gibt also keine weiteren Iterationen mehr. Durch den `step` der Zeile 10 gelangt der Algorithmus in einen neuen Zustand S_4, der auch der Endzustand der Berechnung des Algorithmus ist. Nun wird die Folge A ausgegeben.

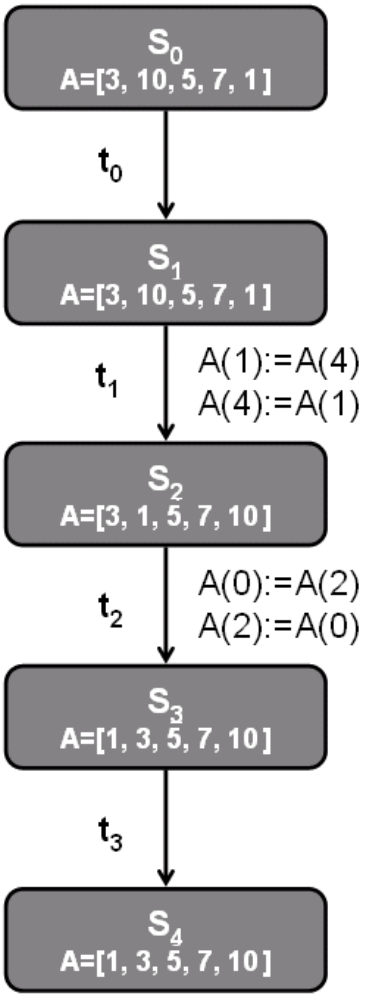

Abbildung 3.4: Berechnung des Sortieralgorithmus

Die Spezifikation des Algorithmus ist minimal. Das bedeutet, dass die höchste Abstraktionsebene benutzt worden ist. Durch die nichtdeterministische Auswahl in Zeile 6 wird die Auswahl der Elemente transparent gehalten. Auch in den Zeilen 8 und 9 werden die Werte zweier Indizes vertauscht, ohne dass beschrieben wird, wie dies geschehen soll. In einer tieferen Abstraktionsebene wird ein temporäres Element benutzt, das den Wert der ersten Indizes zwischenspeichert, damit er nicht überschrieben wird und verloren geht.

3.3 Zusammenfassung

Dieses Kapitel war eine Einführung in die Sprache ASM. In 3.1 wurden die wichtigsten Begriffe und Eigenschaften von ASM sowie auch zwei Werkzeuge für die Ausführung von ASM erläutert. In 3.2 wurde die Sprache des Werkzeugs AsmL näher betrachtet, da diese in dieser Arbeit verwendet wird. Es wurden allerdings nur die Teilbereiche der Sprache AsmL erklärt, die in den folgenden Kapiteln auch benutzt werden.

4. Abbildung von UML Spezifikationen auf ASM

In diesem Kapitel wird beschrieben, wie die Abbildung von Klassendiagrammen und Aktivitätsdiagrammen definiert ist. Um eine bessere Verständigung zu erreichen, werden die Konzepte anhand eines Beispiels erläutert, das in 4.1 eingeführt wird. Um möglichst alle Fälle in Betracht zu ziehen, wird das vorgestellte Beispiel erweitert oder leicht verändert. In 4.2 wird die Abbildung des Klassendiagramms auf ASM beschrieben, indem ASM Datenstrukturen definiert werden, die die Semantik des Klassendiagramms beschreiben. Des Weiteren werden in 4.3 die Methoden der Klassen betrachtet. Eine Methode wird durch die Kombination von Story Patterns und einem Aktivitätsdiagramm modelliert. Die Story Patterns beschreiben das Verhalten der Methode, das Aktivitätsdiagramm den Kontrollfluss der Methode, also die Reihenfolge der Ausführung der Story Patterns. Daher werden zunächst die Story Patterns näher betrachtet. Ein Story Pattern ist eine Graphersetzungsregel, die aus zwei Teilgraphen besteht. Der linke Teilgraph beschreibt die Situation vor der Ausführung des Story Patterns, der rechte Graph die Situation danach. Diese Teilgraphen sind, wie auch die Laufzeitobjektstruktur, Objektdiagramme. Die Semantik eines Objektdiagramms und auch der Story Patterns wird durch ASM Datenstrukturen definiert. Das Ausführen der Story Patterns wird durch eine Menge von ASM Regeln definiert. Bei jeder Ausführung eines Story Patterns muss zuerst eine Anwendungsstelle im Wirtsgraphen gefunden werden. Dies erfordert eine Teilgraphensuche, was ein NP vollständiges Problem ist. Daher deswegen wird diese Regel in einem eigenen Kapitel betrachtet.

Am Ende dieses Kapitels wird beschrieben, wie für jede Methode des Klassendiagramms eine ASM Funktion generiert wird und wie diese implementiert wird. Der Rumpf der ASM Funktion wird durch das Aktivitätsdiagramm der Methode bestimmt. Das Aktivitätsdiagramm beschreibt, wie bereits erwähnt, die Reihenfolge des Ausführung der Story Patterns dieser Methode.

4.1 Beispiel

In den folgenden Paragraphen werden zur besseren Veranschaulichung Beispiele gegeben, die auf dem Klassendiagramm der Abbildung 4.1 aufbauen. Da diese Arbeit im Rahmen des Projekts ISILEIT geschrieben worden ist, sind die wichtigsten Komponenten eine *Station* und ein *Shuttle*, die untereinander interagieren. Um ein reaktives System zu modellieren, werden im Normalfall ein oder mehrere Statecharts benutzt. Da aber im Rahmen dieser Arbeit die Klassendiagramme, Storydiagramme und die Semantik der Story Patterns betrachtet wird, wurden keine Statecharts benutzt. Um das reaktives Verhalten von Klassen zu modellieren, wird ein stark vereinfachter Scheduler benutzt, der Ereignisse verwaltet und behandelt.

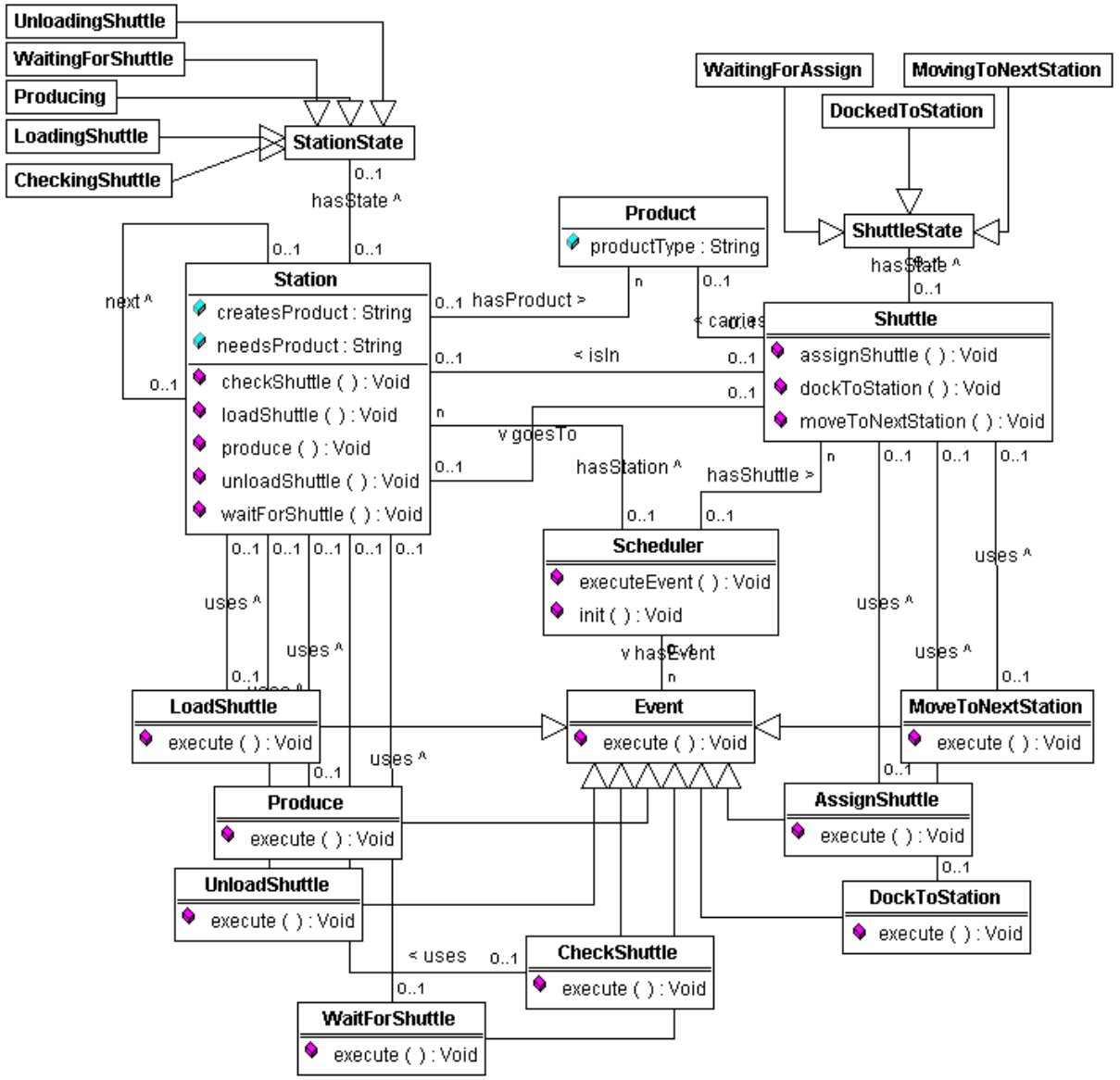

Abbildung 4.1: Klassendiagramm des Beispiels

Eine *Station* kann von einem *Shuttle* besucht werden. Trägt das angekommene *Shuttle* ein Produkt (Klasse *Product*) und ist der Typ des Produktes nutzbar für die *Station*, so wird das *Shuttle* entladen und das Produkt bearbeitet. Dann wird das neue Produkt auf das *Shuttle* geladen und das *Shuttle* kann weiter fahren. Alle Aktionen, die in diesem kleinen Szenario vorkommen, werden durch Ereignisse (Klasse *Event*) realisiert. Es gibt Ereignisse, die von einer *Station* durchgeführt werden, wie z.B. *LoadShuttle*, *CheckShuttle*, und Ereignisse, die von einem *Shuttle* durchgeführt werden, wie z.B. *AssignShuttle*, *MoveToNextStation*. Welches Ereignis mit welcher Komponente in Beziehung steht, wird durch die Assoziation *uses* festgelegt. Ein Objekt der Klasse *Scheduler* hat eine Beziehung zu allen Objekten der Klassen *Station*, *Shuttle* und *Event*. Ist ein *Event* mit dem *Scheduler* verbunden, so kann dieses mittels der Methode *executeEvent* ausgeführt werden. Durch Delegation wird die Methode *execute* der entsprechenden Unterklasse der Klasse *Event* aufgerufen[3]. Diese führt mittels neuer Delegation die entsprechende Methode der Klasse *Station* oder *Shuttle* aus. Die Methoden der Klassen *Station* und *Shuttle* sind also die Implementierungen der Ereignisse. Nachdem ein Ereignis stattgefunden hat, wird es gelöscht.

[3] Es handelt sich hier um das Stategy-Pattern [25].

Jede der Klassen *Station* und *Shuttle* hat eine Menge von Zuständen, die von den Oberklassen *StationState* und *ShuttleState* beschrieben werden. In den Abbildungen 4.2 und 4.3 wird das Statechart der Klassen *Shuttle* und *Station* dargestellt, welches eigentlich im Modell nicht existiert, dessen Verhalten aber mittels der Klasse *Scheduler* stark vereinfacht nachmodelliert wurde.

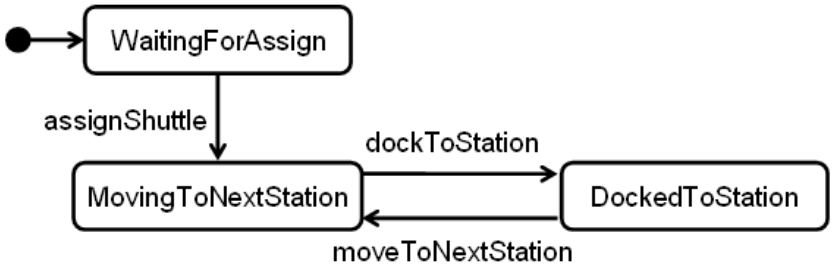

Abbildung 4.2: Statechart der Klasse *Shuttle*

Ein *Shuttle* befindet sich am Anfang im Zustand *WaitingForAssign*. Sobald das *Shuttle* das Signal *AssignShuttle* bekommt, wird sein Zustand geändert. Der neue Zustand *MovingToNextStation* beschreibt, dass das *Shuttle* auf dem Weg zur nächsten *Station* ist, die durch die Assoziation *goesTo* definiert ist. Wenn das *Shuttle* die nächste *Station* erreicht hat, erhält es das Signal *dockToStation* und ändert seinen Zustand in *DockedToStation*. Während das *Shuttle* an der *Station* angedockt bleibt, hat es mit der Station eine *isIn* Beziehung. Um sich wieder auf den Weg zur nächsten *Station* zu machen, muss das *Shuttle* das Signal *moveToNextStation* bekommen.

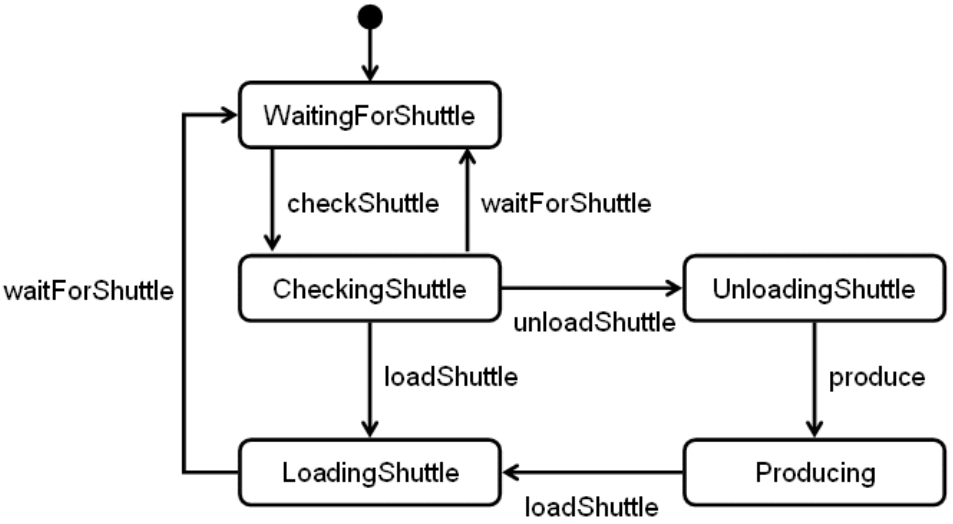

Abbildung 4.3: Statechart der Klasse *Station*

Für die Klasse *Station* gibt es auch ein Statechart, welches im Modell des Beispiels nicht vorkommt, sondern nachmodelliert worden ist. Eine Station befindet sich am Anfang im Zustand *WaitingForShuttle*. Kommt ein *Shuttle* zu dieser *Station*, so wird der Zustand der *Station* mittels des Signals *checkShuttle* in *CheckingShuttle* geändert. In diesem Zustand wird überprüft, ob das angedockte *Shuttle* ein Produkt trägt, das die *Station* gebrauchen könnte. Ist das nicht der Fall und ist auch kein Produkt in der *Station* vorhanden, um das *Shuttle* zu beladen, so fährt das *Shuttle* weiter und die *Station* kehrt zu ihrem alten Zustand zurück. Kann sie es gebrauchen, so wird der Zustand der *Station* in *UnloadingShuttle* geändert. Das Produkt, das sich jetzt in der *Station* befindet, wird verändert und wieder auf das *Shuttle*

geladen, nachdem der Zustand der *Station* sich in *LoadingShuttle* geändert hat. In diesem Zustand befindet sich die *Station* auch, wenn das *Shuttle* kein Produkt trägt, welches der *Station* nützlich sein könnte. Nachdem das *Shuttle* beladen worden ist, darf es weiter zur nächsten *Station* fahren, und die alte *Station* ändert ihren Zustand in ihren initialen Zustand *WaitingForShuttle*.

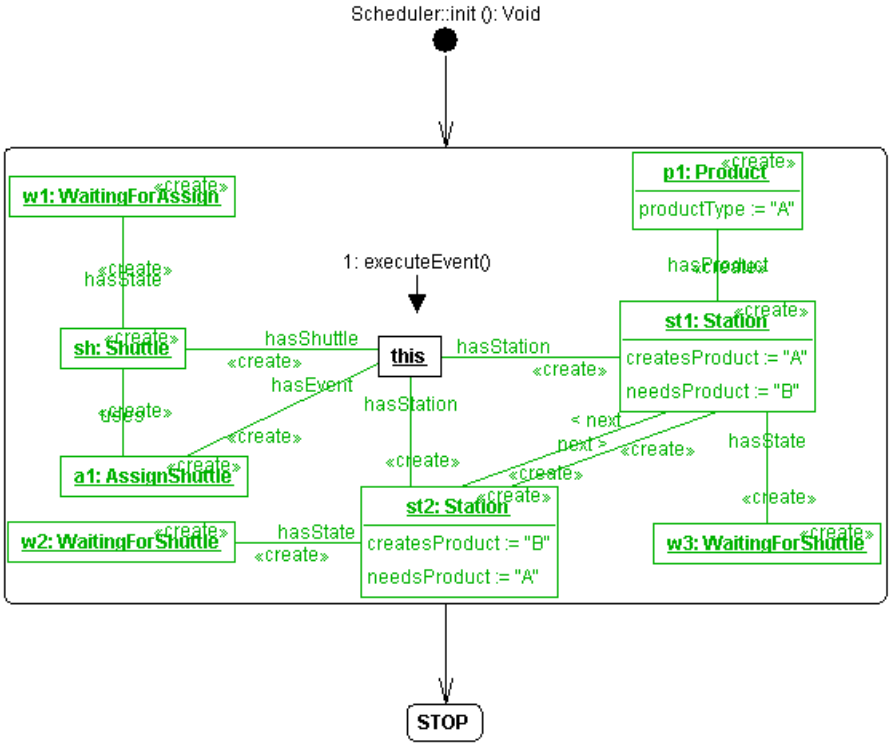

Abbildung 4.4: Die Methode *init*

Der initiale Zustand der Laufzeitobjektstruktur wird durch den Aufruf der Methode *init* der Klasse *Scheduler* erzeugt (Abbildung 4.4). Es existieren ein Shuttle und zwei Stationen, die sich durch die *next* Links ordnen. Die Station *st1* hat ein Produkt *p1*, dessen Typ der Produktionstyp der Station *st1* ist. Das *Shuttle* und die Stationen befinden sich in den Anfangszuständen. Schließlich existiert das Event *AssignShuttle*, um das *Shuttle* in Bewegung zu setzen.

4.2 Abbildung des Klassendiagramms auf ASM

Das Modell eines Systems wird, wie schon erwähnt, durch verschiedene Diagrammarten spezifiziert. Jedes Diagramm beschreibt verschiedene Eigenschaften des Systems. Das Klassendiagramm beschreibt die statischen Eigenschaften eines Systems, d.h. die Struktur der Klassen oder Datentypen und die Beziehungen zueinander. Eine Klasse beschreibt eine Menge von Objekten mit ähnlichen Eigenschaften und besteht aus einem eindeutigen Namen, aus Attributen und Methoden. Die Attribute beschreiben die Eigenschaften einer Klasse und haben einen Namen und einen Typ. Methoden beschreiben das Verhalten einer Klasse und haben einen Namen, eine Liste mit Parametern und ihren Typen und einen Rückgabetyp. Diese Daten bilden die Signatur der Methode. Die Typen der Parameter und der Rückgabe sind die erlaubten Typen der Attribute oder Typen, die durch eine Klasse definiert sind. In der

Abbildung 4.5 wird die Klasse *Station* dargestellt. Im oberen Block wird der Name der Klasse angegeben, dann folgt die Liste der Attributnamen und deren Typen und schließlich die Liste der Methodensignaturen.

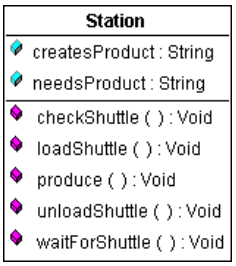

Abbildung 4.5: Die Klasse Station

Klassen können Beziehungen miteinander haben. Eine Art von Beziehung zwischen zwei oder mehreren Klassen ist die Generalisierung oder Vererbung. Generalisierung ist ein Mechanismus, um neue Klassen mit Hilfe bereits bestehender Klassen zu definieren. Die Unterklasse erbt von der Oberklasse alle Attribute und Methoden und kann neue eigene Attribute und Methoden definieren. Dabei können auch geerbte Methoden überschrieben werden(overloading). In Abbildung 4.6 werden die Klassen *Event* und *CheckShuttle* dargestellt. Der Pfeil definiert eine Vererbungsbeziehung zwischen den zwei Klassen. Die Richtung des Pfeils zeigt an, welche Klasse von welcher erbt. Bei der Vererbung überschreibt die Methode *execute* der Klasse *CheckShuttle* die geerbte Methode *execute* der Klasse *Event*.

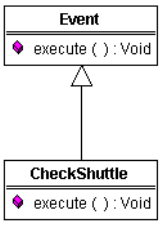

Abbildung 4.6: Die Klasse *CheckShuttle* erbt von der Klasse *Event*. Dabei wird die Methode *execute* überschrieben.

Außer der Generalisierung zwischen zwei Klassen können auch Assoziationen existieren. Assoziationen besitzen einen Namen, eine Leserichtung und für jede Klasse eine Kardinalität und einen Rollennamen. Kardinalitäten an Assoziationen definieren Einschränkungen auf der Instanzebene, Rollennamen beschreiben die Rolle, die ein Objekt in der Beziehung hat. In Abbildung 4.7 werden die Klassen *Station*, *Product* und *Shuttle* dargestellt. Zwischen den drei Klassen gibt es zwei Assoziationen. Die Assoziation *hasProduct* zwischen den Klassen *Station* und *Product* beschreibt, dass eine Instanz der Klasse *Station* mit beliebig vielen Instanzen der Klasse *Product* stehen darf. Analog beschreibt die Assoziation *carries*, dass eine Instanz der Klasse *Shuttle* höchstens mit einer Instanz der Klasse *Product* in Verbindung stehen darf.

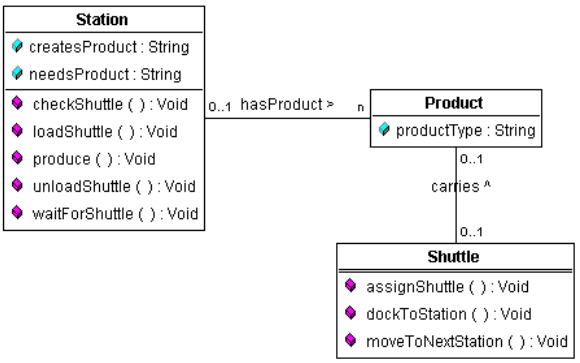

Abbildung 4.7: Ein Teil des Klassendiagramms mit drei Klassen und zwei Assoziationen

In den folgenden Paragraphen werden Strukturen für die Spezifikation des Klassendiagramms in AsmL beschrieben. Anschließend wird für das Beispiel aus Kapitel 4.1 der generierte AsmL Code gezeigt, der aus Initialisierungen der ASM Datenstrukturen des Klassendiagramms besteht.

Struktur des Klassendiagramms

Das Klassendiagramm wird durch Datenstrukturen auf ASM abgebildet. Um eine bessere Sicht der Abbildung der verschiedenen Diagramme auf ASM zu bekommen, wird die Abbildung 2.2 kapitelweise aufgebaut. Die Datenstrukturen, die das Klassendiagramm beschreiben, werden nicht generiert sondern befinden sich statisch im ASM Modell. In Abbildung 4.8 wird dargestellt, dass die ASM Datenstrukturen des ASM Modells (include) die Datenstrukturen enthalten, die dem Klassendiagramm entsprechen.

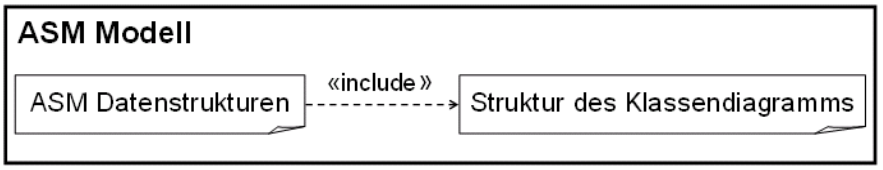

Abbildung 4.8: Struktur des Klassendiagramms im ASM Modell

Die Abbildung des Klassendiagramms in ASM erfolgt durch das Beschreiben des Klassendiagramms aus der Sicht der Graphgrammatiktheorie. Ein Klassendiagramm beschreibt ein Graphschema, das aus einer Menge von Knotenbezeichnungen, einer Menge von Kantenbezeichnungen und einer Menge von Attributnamen besteht. Da das Klassendiagramm eine objektorientierte Spezifikation beschreiben soll, wird noch eine Menge für die Generalisierung und eine Menge für die Struktur der Assoziationen hinzugefügt. Zusätzlich werden in einer Menge die Typen der Attributnamen beschrieben. Tabelle 4.1 stellt die ASM Strukturen dar, die für das Klassendiagramm benötigt werden,.

Strukturen des Klassendiagramms	
MultiInfo	Mögliche Kardinalitäten der Assoziationen
AssocType	Mögliche Assoziationstypen
AttributeType	Mögliche Attributtypen
Node	Struktur der Knoten (Klassenbezeichnungen)
Edge	Struktur der Kanten (Assoziationsbezeichnungen)
NL	Menge aller Knoten

EL	Menge aller Kanten
AttributeLabel	Struktur des Attributs
A	Menge aller Attribute
IsAs	Menge der Vererbungskanten
Assocs	Abbildung der Kantennamen auf ihren Eigenschaften
Attrs	Abbildung der Attribute auf ihren Typen
SIGraph	Struktur des Klassendiagramms

Tabelle 4.1: ASM Strukturen für das Klassendiagramm

Das Klassendiagramm enthält statische Systeminformationen. Aus diesem Grund wird die Struktur, die die Informationen des Klassendiagramms enthält, *SIGraph* (System Information Graph) genannt.

```
01 class SIGraph
02   param as NL * EL * A * IsAs * Assocs * Attrs
```

Die Struktur *SIGraph* besteht aus einem Tupel, das wiederum aus der Menge der Knotenbezeichnungen, Kantenbezeichnungen, Attributnamen, Generalisierungen, Assoziationen und Attributtypen besteht.

Knotenbezeichnungen

Die Struktur *Node*, die aus einem `String` *name* besteht, entspricht dem Namen einer Klasse im Klassendiagramm. Der Name einer Klasse ist eindeutig. Alle Klassennamen werden in eine Struktur *NL* (node labels) eingefügt, die aus einer Menge von Elementen des Typs *Node* besteht, die *elements* heißt.

```
01 class Node
02   name as String
03
04 class NL
05   elements as Set[Node]
```

Kantenbezeichnungen

Die Struktur *Edge*, die aus einem `String` *name* besteht, entspricht dem Namen einer Assoziation im Klassendiagramm. Der Name einer Assoziation ist nicht eindeutig. Die Assoziationen werden durch ihre Eigenschaften unterschieden. Die Eigenschaften einer Assoziation werden in der Struktur *Assoc* beschrieben. Alle Assoziationsnamen werden in eine Struktur *EL* (edge labels) eingefügt, die aus einer Menge von Elementen des Typs *Edge* besteht, die *elements* heißt.

```
01 class Edge
02   name as String
03
04 class EL
05   elements as Set[Edge]
```

Attributsnamen

Die Struktur *AttributeLabel* entspricht dem Namen eines Attributes. Klassen dürfen Attribute mit dem gleichen Namen besitzen. Auch bei einer Vererbung erbt die Unterklasse alle Attribute der Oberklasse. Um diese Eigenschaften der Klassen zu unterstützen, wird

zusammen mit dem Attributnamen auch der Klassenname gesetzt. In der Struktur *A* werden schließlich alle Elemente vom Typ *AttributeLabel* in einer Menge mit dem Namen *elements* gespeichert.

```
01 class AttributeLabel
02   node as Node
03   value as String
04
05 class A
06   elements as Set[AttributeLabel]
```

Generalisierungen

Eine Generalisierung oder Vererbung wird durch ein Tupel von zwei Klassennamen definiert. Die Struktur *IsAs* besteht aus einer Menge von solchen Tupeln. Der erste Klassenname beschreibt die Unterklasse, der zweite die Oberklasse. Die Menge der Tupel trägt den Namen *elements*.

```
01 class IsAs
02   elements as Set[Node * Node]
```

Assoziationen

In der objektorientierten Modellierung sind Assoziationen komplexer als die Kanten eines einfachen Graphen. Aus diesem Grund wird die Struktur *Assocs* definiert, die zu jedem Assoziationsnamen weitere Eigenschaften abbildet. Die Eigenschaften einer Assoziation werden durch zwei Knotenbezeichnungen, zwei Kardinalitäten und eine Menge, die die Typen der Assoziation darstellt. Die zwei Knotenbezeichnungen sind die linken und rechten Klassen (nach der Leserichtung der Assoziation), die als Beziehung diese Assoziation haben. Jede Klasse enthält eine Kardinalität. Die Kardinalitätstypen beschränken sich in *one* und *many*. Schließlich kann die Menge der verschiedenen Typen der Assoziation die Elemente *aggregation* für Aggregation, *qualified* für qualifiziert, *ordered* für geordnete und *sorted* für sortierte Assoziation enthalten. Kompositionen werden nicht direkt unterstützt, da sie sich durch eine Aggregation mit zusätzlichen Bedingungen simulieren lassen. Nur der linke Teil einer Assoziation darf ein Aggregationssymbol haben oder qualifiziert sein. Die Struktur *Assocs* ist eine Abbildung von Kantenbezeichnungen auf das oben beschriebene Tupel.

```
01 class Assocs
02   elements as Edge ->
03   Node * MultiInfo * Set[AssocType] * Node * MultiInfo
```

Die Struktur *MultiInfo* ist eine Menge, die als einzige Elemente *one* und *many* enthält. In der gleichen Art wird auch die Struktur *AssocType* definiert. In dem Tupel wird der Assoziationstyp als eine Menge von *AssocType* Elementen definiert, da beliebige Permutation davon erlaubt ist.

```
01 enum MultiInfo
02   one
03   many
04
05 enum AssocType
06   qualified
07   aggregation
08   ordered
```

Attributtypen

Attributtypen sind beschränkt auf `Boolean`, `Integer`, `String` und `Float`. Es ist nicht erlaubt, Objektreferenzen als Attributtypen zu definieren, da diesen Zweck eine Assoziation erfüllt. Auch komplexe Typen von Attributen werden nicht unterstützt, da diese mit Hilfe von neuen Klassen und Assoziationen realisiert werden können. Die Struktur *Attrs* ist eine Abbildung von Attributnamen auf Attributtypen. Die Attributtypen werden durch die Struktur *AttributeType* definiert, die eine Menge von *INT*, *STRING*, *BOOL* und *FLOAT* ist.

```
01 class Attrs
02   elements as AttributeLabel -> AttributeType
03
04 enum AttributeType
05   INT
06   STR
07   BOOL
08   FLOAT
```

Initialisierung

Bei der Initialisierung des Klassendiagramms werden die ASM Strukturen des Klassendiagramms bewertet. Diese Werte sind abhängig von der Modellspezifikation und werden deswegen von Fujaba generiert. In Abbildung 4.9 wird die Generierung dieser Initialwerte dargestellt.

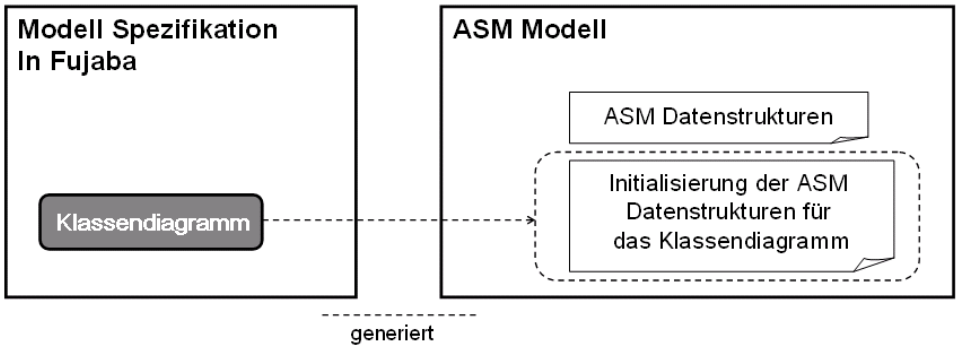

Abbildung 4.9: Generierung der Initialwerte für das Klassendiagramm

Der vollständiger AsmL Code der die Initialisierung des Klassendiagramms des Beispiels aus Kapitel 4.1 beschreibt befindet sich im Anhang. Als erstes werden die Klassenbezeichnungen generiert, also Elemente vom Typ *Node*:

```
01 classNode0 = new Node("Scheduler")
02 …
03 classNode21 = new Node("WaitForShuttle")
```

Nachdem die Elemente kreiert worden sind, werden sie in die Menge *NL* eingefügt, die durch das Element *classNL* repräsentiert wird.

```
04 classNL = new NL({classNode0, …, classNode21})
```

In der gleichen Weise werden auch die Kantenbezeichnungen initialisiert.

```
05 classEdge0 = new Edge("hasState")
06 …
07 classEdge16 = new Edge("uses")
```

Die Klasse *EL*, die durch das Objekt *classEL* repräsentiert wird, enthält alle Kantenbezeichnungen.

```
08 classEL = new EL({classEdge0, …, classEdge16})
```

Die Attribute einer Klasse werden durch ein Tupel gebildet, das aus den Klassennamen und den Namen des Attributes besteht. Klassennamen entsprechen Objekten des Typs *Node* oder Elementen der Menge *NL*.

```
09 classAttributeLabel0 = new AttributeLabel(classNode2, "createsProduct")
10 classAttributeLabel1 = new AttributeLabel(classNode2, "needsProduct")
11 classAttributeLabel2 = new AttributeLabel(classNode6, "productType")
```

Sind alle Attribute erzeugt, so werden sie in die Menge *A* eingefügt, die durch das AsmL Objekt *classA* repräsentiert wird.

```
12 classA = new A({classAttributeLabel0, classAttributeLabel1,
                   classAttributeLabel2})
```

Die Vererbungskanten im Klassendiagramm sind Elemente der Menge *IsAs*. Jedes Element besteht aus zwei Knotenbezeichnungen, der Unterklasse und der Oberklasse.

```
46 classIsAs = new IsAs({(classNode7, classNode1), …, (classNode13,
classNode5)})
```

Die Abbildung der Kantennamen auf die Assoziationseigenschaften wird durch das Objekt *classAssocs* repräsentiert.

```
47 classAssocs = new Assocs({
   classEdge0 |-> (classNode3, one, {}, classNode4, one), …,
   classEdge16 |-> (classNode21, one, {}, classNode2, one)})
```

Um den Typ der Attribute zu definieren, wird die Struktur *Attrs* initialisiert.

```
48 classAttrs = new Attrs({classAttributeLabel0 |-> STR,
classAttributeLabel1 |-> STR, classAttributeLabel2 |-> STR})
```

Nachdem alle Elemente erzeugt worden sind, werden sie schließlich jene in das Tupel, das die Struktur des Klassendiagramms beschreibt, hinzugefügt. Diese Struktur wird durch das Element *classSI* beschrieben.

```
49 classSI = new SIGraph(classNL, classEL, classA, classIsAs, classAssocs,
classAttrs)
```

4.3 Story Patterns, Methoden, Aktivitätsdiagramme

In Kapitel 4.2 wurden die statischen Eigenschaften des Systems betrachtet, die durch das Klassendiagramm spezifiziert worden sind. In diesem Kapitel wird die Abbildung der übrigen Diagramme auf ASM beschrieben und die Methoden betrachtet, also das Verhalten jeder Klasse des Klassendiagramms. Das Verhalten einer Methode wird in ihrer Implementierung spezifiziert. Um eine Methode zu implementieren, werden Story Patterns eingesetzt. Story Patterns sind Graphersetzungsregeln, d. h. durch ihre Anwendung können Änderungen in der

Laufzeitobjektstruktur vorgenommen werden. Die Reihenfolge der Ausführung der einzelnen Story Patterns wird durch ein Aktivitätsdiagramm bestimmt. In Abbildung 4.10 wird die Methode *assignShuttle* der Klasse Shuttle dargestellt. Diese Methode enthält fünf Aktivitäten, zwei davon sind Story Aktivitäten und enthalten ein Story Pattern.

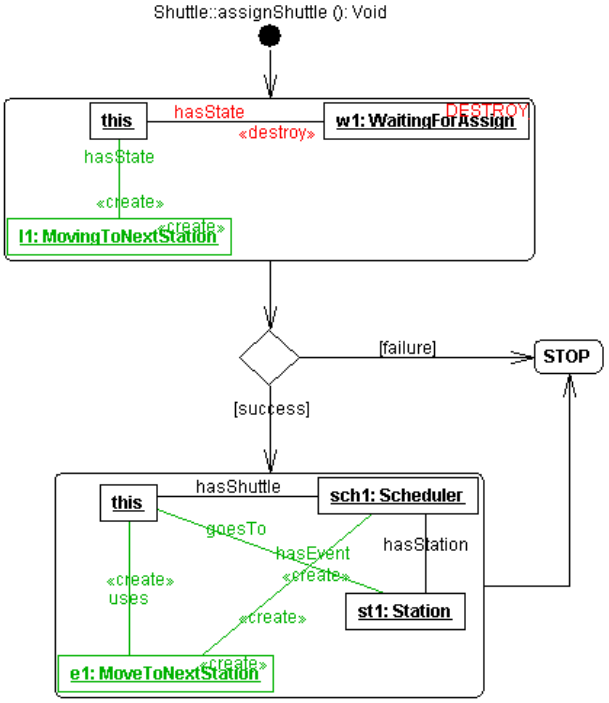

Abbildung 4.10: Die Methode *assignShuttle*

Da die Story Patterns den Kern der Implementierung einer Methode bilden, werden sie zuerst betrachtet. Ein Story Pattern ist eine Graphersetzungsregel, die durch ein Kollaborationsdiagramm beschrieben wird. Dieses Kollaborationsdiagramm kann in zwei Teilgraphen getrennt werden. Der linke Teilgraph beschreibt die Anwendungsstelle der Regel, der rechte Teilgraph beschreibt die Modifikationen, die vorgenommen werden, falls so eine Anwendungsstelle im Wirtsgraphenen existiert. Jeder Teilgraph ist selbst ein Objektdiagramm. Aus diesem Grund wird zunächst die Abbildung von Objektdiagrammen auf ASM betrachtet. Anschließend wird die Abbildung von Story Patterns beschrieben, dann werden die Aktivitätsdiagramme betrachtet. Schließlich wird es möglich sein, die Methoden in ASM abzubilden.

4.3.1 Einführung in die Story Patterns

Story Patterns [13] sind Graphersetzungsregeln, die das dynamische Verhalten eines Systems beschreiben. Sie spezifizieren die Modifikationen an Objekten und deren Beziehungen und damit die Objektstruktur des Systems, die sich von der Laufzeitobjektstruktur oder dem Wirtsgraphen beschreiben lässt. Ein Story Pattern wird durch ein modifiziertes Kollaborationsdiagramm dargestellt. Dieses Diagramm besteht aus Variablen (oder Objekten) und Links, die die Variablen miteinander verbinden. Die Modifizierungen, die nach der Ausführung des Story Patterns im Wirtsgraphen stattfinden sollen, werden von den Elementen, die mit *«create»* oder *«destroy»* markiert sind, modelliert. Ferner ist es möglich, mit Attributzuweisungen die Attributwerte von Objekten zu verändern oder mit Attributbedingungen Bedingungen über Attributwerte zu setzten. Diese Eigenschaften eines

Story Patterns beschreiben die grundlegende Semantik des Story Patterns. Im Kapitel Teilgraphensuche 4.3.5 werden weitere Elemente beschrieben, die die Semantik der Story Patterns erweitern.

Um die Semantik der Story Patterns zu formalisieren, werden diese aus der Sicht der Theorie der Graphersetzungsregeln betrachtet. Ein Story Pattern *G* besteht somit aus zwei Teilgraphen von *G*, dem linken Graphen *LG* und dem rechten Graphen *RG*. Jeder dieser Graphen ist ein Objektdiagramm. Der linke Graph beschreibt die Situation vor der Ausführung des Story Patterns, der rechte die Situation danach. Aus diesem Grund enthält der linke Graph alle Elemente (Objekte und Links) aus *G*, die unverändert bleiben, und alle, die mit *«destroy»* markiert sind. Analog dazu enthält der rechte Graph alle Elemente, die unverändert bleiben, und alle Elemente mit der Markierung *«create»*. In Abbildung 4.11 wird als Beispiel das erste Story Pattern der Methode *assignShuttle* dargestellt. Dieses wird dann in die Graphen *LG* und *RG* geteilt.

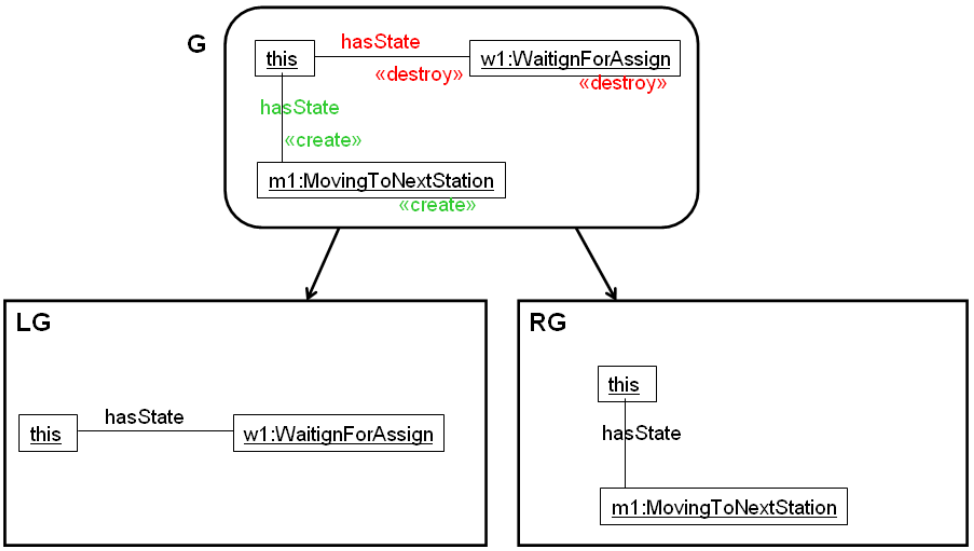

Abbildung 4.11: Der linke Graph *LG* und der rechte Graph *RG* des Story Patterns *G*

Ein Story Pattern führt eine Transformation im Wirtsgraphen durch. Dafür muss zuerst eine Anwendungsstelle im Wirtsgraphen gefunden werden. Eine Anwendungsstelle ist ein Teilgraph des Wirtsgraphen. Der linke Graph *LG* des Story Patterns beschreibt die Bedingungen der Anwendungsstelle, der rechte Graph *RG* die Modifikationen der Anwendungsstelle nach der Ausführung des Story Patterns. Bei der Ausführung einer Graphersetzungsregel auf den Wirtsgraphen wird ein Teilgraph des Wirtsgraphen gesucht, der isomorph zum linken Graphen der Graphersetzungsregel ist. Die Isomorphie wird in [13] beschrieben. Wenn eine Anwendungsstelle, also ein Teilgraph gefunden worden ist, der isomorph zum linken Graphen der Graphersetzungsregel ist, wird der gefundene Teilgraph des Wirtsgraphen den Modifikationen des rechten Graphen entsprechend umgewandelt. Wird kein Teilgraph des Wirtsgraphen gefunden, der isomorph zum linken Graphen *LG* des Story Patterns ist, so bricht die Ausführung des Story Patterns ab.

In den nächsten Paragraphen wird die Struktur eines Objektdiagramms in ASM beschrieben und erklärt, wie diese Strukturen initialisiert werden. Mit Hilfe des Objektdiagramms ist es dann möglich, die Struktur der Story Patterns in ASM zu beschrieben. Nachdem die Strukturen definiert worden sind, kann die Semantik der Story Patterns in ASM abgebildet werden.

4.3.2 Abbildung von Objektdiagrammen auf ASM

Ein Objektdiagramm beschreibt die Struktur und die Beziehungen von Objekten. Die Struktur eines Objektdiagramms wird durch das Klassendiagramm bestimmt. Ein Objekt, das eine Instanz einer Klasse ist, besitzt eine eindeutige Identifikation. Der interne Zustand eines Objektes wird durch die Attribute der entsprechenden Klasse bestimmt. Die Existenz eines Objektes ist unabhängig von der Initialisierung seiner Attribute. Objekte haben Beziehungen zueinander, die im Klassendiagramm durch die Assoziationen definiert sind. Im Objektdiagramm werden diese Beziehungen als Links bezeichnet. Links verbinden zwei Objekte und sind Instanzen einer Assoziation. Falls eine Assoziation geordnet ist, kann der entsprechende Link einen Index besitzen, und falls die Assoziation qualifiziert ist, kann er einen Wert besitzen. Wie bereits erwähnt, sind der linke und rechte Graph eines Story Patterns Objektdiagramme. Auch der Wirtsgraph wird durch ein Objektdiagramm beschrieben.

Struktur des Objektdiagramms

Das Abbilden eines Objektdiagramms auf ASM geschieht wie im Klassendiagramm durch statische ASM Datenstrukturen. Da die Strukturen des Objektdiagramms für alle Modelle gleich bleiben, werden sie nicht generiert. Dieses wird in Abbildung 4.12 dargestellt.

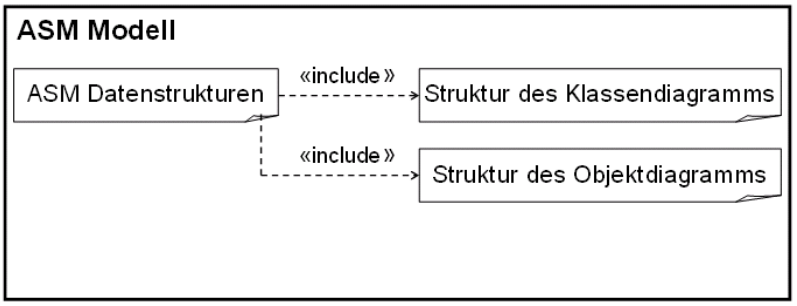

Abbildung 4.12: Struktur des Objektdiagramms im ASM Modell

Ähnlich wie die des Klassendiagramms wird auch die Struktur des Objektdiagramms aufgebaut. Die Objekte des Graphen werden als Knoten dargestellt und die Links als Kanten. Es gibt also eine Menge von Knotenbezeichnungen und eine Menge von Kantenbezeichnungen. Da es sich um ein objektorientiertes Diagramm handelt, müssen weitere Strukturen definiert werden, um die Eigenschaften der Links zu beschreiben. Dass jedes Objekt eine Instanz einer Klasse ist (*instanceOf*-Relation), wird durch eine Abbildung definiert, genau wie die Attributwerte eines Objektes auch. Schließlich werden alle Mengen und Abbildungen durch ein Tupel als eine Struktur definiert, die das Objektdiagramm beschreibt. Die Tabelle 4.2 stellt in kurzer Form die ASM Strukturen eines Objektdiagramms dar.

Struktur des Objektdiagramms	
ExtNode	Struktur der Objektnamen (IDs)
Ind	Struktur der Indizes
Index	Menge aller Indizes
QualAttr	Struktur der qualifizierten Attribute

QualAttrs	Menge aller qualifizierten Attribute
ExtEdge	Struktur der Kanten (Links)
ExtN	Menge aller Objekte
ExtE	Menge aller Kanten
Nl	Abbildung der *instanceOf* Relation zwischen Objekt und Klasse
AttrValue	Struktur der Attributwerte
AttrValues	Menge der Attributwerte
Av	Abbildung eines Objektes und eines Attributes auf einem Attributwert
ExtGraph	Struktur des Objektdiagramms

Tabelle 4.2: ASM Strukturen für das Objektdiagramm

Das Objektdiagramm wird durch die Struktur Ext (extension) repräsentiert. Diese besteht aus einem Tupel, welches wiederum aus der Menge aller Objekte und Links, der Abbildung, die der Relation *instanceOf* entspricht, der Abbildung der Attribute von Objekten auf ihren Wert und der Menge von Indizes und qualifizierten Attributwerte besteht.

```
01 class ExtGraph
02 var param as ExtN * ExtE * Nl * Av * Index * QualAttrs
```

In den folgenden Paragraphen werden die einzelnen Strukturen erläutert.

Knotenmenge

Jedes Objekt (Knoten) des Objektdiagramms bekommt eine Identifikation (ID). Diese Identifikation ist vom Typ String und ist eindeutig für jedes Objekt. Ein Objekt wird durch die Struktur ExtNode definiert. Die Menge aller Objekte bildet die Menge *ExtN*.

```
01 class ExtN
02 var elements as Set[ExtNode]
```

Das Attribut *elements* enthält eine Menge von Elementen des Typs *ExtNode*.

```
01 class ExtNode
02   name as String
```

Kantenmenge

Instanzen der Assoziationen des Klassendiagramms im Objektdiagramm entsprechen den Kanten oder Links. Die Menge aller Links im Objektdiagramm wird als *ExtE* bezeichnet.

```
01 class ExtE
02   var elements as Set[ExtEdge]
```

Diese Menge besteht aus Elementen des Typs *ExtEdge*.

```
01 class ExtEdge
02   name as ExtNode * Edge * Ind * QualAttr * ExtNode
```

Das Attribut *name* der Struktur *ExtEdge* hat den Typ eines Tupels. Die Strukturen *Ind* und *QualAttr* werden in den nächsten Abschnitten definiert. Das Tupel eines Links besteht aus zwei Elementen des Typs *ExtNode*, also dem *source*- und dem *target*-Knoten. Das Element des Typs *Edge* beschreibt, von welcher Assoziation des Klassendiagramms dieser Link

Instanz ist. Das Element des Typs *Ind* bezeichnet den Index des Links, falls die Assoziation, die dem Typ des Links entspricht, geordnet ist. Falls die Assoziation qualifiziert ist, bezeichnet das Element des Typs *QualAttr* den Wert des Links.

Knotenbeschriftung

In der Menge *Nl* wird gespeichert, von welcher Klasse ein Objekt Instanz ist. Dieses entspricht der *instanceOf* Relation zwischen Objekten und ihren Klassen.

```
01 class Nl
02   var elements as ExtNode -> Node
```

Das Attribut *elements* ist eine Abbildung eines Elements des Typs *ExtNode* auf einem Element des Typs *Node*. Dem ersten entspricht das Objekt, dem zweiten entspricht die Klasse.

Bewertung der Attribute

Durch die Menge *Av* werden die Attribute bewertet.

```
01 class Av
02   var elements as ExtNode * AttributeLabel -> AttrValue
```

Die Bewertung erfolgt wieder durch das Attribut *elements*, dessen Typ ein Map ist, das ein Tupel in einem Element des Typs *AttrValue* abbildet. Das Tupel besteht aus dem Objekt und dem Attribut, das bewertet wird. Das Attribut wird durch das Element des Typs *AttributeLabel* angegeben.

```
01 class AttrValue
02   value as String

03 class AttrValues
04   elements as Set[AttrValue]
```

Die Menge *AttrValues* ist die Menge aller Attributwerte.

Index

Geordnete oder sortierte Links können einen Integer Wert enthalten, der dem Index des Links entspricht. Dieser wird durch die Struktur *Ind* beschrieben, die als Attribut eine Variable *value* mit dem Typ `Integer` hat. Die Struktur *Index* enthält alle Indizes.

```
01 class Index
02   var elements as Set[Ind]

03 class Ind
04   var value as Integer
```

Qualifizierte Attribute

Ähnlich wie beim Index werden auch die Werte der Links, die Instanzen von qualifizierten Assoziationen sind, in einer Menge gespeichert. Die Struktur *QualAttr* ist äquivalent mit der Struktur *AttrValue*. Die Menge aller Elemente des Typs *QualAttr* wird durch die Struktur *QualAttrs* beschrieben.

```
01 class QualAttr
02   value as String

03 class QualAttrs
```

```
04   var elements as Set[QualAttr]
```

Initialisierung

Bei der Initialisierung des Objektdiagramms werden die ASM Strukturen des Objektdiagramms bewertet. Diese Werte sind abhängig von der Modellspezifikation und werden daher von Fujaba generiert. Zuerst wird die Laufzeitobjektstruktur initialisiert. Anschließend werden für jedes Story Pattern zwei Objektdiagramme initialisiert. In Abbildung 4.13 wird die Generierung dieser Initialwerte dargestellt.

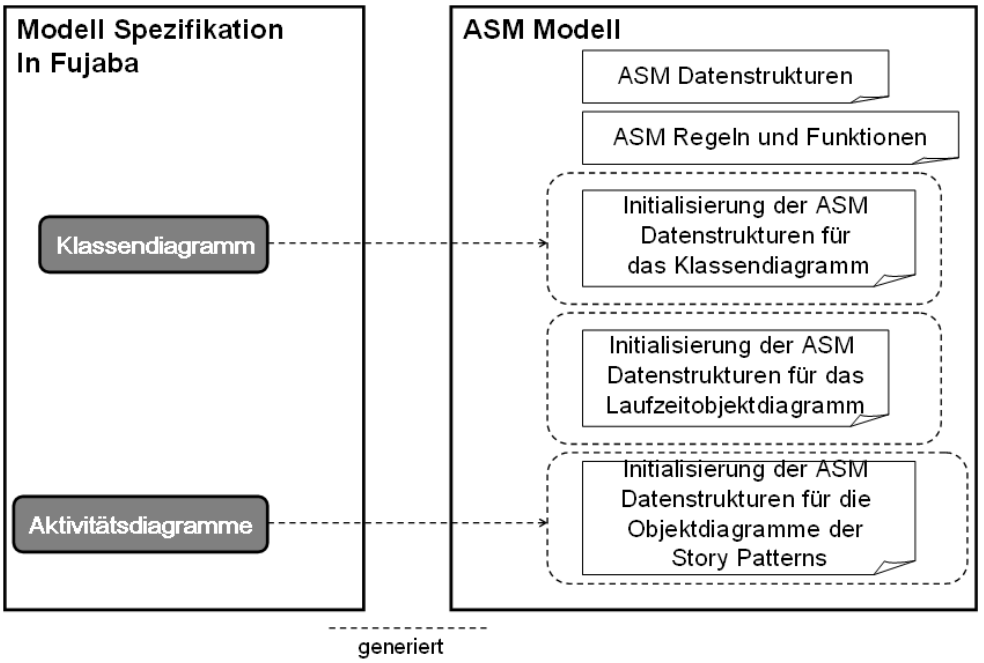

Abbildung 4.13: Generierung der Initialwerte für die Objektdiagramme

Die Laufzeitobjektstruktur ist ein Objektgraph. Vor der Ausführung der Spezifikation enthält der Wirtsgraph nur einen Knoten. Dieser Knoten entspricht dem Objekt, auf dem die Methode aufgerufen wird, die das Ausführen der Spezifikation beginnt. Diese Methode wird durch den Benutzer definiert. Die Ausführung der Methode findet in der Funktion *run* statt.

Im Beispiel des Kapitels 4.1 beginnt die Methode *init* das Ausführen der ASM Spezifikation des Systemmodells. Am Anfang besteht daher der Wirtsgraph nur aus einem Objekt der Klasse *Scheduler.*

```
01 startExtnode = new ExtNode("1")
02 extN = new ExtN({ startExtnode })
03 extInd = new Ind(0)
04 extIndex = new Index({ extInd })
05 extQualAttr = new QualAttr("")
06 extQualAttrs = new QualAttrs({ extQualAttr })
07 extE = new ExtE({ })
08 extNl = new Nl({ startExtnode |-> classNode0 })
09 extAv = new Av({ |-> })
10 extEG = new ExtGraph(extN, extE, extNl, extAv, extIndex, extQualAttrs)
```

In Zeile 1 wird ein Objekt mit der Bezeichnung 1 erzeugt. Knoten, die im Laufe der Zeit kreiert werden, bekommen eine eindeutige Bezeichnung, in Form eines Index. Dieser ist die

Variable *NodeIndex* vom Typ `Integer`. Das erste Objekt des Wirtsgraphen ist, wie oben beschrieben, eine Instanz der Klasse *Scheduler* und bekommt die Bezeichnung 1. Deswegen wird die Variable *NodeIndex* mit 2 initialisiert.

```
01 var NodeIndex as Integer = 2
```

Als nächstes wird in Zeile 2 die Menge aller Knoten gebildet. In den Zeilen 4 bis 6 werden die Mengen *Index* und *QualAttrs* mit den leeren Elementen initialisiert. Da nur ein Objekt im Wirtsgraphen existiert, werden keine Kanten kreiert und die Menge der Kanten wird mit der leeren Menge initialisiert. In Zeile 8 wird die Abbildung *Nl* initialisiert. Die Klasse *Scheduler* entspricht dem Element *classNode0*, daher wird *Nl* mit der Abbildung des Elementes *startExtnode* auf dem Element *classNode0* initialisiert. Die Klasse *Scheduler* enthält keine Attribute, und somit auch ihre Instanzen nicht. Die Abbildung *extAv* wird also in Zeile 9 mit der leeren Abbildung initialisiert. Schließlich werden alle Mengen und Abbildungen für die Initialisierung des Wirtsgraphen *extEG* in Zeile 10 benutzt.

Ein Beispiel für die Initialisierung der Objektdiagramme eines Story Patterns wird im nächsten Kapitel betrachtet.

4.3.3 Abbildung der Story Patterns auf ASM

Wie bereits im 4.3.1 dargelegt, wird ein Story Pattern durch zwei Objektdiagramme beschrieben. Doch zusätzlich zu dem linken und rechten Graphen werden noch weitere Strukturen gebraucht, die für die Ausführung des Story Patterns wichtig sind. Daher wird eine eigene ASM Datenstruktur für die Abbildung der Story Patterns definiert.

Struktur der Story Patterns

Das Abbilden eines Story Patterns auf ASM geschieht, wie auch im Klassen- und Objektdiagramm, durch statische ASM Datenstrukturen. Dieses wird in Abbildung 4.14 dargestellt.

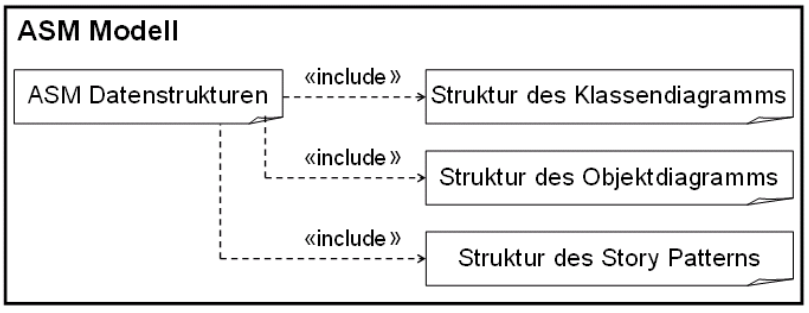

Abbildung 4.14: Struktur der Story Patterns im ASM Modell

Ein Story Pattern *G* besteht aus zwei Objektdiagrammen. Der linke Graph *LG* beschreibt die Bedingungen für die Ausführung von *G*, der rechte Graph *RG* beschreibt die Modifikationen, die an der Anwendungsstelle des Wirtsgraphen vorgenommen werden. Es werden zwei Mengen für die zu löschenden Elemente kreiert, eine für die Knoten und eine für die Kanten. Analog werden auch zwei Mengen für die zu kreierenden Elemente festgelegt. Zusätzlich wird eine Menge definiert, die die unveränderten Knoten enthält. Bei der Ausführung eines Story Patterns ist es nötig, den Teilgraphen des Wirtsgraphen, der der Anwendungsstelle des

Story Patterns entspricht, zu speichern. Für diesen Zweck wird die Struktur *Match* definiert. Die Struktur *Copy* hat eine ähnliche Funktionalität, dient aber dem Speichern der Knoten, die zu dem Teilgraphen, der der Anwendungsstelle entspricht, hinzugefügt werden. Die Tabelle 4.3 stellt in kurzer Form die ASM Strukturen eines Story Patterns dar.

Struktur des Story Patterns	
LG	Linker Regelsatz
RG	Rechter Regelsatz
DelN	Menge der zu löschenden Variablen
DelE	Menge der zu löschenden Links
CoreN	Menge der unveränderbaren Variablen
AddN	Menge der zu kreierenden Variablen
AddE	Menge der zu kreierenden Links
Match	Abbildung eines Elementes aus dem Story Pattern auf einem entsprechenden Element des Wirtsgraphen
Copy	Abbildung der zu kreierenden Variablen auf neuen erzeugten Objekten im Wirtsgraphen
Grr	Struktur des Story Patterns

Tabelle 4.3: ASM Strukturen für das Story Pattern

Ein Story Pattern wird durch die Struktur *Grr* (graphgrammar rewrite rule) definiert. Ein Story Pattern besteht aus zwei Objektgraphen, die durch die Struktur *ExtGraph* definiert werden, aus fünf weiteren Mengen von Knoten und auch aus einer *Match*- und einer *Copy*-Abbildung, die das Binden des linken Teilgraphen definieren. Außerdem enthält jedes Story Pattern einen eindeutigen Namen. Dieser wird durch den Namen des Story Patterns, so wie dieser im Aktivitätsdiagramm angegeben worden ist, und durch einen Index, der die Eindeutigkeit des Namens garantiert, definiert.

```
01 class Grr
02   param as ExtGraph * ExtGraph * DelN * DelE * CoreN * AddN * AddE *
03            Match * Copy
04   name as String
```

Linker und rechter Teilgraph

Ein Story Pattern besteht aus einem linken (*LG*) und einem rechten (*RG*) Objektgraphen. Diese werden durch die Struktur *ExtGraph* definiert. Der linke Graph beschreibt die Situation, die eintreten muss, damit das Story Pattern angewandt wird. Falls so eine Situation im Wirtsgraphen eintritt, wird das Story Pattern ausgeführt, andernfalls nicht.

Modifikationen im Story Pattern

Der Durchschnitt der Menge der Knoten vom linken und rechten Teilgraphen bildet die Elemente, die nach der Anwendung des Story Patterns auf den Wirtsgraphen unverändert bleiben. Sie werden in der Menge *CoreN* definiert.

```
01 class CoreN
02   elements as Set[ExtNode]
```

Die Elemente, die in *LG* aber nicht in *RG* existieren, sind Elemente die nach der Anwendung des Story Patterns gelöscht werden. Dabei können die Knoten aber auch Links sein.

```
01 class DelN
02   elements as Set[ExtNode]

03 class DelE
04   elements as Set[ExtEdge]
```

Als letztes werden die Elemente definiert, die in *RG* existieren aber nicht in *LG*. Diese Elemente werden nach der Anwendung des Story Patterns im Wirtsgraphen erzeugt oder vom *RG* kopiert.

```
01 class AddN
02   elements as Set[ExtNode]

03 class AddE
04   elements as Set[ExtEdge]
```

Match und Copy Funktionen

Um ein Story Pattern anzuwenden, muss zuerst ein Teilgraph des Wirtsgraphen gefunden werden, der isomorph zum linken Teilgraphen ist. Wenn so ein Teilgraph gefunden ist, werden die Knoten und Kanten des Teilgraphen an die Knoten und Kanten des linken Teilgraphen *LG* gebunden. Das erfolgt durch die *Match* Struktur, die Objekte aus dem linken Teilgraphen auf Objekten des Wirtsgraphen abbildet.

```
01 class Match
02   var paramN as ExtNode -> ExtNode
03   var paramE as ExtEdge -> ExtEdge
```

Das Attribut *paramN* ist eine Abbildung des Knotens in LG auf den Knoten des Wirtsgraphen. Das Attribut *paramE* ist ein Map zwischen den Kanten in LG und den Kanten des Wirtsgraphen.

Es bleibt noch eine Struktur zu definieren, die eine Abbildung der Knoten, die in *AddN* gehören, im Wirtsgraphen abbildet. Da zu diesen Knoten keine entsprechenden im Wirtsgraphen existieren, müssen neue erzeugt werden. Dies wird durch das Kopieren dieser Knoten in den Wirtsgraphen erreicht. Dabei wird auf die Identifikation der neuen Knoten geachtet.

```
01 class Copy
02   var param as ExtNode -> ExtNode
```

Initialisierung der Story Patterns

Bei der Initialisierung eines Story Patterns werden die ASM Strukturen des Story Patterns bewertet. Diese Werte sind abhängig von der Modellspezifikation und werden daher von Fujaba generiert. In Abbildung 4.15 wird die Generierung dieser Initialwerte dargestellt.

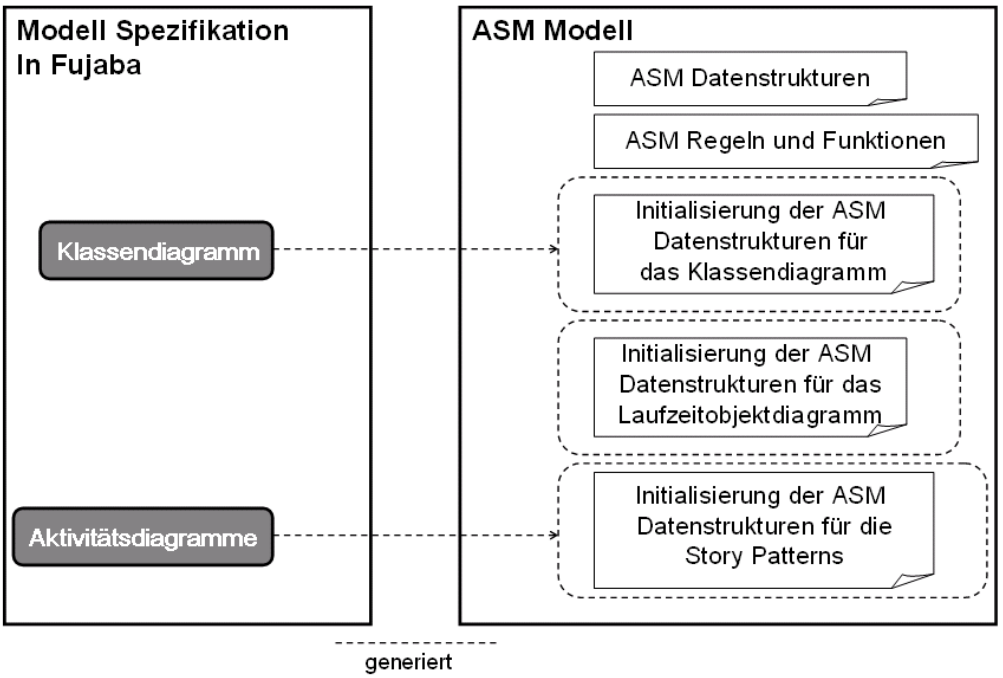

Abbildung 4.15: Generierung der Initialwerte für die Story Patterns

Als Beispiel für die Initialisierung eines Story Patterns wird die Methode *init* betrachtet, die in der Abbildung 4.3 dargestellt wird. Der generierte Code besteht aus drei Teile. Im ersten Teil wird der linke Graph *LG* generiert, im zweiten Teil der, und im dritten Teil die Initialwerte für die übrigen Strukturen, die das Story Pattern beschreiben.

Als nächstes werden Teile des AsmL Codes betrachtet, die die Initialisierung des Story Patterns der Methode *init* beschreiben. Der vollständiger Code befindet sich im Anhang. Folgender AsmL Code ist die Initialisierung des linken Graphen *LG* des Story Patterns der Methode *init*:

```
01 init_2LGNode0 = new ExtNode("thisVariable")
02 init_2LGN = new ExtN({init_2LGNode0})
03 init_2LGInd0 = new Ind(0)
04 init_2LGIndex = new Index({init_2LGInd0})
05 init_2LGQualAttr0 = new QualAttr("")
06 init_2LGQualAttrs = new QualAttrs({init_2LGQualAttr0})
07 init_2LGE = new ExtE({})
08 init_2LGNl = new Nl({init_2LGNode0 |-> classNode0})
09 init_2LGAv = new Av({|->})
10 init_2LGAttrValues = new AttrValues({})
11 init_2LGGraph = new ExtGraph(init_2LGN, init_2LGE, init_2LGNl,
                     init_2LGAv, init_2LGIndex, init_2LGQualAttrs)
```

Die Initialisierung des linken Graphen *LG* ist ähnlich der Initialisierung des Laufzeitobjektdiagramms in 4.3.2. Der Graph besteht aus einem Objekt, dem *this* Objekt. In Zeile 1 wird ein Element des Typs *ExtNode* für dieses Objekt erzeugt. Die Bezeichnung *„thisVariable“* entspricht der Bezeichnung *„this“* der Variable des Story Patterns[4]. In Zeile 2 wird die Menge der Knotenbezeichnungen erzeugt, die als einziges Element den Knoten *init_2LGNode0* enthält. Im Graphen existieren keine Links, und so enthalten die Mengen *init_2LGIndex* und *init_2LGQualAttrs* nur die leeren Elemente. Aus dem gleichen Grund wird auch die Menge der Kantenbezeichnungen *init_2LGE* mit der leeren Menge initialisiert. In

[4] Die Bezeichnung *this* wird aus technischen Gründen, die im Kapitel 5 erläutert werden, modifiziert.

Zeile 8 wird in *init_2LGNl* die Abbildung der Variable *this* auf der Klasse *Scheduler* eingefügt. Da die Variable *this* keine Attribute enthält, bleibt auch die Abbildung *init_2LGAv* und die Menge *init_2LGAttrValues* leer. Schließlich wird das Element *init_2LGGraph* erzeugt, welches dem linken Graphen *LG* des Story Patterns der Methode *init* entspricht.

Als nächstes wird die Initialisierung des linken Graphen *LG* des Story Patterns der Methode *init* betrachtet:

```
12 init_2RGNode0 = new ExtNode("w1")
13 …
14 init_2RGNode7 = new ExtNode("sh")
15 init_2RGN = new ExtN({init_2RGNode0, …, init_2RGNode7})
16 init_2RGInd0 = new Ind(0)
17 init_2RGIndex = new Index({init_2RGInd0})
18 init_2RGQualAttr0 = new QualAttr("")
19 init_2RGQualAttrs = new QualAttrs({init_2RGQualAttr0})
20 init_2RGExtEdge0 = new ExtEdge(init_2RGNode2, classEdge8, init_2RGInd0,
                                  init_2RGQualAttr0, init_2RGNode7)
21 …
22 init_2RGExtEdge10 = new ExtEdge(init_2RGNode7, classEdge0, init_2RGInd0,
                                   init_2RGQualAttr0, init_2RGNode0)
23 init_2RGE = new ExtE({init_2RGExtEdge0, …, init_2RGExtEdge10})
24 init_2RGNl = new Nl({init_2RGNode0 |-> classNode11, …,
                        init_2RGNode7 |-> classNode3})
25 init_2RGAttrValue0 = new AttrValue("A")
26 …
27 init_2RGAttrValue4 = new AttrValue("A")
28 init_2RGAv = new Av({(init_2RGNode3, classAttributeLabel0) |->
    init_2RGAttrValue0, …, (init_2RGNode6, classAttributeLabel1) |->
    init_2RGAttrValue4})
29 init_2RGAttrValues = new AttrValues({init_2RGAttrValue0, …,
                                        init_2RGAttrValue4})
30 init_2RGGraph = new ExtGraph(init_2RGN, init_2RGE, init_2RGNl,
                    init_2RGAv, init_2RGIndex, init_2RGQualAttrs)
```

Die Initialisierung des rechten Graphen *RG* enthält alle Elemente, die unveränderbar bleiben und alle, die neu kreiert werden. In den Zeilen 12 bis 14 werden alle Variablen mit der Markierung *«create»* erzeugt. Die Knoten, die unveränderbar bleiben, werden aus der Initialisierung von *LG* weiterbenutzt. So kommt, in Zeile 15, bei der Initialisierung von *init_2RGN* auch der Knoten *init_2LGNode0* vor. Die Links, die in *RG* enthalten sind, besitzen keinen Index oder Wert, und so enthalten die Mengen *init_2RGIndex* und *init_2RGQualAttrs* nur das leere Element[5]. Als nächstes werden die Links des Graphen als Elemente des Typs *ExtEdge* erzeugt und in die Menge *init_2RGE* eingefügt. In Zeile 24 wird die Abbildung *init_2RGNl* initialisiert, die der Relation *instanceOf* der Variablen entspricht. Auch hier kommt das Element *init_2LGNode0* vor. Die Attributwerte der Variablen werden in den Zeilen 25 bis 27 initialisiert. Für jeden Attributwert wird ein Element des Typs *AttrValue* erzeugt. Die Abbildung der Attribute auf ihrem Attributwert geschieht in Zeile 28 bei der Initialisierung der Abbildung *init_2RGAv*. Alle Attributwerte bilden die Menge *init_2RGAttrValues* (Zeile 29). Schließlich werden alle Mengen und Abbildungen für die Initialisierung des Elements *init_2RGGraph* benutzt, welches dem rechten Graphen des Story Patterns der Methode *init* entspricht.

[5] Am Ende dieses Kapitels wird eine Modifizierung des Beispieles betrachtet, um den Fall einer qualifizierten Assoziation bei der Generierung der Initialwerte des gleichen Story Patterns zu erläutern.

Nach der Generierung der Initialwerte von *LG* und *RG* werden die Initialwerte der übrigen ASM Datenstrukturen, die das Story Pattern der Methode *init* beschreiben, generiert.

```
31 init_2DelN = new DelN({})
32 init_2DelE = new DelE({})
33 init_2CoreN = new CoreN({init_2LGNode0})
34 init_2AddN = new AddN({init_2RGNode0, …, init_2RGNode7})
35 init_2AddE = new AddE({init_2RGExtEdge0, …, init_2RGExtEdge10})
36 init_2Match = new Match({ |-> }, { |-> })
37 init_2Copy = new Copy({ |-> })
38 init_2 = new Grr((init_2LGGraph, init_2RGGraph, init_2DelN, init_2DelE,
                    init_2CoreN, init_2AddN, init_2AddE, init_2Match,
                    init_2Copy), "init_2" )
```

In den Zeilen 31 und 32 werden die Mengen der zu löschenden Elemente mit der leeren Menge initialisiert, da im Story Pattern keine Elemente mit der Markierung *«destroy»* vorkommen. Der einzige Knoten, der bei der Ausführung des Story Patterns unveränderbar bleiben, ist der Knoten *init_2LGNode0*, welcher der *this* Variable entspricht. Anschließend werden die zu kreierenden Elemente erzeugt. In Zeile 34 werden in die Menge *init_2AddN* alle Knotenelemente mit der Markierung *«create»* eingefügt, und in Zeile 35 werden in der Menge *init_2AddE* alle Kantenelemente mit derselben Markierung eingefügt. Die Abbildungen *Match* und *Copy* in den Zeilen 36 und 37 werden mit der leeren Abbildung initialisiert. Schließlich werden alle Mengen und Abbildungen, sowie auch die Elemente *init_2LGGraph* und *init_2RGGraph*, die dem linken, bzw. dem rechten Graphen des Story Patterns entsprechen, für die Initialisierung des Elements *init_2* benutzt, welches dem Story Pattern der Methode entspricht.

Im Anschluss wird ein Teil des Klassendiagramms der Abbildung 4.1 verändert. Zu der Klasse *Station* wird ein neues Attribut *id* hinzugefügt, welches auch bei der Assoziation *hasStation* als qualifizierter Wert benutzt wird. In Abbildung 4.16 werden diese Veränderungen dargestellt.

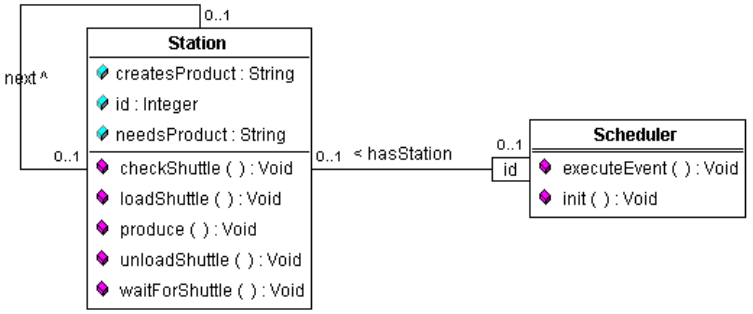

Abbildung 4.16: Veränderungen im Klassendiagramm des Beispiels

Auch das Story Pattern der Methode *init* wird entsprechend verändert. Die Links *hasStation*, die die Stationen *st1* und *st2* mit der Variable *this* verbinden, erhalten jeweils einen Wert für das qualifizierte Attribut *id* (Abbildung 4.17).

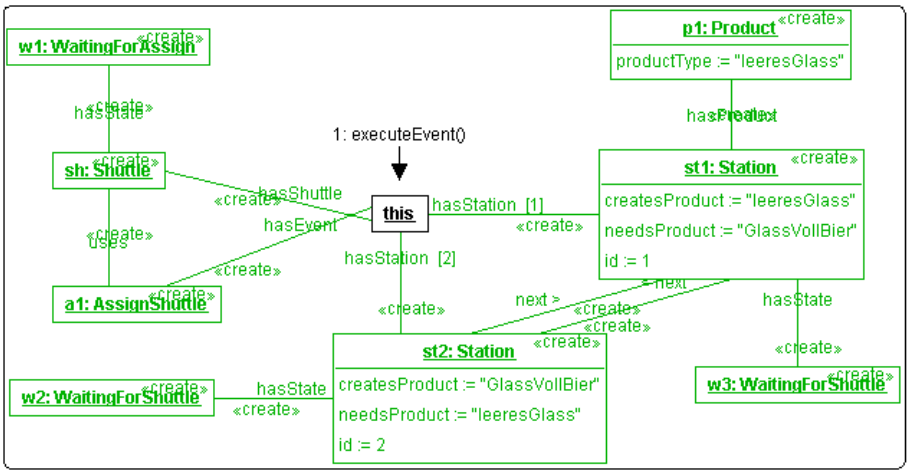

Abbildung 4.17: Veränderungen im Story Pattern der Methode *init*

Nun werden die Unterscheide im generierten Code betrachtet. Zuerst wird der Code der Initialisierung des Klassendiagramms verändert. Für die Klasse Station wird ein weiteres Attribut *id* definiert. Für die Assoziation *hasStation*, die dem Element *classEdge3* entspricht, wird der Eintrag in die Abbildung *Assocs* wie folgt geändert:

```
classEdge3 |-> (classNode0, one, {qualified}, classNode2, one)
```

Bei der Initialisierung des Story Patterns der Methode *this* wird nur *RG* verändert, da die *hasStation* Links als *«create»* markiert sind. Als erstes werden für die qualifizierten Werte zwei neue Elemente des Typs *QualAttr* generiert. Diese werden anschließend in die Menge *init_2RGQualAttrs* eingefügt.

```
40 init_2RGQualAttr0 = new QualAttr("")
   init_2RGQualAttr1 = new QualAttr("1")
   init_2RGQualAttr2 = new QualAttr("2")

41 // QualAttrs
42 init_2RGQualAttrs = new QualAttrs({init_2RGQualAttr0, init_2RGQualAttr1,
                                      init_2RGQualAttr2})
```

Schließlich werden die Elemente *init_2RGExtEdge4* und *init_2RGExtEdge7*, die den Links *hasStation* entsprechen, geändert:

```
48 init_2RGExtEdge4 = new ExtEdge(init_2LGNode0, classEdge3, init_2RGInd0,
                                  init_2RGQualAttr1, init_2RGNode3)
51 init_2RGExtEdge7 = new ExtEdge(init_2LGNode0, classEdge3, init_2RGInd0,
                                  init_2RGQualAttr2, init_2RGNode6)
```

4.3.4 Ausführen von Story Patterns

Nachdem alle nötigen Datenstrukturen erläutert worden sind, ist es nun möglich, ASM Regeln zu definieren, die für das Ausführen eines Story Patterns zuständig sind. Die Semantik eines Story Patterns wird in [13] definiert. In Abbildung 4.18 werden die Regeln und die zusätzlichen Funktionen, die für das Ausführen der Story Patterns nützlich sind im ASM Modell dargestellt.

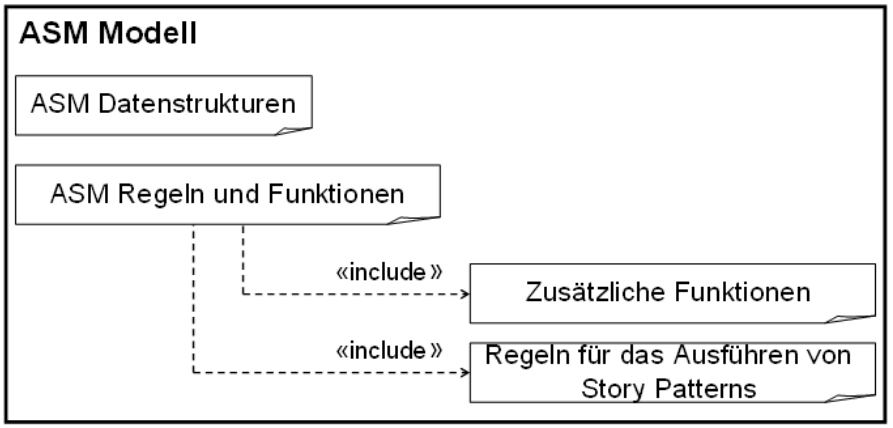

Abbildung 4.18: Regeln und Funktionen für das Ausführen von Story Patterns im ASM Modell

Für das bessere Verständnis der einzelnen Regeln wird hier die Bedeutung einiger häufig benutzter Notationen erläutert. Im Allgemeinen ist mit *G* ein Element des Typs *Grr*, also ein Story Pattern gemeint. Das Laufzeitobjektdiagramm oder der Wirtsgraph wird als ein Element *EG* des Typs *ExtGraph* beschrieben. Schließlich wird mit *S* vom Typ *SIGraph* das Klassendiagramm bezeichnet. In den folgenden Abschnitten werden die wichtigsten Regeln, die für die Ausführung eines Story Patterns notwendig sind, präsentiert.

Grundlegende Semantik

Das Ausführen eines Story Patterns besteht aus drei Schritten. Im ersten Schritt wird eine Teilgraphensuche durchgeführt, um einen Teilgraph des Wirtsgraphen zu finden, der isomorph zum linken Graphen *LG* des Story Patterns ist. Dies geschieht durch das Binden der Variablen des linken Teilgraphen des Story Patterns an Objekte des Teilgraphen des Wirtsgraphen. Nachdem eine Anwendungsstelle für das Story Pattern gefunden worden ist, finden die Modifikationen an der Laufzeitobjektstruktur statt. Der zweite Schritt löscht alle als gelöscht markierten Elemente und addiert entsprechend der dritte Schritt alle zu kreierenden Elemente zum Wirtsgraphen.

Alle drei oben genannten Aktionen werden in der Regel *runGrr* durchgeführt.

```
01 runGrr(G as Grr, EG as ExtGraph, S as SIGraph, thisObject as ExtNode,
      boundObjects as BoundObjects, returnValue as Value, params as
      Parameters) as Boolean=
03   machine
04     let grrName = getGrrName(G)
05     var matchingResult as Boolean = findMatchFor(G, EG, thisObject,
                                       boundObjects, returnValue, params)
06   step
07     if matchingResult then
08       deletionFor(G, EG, S)
09     else
10       skip
11   step
12     if matchingResult then
13       creationFor(G, EG, S, boundObjects)
14     else
15       skip
16   step
17     return matchingResult
```

In dieser Regel werden drei Schritte durchgeführt. In Zeile 5 wird die Regel *findMatchFor* ausgeführt. Diese führt die Teilgraphensuche für das Story Pattern *G* aus, welches als Parameter gegeben wird, und gibt als Ergebnis den `Boolean` *matchingResult* zurück. Ist die Teilgraphensuche erfolgreich, so hat *matchingResult* den Wert `true`, und der nächste Schritt wird durchgeführt. In diesem wird die Regel *deletionFor* für das Story Pattern *G* ausgeführt(Zeile 8). Diese Regel löscht alle Elemente des Story Patterns, welche mit *«destroy»* markiert sind. Nachdem alle zu löschenden Elemente gelöscht worden sind, wird in Zeile 13 die Regel *creationFor* ausgeführt, die alle Elemente mit Markierung *«create»* kreiert. Ist die Teilgraphensuche nicht erfolgreich, so bricht die Ausführung des Story Patterns *G* ab, und die zwei letzten Schritte werden nicht durchgeführt, d.h. der Wirtsgraph wird nicht verändert. In den nächsten Paragraphen werden die einzelnen Schritte näher betrachtet.

Die Regel findMatchFor

Die Regel *FindMatchFor* berechnet für das als Parameter gegebene Story Pattern *G* eine Anwendungsstelle im Wirtsgraphen. Aufgrund ihrer Komplexität, wird sie in einem eigenen Kapitel betrachtet, nämlich dem Kapitel Teilgraphensuche (3.4.5).

Die Regel deletionFor

Nachdem eine Anwendungsstelle für den linken Graph des Story Patterns gefunden worden ist, werden die Modifikationen an der Laufzeitobjektstruktur ausgeführt. Elemente, die im linken Graphen des Story Patterns existieren aber nicht im rechten, werden vom Wirtsgraphen gelöscht. Damit die Konsistenz erhalten bleibt, werden die Mengen *Nl* und *Av* des linken Graphen entsprechend aktualisiert. Die Regel *deletionFor* wird als erstes aufgerufen, und jeweils in einem Schritt der Ausführung werden die Knoten, die Kanten, die Elemente der Abbildung *NodeLabels* und die Attributwerte der gelöschten Knoten entfernt.

```
01 deletionFor(G as Grr, EG as ExtGraph, S as SIGraph) =
02   machine
03     deleteNodesFor(S, G, EG)
04   step
05     deleteEdgesFor(G, EG, S)
06   step
07     deleteNodeLabelsFor(G, EG)
08   step
09     deleteAttributeValuesFor(G, EG)
10   step
11     skip
```

Ohne Beschränkung der Allgemeinheit wird davon ausgegangen, dass Attributwerte von gelöschten Variablen nicht für die Aktualisierung anderer Attribute verwendet werden. Um diesen Fall zu berücksichtigen, würde eine Kopie der zu löschenden Elemente des Story Patterns nötig sein, damit die Attributwerte der zu löschenden Variablen bei der Aktualisierung der Attributwerte erhalten bleiben und zugreifbar sind. Eine andere mögliche Lösung wäre, das Löschen und Kreieren der Elemente, sowie auch das Aktualisieren der Attributwerte in einem ASM Zustand zu erledigen. Das würde den Effekt haben, dass die Aktualisierungen der Mengen definiert werden, die Werte jedoch bis zur nächsten Transitionsregel, die nach der Aktualisierung der Attributwerte stattfinden würde, erhalten bleiben.

Als nächstes werden die einzelnen Regeln der Zeilen 3, 5, 7 und 9 näher betrachtet.

```
01 deleteNodesFor(S as SIGraph, G as Grr, EG as ExtGraph) =
02   let DN = getDelN(G)
```

```
03   let M = getMatch(G)
04   let EN = getExtN(EG)
05   if size(DN.elements)<>0 then
06      EN.elements := EN.elements difference parts(S, G, EG,
                                                 matchSet(DN.elements, EG, M))
07   else
08     skip
```

Die Regel *deleteNodesFor* löscht die Knoten aus dem Wirtsgraphen, die mit den Knoten aus der Menge *DelE* des Story Patterns *G* gebunden sind. In Zeile 5 wird überprüft, ob die Menge *DelN* nicht die leere Menge ist, in Zeile 6 wird der Wert der Menge *EN* (die Menge der Knoten im Wirtsgraphen) aktualisiert. Der neue Wert wird durch die Differenz des alten Wertes von *EN* und der Menge der Knoten des Wirtsgraphen, die mit den Knoten der Menge *DelN* gebunden sind, berechnet. Durch die Funktion *parts* werden alle Kinderknoten berechnet, die mittels Aggregationen mit einem zu löschenden Knoten verbunden sind. Diese Kinderknoten müssen ebenfalls gelöscht werden. Die Funktion *matchSet* gibt für eine Menge von Knoten oder Kanten des Story Patterns die gebundenen Knoten oder Kanten des Wirtsgraphen mittels der Abbildung *M* zurück:

```
01 matchSet (EN as Set[ExtNode], EG as ExtGraph, M as Match) as
                                                    Set[ExtNode] =
02    machine
03      var TempN as Set[ExtNode] = {}
04    step
05      foreach n in EN do
06        if n in dom(m.paramN) then
07          TempN(M.paramN(e)) := true
08    step
09      return TempN
```

Die Funktion *matchSet* bekommt als Parameter eine Menge von Knotenbezeichnungen *EN* aus einem Story Pattern und gibt als Ausgabe eine Menge von Knotenbezeichnungen aus dem Wirtsgraphen. Die Beziehung zwischen beiden Mengen wird durch die Abbildung M, die das Binden der Variablen eines Story Patterns mit Objekten des Wirtsgraphen definiert, beschrieben. In Zeile 3 wird die Menge *TempN*, die schließlich ausgegeben wird, deklariert. In der `foreach`-Schleife der Zeile 5 wird über alle Elemente der Menge *EN* iteriert. Gibt es ein Objekt im Wirtsgraphen, dass mit der Variable n gebunden ist (Zeile 6), so wird dieses Objekt in die Menge *TempN* eingefügt. In Zeile 9 wird die Menge *TempN* durch den `return`-Ausdruck zurückgegeben. Es gibt auch eine analoge Funktion *matchSet* für eine Menge von Kantenbezeichnungen. Aufgrund der identischen Struktur wird hier nur die Signatur der Funktion angegeben:

```
01 matchSet (EE as Set[ExtEdge], EG as ExtGraph, M as Match) as
                                                   Set[ExtEdge] =
02 …
```

Objektorientierte Graphen können Aggregationsbeziehungen haben. Das bedeutet, dass beim Löschen des Vaterknotens einer solchen Beziehung, auch alle Kinderknoten dieses Knotens gelöscht werden müssen. Das Löschen eines Kinderknotens kann wiederum das Löschen neuer Knoten bedeuten, falls dieser mindestens eine Aggregationsbeziehung zu anderen Knoten hat. Wird die Aggregationsbeziehung auch gelöscht, so muss der Kinderknoten nicht gelöscht werden. Die Funktion *parts*, die in der Regel *deleteNodesFor* benutzt wird, berechnet alle über eine Aggregationsbeziehung erreichbare Nachbarknoten einer Menge von Knoten aus dem Wirtsgraphen.

```
01 parts(S as SIGraph, G as Grr, EG as ExtGraph, N as Set[ExtNode]) as
                                                          Set[ExtNode] =
02   machine
03     let DE = getDelE(G)
04     let M = getMatch(G)
05     let EE = getExtE(EG)
06     let SIAssocs = getAssocs(S)
07     let tempE = EE.elements difference matchSet(DE.elements, EG, M)
08     var resultSet as Set[ExtNode] = {}
09   step
10     foreach e in tempE do
11       let (sn, el, i, q, tn) = e.name
12       if sn in N then
13         let (_, _, AssocTyps, _, _) = SIAssocs.elements(el)
14         if aggregation in AssocTyps then
15           resultSet := resultSet union parts(S, G, EG, {tn})
16   step
17     return resultSet
```

Als Parameter bekommt die Funktion *parts* das Klassendiagramm *S*, das Story Pattern *G*, den Wirtsgraphen *EG* und eine Teilemenge *N* von Knoten aus dem Wirtsgraphen. Im Allgemeinen handelt es sich um die Objekte des Wirtsgraphen, die mit den Variablen der Menge *DelE* des Story Patterns *G* gebunden sind. In Zeile 7 wird eine temporäre Menge *tempE* von Links aus dem Wirtsgraphen berechnet. Die Menge *tempE* enthält alle Links des Wirtsgraphen außer denen, die mit der Menge der zu löschenden Links des Story Patterns gebunden sind. In den Zeilen 10 bis 14 wird für jeden Link *e* in *tempE* überprüft, ob er Instanz einer Aggregation ist (Zeile 14). Ist das der Fall, so wird der Kinderknoten *tn* des Knotens *sn* in die Menge *resultSet* eingefügt, die am Ende zurückgegeben wird. In Zeile 15 wird die Menge *parts* rekursiv aufgerufen, um die transitive Hülle der Kinderknoten zu berechnen, die durch eine Aggregationsbeziehung mit Knoten der Menge *tempE* verbunden sind. Schließlich wird in Zeile 17 die Menge *resultSet* zurückgegeben.

Nachdem das Löschen der mit *«destroy»* markierten Variablen des Story Patterns stattgefunden hat, wird als nächstes in der Regel *deletionsFor* die Regel *deleteEdgesFor* aufgerufen.

```
01 deleteEdgesFor(G as Grr, EG as ExtGraph, S as SIGraph) =
02   machine
03     let DE = getDelE(G)
04     let M = getMatch(G)
06     let EE = getExtE(EG)
07     if size(DE.elements)<>0 then
08       EE.elements := EE.elements difference matchSet(DE.elements, EG, M)
09     else
10       skip
11   step
12     if size(DE.elements)<>0 then
         EE.elements := intersectExtEdges(EE, EG, S)
13     else
14       skip
15   step
16     skip
```

Die Regel *deleteEdgesFor* ist verantwortlich für das Löschen der Links, die mit *«destroy»* markiert sind. Durch die Abbildung *M* werden die Kanten des Wirtsgraphen gelöscht, die mit den Elementen der Menge *DelE* des Story Patterns *G* gebunden sind. In Zeile 7 wird überprüft, ob die Menge *DelE* nicht leer ist, in Zeile 8 wird die Menge der Kanten im Wirtsgraphen *ExtE* aktualisiert. Der neue Wert der Menge *ExtE* wird durch Differenz des

alten Wertes von *ExtE* und der Menge der Kanten vom *ExtE*, die durch die Abbildung *M* mit den Elementen der Menge *DelE* gebunden sind, gebildet. Schließlich wird in Zeile 13 eine weitere Aktualisierung der Menge *ExtE* durchgeführt. Es ist möglich, dass in der Menge *ExtE* Kanten existieren, die Knoten verbinden, die in dieser Ausführung des Story Patterns gelöscht worden sind. Diese Kanten müssen ebenfalls gelöscht werden. Deswegen wird der neue Wert der Menge *ExtE* durch den Durchschnitt des alten Wertes und der Menge aller möglichen Kanten in *ExtE* berechnet. Die Menge aller möglichen Kanten in *ExtE* umfasst alle Elemente des kartesischen Produktes *ExtN* x *EL* x *Index* x *AttrValues* x *ExtE*. Die Berechnung des Durchschnitts wird von der Regel *intersectEdges* übernommen.

Um die Konsistenz im Wirtsgraphen zu gewährleisten, müssen für die gelöschten Knotenbezeichnungen auch die entsprechenden Elemente der Abbildung *Nl* gelöscht werden. Aus diesem Grund wird in *deletionsFor* die Regel *deleteNodeLabelsFor* für das Story Pattern *G* aufgerufen.

```
01 deleteNodeLabelsFor(G as Grr, EG as ExtGraph) =
02   let DN = getDelN(G)
03   let M = getMatch(G)
04   let ENl= getNl(EG)
05   if size(DN.elements)<>0 then
06     ENl.elements := { a |-> b | a |-> b in ENl.elements
                        where a notin  matchSet(DN.elements, EG, M) }
07   else
08     skip
```

Die Regel *deleteNodeLabelsFor* löscht die Knotenbezeichnungen der vorher gelöschten Knoten. In Zeile 6 findet eine Aktualisierung der Abbildung *ExtNl* statt. Der neue Wert der Abbildung *ExtNl* enthält nur die Elemente, deren Urbild nicht in der Menge *DelN* enthalten ist.

Das Löschen von Knotenbezeichnungen hat auch zur Folge, dass die Attributwerte der gelöschten Variablen gelöscht werden müssen. Dafür wird die Regel *deleteAttributeValuesFor* aufgerufen.

```
01 deleteAttributeValuesFor(G as Grr, EG as ExtGraph) =
02   let DN = getDelN(G)
03   let M = getMatch(G)
04   let EAv = getAv(EG)
05   if size(DN.elements)<>0 then
06     EAv.elements := { (a, b)|->c | (a, b)|->c in EAv.elements
                        where a notin matchSet(DN.elements, EG, M) }
07   else
08     skip
```

Die Regel *deleteAttributeValuesFor* löscht die Attributwerte der vorher gelöschten Knoten. In Zeile 6 findet eine Aktualisierung der Abbildung *ExtAv* statt. Der neue Wert der Abbildung *ExtAv* enthält nur die Elemente, deren Urbild nicht in der Menge *DelN* enthalten ist.

Die Regel creationFor

Analog zur Regel *deletionFor* wird bei der Regel *creationFor* die Anwendungsstelle des Story Patterns, welches als Parameter zu der Regel gegeben wird, modifiziert, indem neue Elemente hinzugefügt werden. Alle Elemente im Story Pattern, die mit der Markierung *«create»* vorkommen, werden mittels dieser Regel kreiert. Außer diesen Elementen werden auch die Aktualisierungen der Attributwerte aller Variablen berücksichtigt.

```
01 creationFor(G as Grr, EG  as ExtGraph, S  as SIGraph, boundObjects  as
BoundObjects) =
02   machine
03     addNodesFor(G, EG, boundObjects)
04   step
05     addEdgesFor(G, EG, S)
06   step
07     addNodeLabelsFor(G, EG)
08   step
09     updateAttributeValuesFor(G, EG)
10   step
11     skip
```

Nun werden die einzelnen Regeln der Zeilen 3, 5, 7 und 9 näher betrachtet. Die erste Regel, die aufgerufen wird, ist zuständig für das Kreieren neuer Knoten im Wirtsgraphen. Alle Variablen im Story Pattern *G*, die mit *«create»* markiert sind, werden dabei berücksichtigt.

```
01 addNodesFor(G as Grr, EG as ExtGraph, boundObjects as BoundObjects) =
02   machine
03     let AN = getAddN(G)
04     let C = getCopy(G)
05     var Nnew as Set[ExtNode] = {}
06     let EN = getExtN(EG)
07   step
08     if size(AN.elements)<>0 then
09      foreach n in AN.elements do
10        let newn = new ExtNode(asString(NodeIndex))
11        NodeIndex := NodeIndex + 1
12        Nnew := Nnew union {newn}
13        C.param :=  C.param override {n|->newn}
14        boundObjects.elements := boundObjects.elements
                                   override {n|->newn}
15    else
16      skip
17  step
18    if size(AN.elements)<>0 then
19      EN.elements := EN.elements union Nnew
20    else
21      skip
```

Die Regel *addNodesFor* kreiert anhand des Story Patterns *G* neue Knoten im Wirtsgraphen. Diese neuen Knoten sind Kopien der Elemente der Menge *AddN* und werden in die in Zeile 5 deklarierte Menge *Nnew* eingefügt. In Zeile 8 wird überprüft, ob die Menge *AddN* nicht leer ist. <wenn sie nicht leer ist, wird in den Zeilen 9 bis 14 eine `foreach`-Schleife über alle Elemente der Menge *AddN* iteriert. In jeder Iteration wird ein neuer Knoten mit der Identifikation *NodeIndex* kreiert, die ein Integer ist, der in jeder Iteration in Zeile 11 inkrementiert wird. In Zeile 12 findet eine Aktualisierung der Menge N_{new} statt; der neue Wert enthält auch den neu kreierten Knoten new_n. Für diesen neuen Knoten wird auch ein Eintrag in der Abbildung *boundObjects* gemacht, die im Kapitel Teilgraphensuche (4.3.5) näher betrachtet wird. Schließlich wird noch ein Eintrag in der Abbildung *Copy* gemacht, der das Binden des neuen Knoten n_{new} mit dem Knotens *n* des Story Patterns beschreibt. Diese Modifikationen finden in den Zeilen 13 und 14 statt. Nachdem für alle Knoten aus *AddN* ein neuer Knoten im Wirtsgraphen kreiert und in der Menge *Nnew* eingefügt worden ist, wird die Menge *ExtN* in Zeile 19 aktualisiert. Der neue Wert ist eine Vereinigung des alten Wertes und der Menge *Nnew*.

Sind die neuen Objekte im Wirtsgraphen erzeugt, so müssen als nächstes alle Links kreiert werden, die im Story Pattern mit der Markierung *«create»* erscheinen. Dieses realisiert die Regel *addEdgesFor*.

```
01 addEdgesFor(G as Grr, EG as ExtGraph, S as SIGraph) =
02   let AE = getAddE(G)
03   if size(AE.elements)<>0 then
04     machine
05       let EExt = getExtE(EG)
06       let SAssocs = getAssocs(S)
07       let QA = getQualAttrs(EG)
08       let I = getIndex(EG)
09       var Enew = new ExtE({})
10     step
11       foreach e in AE.elements do
12         machine
13           let (sn, el, i, q, tn) = e.name
14           let AssocType = getAssocType(el, S)
15           var qnew as QualAttr
16           var qExists as Boolean = false
17         step
18           if qualified in AssocType then
19             foreach qa in QA.elements do
20               if q.value = qa.value then
21                 qnew := qa
22                 qExists := true
23             if not qExists then
24               QA.elements(q) := true
25               Qnew := q
26           else
27             qnew := extQualAttr
28         step
29           if ordered in AssocTyps then
30             if (i.value = 0) then
31               let EE = getExtE(EG)
32               let maxIndex as Ind = getMaxIndexFrom(EE)
33               maxIndex.value := maxIndex.value + 1
34               Enew.elements := Enew.elements union
       {new ExtEdge(copyMatch(sn, G),el,maxIndex,qnew,copyMatch(tn, G))}
35               I.elements(maxIndex):=true
36             else
37              Enew.elements := Enew.elements union
                {new ExtEdge(copyMatch(sn, G),el,i,qnew,copyMatch(tn, G))}
38              I.elements(i):=true
38           else
40             Enew.elements := Enew.elements union
         {new ExtEdge(copyMatch(sn, G),el,extInd,qnew,copyMatch(tn, G))}
41     step
42       let replace = new ExtE({})
43       foreach e in AE.elements do
44         let (sn, el, i, q, tn) = e.name
45         let (_, srcCard, _, _, tgtCard) = SAssocs.elements(el)
46         if (tgtCard = one or srcCard = one ) then
47           replace.elements := replace.elements union {e}
48         skip
49     step
50       EExt.elements := EExt.elements difference replace.elements
51     step
52       EExt.elements := EExt.elements union Enew.elements
52   else
54     skip
```

Die Regel *addEdgesFor* kreiert anhand des Story Patterns *G* neue Kanten im Wirtsgraphen. Diese neuen Kanten sind Kopien der Kanten der Menge *AddE* des Story Patterns *G*. In Zeile 9 wird die Menge E_{new} deklariert, in der alle neue Kanten hinzugefügt werden. In Zeile 11 beginnt eine `foreach`-Schleife, die über alle Elemente der Menge *AddE* iteriert. In jeder Iteration wird eine neue Kante kreiert und in die Menge N_{new} eingefügt. Dabei wird überprüft, ob die Assoziation, deren Instanz die aktuelle Kante aus *AddE* ist, qualifiziert oder geordnet ist. Ist die Assoziation qualifiziert (Zeile 18), so wird überprüft, ob der Wert *q* des Links(Zeile 13) schon in der Menge *QualAttrs* des Wirtsgraphen existiert. Ist das der Fall, wird der neue Link den qualifizierten Wert bekommen, der im Wirtsgraphen enthalten ist. Ansonsten wird der neue Wert des Links des Story Patterns in die Menge *QualAttrs* des Wirtsgraphen hinzugefügt. Ist die Assoziation geordnet oder sortiert, so muss ein Index berechnet werden, damit der neue Link die korrekte Position in der Liste der Nachbarn, die schon im Wirtsgraphen existieren, bekommt. Falls der Index *i* des Links *e* (Zeile 13) den Wert Null hat, wird in Zeile 32 und 33 das Maximum aller Indizes der Links des Wirtsgraphen, die Instanzen der gleichen Assoziation sind, gebildet und inkrementiert. Wenn der Index ungleich Null ist, wird er für den neuen Link übernommen. Wenn der neu kreierte Link eine Instanz einer to-many Assoziation ist, gibt es nichts Weiteres zu beachten. Ist aber die Assoziation eine to-one, dann muss der Fall betrachtet werden, dass der neue Link ein Objekt verbindet, das schon über diese Assoziation mit einem anderen verbunden ist. In diesem Fall wird der neue Link den alten ersetzen, d.h. der neue Link wird kreiert und der alte wird gelöscht. In Zeile 50 findet eine Aktualisierung der Menge der Kanten des Wirtsgraphen *ExtE* statt, indem vom alten Wert der Menge alle Kanten entfernt werden, die die Funktion *replace* berechnet. Die Funktion *replace* berechnet alle Kanten im Wirtsgraphen, die durch neue ersetzt werden. In Zeile 52 wird dann schließlich die Menge der Kanten aktualisiert. Der neue Wert beinhaltet die neuen Kanten, die durch die Anwendung des Story Patterns *G* kreiert werden.

Nachdem die neuen Links im Wirtsgraphen erzeugt worden sind, muss für die Konsistenz der Abbildung *Nl* gesorgt werden. Dazu werden mittels der Regel *addNodeLabelsFor* in die Abbildung *Nl* des Wirtsgraphen Einträge für die neuen Objekte hinzugefügt.

```
01 addNodeLabelsFor(G as Grr, EG as ExtGraph) =
02   let addN = getAddN(G)
03   if size(addN.elements)<>0 then
04     machine
05       let C = getCopy(G)
06       let RGGrr = getRG(G)
07       let RGNl = getNl(RGGrr)
08       var ENl = getNl(EG)
09     step
10       foreach n in addN.elements do
11         let nodeInExt = C.param(n)
12         let nodeInSI = RGNl.elements(n)
13         ENl.elements := ENl.elements override {nodeInExt |-> nodeInSI}
14   else
15     skip
```

Die Regel *addNodeLabelsFor* kreiert anhand des Story Patterns *G* neue Knotenbezeichnungen im Wirtsgraphen, d.h. Elemente der Abbildung *ExtNl*. In Zeile 13 findet eine Aktualisierung *ExtNl* statt. Der neue Wert der Abbildung *ExtNl* enthält auch die Bezeichnungen der neu kreierten Knoten.

Schließlich werden die Attribute der neuen Objekte in die Abbildung *AttributeValues* des Wirtsgraphen einzugefügt und die Attribute der unveränderten Variablen, d.h. der

Knotenbezeichnungen der Menge *CoreN* des Story Patterns *G*, aktualisiert. Dies geschieht mit der Ausführung der Regel *updateAttributeValuesFor* für das Story Pattern *G*.

```
01 updateAttributeValuesFor(G as Grr, EG as ExtGraph) =
02  let addN = getAddN(G)
03  if size(addN.elements)<>0 then
04     machine
05       let C = getCopy(G)
06       let RGGrr = getRG(G)
07       let RGAv = getAv(RGGrr)
08       var EAv = getAv(EG)
09     step
10       foreach n in addN.elements do
11         let nodeInExt = C.param(n)
12         foreach (a, b) in dom(RGAv.elements) do
13           if ( a = n ) then
14             EAv.elements := EAv.elements override
                              {(nodeInExt, b) |-> RGAv.elements((a,b))}
15   else
16     skip
17   let coreN = getCoreN(G)
18   let M = getMatch(G)
19   if size(coreN.elements)<>0 then
20     machine
21       let RGGrr = getRG(G)
22       let RGAv = getAv(RGGrr)
23       var EAv = getAv(EG)
24     step
25       foreach n in coreN.elements do
26         foreach (a, b) in dom(RGAv.elements) do
27           if ( a = n ) then
28             var attrValueInEG = EAv.elements((M.paramN(n), b))
29             let attrValueInRG = RGAv.elements(a,b)
30             if attrValueInEG = undef then
31               EAv.elements := EAv.elements override
                                 {(M.paramN(n), b) |-> RGAv.elements((a,b))}
32            else
33               if  attrValueInEG.value <> attrValueInRG.value then
34                 attrValueInEG.value := attrValueInRG.value
35   else
36     skip
```

Die Regel *updateAttributeValuesFor* aktualisiert anhand des Story Patterns *G* die Attribute sowohl der alten als auch der neuen Knoten des Wirtsgraphen. In den Zeilen 25 und 26 wird für alle Knoten in der Menge *CodeN* des Story Patterns *G* überprüft, ob Initialisierungen oder Aktualisierungen der Attributwerte stattfinden. Falls das der Fall für einen Knoten n in *CoreN* ist wird in Zeile 30 überprüft, ob der mit n gebundene Knoten im Wirtsgraphen schon einen Wert hat. Wird der Wert des Attributes für diesen Knoten initialisiert, so wird die Abbildung *ExtAv* in Zeile 31 mit Hilfe der Abbildung *M* um diesen Wert erweitert. Handelt es sich um die Aktualisierung des Wertes des Attributes, so wird in Zeile 34 dieser Wert durch den neuen aktualisiert.

4.3.5 Teilgraphensuche

Um ein Story Pattern auszuführen ist es wie schon erwähnt notwendig, dafür eine Anwendungsstelle in der Laufzeitobjektstruktur zu finden. Diese Aufgabe übernimmt die Regel *findMatchFor*. Nachdem diese Regel erfolgreich ausgeführt worden ist, kann ein Teil

der Laufzeitobjektstruktur anhand des Story Patterns modifiziert werden. Das Problem besteht also darin, eine Anwendungsstelle für die Modifikationen zu finden. Diese Anwendungsstelle ist ein Teilgraph des Wirtsgraphen, dem die Laufzeitobjektstruktur entspricht, und soll isomorph zu dem linken Graphen *LG* des Story Patterns sein. Die Eigenschaften der Isomorphe werden in [13] beschrieben. Das Problem der Teilgraphensuche ist in der Graphentheorie unter dem Namen „Subgraph Isomorphism Problem" bekannt und gehört in die Menge der NP vollständigen Probleme [21]. Da das Ziel dieser Arbeit auch die Generierung von ausführbaren ASM Spezifikationen ist, ist ein Einsatz erwünscht, der in der Praxis keinen exponentiellen Aufwand hat. Aus diesem Grund spielt ein optimierter Ansatz für die Teilgraphensuche eine wichtige Rolle. In der Praxis kann der Aufwand der Teilgraphensuche in den meisten Fällen auf polynomielle Zeit und oft sogar auf lineare oder konstante Zeit reduziert werden. Solche Optimierungsalgorithmen [22, 23] wurden in PROGRESS, dem Vorgänger von Fujaba, verwendet. Die Berechnung eines Algorithmus, der einen Teilgraphen eines Graphen anhand bestimmter Bedingungen sucht, wird durch einen Suchbaum repräsentiert, in dem jeder Knoten ein Schritt des Algorithmus ist. In jeder Ebene des Baumes wird eine Variable mit einem Objekt des Teilgraphen gebunden.

Ein naiver Algorithmus könnte wie folgt aussehen:

```
01 foreach n1 in ExtN.elements where nl(n1) = classenname do
02   foreach n2 in ExtN.elements where nl(n2) = classenname do
03     foreach n3 in ExtN.elements where nl(n3) = classenname do
04       if (exist link linkName between n1 and n2) then
05         if (exist link linkname between n2 and n3) then
06           //initialize Match structure for this Story Pattern
```

Obiger Algorithmus hat im schlimmsten Fall exponentielle Laufzeit in Bezug auf die Anzahl der Objekte des Story Patterns. Es gibt verschiedene Möglichkeiten diese Laufzeit zu verbessern. Eine mögliche Optimierung wäre, die Gültigkeit der if-Abfragen für die Links möglichst früh zu überprüfen. So würde obiger Algorithmus wie folgt aussehen:

```
01 foreach n1 in ExtN.elements where nl(n1) = classenname do
02   foreach n2 in ExtN.elements where nl(n2) = classenname do
03     if (exist link linkName between n1 and n2) then
04       foreach n3 in ExtN.elements where nl(n3) = classenname do
05         if (exist link linkname between n2 and n3) then
06           //initialize Match structure for this Story Pattern
```

Durch das Verschieben der if-Abfrage ist es jetzt nicht mehr nötig zu versuchen, die Variable n_3 zu binden, falls keine passenden Objekte in der Objektstruktur für n1 und n_2 existieren oder n_1 und n_2 nicht mit dem Link *linkName* verbunden sind.

Lösungen

Eine Möglichkeit, die Regel *findMatchFor* zu realisieren, ist diese statisch zu implementieren. Das bedeutet einen Algorithmus zu entwerfen, der als Eingabe den linken Graphen eines Story Patterns bekommt und als Ausgabe eine Teilmenge des Wirtsgraphen gibt, die isomorph zu der Eingabe ist. So ein Einsatz hat sich als aufwendig erwiesen, da die Konstrukte der ASM Sprache nicht mächtig genug dafür sind. Eine weitere Möglichkeit wäre, das Suchen des Teilgraphen des Wirtsgraphs durch nichtdeterministische Entscheidungen zu realisieren. Wie im Kapitel 3.2.10 beschrieben bietet ASM mittels des Terms *choose* die Möglichkeit, nichtdeterministische Entscheidungen zu treffen. Doch als Bedingung des *choose* Terms sollte wiederum der Isomorphismus der zwei Graphen definiert werden, und das würde das Problem nicht reduzieren.

Der hier ausgewählte Einsatz ist die Generierung der Regel *findMatchFor*, wobei für jedes einzelne Story Pattern die Reihenfolge der zu bindenden Elementen vordefiniert wird. Das Prinzip, das dabei verwendet wird, ist das „Force-Failure-Prinzip“ [15], das auch in der Teilgraphensuche in Fujaba benutzt wird. Dabei wird versucht, beim Binden der Variablen des linken Graphs eines Story Patterns mit Objekten des Wirtsgraphen möglichst schnell einen Fehler festzustellen. So kann die Suche möglichst früh abgebrochen werden. Ein Unterschied zum Einsatz in Fujaba ist, dass hier nur der linke Graph des Story Patterns betrachtet wird und nicht das ganze Story Pattern. Grund dieses Unterschiedes ist, dass die Regel *findMatchFor* als Ziel hat, eine Anwendungsstelle für das Story Pattern zu finden und nicht das Story Pattern auszuführen. In Fujaba wird durch den generierten Code die Anwendungsstelle gesucht und zugleich das Story Pattern ausgeführt. Durch das Generieren der Regel *findMatchFor* wird das Finden einer günstigen Reihenfolge dem Generator überlassen und ist nicht ein Teil der Aufgabe der Regel selbst, wie bei den zwei anderen schon erwähnten Einsätzen.

Die Generierung der Regel *findMatchFor* hat den Vorteil, dass der Aufwand der Berechnung, die die Reihenfolge des Bindens der Variablen des linken Graphen mit Objekten des Wirtsgraphen bestimmt, vom Generator übernommen wird. Für jedes Story Pattern wird eine möglichst optimierte Reihenfolge berechnet und als Teil der Regel *findMatchFor* hinzugefügt wird. Nachteil dieses Einsatzes ist, dass für jedes Modell die Regel *findMatchFor* anders generiert wird. Bei dem Beweis, dass die Modell Spezifikation äquivalent mit der ASM Spezifikation ist, sollte der Generierungsprozess mitbeachtet werden.

Generierung der Regel *findMatchFor*

Das Generieren der Regel *findMatchFor* erfolgt mittels des „Force-Failure-Prinzips“. Beim Force-Failure Ansatz wird versucht, möglichst früh Fehler beim Binden von Objekten des Wirtsgraphen an Variablen des linken Graphen eines Story Patterns zu erkennen. Tritt ein Fehler auf, kann die Suche abgebrochen werden, da das Story Pattern nicht mehr ausgeführt werden kann. Ein Fehler tritt auf, wenn kein passendes Objekt im Wirtsgraphen gefunden werden kann. Der gesuchte Teilgraph wird sukzessiv über bereits gebundenen Variablen aufgebaut. Das bedeutet, dass in jedem linken Graphen eines Story Patterns mindestens eine gebundene Variable existieren muss. In den meisten Fällen ist es die *this* Variable, also das Objekt, auf das die Methode aufgerufen worden ist, deren Aktion dieses Story Pattern ist. Die Realisierung des Force-Failure Ansatzes erfolgt mittels Prioritäten. Zu jedem Link im linken Graphen eines Story Patterns wird eine Priorität gesetzt, die die Abarbeitungsreihenfolge des Links bezogen auf seine Kosten bestimmt. So wird ein Link mit hoher Priorität gegenüber einem mit niedrigerer Priorität zuerst abgearbeitet. Die Klassifizierung der Links wird vom Generator übernommen.

Die Regel *findMatchFor* hat die folgende Struktur:

```
01 findMatchFor(G as Grr, EG as ExtGraph, thisObject as ExtNode,
     boundObjects as BoundObjects, returnValue as Value,
     params as Parameters) as Boolean =
02   machine
03     let grrName = getGrrName(G)
04     let LG = getLG(G)
05     let LGNodes = getExtN(LG)
06     let LGEdges = getExtE(LG)
07     let LGNl = getNl(LG)
08     let ExtNodes = getExtN(EG)
```

```
09      let ExtEdges = getExtE(EG)
10      let ExtMap = unQual(ExtEdges)
11      let ExtNl = getNl(EG)
12      let GrrMatch = getMatch(G)
13      var result as Boolean = false
14    step
15      …
16    step
17      return result
```

Als Parameter bekommt diese Regel das Story Pattern *G*, für das die Teilgraphensuche erfolgen soll, den Wirtsgraphen *EG*, das *this* Objekt, welches von der Methode bestimmt wird, durch die das Story Pattern *G* ausgeführt wird, die Menge der gebundenen Variablen *boundObjects*, die alle gebundenen Variablen von früheren Ausführungen der Story Patterns enthält, den Wert der *return* Variable, und die Parameter der Methode *params*, die dieses Story Pattern enthält. Als Rückgabe wird ein `Boolean` gegeben, der das Ergebnis der Suche repräsentiert: `true`, wenn die Suche erfolgreich war, und `false`, wenn kein Teilgraph des Wirtsgraphen gefunden worden ist, der isomorph zum linken Graphen *LG* des Story Patterns *G* ist. In den Zeilen 3 bis 12 werden die Mengen definiert, die für die Suche gebraucht werden. Dabei wird in Zeile 10 die Abbildung *ExtMap* durch die Funktion *unQual* über der Menge *ExtEdges* (Menge aller Kantenbezeichnungen) definiert.

Die Suche eines Teilgraphen in einem objektorientierten Graphen erfordert neben der Bedingung, dass die Menge der Kanten des Teilgraphen eine Teilmenge der Menge der Kanten des Wirtsgraphen ist, noch weitere Bedingungen. Es kann vorkommen, dass versucht wird, einen Link des Graphen *LG* mit einem geordneten oder qualifizierten Link des Wirtsgraphen zu binden, obwohl der Link in *LG* keinen Wert oder Index hat. So ein Link soll mit einem Link des Wirtsgraphen gebunden werden, falls beide Instanzen der gleichen Assoziation sind. Um das möglich zu machen, werden im Wirtsgraphen Teilmengen gebildet, ohne dass die Kennzeichner oder Indizes der Links beachtet werden. Um dies zu realisieren, wird in Zeile 10 die Abbildung *ExtMap* gebildet. Dabei wird die Funktion *unQual* benutzt, die wie folgt aussieht:

```
01 unQual(E as ExtE) as ExtNode * Edge * ExtNode -> ExtEdge =
02   machine
03     var tempSet as ExtNode * Edge * ExtNode -> ExtEdge = {|->}
04   step
05     foreach e in E.elements do
06       let (sNode, edge, _, _, tNode) = e.name
07       tempSet:=tempSet override {(sNode, edge, tNode) |-> e}
08   step
09     return tempSet
```

In Zeile 1 wird als Rückgabetyp eine Abbildung deklariert. Diese wird in Zeile 3 als *tempSet* mit der leeren Abbildung initialisiert. In Zeile 5 iteriert die `foreach`-Schleife über alle Kanten des Wirtsgraphen. Für jede Kante wird ein Eintrag in die zurückzugebende Abbildung eingefügt. In Zeile 6 wird ein Pattern benutzt, um auf die Eigenschaften des aktuellen Links zuzugreifen. Von den Eigenschaften des Links werden die zwei Objekte *sNode* und *tNode*, die der Link miteinander verbindet, und die Assoziationsbezeichnung *edge*, deren Instanz der Link ist, ausgewählt. Aus diesen drei Elementen wird ein Tupel gebildet, das auf die aktuelle Kante *e* in Zeile 7 abgebildet wird. Die Abbildung des Tupels auf die Kante wird dann in die Abbildung *tempSet* eingefügt. Schließlich wird *tempSet* von der Funktion zurückgegeben. In Zeile 13 der Struktur der Regel *findMatchFor* wird die boolesche Variable *result* deklariert und mit dem Wert `false` initialisiert. Diese Variable ändert ihren Wert nur auf `true`, wenn

die Teilgraphensuche erfolgreich ist. Der folgende Code beschreibt näher, wie der Code der Teilgraphensuche des einzelnen Story Patterns in der Regel *findMatchFor* angebunden wird.

```
12 …
13     var result as Boolean = false
14   step
15     match grrName with
16       "executeevent1_0" : machine
17                             // AsmL für die Teilgraphensuche
18                             // …
19       "executeevent2_1" : machine
20                             // AsmL für die Teilgraphensuche
21                             // …
22       "init_2" : machine
23                    // AsmL für die Teilgraphensuche
24                    // …
25              …
25       _ : skip
26   step
27     return result
```

In Zeile 15 wird anhand des Namens des Story Patterns *G*, welches in Zeile 3 als *grrName* deklariert wurde, der Code für die Teilgraphensuche von *G* ausgewählt. Der `match`-Ausdruck sucht in den ab Zeile 16 folgenden Namen nach dem Namen des Story Patterns. Wird der Name gefunden, so wird der Code für die Teilgraphensuche des jeweiligen Story Patterns ausgeführt. Existiert der Name des Story Patterns nicht, so wird der `skip`-Befehl der Zeile 25 ausgeführt, und die Teilgraphensuche schlägt fehl.

In den folgenden Paragraphen wird der Code für die Teilgraphensuche der Story Patterns näher untersucht. Zunächst werden die normalen Links betrachtet, also die Links im linken Graphen *LG* des Story Patterns *G*, die eine gebundene Variable mit einer ungebundenen verbinden. Gebundene Variablen werden im gleichnamigen Paragraphen beschrieben. Als nächstes werden die Check-Links betrachtet, die zwei gebundene Variablen verbinden. Weiterhin wird die Semantik der Story Patterns um neue Elemente erweitert. Dabei werden die gebundenen Variablen, Attributbedingungen, die negativen Knoten und negativen Links und optionalen Links eingeführt. Auf die mengenwertige Variablen, sowie auch die Multilinks wird in dieser Arbeit nicht eingegangen. Mengenwertige Variablen können die Multi-Aktivitäten simulieren, die in Kapitel 4.3.7 betrachtet werden.

In den folgenden Abschnitten werden Teile von generierten AsmL Codes präsentiert. Die vollständigen Codes sind im Anhang zu finden.

Normale Links

Die normalen Links sind die Links in einem Story Pattern, die potentiell eine neue Variable an ein Objekt binden. Diese neue Variable kann eventuell über eine zu-1 Assoziation oder eine zu-n Assoziation erreicht werden. Im „Force-Failure-Prinzip“ werden vom Generator die Links bevorzugt, die Instanzen einer zu-1 Assoziation sind. Im Fall der Generierung der Regel *findMatchFor* spielt die Kardinalität der Assoziationen für die Teilgraphensuche keine Rolle, da die Nachbarobjekte nicht direkt über eine *get*-Methode oder einen Iterator zu finden sind, sondern für einen Link alle Nachbarn überprüft werden.

In Abbildung 4.19 wird die Methode *dockToStation* dargestellt. Im ersten Story Pattern wird der Zustand des Shuttles überprüft. Falls dieser *MovingToNextStation* ist, wird er gelöscht und der neue Zustand *DockedToStation* kreiert. Ist die Ausführung dieses Story Patterns

erfolgreich, so wird im zweiten Story Pattern der *goesTo* Link zwischen dem Shuttle und der Station gelöscht und ein neuer Link *isIn* kreiert. Dieser beschreibt, dass das Shuttle nicht mehr zur Station fährt sondern dort angekommen ist. Ist auch dieses Story Pattern erfolgreich, so wird an die Station ein neues Event *CheckShuttle* gehängt.

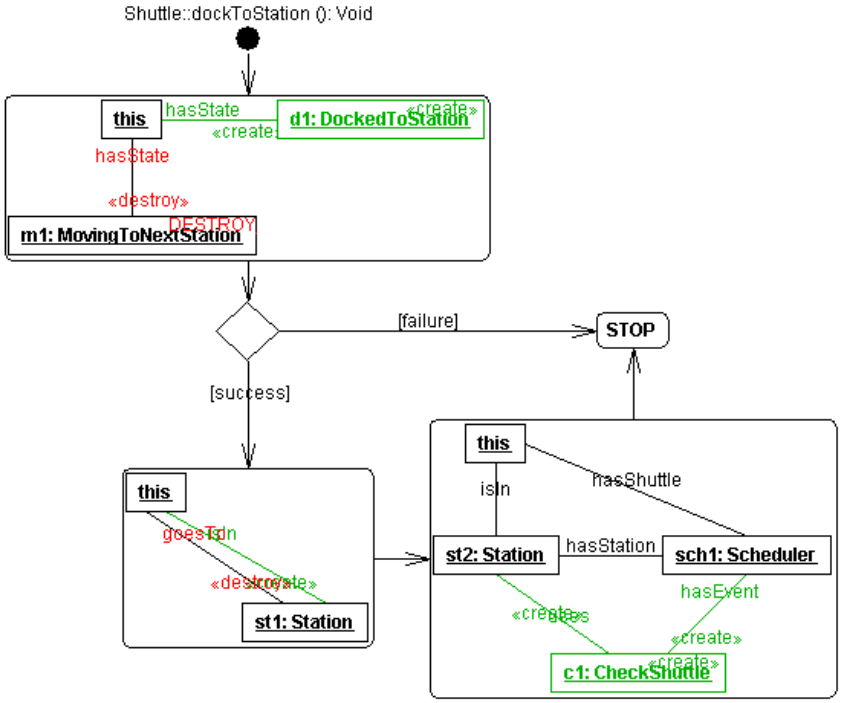

Abbildung 4.19: Die Methode *dockToStation*

Der folgende AsmL Code entspricht dem Teil der Regel *findMatchFor*, der für das Finden der Anwendungsstelle des ersten Story Patterns sorgt. Nachdem durch den *match*-Ausdruck der Name des Story Patterns (hier *dockToStation1*) gefunden ist, wird der folgende Code ausgeführt:

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                          dockToStation1_4LGNode0 |-> thisObject }
03 step
04   //trying to bind m1 with dockToStation1_4LGNode1
05  let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                              dockToStation1_4LGExtEdge0)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     //match found
11     result := true
12     boundObjects.elements := boundObjects.elements override {
                                   dockToStation1_4LGNode1 |-> n0}
13     GrrMatch.paramN := { dockToStation1_4LGNode0 |-> thisObject,
                                     dockToStation1_4LGNode1 |-> n0}
14     GrrMatch.paramE := { dockToStation1_4LGExtEdge0 |-> thisObject,
                            ExtMap((thisObject, getELabelFrom(
                            dockToStation1_4LGExtEdge0 ), n0))}
```

In Zeile 2 wird das *this* Objekt in einer Struktur gespeichert, die alle gebundenen Elemente enthält. Diese Struktur wird später im Paragraphen „Gebundene Variablen“ näher betrachtet.

In Zeile 5 wird versucht, die Variable m_1 des Story Patterns G mit einem Objekt des Wirtsgraphen zu binden. Dies geschieht durch das Aufrufen der Funktion *getMatchObjectTo*. Diese Funktion bekommt als Parameter das Story Pattern *G*, den Wirtsgraphen *EG*, die gebundene Variable, über die die neue Variable gesucht wird, und den Link, der beide verbindet. In diesem Beispiel ist die *this* Variable die gebundene Variable und *dockToStation1_4LGExtEdge0* der Link, der die *this* Variable und die Variable w_1 verbindet. Die Rückgabe der Funktion *getMatchObjectTo* ist ein 3-Tupel, welches das Objekt des Wirtsgraphen, das mit der ungebundenen Variable gebunden werden kann, den Link des Wirtsgraphen, der die zwei Objekte verbindet, und die Assoziation, deren Instanz der Link ist, enthält. In diesem Beispiel ist n_0 das Objekt des Wirtsgraphen, e_0 der Link, welcher das Objekt, das mit der Variable *this* gebunden ist, und das Objekt n_0 verbindet, und $edge_0$ die Assoziation, deren Instanz e_0 ist. In Zeile 6 wird überprüft, ob das Objekt n_0 existiert oder nicht. Ist n_0 gleich *undef*, so bedeutet das, dass kein Objekt im Wirtsgraphen existiert, das mit der Variable m_1 gebunden werden kann. Ist n_0 ungleich *undef*, so ist die Teilgraphensuche erfolgreich. Als nächstes wird die Struktur *boundObjects* aktualisiert. Die Variable m_1 ist nun gebunden, also wird ein Eintrag für sie eingefügt. Schließlich wird die Abbildung *Match* des Story Patterns *G* initialisiert. Diese wird dann für die Ausführung des Story Patterns verwendet. In Zeile 13 werden die neu gebundenen Variablen und deren gebundene Objekte aus dem Wirtsgraphen eingefügt. In Zeile 14 wird das Gleiche auch für die Links gemacht. Dabei wird die Abbildung *ExtMap* benutzt, um den Link aus dem Wirtsgraphen zu bekommen, der mit dem Link des Story Patterns gebunden ist.

Gibt es in dem Story Pattern Constraints, d.h. Bedingungen die zwangsmäßig gelten müssen, damit das Story Pattern ausgeführt werden kann, so werden die überprüft, nachdem für alle Variablen des Story Patterns ein Objekt aus dem Wirtsgraphen gefunden worden ist. Da diese Bedingungen in Form von reinem Java Code definiert werden, werden sie nur teilweise unterstützt. Wie Bedingungen in ASM abgebildet werden, wird im Paragraphen „Attributbedingungen“ näher erläutert.

Enthält eine Variable eine Attributzuweisung, d.h. wird zum Attribut einer Variable im Story Pattern ein Wert zugewiesen, so findet diese Aktualisierung statt, nachdem für alle Variablen des Story Patterns ein Objekt aus dem Wirtsgraphen gefunden worden ist. Ein Beispiel für diesen Fall ist im Kapitel 4.3.8 zu finden.

Folgender Code beschreibt die Funktion *getMatchObjectTo*. Da die Richtung der Links eine wichtige Rolle spielt, gibt es auch eine Funktion *getMatchObjectFrom*, die analog zu der folgenden ist:

```
01 getMatchObjectTo(G as Grr, EG as ExtGraph, n as ExtNode, e as ExtEdge)
                    as ExtNode * ExtEdge * Edge =
02   let LG = getLG(G)
03   let ExtNodes = getExtN(EG)
04   let ExtEdges = getExtE(EG)
05   let ExtMap = unQual(ExtEdges)
06   let (_, edge1, i, q, _) = e.name
07   machine
08     var tempSet as Set[ExtNode * Edge * ExtNode] = {}
09   step
10     foreach n2 in ExtNodes.elements where (n, edge1, n2) in dom(ExtMap)
                                                                    do
11       machine
12         var accept as Boolean = false
13       step
14         if q.value <> "" then
```

```
15          let q2 as QualAttr = getQualAttrFrom(ExtMap((n, edge1, n2)))
16          if q2.value = "" then
17            skip
18          else
19            if q.value=q2.value then
20              accept := true
21        else
22          accept:=true
23      step
24        if accept=true then
25          tempSet((n, edge1, n2)) :=true
26  step
27    if tempSet = {} then
28      return (undef, undef, undef)
29    else
30      let (m1, e1, m2) = select(tempSet)
31      return (m2, ExtMap((m1, e1, m2)), e1)
```

In den Zeilen 2 bis 5 werden Mengen und Abbildungen dargestellt, die für den Ablauf der Funktion wichtig sind. Dabei wird in Zeile 5 auch die Funktion *unQual* benutzt, die im vorherigen Paragraphen beschrieben wurde. In Zeile 6 wird ein Pattern benutzt, um die Eigenschaften des Links *e* in den Konstanten $edge_1$, *i* und *q* zuzuweisen. Dabei ist $edge_1$ die Assoziation deren Instanz der Link *e* ist, *i* der Index des Links und *q* der Wert des qualifizierten Attributs des Links. In Zeile 8 wird die Menge *tempSet* deklariert und mit der leeren Menge initialisiert. Die Menge *tempSet* wird schließlich von der Funktion zurückgegeben. In Zeile 10 werden über eine `for`-Schleife alle Objekte betrachtet, die Nachbarn des als Parameter gegebenen Objekts über einen Link sind, der Instanz der gleichen Assoziation ist, wie der Link, der als Parameter zu der Funktion gegeben worden ist. Ist die Assoziation qualifiziert, so wird überprüft, ob der Wert q_1 des Links *e* gleich dem Wert q_2 des Links ist, der das Objekt *n* mit dem in dieser Iteration gefundenen Objekt n_2 verbindet. Ist das der Fall, so werden die zwei Objekte aus dem Wirtsgraphen und die Assoziation, deren Instanz die Links sind, in die Menge *tempSet* eingefügt (Zeile 25). Ist die Assoziation nicht qualifiziert, so wird für jedes Objekt n_2, welches mit dem Objekt *n* durch den geeigneten Link verbunden ist, ein Eintrag in *tempSet* eingefügt. Nachdem alle Nachbarn des Objektes *n* betrachtet worden sind, wird die Rückgabe vorbereitet. Gibt es keine Nachbarknoten von *n*, die über den Link *e* mit *n* verbunden werden können, so wird das 3-Tupel *(undef, undef, undef)* zurückgegeben (Zeile 28). Existieren Einträge in der Menge *tempSet*, so wird durch die *select* Funktion in Zeile 30 nichdeterministisch ein Element ausgewählt und für diesen das entsprechende Tupel zurückgegeben.

Check-Links

Die Check-Links sind Links in einem Story Pattern, die zwei gebundene Variablen verbinden. Da nach keinen Objekten im Wirtsgraphen gesucht wird, sind sie die kostengünstigsten von allen Links. In Abbildung 4.19, die die Methode *dockToStation* darstellt, enthält das letzte Story Pattern (Kreieren des Objektes c_1) einen Check-Link; doch es steht nicht von Anfang an fest, welcher Link der Check-Link ist. Die Links *isIn* und *hasShutlle* werden zuerst betrachtet, und da sie die gleiche Priorität haben, wird eins von beiden per Zufall ausgewählt. Sei es in diesem Fall der Link *isIn*. Als nächstes werden die Links *hasStation* und *hasShuttle* betrachtet, die auch dieselbe Priorität haben. Wieder wird ein Link von beiden per Zufall ausgewählt. Sei es der Link *hasStation*. Nun sind, falls die Teilgraphensuche soweit erfolgreich war, die Variablen *this*, st_1 und sch_1 gebunden. Was noch überprüft werden muss, ist der Link *hasShuttle*. Da *this* und sch_1 gebunden sind, ist er ein Check-Link.

Folgender AsmL Code beschreibt den Teil der Regel *findMatchFor*, der für das Finden der Anwendungsstelle des letzten Story Patterns sorgt:

```
01 machine
02   …
03   // Binden von st2 mit n0
04     // Binden von sch1 mit n1
05       //check link: hasShuttle
06       var checkResult as Boolean = checkLink(EG, n1,
                              dockToStation3_1LGExtEdge1, thisObject)
07       if checkResult = false then
08         // no match found
09         skip
10       else
11         //match found
12         result := true
13         // Aktualisierung von boundObjects
14         // Initialisierung von Match
```

Die Variablen st_2 und sch_1 werden als normale Variablen im Wirtsgraphen gebunden (Zeilen 4 und 5). Falls sie erfolgreich gebunden werden, bleibt noch die Überprüfung des Links *hasShuttle*. In Zeile 7 wird die Funktion *checkLink* aufgerufen, die für zwei Objekte aus dem Wirtsgraphen und einen Link aus dem Story Pattern überprüft, ob es einen Link im Wirtsgraphen gibt, der mit dem als Parameter gegebenen Link gebunden werden kann. In diesem Fall werden das Objekt n_1, das *this* Objekt und der Link *dockToStation3_1LGExtEdge1* als Parameter gegeben. Die Funktion *checkLink* gibt einen *Boolean* als Parameter. Ist der Rückgabewert `true`, so existiert so ein Link im Wirtsgraphen und die Teilgraphensuche ist erfolgreich. Ist das Ergebnis `false`, so bricht die Teilgraphensuche ab. Wenn die Teilgraphensuche erfolgreich ist, werden als nächstes die Strukturen *boundObjects* und *Match* aktualisiert.

Folgender AsmL Code beschreibt die Funktion *checkLink*:

```
01 checkLink(EG as ExtGraph, n1 as ExtNode, e as ExtEdge, n2 as ExtNode) as
                                                                  Boolean =
02   let ExtEdges = getExtE(EG)
03   let ExtMap = unQual(ExtEdges)
04   let (_, edge1, i, q, _) = e.name
05   machine
06     var result as Boolean = false
07   step
08     if ((n1, edge1, n2) in dom(ExtMap)) then
09       machine
10         var accept as Boolean = false
11       step
12         if q.value <> "" then
13           let q2 as QualAttr = getQualAttrFrom(ExtMap((n1, edge1, n2)))
14           if q2.value = "" then
15             skip
16           else
17             if q.value=q2.value then
18               accept := true
19         else
20           accept:=true
21       step
22         if accept=true then
23           result := true
24     else
25   step
```

```
12      return result
```

Ähnlich wie bei der Funktion *getMatchObjectTo* werden auch hier zunächst Menge und Abbildungen deklariert. Im Gegensatz zu *getMatchObjectTo* müssen hier aber nicht alle Nachbarn betrachtet werden, sondern es muss nur überprüft werden, ob ein Objekt n_2 existiert, das durch einen Link mit dem Objekt n_1 in Verbindung steht. Der Link soll dabei Instanz der gleichen Assoziation sein, deren Instanz auch der Link e ist. Existiert so ein Nachbar, wird `true` zurückgegeben, andernfalls wird `false` zurückgegeben.

Gebundene Variablen

Es kommt öfter vor, dass die Strukturmodifikationen, die in einem Story Pattern beschrieben werden müssen, zu komplex sind. Zur Vereinfachung eines komplexen Story Patterns werden mehrere Story Patterns gebraucht, die einfache Teile dieser komplexen Modifikation beschreiben. Die einzelnen Story Patterns werden dann in ein gemeinsames Story Diagramm eingebunden. Durch das Beschreiben einer Modifikation durch mehrere Story Patterns entsteht ein Problem: Sei *TG* der Teilgraph des Wirtsgraphen, der die Anwendungsstelle des ersten Story Patterns ist. Normalerweise sollten die weiteren Story Patterns, die Teile der gleichen Modifikation sind, den gleichen Teilgraphen *TG* wie der erste als Anwendungsstelle haben. Das ist aber nicht immer garantiert.

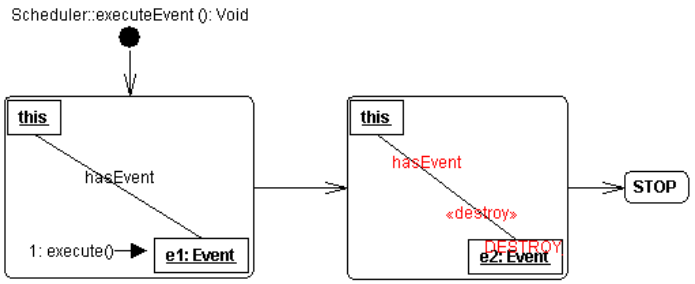

Abbildung 4.20: Die Methode *executeEvent*

In Abbildung 4.20 wird die Methode *executeEvent* dargestellt. Wenn die Methode aufgerufen wird, überprüft sie, ob ein Objekt des Typs *Event* mit dem Scheduler verbunden ist. Falls das der Fall ist, wird dieses Event ausgeführt. Andernfalls passiert nichts. Nach der Ausführung des Events soll dieses vom Scheduler gelöscht werden. Im zweiten Story Pattern der Methode *executeEvent* wird nach einem Objekt des Typs Event im Wirtsgraphen gesucht, das mit dem Objekt, das mit der Variable *this* gebunden ist, verbunden ist. Es wird aber nicht garantiert, dass das gefundene Objekt auch das gleiche ist wie im ersten Story Pattern. In Abbildung 4.21 werden zwei Teilgraphen des Wirtsgraphen betrachtet. Der linke Graph repräsentiert einen Teil des Wirtsgraphen vor der Ausführung der Methode *executeEvent* (Zustand S_i) und der rechte Teilgraph einen Teil des Wirtsgraphen danach (Zustand S_{i+1}). Es wird angenommen, dass beim Ausführen des ersten Story Patterns der Methode das Objekt ev_1 mit der Variable e_1 gebunden wird und die Methode *execute* auf das Objekt e_1 ausgeführt wird.

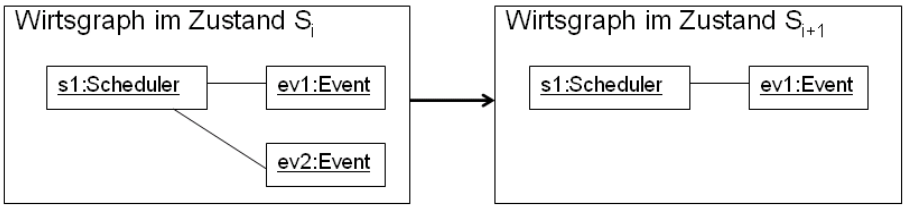

Abbildung 4.21: Ausführung der Methode *executeEvent*

Nachdem das erste Story Pattern erfolgreich ausgeführt worden ist, wird versucht, das zweite Story Pattern auszuführen. Zu diesem Zeitpunkt hat der Wirtsgraph die Form des Zustands S_i. Dann wird versucht, ein Objekt des Typs Event mit dem Objekt *Scheduler* zu verbinden. In diesem Fall wird das Objekt ev_2 mit der Variable e_2 gebunden. Das Ergebnis ist der Zustand S_{i+1}. Zu diesem Zustand existiert immer noch das Objekt ev_1, obwohl das Ereignis des Objektes ausgeführt worden ist. Im Gegensatz dazu wurde das Objekt ev_2 gelöscht und nie ausgeführt.

Um solche Probleme zu überwinden, werden neue Elemente definiert: die gebundenen Variablen. Eine gebundene Variable ist eine Variable in einem Story Pattern, die schon mit einem Objekt des Wirtsgraphen gebunden ist. Die *this* Variable, die o.B.d.A. in jedem Story Pattern enthalten ist, ist auch eine gebundene Variable, da das Objekt des Wirtsgraphen, worauf die Methode aufgerufen wird, bekannt ist. Durch die gebundenen Variablen ist es nun leicht zu garantieren, dass der gleiche Teilgraph des Wirtsgraphen für alle Story Patterns, die Teile der gleichen Modifikation sind, als Anwendungsstelle gefunden wird. In Abbildung 4.22 wird die Methode *executeEvent* dargestellt, mit dem Unterschied, dass im zweiten Story Pattern die Variable e_1 gebunden ist. Eine gebundene Variable ist dadurch zu erkennen, dass neben dem Namen nicht der Typ der Variable angegeben wird. Somit ist garantiert, dass nach der Ausführung des Ereignisses auch das richtige Event gelöscht wird.

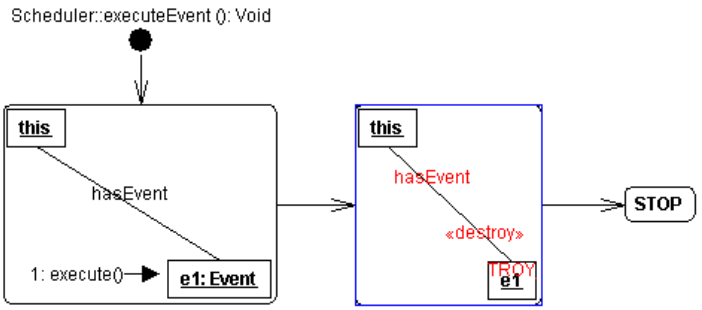

Abbildung 4.22: Verbesserte Version der Methode *executeEvent*

Damit in jedem Story Pattern einer Methode bekannt ist, mit welchem Objekt eine gebundene Variable gebunden ist, wird in [13] für jede Variable eine Information, die besagt, mit welchem Objekt des Wirtsgraphen die Variable gebunden ist, im procedure-call-stack der Methode gespeichert. Da dieses hier nicht möglich ist, wird eine neue Abbildung definiert, die *BoundObjects* heißt. *BoundObjects* ist eine Abbildung einer Variable eines Story Patterns, die den Typ *ExtNode* hat, auf einem Objekt des Wirtsgraphen, das auch vom Typ *ExtNode* ist.

```
01 class BoundObjects
02   var elements as ExtNode -> ExtNode
```

Jede Methode initialisiert ihre eigene *BoundObjects* Abbildung und wird als Parameter an jedem Aufruf der Regel *runGrr* angegeben. Die Regel *runGrr* aktualisiert die Abbildung *BoundObjects*, und diese wird dann vom nächsten Story Pattern weiterverwendet. Im folgenden Code wird der Teil der *findMatchFor*-Regel dargestellt, der für die Teilgraphensuche des ersten Story Patterns der Methode *executeEvent* verantwortlich ist.

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                            executeevent1_0LGNode0 |-> thisObject }
03 step
```

```
04   // Binden von e1 mit n0
05     //match found
06     result := true
07     boundObjects.elements := boundObjects.elements override
                                    {executeevent1_0LGNode1 |-> n0}
08     // Initialisierung von Match
```

In Zeile 2 wird die Variable *executeevent1_0LGNode0*, die der Variable *this* des ersten Story Patterns der Methode *executeEvent* entspricht, mit dem *this* Objekt des Wirtsgraphen gebunden und in der Abbildung *boundObjects* eingefügt. In Zeile 4 wird versucht, die Variable *executeevent1_0LGNode1*, die der Variable e_1 derselben Methode entspricht, mit einem Objekt n_0 des Wirtsgraphen zu binden. Wird die Variable erfolgreich gebunden, so wird die boolesche Variable *result* den Wert `false` behalten, und die Suche ist damit beendet. Wird ein Objekt des Wirtsgraphen gefunden, so ist die Teilgraphensuche erfolgreich. Nach einer erfolgreichen Suche wird die Variable *result* auf `true` gesetzt, und alle neu gebundenen Variablen werden in die Abbildung *boundObjects* eingefügt. Das geschieht in Zeile 7. Schließlich wird die Abbildung *Match* für dieses Story Pattern initialisiert.

Nachdem das erste Story Pattern der Methode *executeEvent* erfolgreich ausgeführt worden ist, wird versucht, eine Anwendungsstelle für das zweite Story Pattern zu finden. Dies ist in diesem Fall trivial, da das Story Pattern aus zwei schon gebundenen Variablen besteht. Es muss also noch der Link zwischen den zwei Variablen überprüft werden. Es handelt sich dabei um einen Check-Link. Um auch auf die schon gebundenen Variablen zugreifen zu können, wird die Abbildung *boundObjects* benutzt. Folgender Code beschreibt den Teil der Regel *findMatchFor*, welcher für die Teilgraphensuche des zweiten Story Patterns der Methode *executeEvent* zuständig ist.

```
01 machine
02  boundObjects.elements := boundObjects.elements override
                                  {executeevent2_1LGNode0 |-> thisObject }
03 step
04   // Überprüfung für das Link hasEvent
05     // Aktualisierung von boundObjects
06     // Initialisierung von Match
```

In Zeile 2 wird die Variable *executeevent2_0LGNode0*, die der Variable *this* des ersten Story Patterns der Methode *executeEvent* entspricht, mit dem *this* Objekt des Wirtsgraphen gebunden und in die Abbildung *boundObjects* eingefügt. In Zeile 4 wird der Check-Link überprüft. Dabei wird die Funktion *checkLink* aufgerufen, die als Parameter die zwei Objekte des Wirtsgraphen bekommt, die mit den zwei gebundenen Variablen des Story Patterns gebunden sind. Um auf diese Objekte zuzugreifen, wird die Abbildung *boundObjects* benutzt. Auf welches Objekt die Variable *executeevent2_1LGExtEdge0* abgebildet wird, ist in Zeile 2 definiert. Für die Variable *executeevent2_1LGExtEdge1* fehlt die Information, ob sie mit dem gleichen Objekt gebunden ist wie die Variable *executeevent1_1LGExtEdge1* des ersten Story Patterns.

Um dieses Problem zu überwinden, wird die Generierung der Initialisierung des linken Objektgraphen *LG* des Story Patterns erweitert. Für jede gebundene Variable wird im vorherigen Story Pattern nach einer Variable des gleichen Namens gesucht, die noch nicht gebunden worden ist. In diesem Fall ist das die Variable *executeevent1_1LGExtEdge1* des ersten Story Patterns der Methode. Die Initialisierung der Variablen des ersten Story Patterns sieht wie folgt aus:

```
01 //ExtNode
```

```
02 executeevent1_0LGNode0 = new ExtNode("thisVariable")
03 executeevent1_0LGNode1 = new ExtNode("e1")
04 //ExtN
05 executeevent1_0LGN = new ExtN({executeevent1_0LGNode0,
                                   executeevent1_0LGNode1})
```

Folgender Code stellt die Initialisierung des zweiten Story Patterns der Methode *executeEvents* dar.

```
01 //ExtNode
02 executeevent2_1LGNode0 = new ExtNode("thisVariable")
03 executeevent2_1LGNode1 = executeevent1_0LGNode1
04 //ExtN
05 executeevent2_1LGN = new ExtN({executeevent2_1LGNode0,
                                   executeevent2_1LGNode1})
```

In Zeile 3 wird für die Variable *executeevent2_1LGNode1* kein neues Element erzeugt, sondern das Element *executeevent1_0LGNode1* des ersten Story Patterns übernommen. Der rechte Objektgraph bleibt unverändert, da eine gebundene Variable nicht die Markierung *«create»* haben darf. Es ist nicht möglich, eine Variable mit einem Objekt des Wirtsgraphen zu binden, wenn diese noch nicht existiert.

Beim Aufruf der Methode *checkLink* ist es nun möglich, auf das gebundene Objekt der Variable *executeevent2_1LGNode1* durch die Abbildung *boundObjects* zuzugreifen. Ist der Rückgabewert der Methode *checkLink* `true`, so ist die Teilgraphensuche erfolgreich gewesen. Als nächstes werden die neu gebundenen Variablen in die Abbildung *boundObjects* eingefügt. In diesem Fall gibt es keine Variablen, die durch die Teilgraphensuche gebunden worden sind. Schließlich wird die *Match*-Abbildung des Story Patterns initialisiert. Auch hier wird das Objekt, das mit der Variable *executeevent2_1LGNode1* gebunden ist, durch die Abbildung *boundObjects* angegeben.

Attributbedingungen

Wie bereits erwähnt, läuft die Teilgraphensuche nach dem Force-Failure Ansatz ab. Bei diesem Ansatz wird versucht, einen Fehler beim Binden der Variablen des linken Graphen des Story Patterns möglichst früh zu entdecken. Eine Attributbedingung ist ein boolescher Ausdruck über einen Attributwert einer Variablen, die gebunden werden soll. Ist die Bedingung erfüllt, so wird die Variable im Wirtsgraphen gebunden. Ist das nicht der Fall, bricht die Suche ab und somit auch das Ausführen der Graphersetzungsregel, die durch das Story Pattern beschrieben wird.

In Fujaba werden solche Bedingungen in Java Code beschrieben. Die syntaktische und semantische Korrektheit der Bedingungen wird dem Benutzer überlassen. Diese Tatsache erzeugt ein Problem für die Übersetzung der Bedingung in ASM. Da eine Abbildung von Java in ASM im Rahmen dieser Arbeit nicht möglich ist, werden Attributbedingungen betrachtet, die eine bestimmte Form haben. In Tabelle 4.4 werden diese Formen dargestellt.

Form der Attributbedingung	Erklärungen
attributeName **op** Konstante	Die Konstante kann ein `Integer`, `Boolean`, `String`, `Float` sein
attributeName **op** Variable	Variable entspricht einer Variable der erlaubten Typen
attributeName **op** Methode	Methode entspricht jeder Methode, die im Klassendiagramm definiert ist, oder einer get/set Methode
op ist ein Vergleichsoperator und kann vom folgenden Typ sein: ==, =<, =>, <, >, !=	

Tabelle 4.4: Erlaubte Formen der Attributbedingungen

Im einfachsten Fall wird ein Attributwert mit einer Konstanten verglichen. Der Vergleichswert wird durch eine Eins-zu-Eins Übersetzung mit Ausnahme des Vergleichsoperators in AsmL geschrieben. In Tabelle 4.5 ist eine Abbildung der Vergleichsoperatoren von Java auf AsmL dargestellt. Natürlich werden nur die Typen von Werten unterstützt, die im Klassendiagramm auch erlaubt sind. Im zweiten Fall wird der Wert eines Attributes mit dem Wert einer Variablen verglichen. Unter einer Variablen ist hier nicht eine Instanz zu verstehen. Auch in diesem Fall findet eine Eins-zu-Eins Übersetzung statt. Im letzten Fall wird der Wert eines Attributes mit dem Rückgabewert einer Methode verglichen. Alle Methoden, die im Klassendiagramm deklariert sind, können auch in dieser Form benutzt werden. Wird die Methode ohne ein Zielobjekt aufgerufen, so wird als Objekt das jeweilige *this* Objekt verwendet.

Da Fujaba hauptsächlich Java Code generiert, wurde dieser auch beim Modellieren verwendet. Für jedes Attribut, dass eine Klasse hat, werden zwei Java Methoden generiert, die in der Modellsicht transparent bleiben. Eine wird als lesende Zugriffsmethode mit dem Präfix *get* versehen, die andere wird als schreibende Zugriffsmethode mit dem Präfix *set* versehen. Wird eine Zugriffsmethode aufgerufen, um den Attributwert mit einem Attributwert eines anderen Objektes zu vergleichen, so wird im AsmL Code die *AttributeValueFrom* Funktion verwendet, um auf den Attributwert des anderen Objektes zuzugreifen.

Vergleichsoperator in Java	Vergleichsoperator in AsmL	Erklärung
==	=	gleich
<	<	kleiner
>	>	größer
=<	<=	kleiner gleich
=>	>=	größer gleich
!=	<>	ungleich

Tabelle 4.5: Vergleichsoperatoren in Java und AsmL

An eine Variable können beliebig viele Attributbedingungen angehängt werden. Wird eine normale Variable mit einer Attributbedingung verknüpft, so werden bei der Ausführung des Story Patterns nur die Objekte des Wirtsgraphen betrachtet, die die Attributbedingung erfüllen. Wird die Variable mit mehreren Attributbedingungen verknüpft, so muss das zu bindende Objekt alle Attributbedingungen erfüllen, d. h. die Attributbedingungen werden untereinander mit dem booleschen Operator UND verknüpft.

In Abbildung 4.23 wird die Methode *checkShuttle* der Klasse *Station* dargestellt. Diese Methode überprüft zuerst den Zustand der Station. Ist dieser *WaitingForShuttle*, so wird als nächstes überprüft (zweites Story Pattern), ob das Shuttle mit einem Produkt beladen ist, das von einem Typ ist, den die Station gebrauchen könnte. Ist das der Fall, so wird das Event *UnloadShuttle* an die Station angehängt. Andernfalls wird überprüft, ob die Station ein Produkt hat. Existiert kein Produkt in der Station, so darf das Shuttle weiterfahren, und die Station kehrt durch das Event *WaitForShuttle* zum Zustand *WaitingForShuttle* zurück. Falls es ein Produkt in der Station gibt, wird das Event *LoadShuttle* an die Station angehängt.

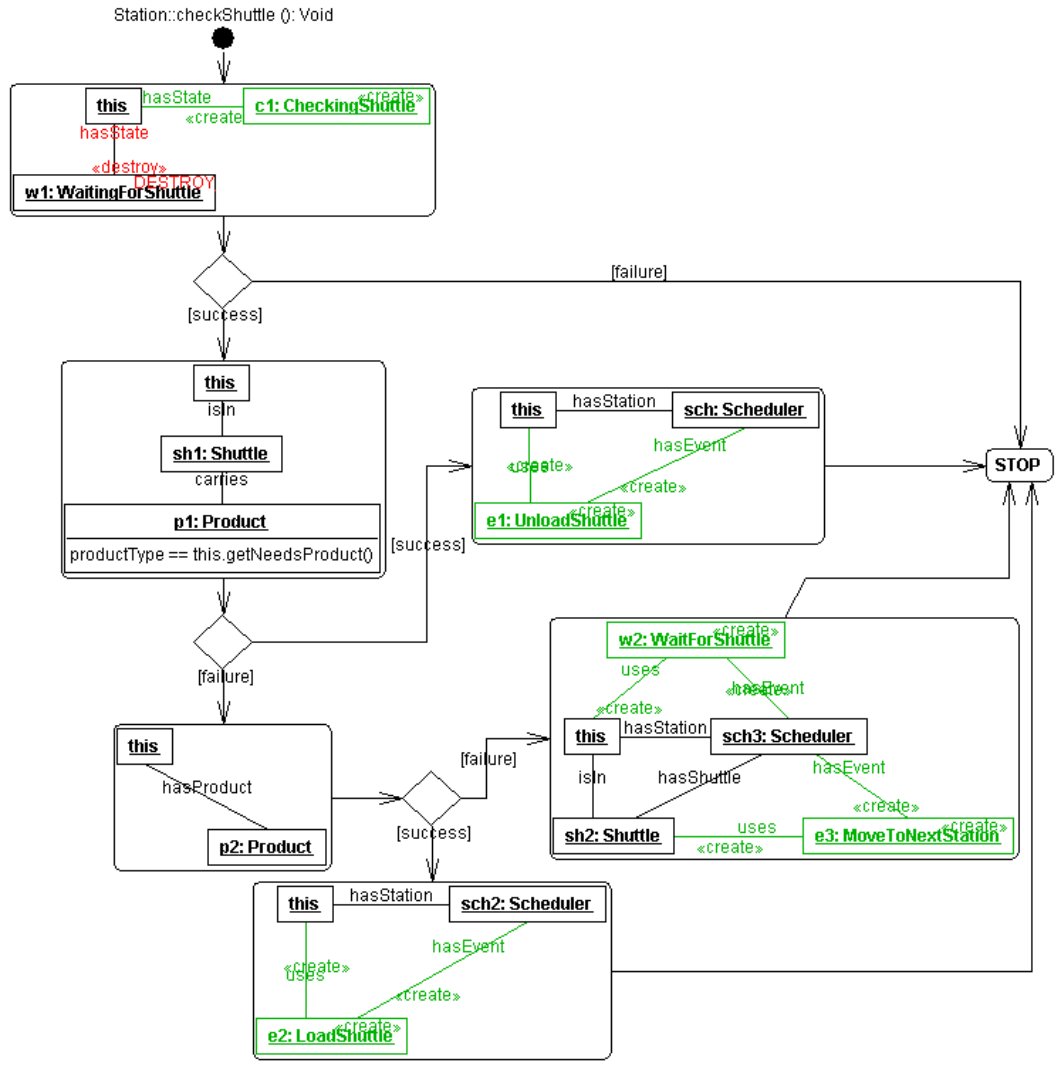

Abbildung 4.23: Die Methode *checkShuttle*

Folgender AsmL Code wird für das zweite Story Pattern generiert, in dem überprüft wird, ob das Shuttle, das an der Station steht, ein Produkt mit einem Typ trägt, welcher der Station nützlich sein könnte.

```
01 machine
02   // Binden von sh1 mit n0
03     // Binden von p1 mit n1
04       if getAttrValueFrom(EG, n1, classAttributeLabel2) = undef then
05         skip
06       else
07         if getAttrValueFrom(EG, n0, classAttributeLabel1) = undef then
```

```
08          skip
09        else
10          if getAttrValueFrom(EG, n1, classAttributeLabel2) =
11                getAttrValueFrom(EG, n0, classAttributeLabel1) then
12            // Aktualisierung von boundObjects
13            // Initialisierung von Match
```

In Zeile 2 wird versucht, die Variable sh_1 mit einem Objekt aus dem Wirtsgraphen zu binden. Gelingt es, so wird in Zeile 3 das Gleiche mit der Variable p_1 versucht. Existiert ein Objekt aus dem Wirtsgraphen auch für diese Variable, so werden als nächstes ab Zeile 4 die Attributbedingungen der Variable p_1 überprüft. Dabei wird zuerst getestet, ob der Attributwert des Objektes n_1, welches mit der Variable sh_1 gebunden ist, ungleich *undef* ist. Ist das der Fall, so wird auch der Attributwert des Objektes p_1 überprüft. Sind beide ungleich *undef*, so werden die Attributwerte der Objekte auf Gleichheit überprüft (Zeile 10). Die *get*-Methode, die den Attributwert der Variable p_1 zurückliefert, wird durch die Funktion *getAttrValueFrom* mit den entsprechenden Parametern ersetzt. Falls die Attributwerte gleich sind, ist die Teilgraphensuche erfolgreich und die Strukturen *boundObjects* und *Match* werden für das Story Pattern *G* aktualisiert.

Ist eine Attributbedingung abhängig von der Existenz anderer Variablen, z.B. beim Vergleich zweier Attributwerte von verschiedenen Variablen, so wird die Überprüfung dieser Attributbedingung eventuell solange in der Teilgraphensuche verschoben, bis beide Variablen mit einem Objekt des Wirtsgraphen gebunden sind.

Negative Knoten und negative Links

Oft ist es erwünscht, außer den Attributbedingungen auch anderen Bedingungen an Variablen zu setzen. Jeder Link *l*, der zwei Variablen v_1 und v_2 miteinander verbindet, ist so eine Bedingung. Damit die Variablen v_1 und v_2 mit den Objekten o_1 und o_2 des Wirtsgraphen gebunden werden, müssen diese Objekte mit einem Link verbunden sein, der Instanz der gleichen Assoziation ist wie der Link *l*. Eine Negation dieser Bedingung würde bedeuten, dass die Variable v_1 nicht mit einer Variable v_2 verbunden sein soll. Um solche Bedingungen spezifizieren zu können, werden die negativen Knoten und negativen Links eingeführt. Die Semantik dieser Elemente wird in [14] ausführlich beschrieben.

Für die Teilgraphensuche gelten die negativen Elemente als zusätzliche Bedingungen für das erfolgreiche Binden einer Variablen an den Wirtsgraphen. Ein negativer Knoten beschreibt Eigenschaften, die eine Variable nicht besitzen darf. Hängen mehrere negative Knoten an einer Variable, so werden sie einzeln unabhängig voneinander überprüft. Im weiteren Verlauf des Kapitels werden die Bedingungen, die durch negative Graphelemente beschrieben werden, als *negative Constraints* bezeichnet. Weitere Begriffe, die im Rahmen der negativen Elemente benutzt werden, sind die *Source-Variable* und *Target-Variable*. Eine Source-Variable ist eine Variable, die einen Link besitzt, der mit einem negativen Knoten verbunden ist. Eine Target-Variable definiert die negative Constraints der Source-Variable. Ein negativer Knoten ist also eine Target-Variable. Ähnlich sind auch die Begriffe *Source-* und *Target-Objekt* zu verstehen. Ein Source-Objekt ist das Objekt vom Wirtsgraphen, welches mit einer Source-Variable gebunden wird, falls diese alle Bedingungen der Teilgraphensuche erfüllt. Ein Target-Objekt ist das Objekt im Wirtsgraphen, welches durch einen Link mit dem Source-Objekt verbunden ist. Der Typ des Target-Objektes muss gleich dem Typ der Target-Variable sein.

Im einfachsten Fall wird ein negativer Knoten mit einer Variablen verbunden. Ist diese Target-Variable ungebunden, so darf das entsprechende Source-Objekt im Wirtsgraphen

keine Links zu Target-Objekten haben, deren Typ durch den negativen Knoten definiert wurde. Mit anderen Worten: Das Source-Objekt darf keine Links zu Instanzen vom Typ der Target-Variable haben. Ist die Target-Variable gebunden, so muss sie mit Attributbedingungen verknüpft sein, damit ihre Semantik Sinn macht. Eine negative gebundene Variable ohne Attributbedingungen führt zu einer widersprüchlichen Semantik. Der Widerspruch ergibt sich dadurch, dass eine gebundene Variable aufgrund der Semantik der gebundenen Variablen als existent gilt, die Negation der gebundenen Variable fordert aber, dass sie nicht existieren darf. Ist aber eine negative gebundene Variable mit Attributbedingungen verknüpft, so bezieht sich die „Nicht-Existenz“ der Negation nicht auf die Variable selbst, sondern auf ihre Attributbedingungen.

In der Abbildung 4.24 wird die Methode *loadShuttle* etwas modifiziert abgebildet. Diese Methode wird ausführlich in 4.2.7 beschrieben. Im Zusammenhang mit den negativen Elementen wird nur das dritte Story Pattern betrachtet. In diesem Story Pattern ist die Station *this* mit einem Produkt p_1 und mit einem Shuttle sh_1 verbunden, das wiederum mit einer negativen Variable des Typs *Product* verbunden ist. Dies bedeutet, dass die Station mit einem Shuttle verbunden sein soll, das nicht mit einem Produkt beladen ist. Die Source-Variable ist in diesem Fall sh_1 und die Target-Variable p_2.

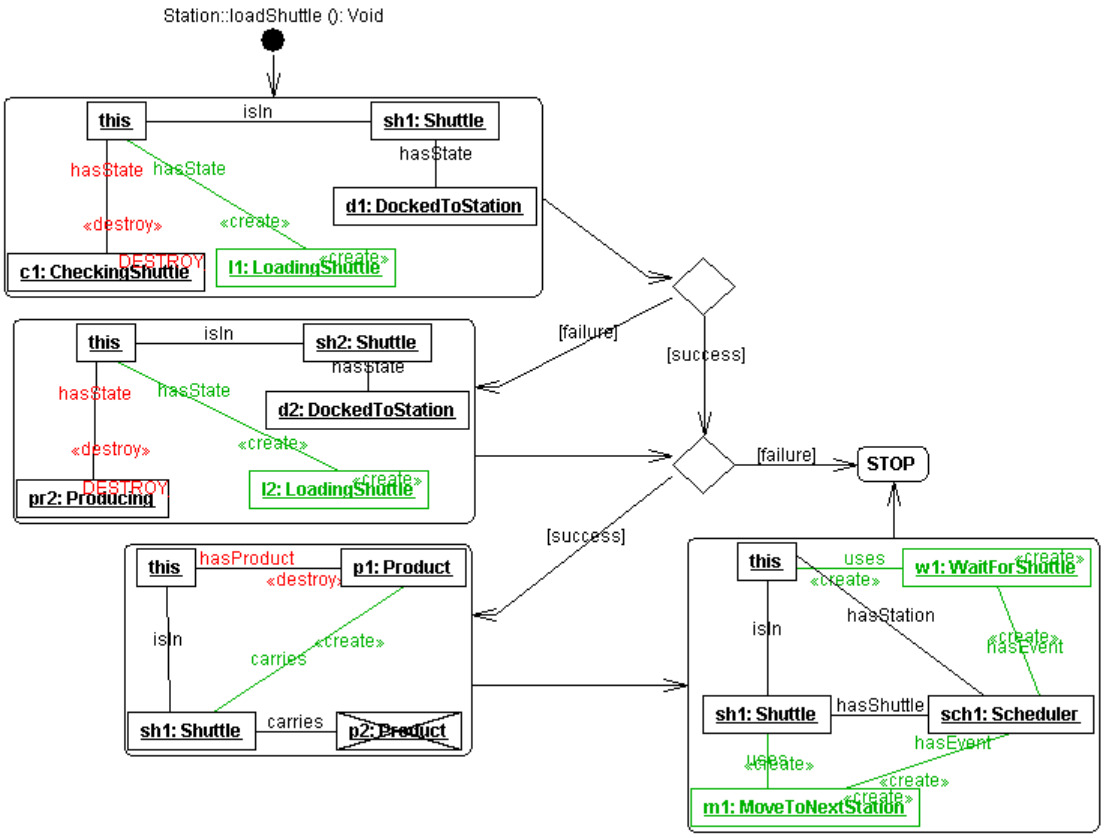

Abbildung 4.24: Die Methode *loadShuttle* mit einer negativen Variable

Der AsmL Code dieses Story Patterns sieht wie folgt aus:

```
01 machine
02   // Binden von sh1 mit n0
03   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                                    loadshuttle2_14LGExtEdge1)
04     // check negative variable
05     let (n1, e1, edge1) = getMatchObjectTo(G, EG, n1,
```

```
                                    loadshuttle2_14LGExtEdge2)
06      if n1 <> undef then
07        //no match found
08        skip
09      else
10        // Binden von p1 mit n2
11          // Aktualisierung von boundObjects
12          // Initialisierung von Match
```

Da die Variable sh_1 als Bedingung die negative Variable p_2 enthält, hat der Link *isIn* höhere Priorität als der Link *hasProduct*. Aus diesem Grund wird beabsichtigt, als erstes die Variable sh_1 zu binden. In Zeile 2 wird versucht die Variable sh_1 mit einem Objekt aus dem Wirtsgraphen zu binden. Existiert so ein Objekt, so werden als nächstes die Bedingungen überprüft. Laut Story Pattern darf das Objekt n_0 kein Objekt des Typs *Product* über einem Link des Typs *carries* als Nachbar haben. Dies wird in den Zeilen 5 und 6 überprüft. Existiert ein Objekt n_1, so wird die Suche abgebrochen, da die Bedingung nicht erfüllt ist. Existiert so ein Objekt nicht, so läuft die Teilgraphensuche weiter nach dem bekannten Konzept.

Weiterhin ist es möglich, eine ungebundene Target-Variable mit *«create»* zu markieren. Dies verändert die Semantik des negativen Knotens, indem sie um das Kreieren eines neuen Knotens erweitert wird. Existiert also ein Target-Objekt für die entsprechende Target-Variable, so wird die Regel nicht ausgeführt. Existiert aber kein Target-Objekt, so wird die Regel ausgeführt, und da die Target-Variable mit dem Modifizierer *«create»* markiert war, wird dafür ein neues Objekt im Wirtsgraphen erzeugt.

Eine ungebundene Target-Variable darf mit Attributbedingungen verknüpft sein. Semantisch bedeutet das, dass es kein Target-Objekt geben darf, das diese Attributbedingungen erfüllt. Das Source-Objekt darf mit Instanzen des Typs der Target-Variable durch einen Link verbunden sein, doch dieses Objekt darf mindestens eine der Attributbedingungen der Target-Variable nicht erfüllen. In Abbildung 4.25 wird das Story Pattern des letzten Beispiels verändert dargestellt. Die Variable p_2 enthält jetzt die Attributbedingung, dass der Typ des Produktes nicht *A* sein darf.

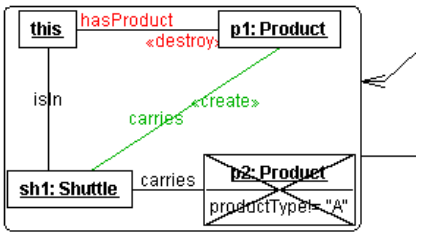

Abbildung 4.25: Negative ungebundene Variable mit Attributbedingung

Folgender Code ist ein Teil des obigen Codes und beschreibt das Binden der Variable p_2 und das Überprüfen ihrer Bedingung:

```
10      …
11      let (n1, e1, edge1) = getMatchObjectTo(G, EG, n1,
                                    loadshuttle2_14LGExtEdge2)
12      if n1 <> undef then
13        if getAttrValueFrom(EG, n1, classAttributeLabel2) = “A” then
14          // no match found
15        else
16          …
```

In Zeile 11 wird versucht, die Variable p_2 mit einem Objekt aus dem Wirtsgraphen zu binden. Existiert ein Objekt (Zeile 12), welches mit der Variable p_2 gebunden werden könnte, so muss zuerst die Attributbedingung getestet werden. In Zeile 13 wird überprüft, ob das Attribut *productType* des gefundenen Objektes n_1 gleich *A* ist. Ist das der Fall, so bricht die Teilgraphensuche ab. Andernfalls wird versucht, die Variable p_1 zu binden.

Ist die negative Variable mit Attributbedingungen gebunden, so darf das entsprechende Target-Objekt mindestens eine der Attributbedingungen nicht erfüllen. Tut sie es aber doch, so wird die Teilgraphensuche abgebrochen und das Story Pattern nicht ausgeführt. In Abbildung 4.26 wird dasselbe Story Pattern wie oben dargestellt, allerdings ist die Variable p_2 jetzt gebunden.

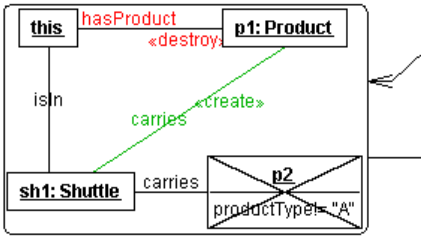

Abbildung 4.26: Negative gebundene Variable mit Attributbedingung

Folgender Code entspricht nur den Änderungen des in Abbildung 4.26 dargestellten Story Patterns:

```
10      …
11      if getAttrValueFrom(EG, boundObjects.elements(someVariable),
                            classAttributeLabel2) = "A" then
12        // no match found
13      else
14        …
```

In diesem Fall ist die Variable p_2 bereits gebunden, es wird also direkt der Wert des Attributes *productType* überprüft. Der Zugriff auf das Objekt, welches mit der Variable p_2 gebunden ist, geschieht mittels der Abbildung *boundObjects*.

Die Semantik der negativen Knoten ist auch abhängig von der Source-Variable. Ist diese als optional oder mengenwertig markiert, so ist die Semantik der Target-Variable verschieden. In [14] werden noch die negativen mengenwertigen und negativen optionalen Variablen untersucht. Für die Zwecke dieser Arbeit werden nur die oben beschriebenen Fälle betrachtet.

Auf ähnliche Weise wie die negativen Knoten werden auch die negativen Links definiert. Ein negativer Link zwischen zwei Variablen in einem Story Pattern hat die Semantik, dass zwischen Source- und Target-Objekt so ein Link nicht existieren darf. In Abbildung 4.27 wird die Negation der Variable p_2 durch einen negativen Link modelliert.

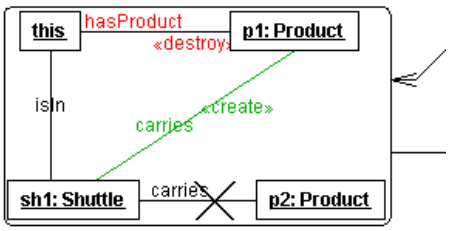

Abbildung 4.27: Negativer Link

Der generierte Code ist derselbe, wie im Fall einer ungebundenen negativen Variable, da die Semantik äquivalent ist. Auch für die negativen Links existieren unterschiedliche Semantiken, deren Interpretationen abhängig von den Variablen sind, die durch die negativen Links verbunden sind. Jeder dieser Fälle kann auf ähnliche Weise wie oben beschrieben in ASM übersetzt werden; daher werden sie nicht betrachtet.

Optionale Variablen

Ein weiteres Element, welches in einem Story Pattern vorkommen kann, ist die optionale Variable. Die Semantik einer optionalen Variable ist, dass sie bei der Teilgraphensuche nicht unbedingt gebunden werden muss. Existiert ein Objekt im Wirtsgraphen, das mit der optionalen Variablen gebunden werden kann, so wird es gebunden, und die Teilgraphensuche wird fortgesetzt. Existiert kein Objekt im Wirtsgraphen, welches mit der optionalen Variable gebunden werden kann, wird die Teilgraphensuche trotzdem fortgesetzt, und die optionale Variable bleibt ungebunden. In Abbildung 4.28 wird das Story Pattern aus dem obigen Paragraphen dargestellt, mit dem Unterschied, dass die Variable p_2 optional ist. Diese Modellierung setzt voraus, dass ein Shuttle mehrere Produkte tragen darf.

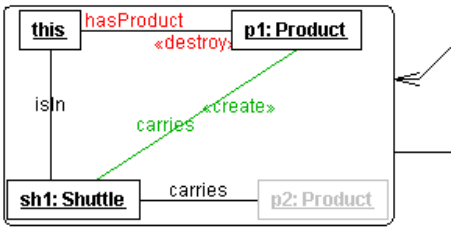

Abbildung 4.28: Optionale Variable

Folgender Code beschreibt den Teil der *FindMatchFor* Regel für das Story Pattern der Abbildung 4.28:

```
01 machine
02   // Binden von sh1 mit n0
03   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                                    loadshuttle2_14LGExtEdge1)
04     // check optional variable
05     let (n1, e1, edge1) = getMatchObjectTo(G, EG, n1,
                                  loadshuttle2_14LGExtEdge2)
06     //Binden von p1 mit n2
07       // Aktualisierung von boundObjects
08       if n1 <> undef then
09         boundObjects.elements := boundObjects.elements override
                                  { someVariable |-> n1 }

10       // Initialisierung von Match für die Variablen
11       if n1 <> undef then
12         GrrMatch.paramN := GrrMatch.paramN union
```

```
                          { someVariable |-> n1 }
13      // Initialisierung von Match für die Links
14      if n1 <> undef then
15        GrrMatch.paramE := GrrMatch.paramE union
             { someLink |-> ExtMap((n1,
             getELabelFrom(loadshuttle2_14LGExtEdge0), thisObject)) }
```

In Zeile 2 wird versucht, die Variable sh_1 mit einem Objekt aus dem Wirtsgraphen zu binden. Ist dies erfolgreich, so wird als nächstes versucht, die optionale Variable p_2 zu binden. Da die Variable optional ist, wird nicht überprüft, ob ein Objekt im dem Wirtsgraphen existiert, das mit dieser Variable gebunden werden kann. Dann wird versucht, die Variable p_1 zu binden. Falls das gelingt, ist die Teilgraphensuche erfolgreich, und die Strukturen *boundObjects* und *Match* werden aktualisiert. Da nicht sicher ist, ob für die optionale Variable p_2 ein gebundenes Objekt existiert, wird, bevor für sie ein Eintrag in den Strukturen *boundObjects* und *Match* eingefügt wird, überprüft, ob das Objekt n_1 existiert oder nicht. Falls das Objekt existiert, werden in den Zeilen 8, 11 und 14 die entsprechenden Einträge in den Strukturen eingefügt.

4.3.6 Methoden (Teil 1)

Die Methoden werden im Klassendiagramm deklariert und beschreiben das Verhalten einer Klasse. Das Verhalten einer Methode wird durch Story Patterns definiert, die Reihenfolge der Ausführung der einzelnen Story Patterns einer Methode wird durch ein Aktivitätsdiagramm bestimmt. Ein Aktivitätsdiagramm beschreibt also den Kontrollfluss einer Methode. Eine Methode besteht aus einem Namen, einer Liste von Parameternamen und ihren Typen und aus einem Rückgabetyp. Objektorientierte Spezifikationen erlauben das Überladen und das Überschreiben von Methoden. Überladen einer Methode bedeutet, dass in einer Klasse zwei oder mehrere Methoden denselben Namen haben dürfen, aber unterschiedliche Signaturen, d.h. verschiedene Parameterlisten oder Rückgabeparamtertypen, haben müssen. Überschreiben einer Methode bedeutet, dass die Unterklasse *B* einer Klasse *A* eine Methode der Klasse *A* deklariert, allerdings mit unterschiedlicher Implementierung. Beim Ausführen einer überschriebenen Methode wird deren Implementierung vom Laufzeittyp des Objektes bestimmt, auf das die Methode ausgeführt worden ist. Im Beispiel des Kapitels 4.1 wird die Methode *execute* der Klasse *Event* von ihren Unterklassen überschrieben. D.h. jede Unterklasse der Klasse *Event* hat ihre eigene Implementierung, die die Implementierung der Klasse *Event* erweitert.

In [13] wird beschrieben, wie mit Hilfe von Story Patterns und Story Diagrammen das Ausführen einer Methode spezifiziert wird. Dabei wird der Ablauf des Ausführens einer Methode im Detail nachmodelliert, indem ein procedure-call-stack Objekt in einem speziellen Story Pattern angelegt wird, um Objekte der internen Funktionalität der Methode zu verwalten. An dieses Objekt werden für jedes Ausführen der Methode in Form von Variablen Informationen über lokale Variablen, Variablen der Story Patterns der Methode u.s.w. gehängt. Beim Ausführen der Methode werden mit Hilfe von zusätzlichen Story Patterns diese Informationen verändert. Um diese Details zu umgehen, wird jede Methode des Klassendiagramms auf eine ASM Funktion abgebildet. Das bedeutet, dass für jede Methode eine ASM Funktion generiert wird[6], was den Vorteil hat, dass die Verwaltung des procedure-call-stack Objekts von ASM übernommen wird. Die Rekursion wird unterstützt, da ASM Funktionen rekursiv aufgerufen werden können. Um das Überschreiben einer Methode zu

[6] Der AsmL Code wird in C++ übersetzt und ausgeführt. Daher ist die Spezifikation des procedure-call-stack in [13], die die Funktionen von Java oder C++ nachmodelliert, auch in AsmL gleich.

unterstützen, wird für jede Methode ein neuer Funktionsname definiert. Der neue Name der Funktion besteht aus den Namen der Methode und des Rückgabetyps. Um das Überladen einer Methode zu unterstützen, wird noch der Name der Klasse hinzugefügt, in der die jeweilige Methode deklariert ist. Die Parameternamen der Methode werden in die Funktion übernommen. Der Typ des jeweiligen Parameters kann entweder einer der primitiven Typen oder eine Klasse aus dem Klassendiagramm sein. Außer den Parametern der Methode werden auch noch weitere Parameter in der Funktion definiert, die für den Ablauf der Funktion wichtig sind. Bei einem Methodenaufruf in einer objektorientierten Spezifikation existiert immer ein Ziel-Objekt, also ein Objekt des Wirtsgraphen, auf das die Methode ausgeführt wird. Falls so ein Objekt nicht vorhanden ist, wird das *this* Objekt als Ziel-Objekt genommen. Das Ziel-Objekt wird als Parameter in der Funktion gegeben. Der Name dieses Parameters ist *thisObject* und sein Typ ist *ExtNode*. Schließlich werden noch der Laufzeitobjektgraph und das Klassendiagramm als Parameter gegeben. Beide sind notwendig für die Ausführung der einzelnen Story Patterns, die die Methode enthält, aber auch für die Funktion an sich. Im Allgemeinen sieht die Deklaration einer Funktion, die einer Methode mit einem Rückgabewert entspricht, wie folgt aus:

```
r_cn_mn (EG as ExtGraph, S as SIGraph, thisObject as ExtNode, p1 as Value,
p2 as Value, …) as Value =
 ...
```

In der obigen Deklaration ist *r* der Rückgabetyp der Methode *mn*, die in der Klasse *cn* deklariert wurde. Der Parameter *EG* entspricht dem Wirtsgraphen, *S* dem Klassendiagramm und das *thisObject* dem Objekt, auf das die Methode aufgerufen wird. Danach folgen die Parameter p_i der Methode *mn*. Die Parameter sind vom Typ *Value*, der wie folgt definiert wird:

```
01 class Value
02   var element as ExtNode
03   var value as String
```

Ein Element der Struktur *Value* entspricht dem Wert einer Variablen im Modell. Ist der Wert vom Standardtyp, so wird dieser in `String` konvertiert und das Attribut *value* initialisiert. Ist der Typ des Wertes durch eine Klasse definiert, ist also der Wert ein Objekt des Wirtsgraphen, so wird das Attribut *element* mit dem entsprechenden Element initialisiert. Falls die Methode einen Wert zurückgibt, wird dieser auch vom Typ *Value* sein. Gibt die Methode keinen Wert zurück, so sieht die Deklaration der Funktion wie folgt aus:

```
void_cn_mn (EG as ExtGraph, S as SIGraph, thisObject as ExtNode, p1 as
Value, p2 as Value, …) =
 ...
```

Wie schon am Anfang des Kapitels 4 erwähnt, besteht eine Methode aus Story Patterns. Deren Ausführung beschreibt das Verhalten der Methode. Die Reihenfolge der Ausführung der einzelnen Story Patterns wird durch einen Kontrollfluss beschrieben, der mittels Aktivitätsdiagrammen spezifiziert wird. Um den Rumpf der Methode zu generieren, ist es notwendig, die Aktivitätsdiagramme auf ASM abzubilden.

4.3.7 Abbildung von Aktivitätsdiagrammen auf ASM

Aktivitätsdiagramme werden in der Softwareentwicklung für die Erläuterung der Funktionalität von Methoden eingesetzt. Ein Aktivitätsdiagramm besteht aus Aktivitäten und Transitionen. Eine Aktivität repräsentiert das Ausführen einer Operation, Transitionen beschreiben den Kontrollfluss zwischen den Aktivitäten. In Fujaba [15] werden die Aktivitäten eines Aktivitätsdiagramms durch Story Patterns beschrieben. Aus diesem Grund werden sie auch Story Diagramme genannt. Ein Aktivitätsdiagramm ist eine Kontrollstruktur von mehreren Story Patterns, es beschreibt also, in welcher Reihenfolge die eingebundenen Story Patterns ausgeführt werden.

Ein Aktivitätsdiagramm in Fujaba ist mit einer Methode assoziiert. Der Name des Diagramms wird durch die Konkatenation des Namens der Klasse und der Signatur der Methode definiert. Die Signatur einer Methode besteht aus dem Namen der Methode und den Eingabe- und Ausgabeparametern. Jedes Aktivitätsdiagramm hat einen Startknoten, mindestens eine Aktivität und mindestens einen Endknoten. Die Verbindung der einzelnen Knoten erfolgt durch Transitionen. Eine Aktivität kann eine oder mehrere ausgehende Transitionen haben. Falls mehrere ausgehende Transitionen existieren, wird die Benutzung von disjunkten Wächterbedingungen benötigt, um einen deterministischen Ablauf zu sichern. Ferner existieren Knoten, die keine Aktivität haben und nur für Fallunterscheidungen benutzt werden. Diese Knoten besitzen eine eingehende Transition und mindestens zwei ausgehende Transitionen. Auch hier müssen die ausgehenden Transitionen disjunkte Wächterbedingungen haben, um einen deterministischen Ablauf des Aktivitätsdiagramms zu sichern. Wie bereits erwähnt, enthält eine Aktivität ein Story Pattern. Wenn eine Aktivität ausgeführt wird, wird das beinhaltete Story Pattern ausgeführt. Das Story Pattern kann entweder anwendbar sein oder nicht. Aus diesem Grund existieren als Wächterbedingungen für Transitionen noch zwei weitere Ausdrücke, *Success* und *Failure*. Falls das Story Pattern anwendbar ist, wird der *Success*-Transition gefolgt, sonst der *Failure*-Transition.

Aus dem Aktivitätsdiagramm wird der Rumpf der Funktion, die einer Methode des Klassendiagramms entspricht, generiert. Dieses Konzept hat ähnliche Vorteile und Nachteile wie die Generierung der Regel *findMatchFor*. Sie werden in 4.3.5 erläutert. In [16] werden zwei Konzepte beschrieben, wie Java Code von einem Aktivitätsdiagramm generiert werden kann. Das erste Konzept ist das “Interpreter-Konzept”, welches auch früher in Fujaba im Einsatz war. Nach diesem Konzept bekommt jede Aktivität eine eindeutige Nummer. Der Quellcode jeder Aktivität wird in einen *case*-Block eines *switch-case*-Kostruktes geschachtelt. Anhand der Nummer jeder Aktivität ist es möglich, die erwünschte Aktivität auszuführen. Das *switch-case*-Konstrukt befindet sich innerhalb einer *while*-Schleife, die jedes Mal überprüft, ob das Aktivitätsdiagramm beendet ist, das heißt, ob ein Endknoten erreicht ist. Falls das der Fall ist, wird die *while*-Schleife verlassen. Wenn nicht, wird in einer Variablen die Nummer der als nächstes auszuführenden Aktivität gelesen und die entsprechende Aktivität ausgeführt. Das zweite Konzept basiert auf der Wohlgeformtheit der Aktivitätsdiagramme. Wohlgeformtheit eines Aktivitätsdiagramms bedeutet, dass das Aktivitätsdiagramm aus einer Kombination der Kontrollstrukturen Selektion, Repetition und Sequenz aufgebaut sein kann. Falls ein Diagramm diese Bedingung erfüllt, ist es möglich, dafür Java Quellcode durch *if-then-else* und *while*-Kontrollstrukturen zu generieren. Um die Kontrollstrukturen im Aktivitätsdiagramm zu erkennen, wird das Diagramm in einen ungerichteten, knoten- und kantenmarkierten Graphen umgewandelt, wobei die Knoten die Aktivitäten und die Kanten die Transitionen sind. Ein Graph Parser übersetzt dann den Graphen in seine syntaktische Struktur, und aus dem sich ergebenden Ableitungsbaum ist es dann möglich, Kontrollstrukturen in Java zu erzeugen.

Die Gründe dafür, dass in Fujaba das zweite Konzept implementiert worden ist, sind zwei: Erstens hat der Java Quellcode des "Übersetzer-Konzepts" die Form eines von einem Programmierer geschriebenen Quelltextes, weil die syntaktischen Kontrollstrukturen des Aktivitätsdiagramms übersetzt worden sind, und nicht die semantischen, wie im Fall des "Interpreter-Konzepts". Der zweite Grund ist, dass ein Ziel von Fujaba die (Re-)Konstruktion von Aktivitätsdiagrammen aus Java Quelltext ist. Bei Verwendung des "Interpreter-Konzepts" steigt mit jedem erneuten Generieren und Wiedereinlesen der Quelltexte die Anzahl der Aktivitäten monoton. Das ist ein unerwünschtes Ergebnis. Ziel dieser Diplomarbeit ist es, Abbildungen von UML Spezifikationen in ASM zu konzipieren, um eine Spezifikation auf eine einheitliche Abstraktionsebene zu bekommen, so dass der generierte Quelltext als Eingabe für ein Model-Checker vorbereitet werden kann. Daher ist es nicht wichtig, dass der generierte Quelltext einer manuellen Programmierung entspricht. Ferner ist es auch nicht Ziel, aus einem ASM Quelltext Aktivitätsdiagramme zu erzeugen. Aus diesen Gründen wurde für die Generierung der Aktivitätsdiagramme das "Interpreter-Konzept" ausgewählt.

Anhand eines Beispiels wird nun erläutert, wie aus einem Aktivitätsdiagramm der Rumpf einer Funktion, die einer Methode der Spezifikation entspricht, generiert wird. In Abbildung 4.29 wird die Methode *loadShuttle* der Klasse *Station* des Beispiels aus Kapitel 4.1 dargestellt.

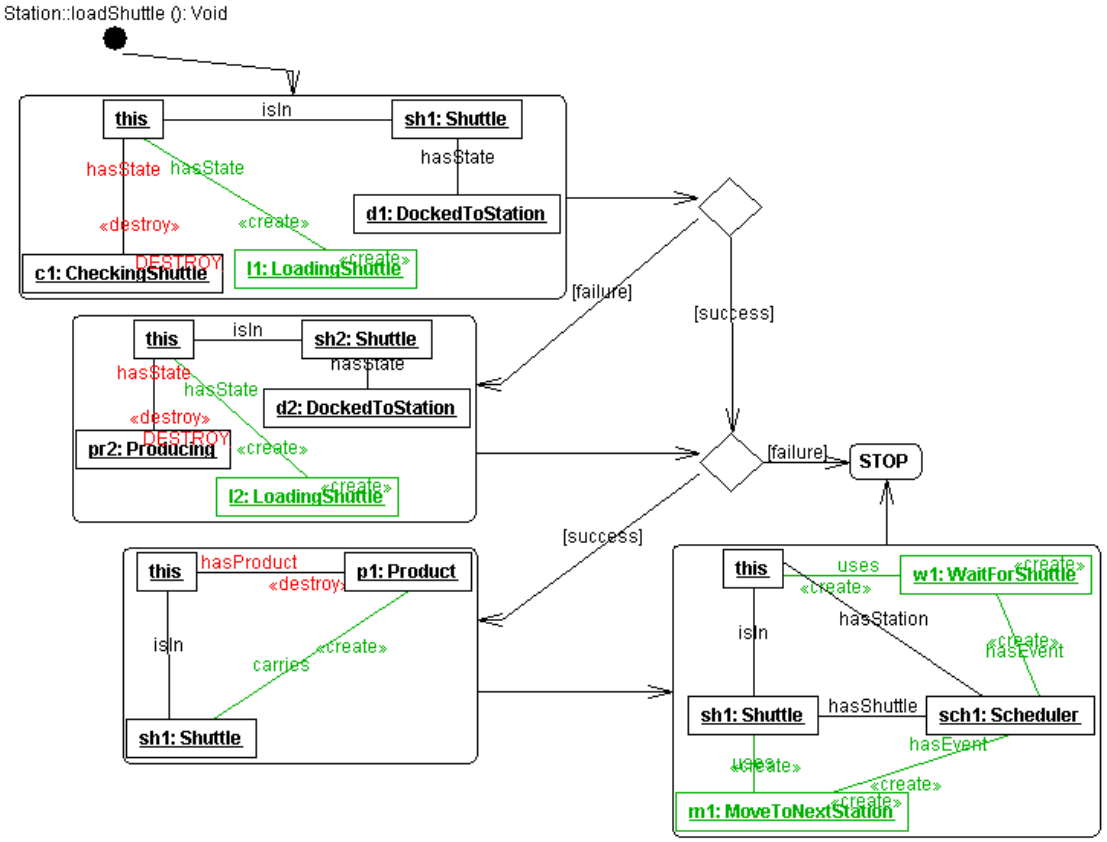

Abbildung 4.29: Die Methode *loadShuttle*

Die Methode besteht aus vier Story Patterns, aus einer Startaktivität, zwei NOP Aktivitäten und einem Endknoten. Im ersten Story Pattern, das mit $load_1$ bezeichnet wird, werden die Zustände der Station und des Shuttles, welches an der Station steht, überprüft. Ein Shuttle darf beladen werden, wenn es sich im Zustand *DockedToStation* befindet, und eine Station

darf ein Shuttle mit einem Produkt beladen, falls sie im Zustand *CheckingShuttle* ist. Ist diese Situation nicht im Wirtsgraphen vorhanden, so wird dieses Story Pattern nicht ausgeführt. Da die Aktivität des Story Patterns $load_1$ fehlgeschlagen ist, wird über den NOP Knoten und die *Failure*-Transition das zweite mit $load_2$ bezeichnete Story Pattern erreicht, das eine neue Bedingung für die Zustände der Station und des Shuttles stellt. Dieses beschreibt, wie auch im Statechart der Abbildung 4.3 dargestellt wird, dasselbe wie $load_1$, mit dem Unterschied, dass die Station im Zustand *Producing* sein muss. Ist auch dieser Fall nicht im Wirtsgraphen vorhanden, so wird über den zweiten NOP Knoten und die *Failure*-Transition der Endknoten erreicht, und somit ist die Ausführung der Methode beendet. Kommt eine der beiden Situationen im Wirtsgraphen vor, so wird eines der beiden Story Patterns aufgerufen. In beiden Fällen wird der alte Zustand der Station gelöscht und durch die Variable l_1 der neue Zustand *LoadingShuttle* an das Objekt der *Station* angehängt. Als nächstes wird, unabhängig davon, welches der beiden Story Patterns erfolgreich war, über die *Success*-Transition des zweiten NOP Knotens das dritte Story Pattern, welches $load_3$ genannt wird, erreicht. Dieses belädt das Shuttle mit dem Produkt der Station. Danach wird über eine einfache Transition das vierte Story Pattern, $load_4$ genannt, erreicht. In diesem Story Pattern werden an die Station und das Shuttle neue Events angehängt. Schließlich wird der Endknoten erreicht, und die Ausführung der Methode ist beendet.

Der AsmL Code dieses Aktivitätsdiagramms sieht wie folgt aus:

```
01 var asml_nextStep as Integer = 0
02 var asml_running as Boolean = true
03 var asml_GrrResult as Boolean = false
04 var returnValue = new Value(undef)
05 let params = new Parameters({|->})
06 var boundObjects = new BoundObjects({ |-> })
07 while asml_running do
08   machine
09     match asml_nextStep with
10       0: machine
11            asml_nextStep := 1
12          step
13            skip
14       1: machine
15            asml_GrrResult := runGrr(load1, EG, S, thisObject,
                                       boundObjects, returnValue, params)
16          step
17            asml_nextStep := 2
18          step
19            skip
20       2: machine
21            //nop activity
22            if not asml_GrrResult then  //failure
23              asml_nextStep := 3
24            if asml_GrrResult then  //success
25              asml_nextStep := 4
26          step
27            skip
28       3: machine
29            asml_GrrResult := runGrr(load2, EG, S, thisObject,
                                       boundObjects, returnValue, params)
30          step
31            asml_nextStep := 4
32          step
33            skip
34       4: machine
35            //nop activity
```

```
36            if not asml_GrrResult then  //failure
37              asml_nextStep := 5
38            if asml_GrrResult then  //success
39               asml_nextStep := 6
40          step
41            skip
42       5: machine
43            //stop activity
44            asml_running := false
45          step
46            skip
47       6: machine
48            asml_GrrResult := runGrr(load3, EG, S, thisObject,
                                       boundObjects, returnValue, params)
49          step
50            asml_nextStep := 7
51          step
52            skip
53       7: machine
54            asml_GrrResult := runGrr(load4, EG, S, thisObject,
                                       boundObjects, returnValue, params)
55          step
56            asml_nextStep := 5
57          step
58            skip
59   step
60     skip
```

In Zeile 1 wird die Variable *asml_nextStep* deklariert und mit dem Wert 0 initialisiert. Die Variable *asml_nextStep* weist immer auf die nächste Aktivität hin. Mit 0 wird immer die Startaktivität identifiziert. In Zeile 2 wird die Variable *asml_running* deklariert und mit `false` initialisiert. Solange kein Endknoten des Aktivitätsdiagramms erreicht worden ist, wird auch der Wert dieser Variable nicht geändert. In Zeile 3 wird die Variable *asml_GrrResult* deklariert. Diese bekommt bei jedem Aufruf der Regel *runGrr*, die ein Story Pattern ausführt, ihren Rückgabewert, der besagt, ob das Story Pattern erfolgreich ausgeführt worden ist oder nicht. In den Zeilen 4 bis 6 werden zwei Variablen deklariert, die in diesem Zusammenhang nicht wichtig sind. Sie werden nur deklariert, weil Sie als Parameter zur Regel runGrr gegeben werden. In Zeile 7 beginnt die `while`-Schleife, die solange iteriert, bis die Variable *asml_running* auf `false` gesetzt wird. Als nächstes folgt in Zeile 9 ein `match`-Ausdruck über die Variable *asml_nextStep*. Anhand ihres Wertes wird auch die entsprechende Aktivität ausgeführt. Am Anfang ist ihr Wert 0, es wird also die erste Aktivität ausgeführt, die den Wert 0 auf 1 ändert. In der nächsten Iteration der Schleife wird die Aktivität 1 ausgeführt, die der Aktivität mit dem Story Pattern $load_1$ entspricht. Nun wird in Zeile 15 die Regel *runGrr* und als Parameter das Story Pattern $load_1$ ausgeführt. Das Ergebnis der Ausführung wird in der Variable *asml_GrrResult* gespeichert, und als nächste Aktivität wird die Nummer 2 gesetzt, die dem ersten NOP Knoten entspricht. In dem NOP Knoten in Zeile 20 wird der Wert der Variable überprüft und entsprechend die nächste Aktivität gesetzt. So wird für den Fall der Failure Transition die Aktivität 3 und für den Fall der Success Transition die Aktivität 4 in die Variable *asml_nextStep* gesetzt. Das Ganze wiederholt sich solange, bis die Aktivität 5 erreicht worden ist. Dort bekommt die Variable *asml_running* den Wert `false`, und somit wird die Schleife verlassen.

Wächterbedingungen (*Guards*)

Neben den ausgehenden Transitionen *[Success]* und *[Failure]*, die abhängig von der Ausführung des Story Patterns sind, ist es möglich, weitere ausgehende Transitionen zu

modellieren. Diese erfordern dann, dass an den Transitionen disjunkte Wächterbedingungen vorhanden sind. In Abbildung 4.25 wird ein Beispiel eines NOP Knotens, aus dem vier Transitionen ausgehen, dargestellt. Die ersten drei Transitionen besitzen die Bedingungen B_1, B_2 und B_3, während die letzte Bedingung den Term *else* beinhaltet und dem Fall entspricht, dass keine andere Bedingung erfüllt ist. Eine Bedingung muss ein boolescher Ausdruck sein.

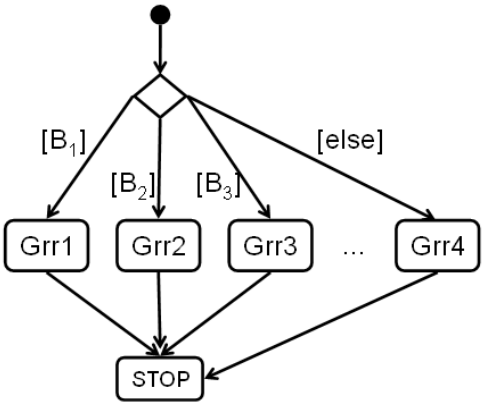

Abbildung 4.30: Mehrere ausgehende Transitionen mit Wächterbedingungen

Der generierte Code des Aktivitätsdiagramms der Abbildung 4.30 für den NOP Knoten sieht nun wie folgt aus:

```
19 n: machine
20      //nop activity
21      if B1 then
22        asml_nextStep := n1
23      if B2 then
24        asml_nextStep := n2
25      if B3 then
26        asml_nextStep := n3
27      else
28        asml_nextStep := n4
29      step
30        skip
```

In Zeile 19 beginnt der Code der Aktivität *n*, die dem NOP Knoten entspricht. In Zeile 21 wird die erste Bedingung B_1 genannt, die die Variable *asml_nextStep* auf n_1 setzt. Die Aktivität n_1 führt dann das Story Pattern Grr_1 aus. Entsprechend werden auch die anderen Bedingungen bis zum *else*-Ausdruck in Zeile 27 generiert. Die Bedingungen B_i sind boolesche Ausdrücke und werden in Java Code geschrieben, ähnlich wie bei den Attributbedingungen im Kapitel 4.3.5.

Multi-Aktivitäten

Eine Multi-Aktivität ist eine Aktivität, die das enthaltene Story Pattern solange aufruft, bis keine Anwendungsstelle mehr im Wirtsgraphen gefunden werden kann. D.h., solange im Wirtsgraphen ein Teilgraph existiert, der isomorph zum linken Graphen des Story Patterns ist, das in der Aktivität enthalten ist, wird dieses Story Pattern ausgeführt. Darüber hinaus können an die Multi-Aktivität die Transitionen *[each time]* und *[end]* geknüpft werden. Durch die Transition *[each time]* können für jede Anwendungsstelle individuelle Handlungen angesprungen werden. Wird keine mögliche Anwendungsstelle mehr gefunden, so wird die Multi-Aktivität über die *[end]* Transition verlassen.

Wie in [13] beschrieben wird, gibt es zwei Ansätze, die die Semantik von Multi-Aktivitäten beschreiben. Beide Semantiken beschäftigen sich mit der Verwaltung der Anwendungsstellen, so dass garantiert wird, dass alle Anwendungsstellen im Wirtsgraphen gefunden werden und dass die Multi-Aktivität auch terminiert. Bei der *fresh-matches* Semantik werden die gefundenen Anwendungsstellen gespeichert, damit garantiert wird, dass nur neue Anwendungsstellen gefunden werden. Dies kann aber zu Terminierungsproblemen führen, da das Ausführen des enthaltenen Story Patterns neue Anwendungsstellen erzeugen könnte. Bei der *pre-select* Semantik werden alle Anwendungsstellen vor der Ausführung des Story Patterns berechnet und gespeichert. So entstehen keine Terminierungsprobleme, es muss aber nach jeder Ausführung des Story Patterns überprüft werden, ob noch alle Anwendungsstellen gültig sind. Weiterhin könnte die Ausführung des Story Patterns neue Anwendungsstellen erzeugen, die nicht in der Menge der vorberechneten Anwendungsstellen enthalten sind. Dies steht im Konflikt mit der Definition der Multi-Aktivität. Ein weiteres Argument, das gegen die *pre-select* Semantik spricht, ist, dass die Implementierung sehr aufwendig sein könnte, da alle Anwendungsstellen im Voraus berechnet werden müssen. Aus diesen Gründen wurde die *fresh-matches* Semantik für die Multi-Aktivitäten ausgewählt.

Als nächstes wird anhand eines Beispiels die Generierung von AsmL Code für eine Multi-Aktivität dargestellt. Dabei wird die Methode *executeEvent* des Beispiels aus 4.1 modifiziert. In Abbildung 4.31 ist die modifizierte Methode executeEvent dargestellt.

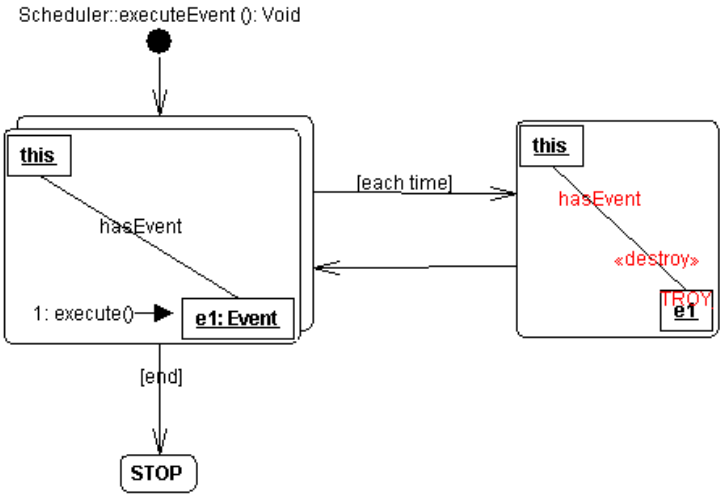

Abbildung 4.31: Die modifizierte Methode *executEvent*

Die erste Aktivität der Methode ist jetzt eine Multi-Aktivität. Das in der Multi-Aktivität enthaltene Story Pattern, welches mit $executeEvent_1$ bezeichnet wird, wird solange ausgeführt, bis kein Event mehr am Scheduler angehängt ist. Falls so ein Objekt der Klasse *Event* mit dem Scheduler in Verbindung steht, wird die Methode *execute* auf diesem Objekt ausgeführt. Wird für dieses Story Pattern eine Anwendungsstelle im Wirtsgraphen gefunden, so wird der *[each time]*-Transition gefolgt und das zweite Story Pattern, welches mit $executeEvent_2$ bezeichnet wird, ausgeführt. Dabei wird das gebundene Objekt e_1 des Wirtsgraphen gelöscht und das Story Pattern über die normale Transition verlassen. Existiert kein Objekt der Klasse *Event* mehr am Scheduler, so wird die Multi-Aktivität über die *[end]*-Transition verlassen, und die Ausführung der Methode *executeEvent* ist beendet.

Der generierte AsmL Code des Aktivitätsdiagramms der Abbildung 4.31 sieht wie folgt aus:

```
01 // Deklarationen
```

```
02 while asml_running do
03   machine
04     match asml_nextStep with
05       0: // Startaktivität
06       1: machine
07            let foundM = new foundMatches({})
08            asml_GrrResult := runIterGrr(executeEvent1, EG, S,
                    thisObject, boundObjects, returnValue, params, foundM)
09          step
10            if asml_GrrResult then
11              asml_nextStep := 2 // each time
12            else
13              asml_nextStep := 3 // end
14          step
15            skip
16       2: // Aktivität mit executeEvent2
17       3: // Stop Aktivität
```

Bis auf Zeile 6 entspricht der generierte Code dem Konzept, welches am Anfang des Kapitels beschrieben wurde. In Zeile 6 wird die Multi-Aktivität implementiert. In den Zeilen 7 und 8 finden die Vorbereitungen für die Ausführung der Regel *runIterGrr* und das Ausführen dieser Regel statt. Diese Regel wird im Anschluss an diesen Paragraphen näher erläutert. Das Ergebnis der Regel wird in *asml_GrrResult* zugewiesen. Wurde die Regel erfolgreich ausgeführt, so wird der *[each time]*-Transition gefolgt, die zur Aktivität 2 (Zeile 16) führt, anderfalls wird der *[end]*-Transition gefolgt, die zur Aktivität 3 (Zeile 17) führt, die auch dem Endknoten entspricht.

Wie bereits erwähnt, ist es nötig, die schon gefundenen Anwendungsstellen zu speichern. Dafür wird eine neue Struktur definiert:

```
01 class foundMatches
02   var elements as Set[Match]
```

Die Struktur *foundMatches* enthält ein Attribut vom Typ einer Menge von Elementen vom Typ *Match*. Die Abbildung *Match* wird von einem Story Pattern benutzt, um das Binden seiner Variablen mit Objekten des Wirtsgraphen zu speichern. *Match* enthält also indirekt die Anwendungsstelle des Story Patterns. In Zeile 14 des generierten Aktivitätsdiagramms wird die Menge *foundM* vom Typ *foundMatches* deklariert und mit der leeren Menge initialisiert. In Zeile 15 wird das in der Multi-Aktivität enthaltene Story Pattern ausgeführt, dieses Mal aber durch die Regel *runIterGrr*. Ein erster Unterschied zu der Regel *runGrr* ist, dass *runIterGrr* als Parameter auch die Menge *foundM* bekommt. Die Regel *runIterGrr* sieht wie folgt aus:

```
01 runIterGrr(G as Grr, EG as ExtGraph, S as SIGraph, thisObject as
       ExtNode, boundObjects as BoundObjects, returnValue as Value,
       params as Parameters, foundM as foundMatches) as Boolean =
02   machine
03      var matchingResult as Boolean = findMatchFor(G, EG, thisObject,
                                        boundObjects, returnValue, params)
04   step
05     if matchingResult then
06       let M = getMatch(G)
07       if isInFoundMatches(M, foundM) then
08         matchingResult := false
09       else
10         foundM.elements(M):=true
11   step
12     if matchingResult then
```

```
13        deletionFor(G, EG, S)
14      else
15        skip
16    step
17      if matchingResult then
18        creationFor(G, EG, S, boundObjects)
19      else
20        skip
21    step
22      return matchingResult
```

Die Regel *runIterGrr* ist der Regel *runGrr* ähnlich. Die Unterschiede bestehen darin, dass diese Regel überprüft, ob die von der Regel *findMatchFor* gefundenen Anwendungsstellen schon in der Vergangenheit gefunden worden sind oder nicht. Die schon gefundenen Anwendungsstellen existieren in der Menge *foundM*, die als Parameter zu der Regel gegeben wird. In Zeile 3 wird die Regel *findMatchFor* für das Story Pattern *G* ausgeführt, und in *matchingResult* wird das Ergenbis zugewiesen. Wird eine Anwendungsstelle gefunden, so wird für diese die Abbildung *M* (Zeile 6) durch die Funktion *isInFoundMatches* überprüft. Die Funktion *isInFoundMatches* stellt fest, ob die als Parameter gegebene *Match*-Abbildung in der Menge *foundM* enthalten ist. Ist das der Fall, so wird davon ausgegangen, dass keine weiteren Anwendungsstellen im Wirtsgraphen existieren, und die Variable *matchingResult* wird auf `false` gesetzt, damit das Story Pattern nicht mehr ausgeführt wird. Ist die Abbildung *M* nicht in *foundM* enthalten, so handelt es sich um eine neue Anwendungsstelle. In Zeile 10 wird die neue Anwendungsstelle in *foundM* gespeichert. Der übrige Verlauf der Regel ist mit der Regel *runGrr* identisch. Folgender Code beschreibt die Funktion *isInFoundMatches*:

```
01 isInFoundMatches(M as Match, foundM as foundMatches) as Boolean =
02   machine
03     var result = false
04   step
05     foreach m in foundM.elements do
06       if m.paramN = M.paramN and m.paramE = M.paramE then
07         result := true
08   step
09     return result
```

Die Funktion *isInFoundMatches* bekommt als Parameter die zu überprüfende Abbildung *M* des Typs *Match* und die Menge *foundM* des Typs *foundMatches*, die der Menge der schon gefundenen Anwendungsstellen entspricht. In Zeile 5 werden durch eine `foreach`-Schleife alle Elemente der Menge *foundM* durchlaufen. Sind die Einträge einer Abbildung *m* mit den Einträgen der Abbildung *M* identisch, so ist die Abbildung *M* in der Menge enthalten, und es wird `true` zurückgegeben. Sind sie nicht identisch, handelt es sich um eine neue Anwendungsstelle und es wird `false` zurückgegeben.

4.3.8 Methoden (Teil 2)

Nachdem die Abbildung der Aktivitätsdiagramme auf ASM erläutert worden ist, ist es nun möglich, eine Methode aus dem Klassendiagramm des Modells als eine AsmL Funktion zu generieren. Wie bereits in 4.3.6 erwähnt, werden Methoden durch Story Patterns und Aktivitätsdiagramme modelliert. Daher war es notwendig, erst die Story Patterns und die Aktivitätsdiagramme auf ASM abzubilden. In Abbildung 4.32 wird die Generierung von ASM Funktionen dargestellt, die Methoden entsprechen, die durch ein Aktivitätsdiagramm implementiert sind.

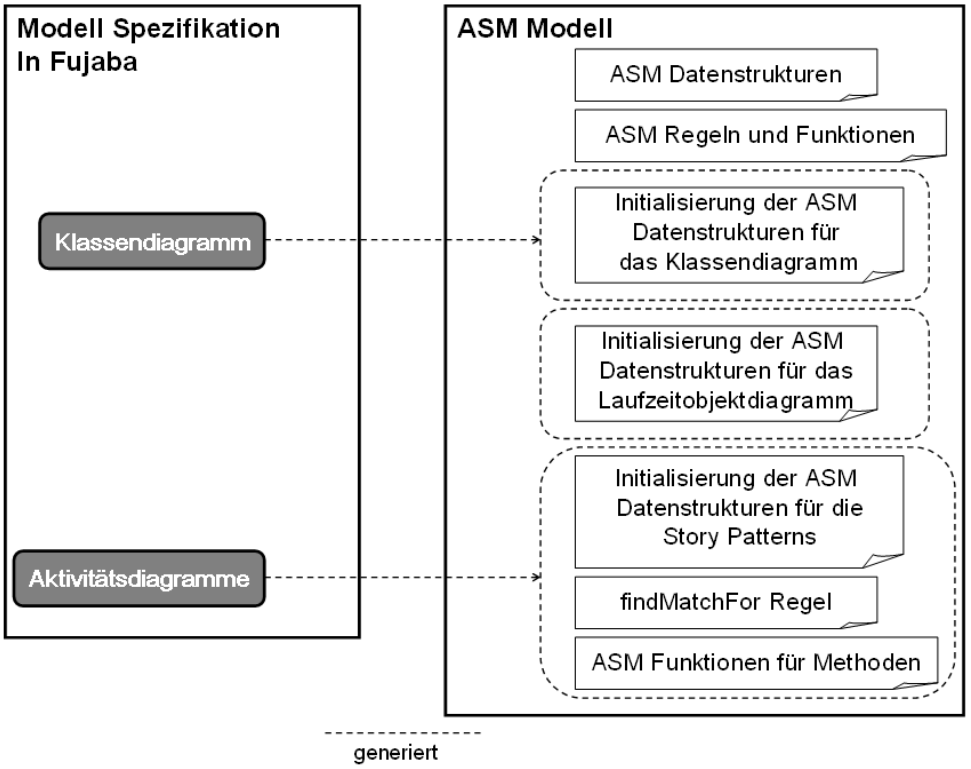

Abbildung 4.32: Generierung der Methoden durch ASM Funktionen

Die Implementierung einer Methode des Klassendiagramms wird in Fujaba durch ein Aktivitätsdiagramm modelliert. Da Story Patterns das Verhalten der Methoden beschreiben, werden zuerst aus dem Aktivitätsdiagramm die Initialwerte der Datenstrukturen der Story Patterns generiert. Anschließend wird dafür die Regel *findMatchFor* generiert. Schließlich ist es möglich, eine ASM Funktion für jede Methode des Klassendiagramms zu generieren. Dazu wird der Kontrollfluss des Aktivitätsdiagramms wie in 4.3.7 beschrieben benutzt. In Abbildung 4.33 wird die Methode *produce* der Klasse *Station* aus dem Beispiel des Kapitels 4.1 dargestellt.

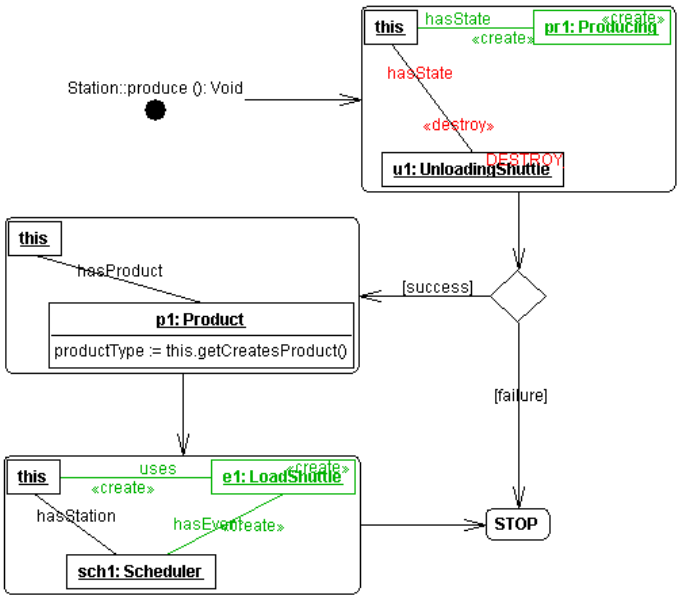

Abbildung 4.33: Die Methode *produce*

Im ersten Story Pattern der Methode *produce* wird der Zustand der Station überprüft. Ist die Station im Zustand *UnloadingShuttle*, so wird dieser gelöscht und an das *this* Objekt der neue Zustand *Producing* angehängt. Falls die Station nicht im Zustand *UnloadingShuttle* ist, wird das Ausführen der Methode beendet. Nachdem der Zustand der Station geändert worden ist, wird das Produkt p_1, welches sich auf dem Shuttle befand, verändert. Dabei wird der Typ des Produktes geändert, indem das Attribut *productType* der Variable p_1 mit dem Typ von Produkten, die diese Station produziert, zugewiesen wird. Nachdem das Produkt in der Station „bereit“ ist, wird im letzten Story Pattern der Methode an die Station das Event *LoadShuttle* angehängt.

Der AsmL Code der Methode *produce* besteht aus der Deklaration der Funktion, wie in Kapitel 4.3.6 definiert, und aus dem Rumpf der Funktion, der in Kapitel 4.3.7 definiert wurde, und sieht wie folgt aus:

```
01 void_Station_produce(EG as ExtGraph, S as SIGraph, thisObject as
                        ExtNode) =
02 var asml_nextStep as Integer = 0
03 var asml_running as Boolean = true
04 var returnValue = new Value(undef)
05 let params = new Parameters({|->})
06 var asml_GrrResult as Boolean = false
07 var boundObjects = new BoundObjects({ |-> })
08 while asml_running do
09 machine
10   match asml_nextStep with
11      0: // Startaktivität
12      1: // Aktivität mit produce1_16
13      2: // NOP Aktivität
14      3: // Aktivität mit produce2_17
15      5: // Aktivität mit produce3_18
16      4: // Stop Aktivität
```

In Zeile 1 wird die Funktion der Methode *produce* deklariert. Der Name der Funktion *void_Station_produce* besteht aus dem Rückgabeparameter der Methode *produce*, der Klasse, in der die Methode im Klassendiagramm deklariert wird und schließlich aus dem Namen der Methode. Die Parameter der Funktion bestehen aus den Strukturen des Wirtsgraphen *EG*, des Klassendiagramms *S* und aus dem *thisObjekt*, welches das Objekt repräsentiert, worauf die Methode aufgerufen wird. Ab Zeile 2 fängt die Implementierung der Funktion an, die aus dem Aktivitätsdiagramm der Methode generiert wird, die in Kapitel 4.3.7 noch beschrieben wird. In Zeile 4 wird die Variable *returnValue* deklariert. Diese Variable vom Typ *Value* entspricht der Rückgabe der Funktion. In der nächsten Zeile wird die Abbildung *params* deklariert. Sie hat den Typ *Parameters*, der wie folgt definiert wird:

```
class Parameters
  var elements as String -> Value
```

Die Abbildung *params* enthält also für jedes Parameter der Methode einen Eintrag. Ein Eintrag besteht aus dem Namen des Parameters vom Typ `String` und einem Wert vom Typ *Value*. Ist der Name des Parameters bekannt, so ist es möglich, auf den Wert des Parameters mittels der *params* Abbildung zuzugreifen.

In den folgenden Abschnitten wird das Beispiel aus dem Kapitel 4.1 leicht modifiziert. Dabei werden zu der Klasse *Product* zwei Methoden hinzugefügt. Beide Methoden entsprechen den

Zugriffsmethoden, die bei der Generierung des Java Codes für jedes Attribut einer Klasse erzeugt werden. In Abbildung 4.34 wird die erweiterte Klasse *Product* gezeigt.

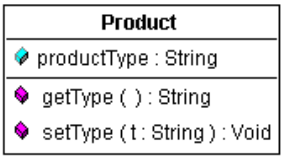

Abbildung 4.34: Die modifizierte Klasse *Product*

Die Methode *getType* wird in Abbildung 4.35 dargestellt. Diese besteht nur aus einem Story Pattern, in dem keine Modifikationen vorkommen, und einem Stop Knoten, der als Rückgabe einen Java Ausdruck hat. Dieser Ausdruck ist der Aufruf der Zugriffsmethode *getProductType* auf dieses *this* Objekt. Was also zurückgegeben wird, ist der aktuelle Wert des Attributes *productType* der Variable *this*.

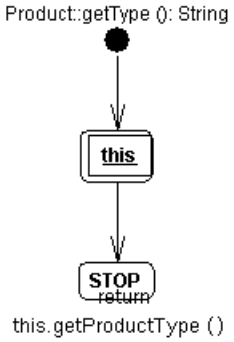

Abbildung 4.35: Die Methode *getType*

Der AsmL Code dieser Methode sieht wie folgt aus:

```
01 string_Product_getType(EG as ExtGraph, S as SIGraph, thisObject as
                          ExtNode) as Value =
02   // Deklarationen
03   while asml_running do
04     machine
05       match asml_nextStep with
06          0: // Startaktivität
07          1: // Aktivität mit getType_35
08          2: machine
09               //stop activity
10               asml_running := false
11               returnValue.value := getAttrValueFrom(EG, thisObject,
                                      classAttributeLabel2)
12             step
13               skip
14      step
15        return returnValue
```

In Zeile 1 wird die Funktion deklariert. Da die Methode *getType* den Rückgabetyp `String` hat, enthält der Name der Funktion den Term *string*. Als Rückgabewert wird der Typ *Value* gesetzt. Die Implementierung der Funktion folgt dem bekannten Konzept bis zur Zeile 8. In der Stop Aktivität des Aktivitätsdiagramms der Methode *getType* ist eine Rückgabe definiert. Der Wert, der von der Funktion zurückgegeben wird, wird in Zeile 11 gesetzt. Da es sich um einen primitiven Typ handelt (`String`) wird das Attribut *value* von *returnValue* gesetzt. Der

Wert des Attributes *productType* wird durch die Funktion *getAttrValueFrom* gegeben, die die Zugriffsmethode *getProductType* ersetzt. Schließlich wird in Zeile 15 der Rückgabewert zurückgegeben.

Nun wird die Methode *setType* der Klasse *Product* (Abbildung 4.34) betrachtet. Die Methode *setType* wird in Abbildung 4.36 abgebildet und hat ein Parameter *t* vom Typ `String`. Die Methode besteht aus einem Story Pattern, in dem es eine Attributzuweisung an die Variable *this* gibt. Dem Attribut *productType* wird der Wert *t* zugewiesen.

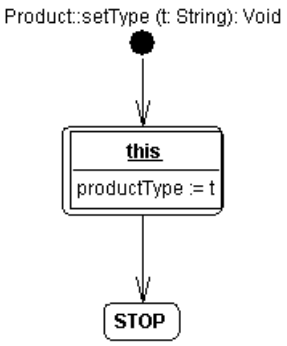

Abbildung 4.36: Die Methode *setType*

Der AsmL Code der Methode *setType* sieht wie folgt aus:

```
01 void_Product_setType(EG as ExtGraph, S as SIGraph, thisObject as
                        ExtNode, t as Value) =
02   // Deklarationen
03   let params = new Parameters({“t”|-> t})
04   while asml_running do
05     machine
06       match asml_nextStep with
07          0: // Startaktivität
08          1: // Aktivität mit setType_36
09          2: // Stop Aktivität
```

Vor dem Aufruf der Methode wird ein Element vom Typ *Value* für die Parameter *t* kreiert. Dabei wird für dieses Element das Attribut *value* mit dem Wert *t* initialisiert. In Zeile 1 wird also die Funktion der Methode *setType* deklariert, die neben den Standardparametern noch den Parameter *t* vom Typ *Value* hat. In Zeile 3 wird die Abbildung *params* deklariert und mit dem Eintrag initialisiert, der dem Parameter *t* entspricht. Die Abbildung *params* wird bei jeder Ausführung eines Story Patterns als Parameter zu der entsprechenden Regel gegeben. So kann, nachdem die Teilgraphensuche erfolgreich war, in der Regel *findMatchFor* folgende Zuweisung gemacht werden:

```
…
setAttrValueFrom(setType_8, thisObjekt, classAttributeLabel2,
                                        params.elements("t").value)
…
```

Dabei wird die Regel *setAttrValueFrom* benutzt, die als Parameter das Story Pattern *setType*, das Objekt aus dem Wirtsgraphen, das mit der Variable *this* gebunden ist (hier *thisObject*), das Attribut, welches aktualisiert wird, und schließlich den neuen Wert des Attributes bekommt. Da der Wert des Attributs dem Wert des Parameters *t* entspricht, wird dafür die Abbildung *params* benutzt.

Kollaborations-Nachrichten

In Story Patterns ist es möglich, Kollaborations-Nachrichten zu definieren. Diese erweitern die Semantik eines Story Patterns, wie in [18] beschrieben wird, und erlauben das Ausführen von Methoden auf Objekten. Kollaborations-Nachrichten bestehen aus dem Kollaborations-Text und dem Kollaborations-Pfeil, der in diesem Zusammenhang unwichtig ist. Der Kollaborations-Text besteht aus einer Zeitablauf-Nummer und einer Anweisung. Die Zeitablauf-Nummern beschreiben die Reihenfolge der Abarbeitung der Kollaborations-Nachrichten. Eine Anweisung enthält ausführbaren Java Code. Da eine Anweisung einen beliebigen Code darstellen darf, ist es notwendig, eine Begrenzung festzulegen. Aus diesem Grund werden nur Methodenaufrufe auf Objekte erlaubt. Die Methoden sind entweder im Klassendiagramm deklariert, oder es handelt sich um lesende Zugriffsmethoden mit dem Präfix *get* (vgl. Kapitel 4.2.5 in „Attributbedingungen“). Wird die Methode auf kein bestimmtes Objekt aufgerufen, so wird davon ausgegangen, dass es sich um das Objekt *this* handelt. Im Folgenden wird die Methode *execute* der Klasse *AssignShuttle* betrachtet, die in Abbildung 4.37 dargestellt wird.

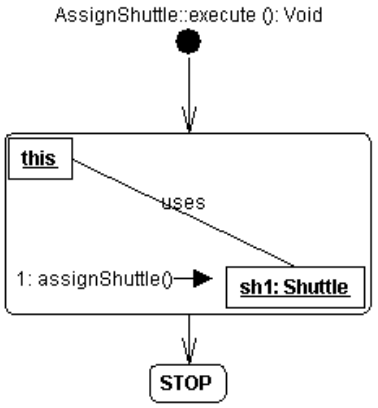

Abbildung 4.37: Die Methode *execute* der Klasse *AssignShuttte*

Die Methode *execute* der Klasse *AssignShuttle* hat ein Story Pattern, das auf der Variable sh_1, welche mit dem Event *this* in Verbindung steht (die Klasse *AssignShuttle* ist Unterklasse der Klasse *Event*), die Methode *assignShuttle* aufruft. Das Objekt, worauf die Methode aufgerufen wird, wird durch einen Pfeil gekennzeichnet. Der AsmL Code der Methode *execute* sieht wie folgt aus:

```
01 void_AssignShuttle_execute(EG as ExtGraph, S as SIGraph, thisObject as
                              ExtNode) =
02   // Deklarationen
03   while asml_running do
04     machine
05       match asml_nextStep with
06         0: // Startaktivität
07         1: machine
08              asml_GrrResult := runGrr(executeassignshuttle_26, EG, S,
                          thisObject, boundObjects, returnValue, params)
09            step
10              void_Shuttle_assignShuttle(EG, S, boundObjects.elements(
                                      executeassignshuttle_26LGNode1))
11            step
12              asml_nextStep := 2
13            step
14              skip
15         2: // Stop Aktivität
```

In Zeile 7 wird die Aktivität, die das einzige Story Pattern der Methode enthält, implementiert. Zuerst wird das Story Pattern *executeassignshuttle_26* ausgeführt (Zeile 8). Anschließend wird in der gleichen Aktivität die Funktion *void_Shuttle_assignShuttle* auf dem Objekt, das mit der Variable *executeassignshuttle_26LGNode1* gebunden ist, ausgeführt (Zeile 10). Der Aufruf der Funktion *void_Shuttle_assignShuttle* entspricht der Kollaborations-Nachricht, in der die Methode *assignShuttle* auf der Variable sh_1 aufgerufen wird.

Überschreiben von Methoden

Wie bereits in Kapitel 4.3.6 erwähnt, ist es in objektorientierten Spezifikationen möglich, Methoden zu überschreiben (*overloading*). In Abbildung 4.38 werden die Klassen *Scheduler*, *Event* und die Unterklassen der Klasse *Event* dargestellt. Die Klasse *Scheduler* beinhaltet die Methode *executeEvent* (Abbildung 4.22), die in ihrer Implementierung auf einer Instanz der Klasse *Event* die Methode *execute* der Klasse *Event* aufruft. Die Methode *execute* ist aber von den Unterklassen der Klasse *Event* überschrieben, d.h. sie wird anders implementiert. Beim Aufruf der Methode *executeEvent* der Klasse *Scheduler*, sowie auch beim Aufruf der Methode *execute* der Klasse *Event* muss der Laufzeittyp der Instanz überprüft werden, um festzustellen, ob diese Instanz einer Unterklasse der Klasse *Event* ist. Entsprechend der Instanz muss also auch die korrekte Methode *execute* aufgerufen werden.

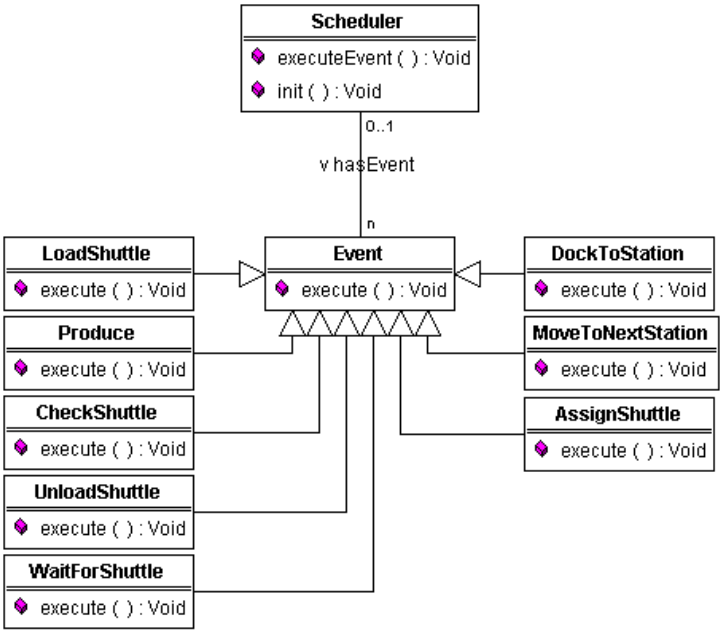

Abbildung 4.38: Die Klasse *Event* und ihre Unterklassen

Folgender AsmL Code stellt die Implementierung der Methode *executeEvent* (Abbildung 4.22) als eine AsmL Funktion dar:

```
01 void_Scheduler_executeEvent(EG as ExtGraph, S as SIGraph, thisObject as
                               ExtNode) =
02   // Deklarationen
03   while asml_running do
04     machine
05       match asml_nextStep with
06         0: // Startaktivität
07         1: machine
08              asml_GrrResult := runGrr(executeevent1_0, EG, S,
```

```
                              thisObject, boundObjects, returnValue, params)
09            step
10              //polymorphic method
11              let extNl = getNl(EG)
12              let n = extNl.elements( boundObjects.elements(
                                        executeevent1_0LGNode1) )
13              //class Event
14              if (n = classNode1) then
15                void_Event_execute( EG, S, boundObjects.elements(
                                      executeevent1_0LGNode1) )
                                boundObjects.elements( executeevent1_0LGNode1) )
16              …
17              // Überprüfungen für alle Unterklassen der Klasse Event
18              …
19              asml_nextStep := 2
20            step
21              skip
22          2: // Aktivität mit executeevent2_1
23          3: // Stop Aktivität
```

Ab Zeile 7 wird die Aktivität, die das erste Story Pattern der Methode enthält, implementiert. Dieses Story Pattern hat eine Kollaborations-Nachricht. Nachdem also das Story Pattern ausgeführt worden ist (Zeile 8), wird die Kollaborations-Nachricht implementiert. Diese besteht aus dem Ausführen der Methode *execute* auf der Variable e_1. Da e_1 eine Instanz der Klasse *Event* ist, muss der Laufzeittyp des Objektes, das mit der Variable gebunden wird, überprüft werden. Um den Typ des Objektes zu bekommen, wird die Abbildung *Nl* benutzt (Zeile 11 und 12), die Objekte auf Klassen abbildet. Von Zeile 14 bis 18 wird überprüft, von welcher Klasse das Objekt *executeevent1_0LGNode1* Instanz ist. Dabei werden die Unterklassen der Klasse *Event* und die Klasse *Event* selbst betrachtet. Für jede verschiede Klasse wird auch die entsprechende Funktion aufgerufen. Anschließend wird in Zeile 22 die Aktivität des zweiten Story Patterns und schließlich in Zeile 23 die Stop Aktivität implementiert.

4.3.9 Ausführen der Spezifikation

Wie schon erwähnt, ist es möglich, die generierte AsmL Spezifikation des Modells auszuführen und dadurch das System zu simulieren. Um eine AsmL Spezifikation auszuführen, ist es nötig, die Funktion *run* zu implementieren. Diese Funktion wird generiert, da für jede Spezifikation eine vom Benutzer gegebene Methode definiert werden muss, mit der die Ausführung beginnt. Für das Beispiel aus Kapitel 4.1 sieht die Funktion *run* wie folgt aus:

```
01 run() =
02   machine
03     let EG = extEG
04     let S = classSI
05   step
06     void_Scheduler_init(EG, S, startExtnode)
07   step
08     skip
```

Der schon initialisierte Wirtsgraph wird in Zeile 3, das Klassendiagramm in Zeile 4 deklariert. In Zeile 6 wird die Funktion *void_Scheduler_init* aufgerufen, die der Methode *init* der Klasse *Scheduler* entspricht. Mit dem Aufruf dieser Funktion beginnt die Ausführung der eigentlichen Spezifikation.

Es ist leicht zu sehen, dass die Berechnung der Spezifikation aus drei Zuständen besteht. Im ersten Zustand enthält der Wirtsgraph seinen initialen Wert, der in diesem Fall die Instanz *startExtnode* der Klasse *Scheduler* ist. Der zweite Zustand enthält die Menge der Aktualisierungen, die aus dem Aufruf der Funktion *void_Scheduler_init* stammen, da diese wiederum eine neue ASM Maschine definiert. Durch diese Maschine werden neue Zustände definiert, die eigene Aktualisierungen im Wirtsgraphen vornehmen oder wiederum durch den Aufruf einer Funktion eine neue Maschine definieren. Nachdem der Aufruf der Funktion *void_Scheduler_init* beendet ist, wird der Endzustand der Berechnung erreicht, und auch das Ausführen der Spezifikation ist abgeschlossen.

4.4 Zusammenfassung

In diesem Kapitel wurde die Abbildung vom UML Klassendiagrammen und Aktivitätsdiagrammen auf ASM beschrieben. Um die Semantik des Klassendiagramms auf ASM abzubilden, wurde eine Menge von Datenstrukturen definiert, die durch das jeweilige Klassendiagramm initialisiert wird. Die Aktivitätsdiagramme modellieren mittels Story Patterns die Implementierung der Methoden, die im Klassendiagramm deklariert werden. Daher wurden zunächst die Story Patterns betrachtet. Ein Story Pattern ist eine Graphersetzungsregel und besteht aus zwei Objektgraphen. Der linke Graph beschreibt die Anwendungsstelle des Story Patterns im Wirtsgraphen, der rechte Graph beschreibt die Situation nach der Ausführung des Story Patterns. Die Semantik eines Objektdiagramms wurde durch eine Menge von Datenstrukturen in ASM definiert. Um die Semantik der Story Patterns in ASM zu definieren wurden außer den Datenstrukturen auch Regeln definiert, die für das Ausführen eines Story Patterns eingesetzt werden. Die Ausführung eines Story Patterns erfolgt in drei Schritten. Im ersten Schritt wird eine Teilgraphensuche durchgeführt, um eine Anwendungsstelle im Wirtsgraphen zu finden. Existiert eine Anwendungsstelle, so werden im zweiten Schritt alle zu löschenden Elemente des Story Pattern aus dem Wirtsgraphen gelöscht. Entsprechend werden im dritten Schritt alle zu kreierenden Elemente des Story Patterns im Wirtsgraphen kreiert. Dabei werden auch die Attributwerte der Objekte aktualisiert. Die Story Patterns werden in einem Aktivitätsdiagramm eingesetzt, um das Verhalten der Methode zu beschreiben. Jede Methode wird in einer ASM Funktion abgebildet, deren Rumpf vom Aktivitätsdiagramm bestimmt wird, da dieses den Kontrollfluss der Methode modelliert. Schließlich entsteht eine ausführbare Spezifikation des Systemmodells in ASM.

5. Technische Umsetzung

Im diesem Kapitel wird die technische Umsetzung der in Kapitel 4 vorgestellten Konzepte in einem groben Überblick beschrieben. Anschließend wird die Ausführung eines generierten AsmL Codes mit Hilfe des Tools AsmL erläutert.

Architektur

Die Klassen, die zur technischen Umsetzung dieser Diplomarbeit benötigt werden, sind in einem Paket (*de.uni_paderborn.fujaba.asm*) in die Fujaba Architektur integriert worden. Beziehungen der Klassen dieses Pakets zu Klassen aus den übrigen Paketen von Fujaba sind als Referenzen modelliert worden, um zukünftig dieses Paket als Modul leichter implementieren zu können.

Die Hauptklasse *ASMCreator* des Pakets *de.uni_paderborn.fujaba.asm* ist als Singelton Klasse[7] implementiert und entspricht einem Werkzeug mit eigener graphischer Oberfläche (Klasse *ASMGui*) und einem Generator (Klasse *AsmLGenerator*). In Abbildung 5.1 werden die wichtigsten Klassen des Pakets dargestellt.

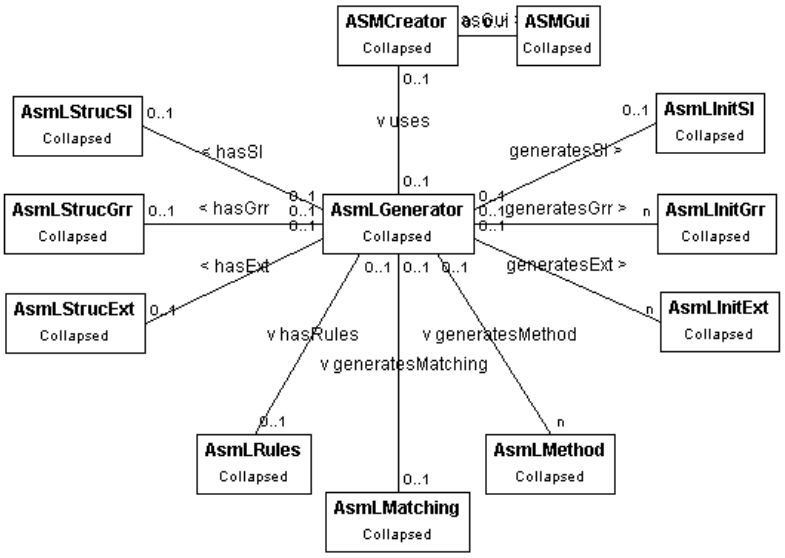

Abbildung 5.1:Die wichtigsten Klassen des ASMCreator

Die Klasse *AsmLGenerator* ist zuständig für die Generierung von AsmL Code. Diese Klasse hat Beziehungen zu mehreren Klassen, die jeweils einen Teil des AsmL Codes generieren. Zunächst werden die statischen Strukturen (Kapitel 4.2, 4.3.2, 4.3.3) des Klassendiagramms (Klasse *AsmLStrucSI*), des Objektdiagramms (Klasse *AsmLStrucExt*) und der Story Patterns (Klasse *AsmLStructGrr*) generiert. Danach werden die Initialwerte für jede Struktur generiert. Dies geschieht mittels der Init-Klassen (*AsmLInitSI*, *AsmLInitExt*, *AsmLInitExt*). Die Regel *findMatchFor* wird in der Klasse *AsmLMatching* generiert und die Methoden in der Klasse *AsmLMethod*. Schließlich werden durch die Klasse *AsmLRules* die statischen Regeln und Funktionen in den generierten AsmL Code eingefügt.

[7] Es handelt sich hier um das Singelton-Pattern [24].

Jede Diagrammstruktur wird in einer Art Metamodell modelliert. Abbildung 5.2 stellt das Metamodell eines Objektgraphen dar. Für jede Struktur existiert eine Klasse. Die Beziehungen zwischen den Strukturen werden durch Assoziationen zwischen den Klassen modelliert. Strukturen, die einer Klasse des UML Metamodells entsprechen, wie zB. die Struktur *AsmLExtNode*, die der Menge der Objekte des Objektgraphen entspricht, werden durch eine Referenz mit der entsprechenden Klasse des UML Metamodells verbunden. Im Fall der Klasse *AsmLExtNode* wird die Klasse *UMLObject* referenziert. Dadurch werden keine Änderungen in den übrigen Paketen von Fujaba vorgenommen, was den Vorteil hat, dass das Paket *de.uni_paderborn.fujaba.asm* problemlos als Modul implementiert werden kann.

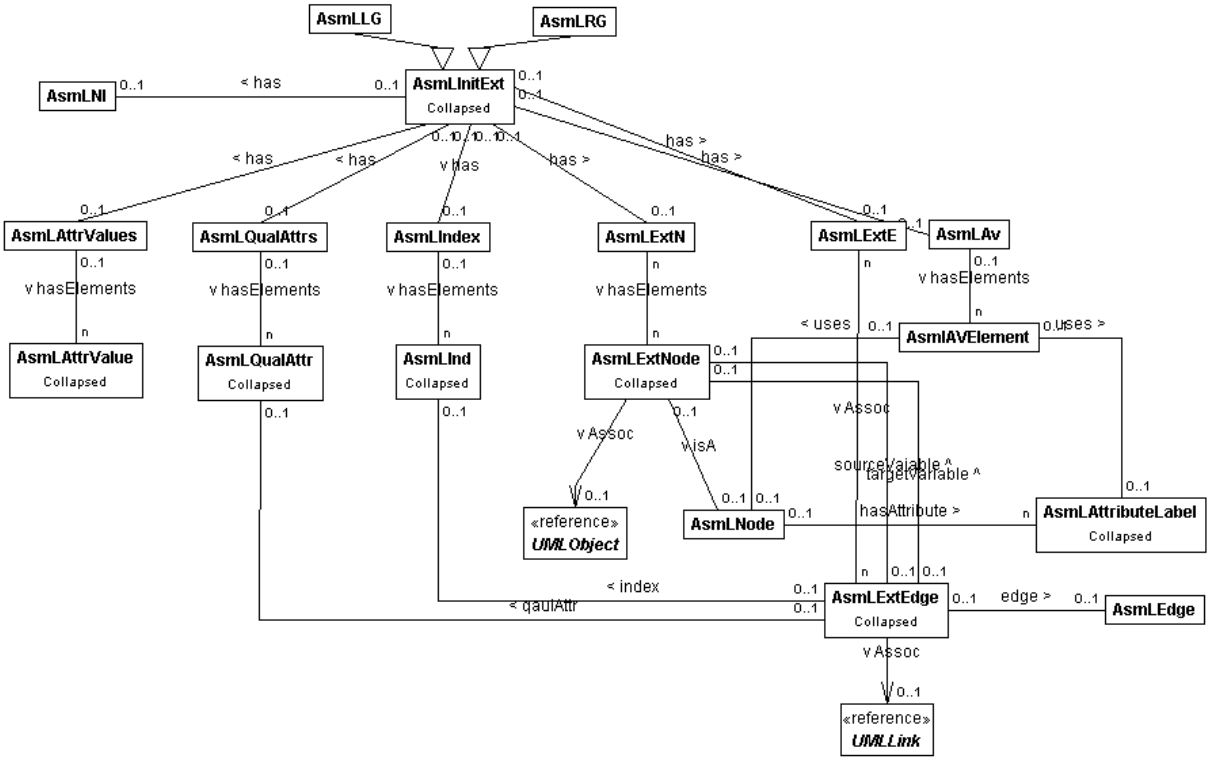

Abbildung 5.2:Metamodell eines Objektgraphen

Um AsmL Code von einem Projekt zu generieren, wird das Werkzeug ASM Creator geladen. Am Anfang muss der Benutzer eine Methode einer Klasse auswählen, die der Anfangsmethode der Spezifikation entspricht. Im Beispiel des Kapitels 4.1 entspricht das der Methode *init*. Danach ist es möglich, den AsmL Code zu generieren und in einem Texteditor anzuzeigen oder in einer Datei zu speichern.

Das Werkzeug AsmL

Der von ASM Creator generierte AsmL Code entspricht einer ausführbaren Spezifikation des Modells. Es ist also möglich, die Spezifikation auszuführen und sie auf Korrektheit zu überprüfen. Das entspricht auch einer Simulation des Modells in früheren Phasen des Entwurfs. Für diesen Zweck wird zusammen mit den in Kapitel 4 beschriebenen Funktionen auch die Funktion *output* zur Verfügung gestellt, die den aktuellen Zustand der Berechnung, also die Werte der Mengen und Abbildungen des Wirtsgraphen, ausgibt.

Das Werkzeug AsmL wird als Add-In für das kommerzielle Programm Visual Studio 6 von Microsoft geliefert. Damit ist es möglich, den generierten Code zu kompilieren und

auszuführen. Bei der Ausführung der Spezifikation wird dafür C++ Code generiert und anschließend ausgeführt. Das bedeutet, dass bestimmte Schlüsselwörter, wie zB. `this` oder `type`, nicht im AsmL Code vorkommen dürfen, da diese Fehler bei der Übersetzung erzeugen könnten. Die Version des Werkzeugs AsmL, mit dem die in dieser Arbeit vorgestellten Konzepte entwickelt worden sind, ist 15v11h2. Die aktuelle Version 15v12d weist mehrere syntaktische Unterschiede zu der älteren Version auf, die aber nicht zu Fehlern bei der Kompilierung führen. Außerdem gibt es ein Werkzeug, mit dem die Ausführung der Spezifikation visualisiert werden kann.

6. Beispielsitzung

In diesem Abschnitt wird anhand des Beispiel aus Kapitel 4.1 gezeigt, wie das in Fujaba integrierte Werkzeug ASM Creator den AsmL Code für das Systemmodell generiert und wie dieser mit dem Werkzeug AsmL kompiliert und ausgeführt wird.

Generierung von AsmL Code

Nachdem das System modelliert wurde, ist es möglich mit dem Aufruf des Werkzeugs ASM Creator (Abbildung 6.1) AsmL Code zu generieren. Das Werkzeug besteht aus dem Menüeintrag *File*, der das Speichern des generierten Codes in einer Datei ermöglicht, dem Menüeintrag *Code*, der das Generieren und Anzeigen des AsmL Codes für das aktuelle Projekt ermöglicht, und dem Menüeintrag *Status*, der das Löschen der Status-Fläche erlaubt. In der Statusfläche wird ein Protokoll der Generierungsprozesse gehalten.

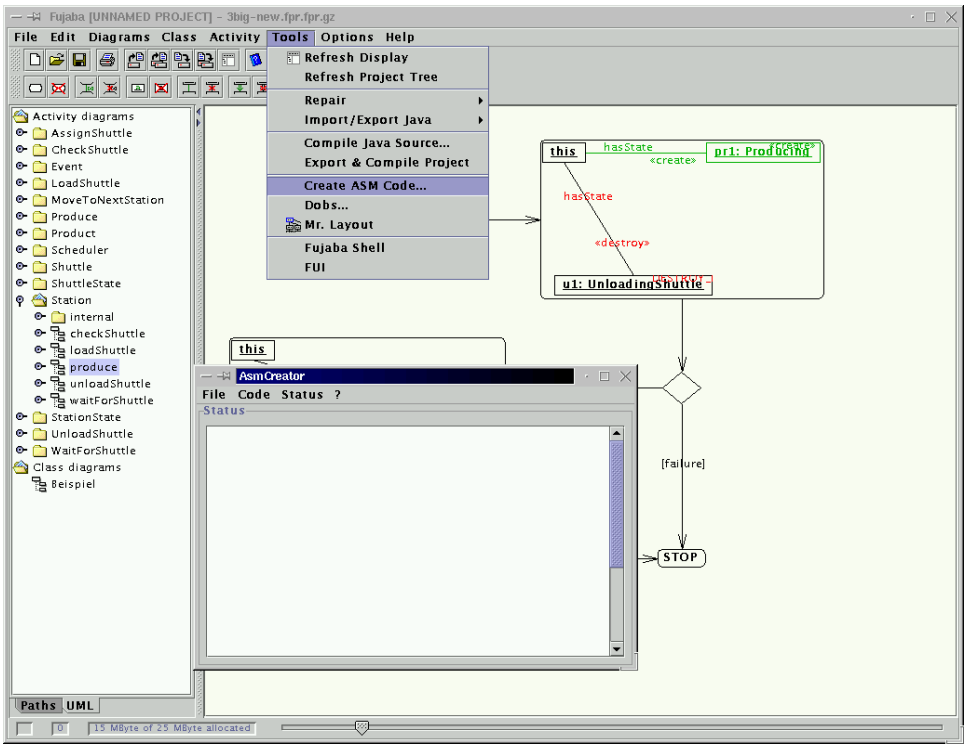

Abbildung 6.1: Die Fujaba Umgebung und das Werkzeug ASM Creator

Wird das Werkzeug zum ersten Mal geladen, so erscheint ein Fenster, in dem der Benutzter die Methode auswählen kann, mit der das Ausführen der Spezifikation beginnt(Start-Methode). Wird keine Methode ausgewählt, so kann die Spezifikation nicht ausgeführt werden. Für das Beispiel des Kapitels 4.1 wird die Methode *init* der Klasse *Scheduler* ausgewählt. Nachdem die Methode ausgewählt wurde, wird der AsmL Code für das aktuelle Projekt generiert. Über den Menüeintrag *Code* ist es möglich den generierten Code anzuzeigen (Abbildung 6.2). Um den generierten Code mit dem Werkzeug AsmL zu kompilieren und auszuführen ist es nötig, ihn in einer Datei zu speichern. Die generierten AsmL Dateien bekommen die Erweiterung *asml*.

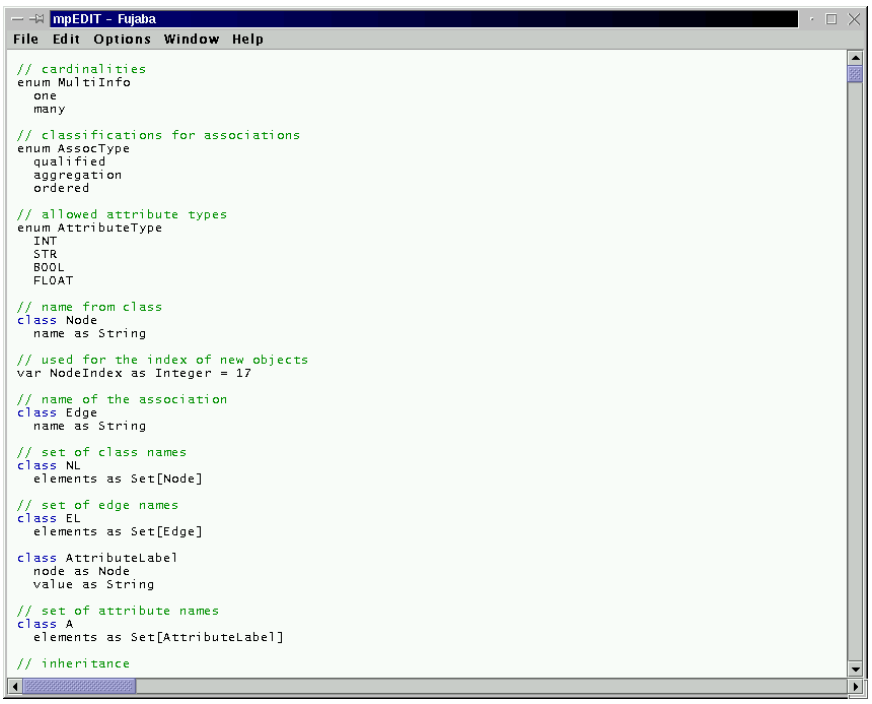

Abbildung 6.2: Der AsmL Code wird mit einem in Fujaba integrierten Editor angezeigt.

Kompilieren und Ausführen

Um einen AsmL Code zu kompilieren und anschließend auszuführen, muss das Werkzeug AsmL an die Entwicklungsumgebung Visual Studio 6.0 gebunden sein [8]. Ist das der Fall, so kann ein neues AsmL Projekt geöffnet werden. Der Name des AsmL Projektes entspricht dem Namen der AsmL Datei, die den Code enthält. Diese kann durch die generierte Datei ersetzt werden. Nun ist es möglich, den AsmL Code zu kompilieren, indem der entsprechende Eintrag im Menü *Erstellen* ausgewählt wird (Abbildung 6.3).

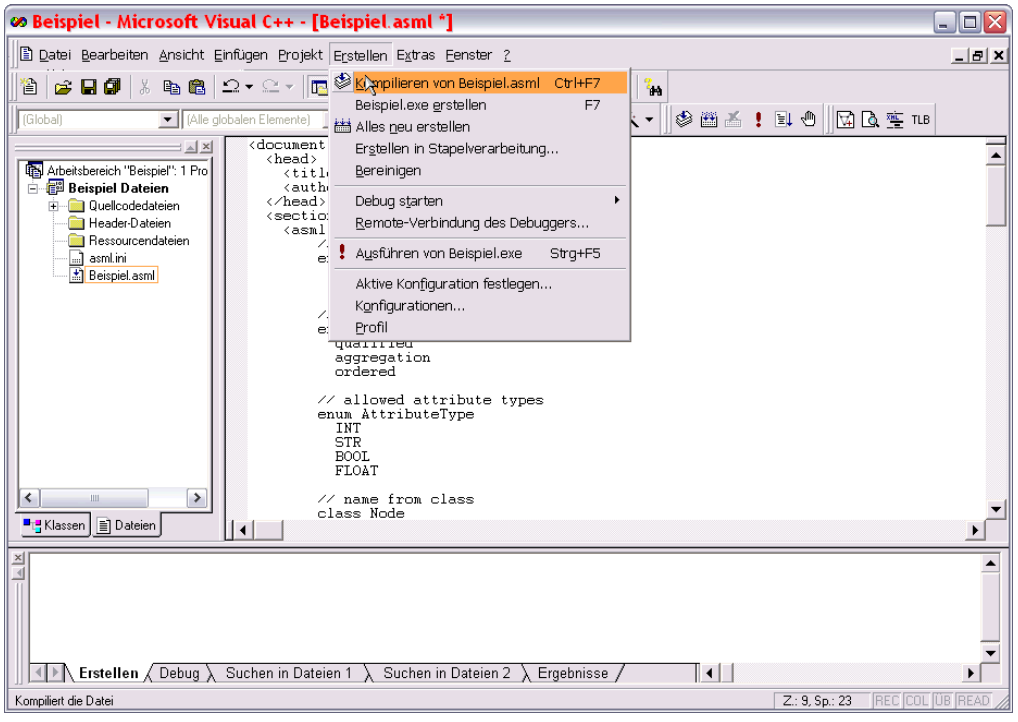

Abbildung 6.3: Die Entwicklungsumgebung von Visual Studio 6.0

Ist die Kompilierung der Spezifikation fehlerfrei, so ist es möglich, die Spezifikation auszuführen, indem der entsprechende Eintrag im Menü *Erstellen* ausgewählt wird. In der Regel gibt es im generierten AsmL Code keine Ausgaben, es wird jedoch die Funktion *output* zur Verfügung gestellt, die eine Ausgabe der Inhalte der Strukturen, die den Wirtsgraphen beschreiben, ermöglicht (Abbildung 6.4). Dabei soll der Benutzer den generierten Code entsprechend modifizieren. Für das Beispiel des Kapitels 4.1 würde das eine Änderung der Methode *run* bedeuten.

```
01 run() =
02   machine
03     let EG = extEG
04     let S = classSI
05   step
06     output(EG, “”)
07     void_Scheduler_init(EG, S, startExtnode)
08   step
09     output(EG, “”)
10     void_Scheduler_executeEvent(EG, S, startExtnode)
11   step
02     output(EG, “”)
13     …
```

In Zeile 6 wird die Funktion *output* mit dem Wirtsgraphen EG als Parameter ausgeführt. Die Ausgabe zeigt die initialen Werte des Wirtsgraphen an. In diesem Zustand der Berechnung ist das einzige Objekt im Wirtsgraphen eine Instanz der Klasse, die die Start-Methode enthält. Im gleichen Zustand wird auch die Start-Methode aufgerufen, die in diesem Fall die Methode *init* der Klasse *Scheduler* ist. Im nächsten Zustand wird die Funktion *output* erneut aufgerufen. Die Ausgabe für diesen Fall zeigt die Abbildung 6.4.

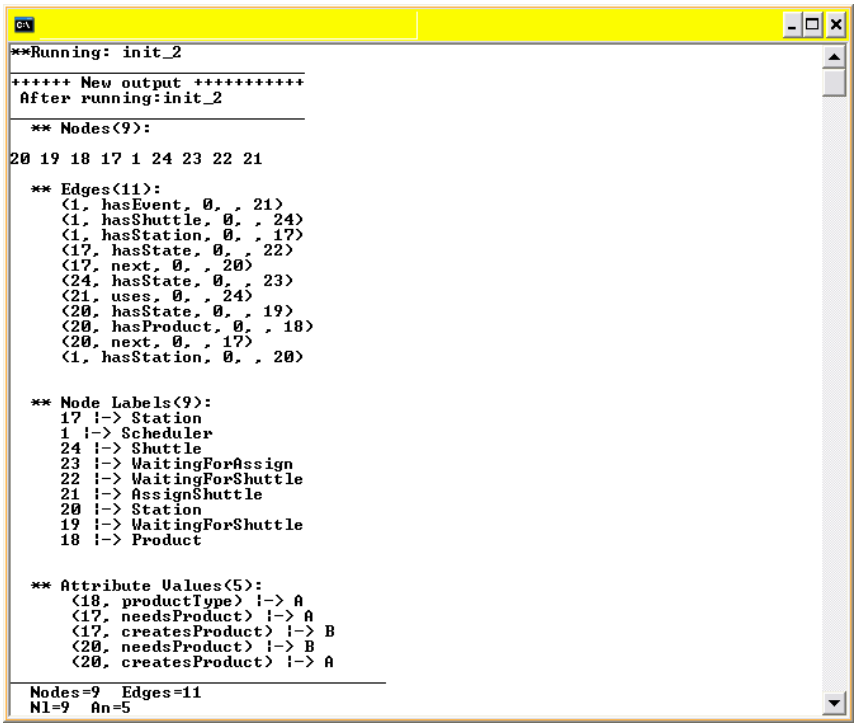

Abbildung 6.4: Textbasierte Ausgabe der Funktion *output*

Weiterhin kann der Benutzter die Funktion *run* modifizieren, indem er die Methode *executeEvent* der Abbildung 4.20 in neuen Zuständen aufruft (ab Zeile 10). Jeder Zustand der

Berechnung gibt so eine neue Ausgabe des Wirtsgraphen, und es ist möglich, schrittweise das Ausführen der Spezifikation zu betrachten.

Wie schon erwähnt, stellt die neue Version des Werkzeugs AsmL eine graphische Oberfläche zur Verfügung, mit der die Berechnung einer ausführbaren Spezifikation in AsmL visualisiert werden kann. Aufgrund mangelnder Dokumentation war es jedoch nicht möglich von dieser Option Gebrauch zu machen.

7. Zusammenfassung und Ausblick

In dieser Arbeit wurde die formale Verifikation im Entwicklungsprozess eines Systems betrachtet. Dabei wurde das System, wie in Kapitel 2 beschrieben, durch die Benutzung von verschiedenen Abstraktionsebenen auf einem ASM Modell abgebildet, welches als Zwischenschritt für die Anbindung an einem Model-Checker diente. Ziel dieser Arbeit war die Abbildungen von UML Klassendiagramme, Aktivitätsdiagramme und der Semantik der Story Patterns auf ASM. Diese Abbildungen definieren eine ASM Spezifikation, die ein Teil der niedrigste Abstraktionsebene des ASM Modells ist. In Kapitel 3 wurde eine Einführung in ASM und das Werkzeug AsmL gegeben, indem die wichtigsten Begriffe und Eigenschaften von ASM und die zu dieser Arbeit verwendete Teilbereiche der Sprache AsmL erläutert wurden. In Kapitel 4 wurden die Konzepte der Abbildung von UML Spezifikationen auf ASM beschrieben. Zunächst wurde die Abbildung der statischen Eigenschaften des Systems betrachtet, die durch das Klassendiagramm modelliert werden. Die Methoden der Klassen beschreiben das Verhalten des Systems. Da eine Methode durch den Einsatz von Story Patterns und eines Aktivitätsdiagramms modelliert wird, wurden zuerst die Story Patterns und anschließend die Aktivitätsdiagramme betrachtet. Nachdem die Abbildung der Semantik der Story Patterns und der Aktivitätsdiagramme auf ASM beschrieben wurde, wurde ein Konzept vorgestellt, wie aus einer Methode des Klassendiagramms eine ASM Funktion generiert werden kann. Die technische Umsetzung wurde in Kapitel 5 beschrieben. Dabei wurde das Werkzeug ASM Creator vorgestellt. Schließlich wurde in Kapitel 6 eine Beispielsitzung beschrieben.

In den folgenden Abschnitten werden drei Themen betrachtet: Im ersten Abschnitt wird eine mögliche Abbildung von Java Code auf ASM betrachtet. Im zweiten Abschnitt wird die Realisierung der Abstraktionsebenen mit ASM Metamodellen näher betrachtet. Schließlich wird im letzten Abschnitt die Anbindung an den Model-Checker SMV beschrieben.

Unterstützung von Java Ausdrücken

Wie bereits erwähnt, ist es möglich, bei der Modellierung des Systems Java Ausdrücke zu verwenden, z.B. bei den Kollaborations-Nachrichten oder Attributbedingungen. Weiterhin kann in einer Aktivität eines Aktivitätsdiagramms anstelle eines Story Patterns direkt Java Code angegeben werden. Voraussetzung für die Unterstützung solcher Ausdrücke im ASM Modell ist die Abbildung von Java Code auf ASM. In [6] wird die Semantik von Java durch die Spezifikation eines Java Interpreters in ASM beschrieben. Dabei wird die Benutzung einer Klasse in drei Phasen unterteilt. In der ersten Phase (parsing) wird die Klasse eingelesen und ein abstrakter Syntaxbaum konstruiert. In der zweiten Phase (statische Phase) findet die Überprüfung der syntaktischen Korrektheit der Klasse statt. Schließlich wird in der dritten Phase (dynamische Phase) der Code der Klasse geladen und ausgeführt, indem Ausdrücke bewertet werden und die Laufzeitobjektstruktur verändert wird. Die dynamische Semantik von Java, die der dynamischen Phase entspricht, wird in fünf Teilsprachen unterteilt, die jeweils durch eine Erweiterung der letzten gebildet werden. Die erste Sprache umfasst die Anweisungen und Ausdrücke der primitiven Typen und stellt den notwendigen Kern dar. Die zweite Sprache enthält die Java Klassen und definiert so eine objektbasierte Teilsprache von Java. Als nächstes werden die objektorientierten Konzepte von Java betrachtet. Die vierte Sprache unterstützt das Exception Konzept. Schließlich wird durch die fünfte Sprache die Nebenläufigkeit in Java unterstützt. Weiterhin wird in [5] eine Spezifikation der Java Virtual Machine (JVM) in ASM gegeben. Dabei werden die Konzepte aus [6] weiterverwendet.

Abstraktionsebenen mit ASM Metamodellen

In Kapitel 2 wurde beschrieben, wie ein ASM Modell aus der Systemspezifikation mit Hilfe von ASM Metamodellen erzeugt wird. In Abbildung 2.1 wird das Erzeugen dieses Modells dargestellt. Im folgenden wird dieser Vorgang etwas detaillierter beschrieben. Wie bereits erwähnt, werden aus der Modellspezifikation eine Kodierung des Systems in Form von ASM Datenstrukturen und Funktionen generiert. Für ein Statechart z.B. könnten diese die Menge der Zustände, die Menge der Transitionen und die Menge der Ereignisse sein. Die Kodierung des Systems wird mit einem ASM Metamodell gebunden, welches als Interpreter dieser Daten zu verstehen ist. Das ASM Metamodell bildet zusammen mit der Kodierung des Systems das ASM Modell der Systemspezifikation.

Ein ASM Metamodell [25] entspricht der Abstraktionsebene des ASM Modells. In der höchsten Abstraktionsebene, die dem ASM Metamodell I entspricht, werden Prozesse als eine Black-Box betrachtet. In Tabelle 6.1 werden Beispiele von Abstraktionen von ASM Metamodellen dargestellt. Solche Abstraktionen werden durch den `choose`-Ausdruck, der einer nichtdeterministischen Auswahl entspricht, realisiert. In der linken Spalte wählt die Regel *selectEvent* nichtdeterministisch ein Element aus der Menge *acceptable_events* aus, die alle Ereignisse beinhaltet, die der Prozess akzeptiert. Weiterhin werden, um möglichst viele Implementierungsdetails auszublenden, Bedingungen durch die Regel *evalGuard* mittels einer nichtdeterministischen Auswahl aus der Menge `{true, false}` ausgewertet. Die Regel *executeAction*, die eine Aktion ausführt, entspricht der leeren Regel `skip`. Das Ergebnis ist ein ASM Metamodell, das die Semantik eines Statecharts beschreibt, während alle Implementierungsdetails der Ereignissen, Bedingungen und Aktionen nicht betrachtet werden.

ASM Metamodell I

```
// evaluate specified quard
evalGuard(g as guard) as Boolean =
  choose v in {true, false}

// select event for dispatching
// choose any possible one
selectEvent() as Event =
  choose e in acceptable_events

// execute specified action
executeAction(a as Action) =
  skip
```

ASM Metamodell II

```
// evaluate specified quard
evalGuard(g as guard) as Boolean =
  choose v in {true, false}

// select event for dispatching
// take first event (FIFO queue)
selectEvent() as Event =
  a.events: = tail(a.events)
  return a.events(0)

// execute specified action
executeAction(a as Action) =
  skip
```

Tabelle 6.1: Beispiel von Abstraktionen in den ASM Metamodellen I und II

Auf dieser sehr hohen Abstraktionsebene kann nur eine kleine Menge von Eigenschaften des Systems verifiziert werden, doch sie bildet eine gute Grundlage, um weitere Verfeinerungen in Betracht zu ziehen, die den ASM Metamodellen niedrigerer Abstraktionsebenen entsprechen. In der rechten Spalte der Tabelle 6.1 werden Abstraktionen des ASM Metamodells II dargestellt. Hier wird in der Regel *selectEvent* das erste Element der Warteschlange ausgewählt und zurückgegeben. Dies ist eine Verfeinerung im Vergleich zum ASM Metamodell I, da konkret angegeben wird, welches Element ausgewählt wird und gleichzeitig gilt, dass dieses Element in der Menge *acceptable_events* enthalten ist.

In dieser Art wird das grundlegende ASM Metamodell weiter verfeinert, indem alle Implementierungsdetails in Betracht gezogen werden, wie Objekte und ihre Hierarchie, dieSemantik der Ausführung von Aktionen, die Auswertung von Bedingungen mittels Attributen von Objekten und ihren Werten u.s.w. Alle diese Eigenschaften erweitern die ASM Spezifikation um neue Strukturen und Regeln. Die niedrigeren Abstraktionsebenen beinhalten also mehrere Details des Modells. Die in dieser Arbeit beschriebenen Konzepte werden in den tieferen Ebenen der Abstraktion benutzt, da sie sehr detailliert die Implementierung des Systems beschreiben. Das Ergebnis ist eine Menge von ASM Metamodellen, die die Verifikation des Systems in verschiedenen Abstraktionsebenen erlauben, indem jeweils eine bestimmte Menge von Eigenschaften betrachtet wird.

Anbindung an den Model-Checker

Wie bereits in 3.1.6 erwähnt, bietet das Werkzeug ASM Workbench eine Anbindung an den Model-Checker SMV [10]. In diesem Kapitel wird beschrieben, welche Probleme es bei der Transformation des ASM Modells in das Model-Checker Eingabeformat gibt (vgl. Abbildung 1.1). Im Allgemeinen ist die formale Verifikation mit ASM nicht möglich, da ASM Programme Transitionssysteme mit unendlich vielen Zuständen erlauben, während Model-Checker nur mit Systemen mit endlich vielen Zuständen arbeiten können. Aus diesem Grund kann die Korrektheit des gesamten Modells nicht durch die formale Verifikation mittels Model-Checking überprüft werden. Diese bietet aber einen viel effizienteren Weg, das System zu überprüfen, als die Validierung des Systems mittels Simulation. Es handelt sich um eine systematische Überprüfung, die in den früheren Phasen des Entwurfs sehr hilfreich sein kann.

Der Model-Checker SMV benutzt die Technik *symbolic model checking*. Bei dieser Technik wird die Menge der Zustände und Transitionen als BDDs (*Binary Decision Diagrams*) repräsentiert. Dadurch wird die Berechnung des Model-Checkers effizienter und erlaubt auch das Testen von größeren Systemen. Die Eingabe für das Werkzeug SMV wird in einer eigenen Sprache geschrieben, die SMV Sprache heißt. Die Eingabe besteht aus der Modellspezifikation und aus Constraints, die bei der Verifikation auf Korrektheit überprüft werden. Die Ausgabe ist entweder eine Aussage der Art *„specification S ist true“* oder ein Berechnungsweg, der der Spezifikation nicht entspricht.

ASM Spezifikationen sind, wie auch die SMV Modellbeschreibungen, Spezifikationen von Transitionssystemen. Trotzdem unterscheiden sich ASM von SMV in vier wichtigen Punkten:

1. Da ASM in der Modellspezifikation das Definieren von Strukturen mit unendlich vielen Elementen erlauben, können ASM Systeme mit unendlich vielen Zuständen definieren. Im Gegensatz dazu können Model-Checkers nur Transitionssysteme verifizieren, die endlich viele Zustände haben, und bei denen jeder Zustand eine endliche Menge von Variablen enthält, die wiederum Werte aus endlichen Mengen enthalten.

2. Die Spezifikation von Transitionen in ASM und SMV ist verschieden. In SMV werden die Transitionen mittels `next`-Ausdrücken definiert. Ein `next`-Ausdruck definiert den Wert einer Zustandsvariable für den nächsten Zustand. In ASM dagegen ist es möglich, mehrere Aktualisierungen einer Variable in einem Zustand zu definieren, oder auch keine, was bedeuten würde, dass die Variable ihren Wert nicht ändert.

3. Wie bereits in 3.1.2 erwähnt, ist eine Zustandsvariable in ASM eine Funktion ohne Parameter. In ASM werden also anstelle von Zustandsvariablen, wie in SMV, Funktionen benutzt.

4. ASM bietet externe Funktionen (3.1.4) an, die einen Einfluss der Umgebung spezifizieren. Solche Funktionen machen die Berechnung nichtdeterministisch. In SMV existieren keine Konstrukte, die eine nichtdeterministische Wahl beschreiben.

Das in Punkt 1 beschriebene Problem wird durch die Einführung von *finiteness-constraints* gelöst. Für jede dynamische oder externe Funktion muss der Benutzer ein *finiteness-constraint* definieren, welches den Wertebereich der Funktion auf eine endliche Menge beschränkt, die auch durch Parameter der Funktion definiert werden darf. Für externe Funktionen gelten die *finiteness-constraints* als Annahmen, während sie für dynamische als Anforderungen an das Verhalten des Systems gelten. Die Punkte 2 und 3 werden durch ein Übersetzungsschema gelöst, welches als nächstes beschrieben wird. Schließlich wird die Modellierung von Nichtdeterminismus (Punkt 4) nicht direkt unterstützt. Trotzdem ist es möglich, den `choose`-Ausdruck durch externe Funktion zu ersetzen und so eine äquivalente Struktur zu erreichen.

In [10] wird ein Übersetzungsschema vorgestellt, um ein ASM-Modell in eine SMV Eingabe zu transformieren. Dabei wird versucht, die oben genannten Unterschiede zu überbrücken, indem eine Teilmenge ASM_0 von ASM gebildet wird, die SMV sehr ähnlich ist. Im Allgemeinen besteht ein SMV-Programm aus einer Menge von Modulen. Jedes Modul besteht aus Deklarationen, Zuweisungen und der Spezifikation. Im Bereich der Deklarationen, der durch das Schlüsselwort *VAR* angegeben wird, werden die Zustandsvariablen und ihre Wertbereiche deklariert. Im Bereich der Zuweisungen, der durch das Schlüsselwort *ASSIGN* angegeben wird, werden Zuweisungen zu den Zustandsvariablen deklariert. Es gibt zwei Arten von Zuweisungen. Die *init*-Zuweisungen bestimmen den Wert einer Zustandsvariablen im initialen Zustand. Die Menge der *init*-Zuweisungen beschreibt also den initialen Zustand. Die *next*-Zuweisungen bestimmen den Wert einer Zustandsvariablen für den nächsten Zustand, abhängig von ihrem Wert im aktuellen Zustand. Schließlich werden im Bereich der Spezifikation, der durch das Schlüsselwort *SPEC* angegeben wird, CTL Eigenschaften definiert, die für dieses Modul überprüft werden müssen.

In ASM_0 werden nur Methoden erlaubt, die keine Parameter und nur eine kleine Anzahl von statischen Funktionen haben. Die dynamischen und externen Funktionen, die keine Parameter enthalten, können also auf SMV Zustandsvariablen abgebildet werden. Im Bereich der Deklarationen (*VAR*) werden also einfach die Zustandsvariablen mit Wertebereichen deklariert, die durch die *finiteness-constraints* gegeben sind. Als nächstes wird der Bereich der Zuweisungen (*ASSIGN*) definiert. Aus jeder dynamischen Funktion werden ein init- und ein `next`-Ausdruck generiert. Der `init`-Ausdruck wird durch die ASM-Initialisierung der Funktion bewertet. Der `next`-Ausdruck wird in zwei Schritten erzeugt. Im ersten Schritt wird das ASM-Programm in einem Block von bewachten Aktualisierungen anhand vorgegebener Regeln transformiert. Im zweiten Schritt werden alle bewachten Aktualisierungen einer Funktion gesammelt und im `next`-Ausdruck dieser Funktion geschrieben. Dabei wird darauf geachtet, dass keine Konflikte entstehen.

Durch dieses Übersetzungsschema kann nur eine begrenzte Menge von ASM Programmen in SMV übersetzt werden. In [10] wird dieses Schema erweitert, so dass fast jedes endliche ASM Programm in SMV übersetzt werden kann. Das einzige Element, das nicht übersetzt werden kann, ist der `choose`-Ausdruck, für den kein entsprechendes Konstrukt in SMV existiert. Trotzdem ist es, wie bereits erwähnt, möglich, die `choose`-Ausdrücke durch externe Funktionen zu ersetzen.

Die Übersetzung des Beispiels 4.1 in SMV ist theoretisch machbar. Die Sprache AsmL erlaubt die Definition von unendlichen Mengen nicht, d.h. ein AsmL Programm hat eine endliche Anzahl von Zuständen. Somit ist das im ersten Punkt beschriebene Problem der Inkompatibilität gelöst. Die Punkte 2 und 3 können durch das Übersetzungsschema gelöst werden. Schließlich gibt es noch das Problem der `choose`-Ausdrücke. Diese Ausdrücke werden bei der Regel *findMatchFor* benutzt, um aus einer Menge von Objekten des Wirtsgraphen, die mit einer gegebenen Variablen gebunden werden können, nur eins auszuwählen.

Literaturverzeichnis

[1] Yuri Gurevich*:* Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*. Oxford University Press, 1995.

[2] Yuri Gurevich.: *May 1997 Draft of the ASM Guide*. CSE. Technical Report CSE-TR-336-97, EECS Department, University of Michigan-Ann Arbor, 1997.

[3] Yuri Gurevich.: *Sequential Abstract State Machines Capture Sequential Algorithms*. ACM Transactions on Computational Logic, vol. 1, no. 1, July 2000, 77-111. © Copyright 2000 by ACM, Inc..

[4] Yuri Gurevich. Evolving Algebras.: *A Tutorial Introduction.* Bulletin of EATCS, 43:264-284, 1991.

[5] Egon Börger, Wolfram Schulte.: *Defining the Java Machine as Platform for Provably Correct Java Compilation.* In J. Alves-Foss, editor, Formal Syntax and Semantics of Java(tm), Springer LNCS, to appear. 1998.

[6] E. Börger and W. Schulte.: *A programmer friendly modular definition of the semantics of Java.* In J. Alves-Foss, ed., "Formal Syntax and Semantics of Java", Springer LNCS 1523, 1998.

[7] Yuri Gurevich, Wolfram Schulte, Colin Campbell, Wolfgang Grieskamp. AsmL: *The Abstract State Machine Language Version 1.5* (2001)
http://research.microsoft.com/foundations/AsmL/default.html

[8] Foundations of Software Engineering (FSE) Microsoft Research. *An Introduction to ASML 1.5* (2001)
http://research.microsoft.com/foundations/AsmL/default.html

[9] http://www.microsoft.com/fse

[10] Giuseppe Del Castillo. *Towards Comprehensive Tool Support for Abstract State Machines: The ASM Workbench Tool Environment and Architecture.* In: D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, eds., Applied Formal Methods -- FM-Trends 98, Springer LNCS 1641, 1999, 311--325.

[11] Hans J. Köhler, Ulrich Nickel, Jörg Niere, Albert Zündorf.: *Integrating UML Diagrams for Production Control Systems*. In Proc. Ofthe 22th International Conference on Software Engineering, Limerick, Ireland. ACM Press, 2000.

[12] Holger Giese, Martin Kardos, Ulrich Nickel.: *Integrating Verification in a Design Process for Distributed Production Control Systems.* Second International Workshop on Integration of Specification Techniques for Applications in Engineering (INT 2002). Grenoble, France, April 2002.

[13] Zündorf A.: *Rigorous Object Oriented Software Development*; Habilitation Thesis, University of Paderborn (2001) (1996)

[14] Kan-Hung Wan, Kan-Sin Wan.: *Java Codegenerierung für Negative-Knoten und Multilinks einer Grapgersetzungsregel.* University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Studienarbeit, August 2001

[15] Thorsten Fischer, Jörg Niere, Lars Torunski.: *Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling* University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, July 1998.

[16] Thomas Klein.: *Rekonstruktion von UML-Aktivitäts- und Kollaborationsdiagrammen aus UML-Quelltexten.* University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit, Oktober 1999

[17] From UML to JAVA and Back Again (FUJABA): http://www.fujaba.de

[18] Hans Josef Köhler.: *Codegenerierung für UML-Collaborations-, -Sequenz- und -Statechart-Diagramme.* University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany, Diplomarbeit

[19] OMG Unified Modeling Language Specification Version 1.3 (1999)

[20] Gabriel Valiente, Conrado Matrinez.: *An algorithm for graph pattern-matching.* In Proc. Fourth South American Workshop on String Processing, volume 8 of International nformatics Series, Carleton University Press (1997), pp. 180-197.

[21] K. Mehlhorn: *Graph Algorithms and NP-Completeness.* Springer-Verlag, 1994

[22] Albert Zündorf.: *Graph Pattern Matching in PROGRESS.* in: J. Cuny, H. Ehrig, G. Engels, G. Rozenberg (eds.): Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science, Willamsburg, November 1994, Springer, LNCS 1073 (1996)

[23] Albert Zündorf.: *A Heuristic for the Subgraph Isomorphism Problem in Executing PROGRESS.* Technical Report AIB 93-5, RWTH Aachen, Germany (1993)

[24] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.: *Design Patterns.* Addison-Wesley, 1995.

[25] Holger Griese, Martin Kardos, Ulrich Nickel: *Towards Design Verification and Validation at Multiple Levels of Abstraction*

[26] Abstract State Machines: http://www.eecs.umich.edu/gasm/

Anhang

Initialisierungen

Initialisierung des Klassendiagramms (Abbildung 4.1):

```
01 classNode0 = new Node("Scheduler")
02 classNode1 = new Node("Event")
03 classNode2 = new Node("Station")
04 classNode3 = new Node("Shuttle")
05 classNode4 = new Node("ShuttleState")
06 classNode5 = new Node("StationState")
07 classNode6 = new Node("Product")
08 classNode7 = new Node("AssignShuttle")
09 classNode8 = new Node("LoadShuttle")
10 classNode9 = new Node("UnloadShuttle")
11 classNode10 = new Node("Produce")
12 classNode11 = new Node("WaitingForAssign")
13 classNode12 = new Node("DockedToStation")
14 classNode13 = new Node("WaitingForShuttle")
15 classNode14 = new Node("Producing")
16 classNode15 = new Node("LoadingShuttle")
17 classNode16 = new Node("UnloadingShuttle")
18 classNode17 = new Node("MovingToNextStation")
19 classNode18 = new Node("MoveToNextStation")
20 classNode19 = new Node("CheckShuttle")
21 classNode20 = new Node("CheckingShuttle")
22 classNode21 = new Node("WaitForShuttle")
23 classNL = new NL({classNode0, classNode1, classNode2, classNode3,
                    classNode4, classNode5, classNode6, classNode7,
                    classNode8, classNode9,  classNode10, classNode11,
                    classNode12, classNode13, classNode14,  classNode15,
                    classNode16, classNode17, classNode18, classNode19,
                    classNode20, classNode21})
24 classEdge0 = new Edge("hasState")
25 classEdge1 = new Edge("hasState")
26 classEdge2 = new Edge("hasShuttle")
27 classEdge3 = new Edge("hasStation")
28 classEdge4 = new Edge("isIn")
29 classEdge5 = new Edge("carries")
30 classEdge6 = new Edge("next")
31 classEdge7 = new Edge("uses")
32 classEdge8 = new Edge("uses")
33 classEdge9 = new Edge("uses")
34 classEdge10 = new Edge("uses")
35 classEdge11 = new Edge("hasEvent")
36 classEdge12 = new Edge("hasProduct")
37 classEdge13 = new Edge("uses")
38 classEdge14 = new Edge("uses")
39 classEdge15 = new Edge("goesTo")
40 classEdge16 = new Edge("uses")
41 classEL = new EL({classEdge0, classEdge1, classEdge2, classEdge3,
                   classEdge4, classEdge5, classEdge6, classEdge7,
                   classEdge8, classEdge9, classEdge10, classEdge11,
                   classEdge12, classEdge13, classEdge14, classEdge15,
                   classEdge16})
42 classAttributeLabel0 = new AttributeLabel(classNode2, "createsProduct")
43 classAttributeLabel1 = new AttributeLabel(classNode2, "needsProduct")
```

```
44 classAttributeLabel2 = new AttributeLabel(classNode6, "productType")
45 classA = new A({classAttributeLabel0, classAttributeLabel1,
                   classAttributeLabel2})
46 classIsAs = new IsAs({(classNode7, classNode1), (classNode19,
                          classNode1), (classNode20, classNode5),
                         (classNode12, classNode4), (classNode8,
                          classNode1), (classNode15,  classNode5),
                         (classNode18, classNode1), (classNode17,
                          classNode4), (classNode10, classNode1),
                         (classNode14, classNode5), (classNode9,
                          classNode1), (classNode16, classNode5),
                         (classNode21, classNode1), (classNode11,
                          classNode4), (classNode13, classNode5)})
47 classAssocs = new Assocs({
       classEdge0 |-> (classNode3, one, {}, classNode4, one),
       classEdge1 |-> (classNode2, one, {}, classNode5, one),
       classEdge2 |-> (classNode0, one, {}, classNode3, many),
       classEdge3 |-> (classNode0, one, {}, classNode2, many),
       classEdge4 |-> (classNode3, one, {}, classNode2, one),
       classEdge5 |-> (classNode3, one, {}, classNode6, one),
       classEdge6 |-> (classNode2, one, {}, classNode2, one),
       classEdge7 |-> (classNode8, one, {}, classNode2, one),
       classEdge8 |-> (classNode7, one, {}, classNode3, one),
       classEdge9 |-> (classNode9, one, {}, classNode2, one),
       classEdge10 |-> (classNode10, one, {}, classNode2, one),
       classEdge11 |-> (classNode0, one, {}, classNode1, many),
       classEdge12 |-> (classNode2, one, {}, classNode6, many),
       classEdge13 |-> (classNode18, one, {}, classNode3, one),
       classEdge14 |-> (classNode19, one, {}, classNode2, one),
       classEdge15 |-> (classNode3, one, {}, classNode2, one),
       classEdge16 |-> (classNode21, one, {}, classNode2, one)})
49 classSI = new SIGraph(classNL, classEL, classA, classIsAs, classAssocs,
                         classAttrs)
48 classAttrs = new Attrs({classAttributeLabel0 |-> STR,
                           classAttributeLabel1 |-> STR,
                           classAttributeLabel2 |-> STR})
```

Initialisierung des Story Patterns der *init* Methode (Abbildung 4.4):

```
01 // Nodes
02 startExtnode = new ExtNode("1")
03 // extN
04 extN = new ExtN({ startExtnode })
05 // Ind
06 extInd = new Ind(0)
07 // Index
08 extIndex = new Index({ extInd })
09 // QualAttr
10 extQualAttr = new QualAttr("")
11 // QualAttrs
12 extQualAttrs = new QualAttrs({ extQualAttr })
13 // Edges
14 // ExtE
15 extE = new ExtE({ })
16 // Nl
17 extNl = new Nl({ startExtnode |-> classNode0 })
18 // AttrValue
19 // AttrValues
20 // Av
21 extAv = new Av({ |-> })
22 // ExtGraph
```

```
23 extEG = new ExtGraph(extN, extE, extNl, extAv, extIndex, extQualAttrs)
24 //ExtNode
25 init_2RGNode0 = new ExtNode("w1")
26 init_2RGNode1 = new ExtNode("w2")
27 init_2RGNode2 = new ExtNode("a1")
28 init_2RGNode3 = new ExtNode("st1")
29 init_2RGNode4 = new ExtNode("w3")
30 init_2RGNode5 = new ExtNode("p1")
31 init_2RGNode6 = new ExtNode("st2")
32 init_2RGNode7 = new ExtNode("sh")
33 //ExtN
34 init_2RGN = new ExtN({init_2RGNode0, init_2RGNode1, init_2RGNode2,
                        init_2RGNode3, init_2RGNode4, init_2RGNode5,
                        init_2LGNode0, init_2RGNode6, init_2RGNode7})
35 // Ind
36 init_2RGInd0 = new Ind(0)
37 // Index
38 init_2RGIndex = new Index({init_2RGInd0})
39 // QualAttr
40 init_2RGQualAttr0 = new QualAttr("")
41 // QualAttrs
42 init_2RGQualAttrs = new QualAttrs({init_2RGQualAttr0})
43 // ExtEdge
44 init_2RGExtEdge0 = new ExtEdge(init_2RGNode2, classEdge8, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode7)
45 init_2RGExtEdge1 = new ExtEdge(init_2RGNode3, classEdge1, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode4)
46 init_2RGExtEdge2 = new ExtEdge(init_2RGNode3, classEdge12, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode5)
47 init_2RGExtEdge3 = new ExtEdge(init_2RGNode3, classEdge6, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode6)
48 init_2RGExtEdge4 = new ExtEdge(init_2LGNode0, classEdge3, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode3)
49 init_2RGExtEdge5 = new ExtEdge(init_2LGNode0, classEdge11, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode2)
50 init_2RGExtEdge6 = new ExtEdge(init_2LGNode0, classEdge2, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode7)
51 init_2RGExtEdge7 = new ExtEdge(init_2LGNode0, classEdge3, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode6)
52 init_2RGExtEdge8 = new ExtEdge(init_2RGNode6, classEdge1, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode1)
53 init_2RGExtEdge9 = new ExtEdge(init_2RGNode6, classEdge6, init_2RGInd0,
                                 init_2RGQualAttr0, init_2RGNode3)
54 init_2RGExtEdge10 = new ExtEdge(init_2RGNode7, classEdge0, init_2RGInd0,
                                  init_2RGQualAttr0, init_2RGNode0)
55 // ExtE
56 init_2RGE = new ExtE({init_2RGExtEdge0, init_2RGExtEdge1,
       init_2RGExtEdge2, init_2RGExtEdge3, init_2RGExtEdge4,
       init_2RGExtEdge5, init_2RGExtEdge6, init_2RGExtEdge7,
       init_2RGExtEdge8, init_2RGExtEdge9, init_2RGExtEdge10})
57 // Nl
58 init_2RGNl = new Nl({init_2RGNode0 |-> classNode11, init_2RGNode1 |->
            classNode13, init_2RGNode2 |-> classNode7, init_2RGNode3 |->
            classNode2, init_2RGNode4 |-> classNode13, init_2RGNode5 |->
            classNode6, init_2LGNode0 |-> classNode0, init_2RGNode6 |->
            classNode2, init_2RGNode7 |-> classNode3})
59 // AttrValue
60 init_2RGAttrValue0 = new AttrValue("leeresGlass")
61 init_2RGAttrValue1 = new AttrValue("GlassVollBier")
62 init_2RGAttrValue2 = new AttrValue("leeresGlass")
63 init_2RGAttrValue3 = new AttrValue("GlassVollBier")
64 init_2RGAttrValue4 = new AttrValue("leeresGlass")
65 // Av
```

```
66 init_2RGAv = new Av({(init_2RGNode3, classAttributeLabel0) |->
    init_2RGAttrValue0, (init_2RGNode3, classAttributeLabel1) |->
    init_2RGAttrValue1, (init_2RGNode5, classAttributeLabel2) |->
    init_2RGAttrValue2, (init_2RGNode6, classAttributeLabel0) |->
    init_2RGAttrValue3, (init_2RGNode6, classAttributeLabel1) |->
    init_2RGAttrValue4})
67 //AttrValues
68 init_2RGAttrValues = new AttrValues({init_2RGAttrValue0,
                   init_2RGAttrValue1, init_2RGAttrValue2,
                   init_2RGAttrValue3, init_2RGAttrValue4})
69 init_2RGGraph = new ExtGraph(init_2RGN, init_2RGE, init_2RGNl,
                   init_2RGAv, init_2RGIndex, init_2RGQualAttrs)
70 // ****************************************
71 //DelN
72 init_2DelN = new DelN({})
73 //DelE
74 init_2DelE = new DelE({})
75 //CodeN
76 init_2CoreN = new CoreN({init_2LGNode0})
77 //AddN
78 init_2AddN = new AddN({init_2RGNode0, init_2RGNode1, init_2RGNode2,
                         init_2RGNode3, init_2RGNode4, init_2RGNode5,
                         init_2RGNode6, init_2RGNode7})
79 //AddE
80 init_2AddE = new AddE({init_2RGExtEdge0, init_2RGExtEdge1,
        init_2RGExtEdge2, init_2RGExtEdge3, init_2RGExtEdge4,
        init_2RGExtEdge5, init_2RGExtEdge6, init_2RGExtEdge7,
        init_2RGExtEdge8, init_2RGExtEdge9, init_2RGExtEdge10})
81 // initialize Match with empty
82 init_2Match = new Match({ |-> }, { |-> })
83 // initialize Copy with empty
84 init_2Copy = new Copy({ |-> })
85 // Grr structur for init_2
86 init_2 = new Grr((init_2LGGraph, init_2RGGraph, init_2DelN, init_2DelE,
                    init_2CoreN, init_2AddN, init_2AddE, init_2Match,
                    init_2Copy), "init_2" )
```

Segmente aus der Regel *findMatchFor*

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des letzten Story Patterns der Methode *dockToStation* ist (Abbildung 4.19):

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                         dockToStation3_4LGNode0 |-> thisObject }
03 step
04   //trying to bind st2 with dockToStation3_4LGNode1
05   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                              dockToStation3_4LGExtEdge0)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     //trying to bind sch1 with dockToStation3_4LGNode2
11     let (n1, e0, edge0) = getMatchObjectFrom(G, EG, thisObject,
                              dockToStation3_4LGExtEdge2)
12     if n1 = undef then
13       //no match found
14       skip
15     else
```

```
16       //check link: hasShuttle
17       var checkResult as Boolean = checkLink(EG, n1,
                          dockToStation3_1LGExtEdge1, thisObject)
18       if checkResult = false then
19         // no match found
20         skip
21       else
22         //match found
23         result := true
24         boundObjects.elements := boundObjects.elements override {
                                  dockToStation3_4LGNode1 |-> n0,
                                  dockToStation3_4LGNode2 |-> n1}
25         GrrMatch.paramN := { dockToStation3_4LGNode0 |-> thisObject,
                                dockToStation3_4LGNode1 |-> n0,
                                dockToStation3_4LGNode2 |-> n1}
26         GrrMatch.paramE := { dockToStation3_4LGExtEdge0 |->
                               ExtMap((thisObject, getELabelFrom(
                               dockToStation3_4LGExtEdge0 ), n0)),
                               dockToStation3_4LGExtEdge2 |->
                               ExtMap((n1, getELabelFrom(
                               dockToStation3_4LGExtEdge2 ), n0)),
                               dockToStation3_4LGExtEdge1 |->
                               ExtMap((thisObject, getELabelFrom(
                               dockToStation3_4LGExtEdge1 ), n2))}
```

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des ersten Story Patterns der Methode *executeEvent* ist (Abbildung 4.22):

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                             executeevent1_0LGNode0 |-> thisObject }
03 step
04   //trying to bind e1 with executeevent1_0LGNode1
05   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                                             executeevent1_0LGExtEdge0)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     //match found
11     result := true
12     boundObjects.elements := boundObjects.elements override
                                    {executeevent1_0LGNode1 |-> n0}
13     GrrMatch.paramN := {executeevent1_0LGNode0 |-> thisObject,
                           executeevent1_0LGNode1 |-> n0}
14     GrrMatch.paramE := {executeevent1_0LGExtEdge0 |->
        ExtMap((thisObject, getELabelFrom(executeevent1_0LGExtEdge0), n0))}
```

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des zweiten Story Patterns der Methode *executeEvent* ist (Abbildung 4.22):

```
01 machine
02  boundObjects.elements := boundObjects.elements override
                                  {executeevent2_1LGNode0 |-> thisObject }
03 step
04   //check link: hasEvent
05   var checkResult as Boolean = checkLink(EG,
                           boundObjects.elements(executeevent2_1LGNode0),
```

```
                          executeevent2_1LGExtEdge0,
                          boundObjects.elements(executeevent2_1LGNode1))
06   if checkResult = false then
07     // no match found
08     skip
09   else
10     result := true
11     boundObjects.elements := boundObjects.elements override { |-> }
12     GrrMatch.paramN := {executeevent2_1LGNode0 |-> thisObject,
                          executeevent2_1LGNode1 |->
                          boundObjects.elements(executeevent2_1LGNode1)}
13     GrrMatch.paramE := {executeevent2_1LGExtEdge0 |->
              ExtMap((thisObject, getELabelFrom(executeevent2_1LGExtEdge0),
              boundObjects.elements(executeevent2_1LGNode1)))}
```

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des zweiten Story Patterns der Methode *checkShuttle* ist (Abbildung 4.23):

```
01 machine
02   boundObjects.elements := boundObjects.elements override
                 { checkShuttle2_5LGNode0 |-> thisObject }
03 step
04   //trying to bind sh1 with checkShuttle2_5LGNode1
05   let (n0, e0, edge0) = getMatchObjectFrom(G, EG, thisObject,
                                checkShuttle2_5LGExtEdge0)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     //match found
11     //trying to bind p1 with checkShuttle2_5LGNode2
12     let (n1, e1, edge1) = getMatchObjectTo(G, EG, n0,
                                checkShuttle2_5LGExtEdge1)
13     if n1 = undef then
14       //no match found
15       skip
16     else
17       if getAttrValueFrom(EG, n1, classAttributeLabel2) = undef then
18         skip
19       else
20         if getAttrValueFrom(EG, n0, classAttributeLabel1) = undef then
21           skip
22         else
23           if getAttrValueFrom(EG, n1, classAttributeLabel2) =
24                getAttrValueFrom(EG, n0, classAttributeLabel1) then
25             //match found
26             result := true
27             boundObjects.elements := boundObjects.elements override
                              { checkShuttle2_5LGNode1 |-> n0,
                                checkShuttle2_5LGNode2 |-> n1}
28             GrrMatch.paramN := { checkShuttle2_5LGNode0 |->
                  thisObject, checkShuttle2_5LGNode1 |-> n0,
                  checkShuttle2_5LGNode2 |-> n1}
29             GrrMatch.paramE := { checkShuttle2_5LGExtEdge0 |->
                  ExtMap((n0, getELabelFrom(
                  checkShuttle2_5LGExtEdge0), thisObject)),
                  checkShuttle2_5LGExtEdge1 |-> ExtMap((n0,
                  getELabelFrom(checkShuttle2_5LGExtEdge1), n1))}
```

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des zweiten Story Patterns der Methode *loadShuttle* (Abbildung 4.24) mit einer negativen Variable ist:

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                          loadshuttle2_14LGNode0 |-> thisObject }
03 step
04   //trying to bind sh11 with loadshuttle2_14LGNode2
05   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                                     loadshuttle2_14LGExtEdge1)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     // check negative variable
11     let (n1, e1, edge1) = getMatchObjectTo(G, EG, n1,
                                  loadshuttle2_14LGExtEdge2)
12     if n1 <> undef then
13       //no match found
14       skip
15     else
16       //match found
17       //trying to bind p1 with loadshuttle2_14LGNode1
18       let (n2, e2, edge2) = getMatchObjectFrom(G, EG, thisObject,
                                    loadshuttle2_14LGExtEdge0)
19       if n2 = undef then
20         //no match found
21         skip
22       else
23         //match found
24         result := true
25         boundObjects.elements := boundObjects.elements override
                                {loadshuttle2_14LGNode1 |-> n2,
                                 loadshuttle2_14LGNode2 |-> n0}
26         GrrMatch.paramN := {loadshuttle2_14LGNode0 |-> thisObject,
                               loadshuttle2_14LGNode1 |-> n2,
                               loadshuttle2_14LGNode2 |-> n0}
27         GrrMatch.paramE := {loadshuttle2_14LGExtEdge1 |->
              ExtMap((thisObject, getELabelFrom(loadshuttle2_14LGExtEdge1),
              n2)), loadshuttle2_14LGExtEdge0 |-> ExtMap((n0,
              getELabelFrom(loadshuttle2_14LGExtEdge0), thisObject))}
```

Segment der Regel *findMatchFor*, das zuständig für die Teilgraphensuche des zweiten Story Patterns der Methode *loadShuttle* mit einer optionalen Variable ist (Abbildung 4.28):

```
01 machine
02   boundObjects.elements := boundObjects.elements override {
                          loadshuttle2_14LGNode0 |-> thisObject }
03 step
04   //trying to bind sh11 with loadshuttle2_14LGNode2
05   let (n0, e0, edge0) = getMatchObjectTo(G, EG, thisObject,
                                     loadshuttle2_14LGExtEdge1)
06   if n0 = undef then
07     //no match found
08     skip
09   else
10     // check optional variable
11     let (n1, e1, edge1) = getMatchObjectTo(G, EG, n1,
                                  loadshuttle2_14LGExtEdge2)
17     //trying to bind p1 with loadshuttle2_14LGNode1
18     let (n2, e2, edge2) = getMatchObjectFrom(G, EG, thisObject,
```

```
                                    loadshuttle2_14LGExtEdge0)
19      if n2 = undef then
20        //no match found
21        skip
22      else
23        //match found
24        result := true
25        boundObjects.elements := boundObjects.elements override
                                  {loadshuttle2_14LGNode1 |-> n2,
                                   loadshuttle2_14LGNode2 |-> n0}
26        if n1 <> undef then
27          boundObjects.elements := boundObjects.elements override
                                  { someVariable |-> n1 }

28        GrrMatch.paramN := {loadshuttle2_14LGNode0 |-> thisObject,
                                 loadshuttle2_14LGNode1 |-> n2,
                                 loadshuttle2_14LGNode2 |-> n0}
29        if n1 <> undef then
30          GrrMatch.paramN := GrrMatch.paramN union
                             { someVariable |-> n1 }
31        GrrMatch.paramE := {loadshuttle2_14LGExtEdge1 |->
              ExtMap((thisObject, getELabelFrom(loadshuttle2_14LGExtEdge1),
              n2)), loadshuttle2_14LGExtEdge0 |-> ExtMap((n0,
              getELabelFrom(loadshuttle2_14LGExtEdge0), thisObject))}
32        if n1 <> undef then
33          GrrMatch.paramE := GrrMatch.paramE union
              { someLink |-> ExtMap((n1,
              getELabelFrom(loadshuttle2_14LGExtEdge0), thisObject)) }
```

Aktivitätsdiagramme

Das Aktivitätsdiagramm der modifizierten Methode executeEvent in AsmL Code (Abbildung 4.31):

```
01 var asml_nextStep as Integer = 0
02 var asml_running as Boolean = true
03 var asml_GrrResult as Boolean = false
04 var returnValue = new Value(undef)
05 let params = new Parameters({|->})
06 var boundObjects = new BoundObjects({ |-> })
07 while asml_running do
08   machine
09     match asml_nextStep with
10       0: machine
11            asml_nextStep := 1
12          step
13            skip
14       1: machine
15            let foundM = new foundMatches({})
16            asml_GrrResult := runIterGrr(executeEvent1, EG, S,
                    thisObject, boundObjects, returnValue, params, foundM)
17          step
18            if asml_GrrResult then
19              asml_nextStep := 2 // each time
20            else
21              asml_nextStep := 3 // end
22          step
23            skip
24       2: machine
25            asml_GrrResult := runGrr(executeEvent2, EG, S, thisObject,
```

```
                                        boundObjects, returnValue, params)
26          step
27            asml_nextStep := 1
28          step
29            skip
30       3: machine
31            //stop activity
32            asml_running := false
33          step
34            skip
35   step
36     skip
```

Funktionen

Die Funktion *void_Station_produce*, die der Methode *produce* der Klasse *Station* entspricht (Abbildung 4.33):

```
01 void_Station_produce(EG as ExtGraph, S as SIGraph, thisObject as
                        ExtNode) =
02 var asml_nextStep as Integer = 0
03 var asml_running as Boolean = true
04 var returnValue = new Value(undef)
05 let params = new Parameters({|->})
06 var asml_GrrResult as Boolean = false
07 var boundObjects = new BoundObjects({ |-> })
08 while asml_running do
09 machine
10   match asml_nextStep with
11      0: machine
12           asml_nextStep := 1
13         step
14           skip
15      1: machine
16           asml_GrrResult := runGrr(produce1_16, EG, S, thisObject,
                                      boundObjects, returnValue, params)
17         step
18           asml_nextStep := 2
19         step
20           skip
21      2: machine
22           //nop activity
23           skip
24           if asml_GrrResult then  //success
25             asml_nextStep := 3
26           if not asml_GrrResult then  //failure
27             asml_nextStep := 4
28         step
29           skip
30      3: machine
31           asml_GrrResult := runGrr(produce2_17, EG, S, thisObject,
                                      boundObjects, returnValue, params)
32         step
33           asml_nextStep := 5
34         step
35           skip
36      5: machine
37           asml_GrrResult := runGrr(produce3_18, EG, S, thisObject,
                                      boundObjects, returnValue, params)
38         step
```

```
39          asml_nextStep := 4
40        step
41          skip
42     4: machine
43          //stop activity
44          asml_running := false
45        step
46          skip
47  step
48    skip
```

Die Funktion *string_Product_getType*, die der Methode *getType* der modifizierten Klasse *Product* entspricht (Abbildung 4.35):

```
01 string_Product_getType(EG as ExtGraph, S as SIGraph, thisObject as
                          ExtNode) as Value =
02   var asml_nextStep as Integer = 0
03   var asml_running as Boolean = true
04   var returnValue = new Value(undef)
05   let params = new Parameters({|->})
06   var asml_GrrResult as Boolean = false
07   var boundObjects = new BoundObjects({ |-> })
08   while asml_running do
09     machine
10       match asml_nextStep with
11          0: machine
12               asml_nextStep := 1
13             step
14               skip
15          1: machine
16               asml_GrrResult := runGrr(getType_35, EG, S, thisObject,
                                      boundObjects, returnValue, params)
17             step
18               asml_nextStep := 2
19             step
20               skip
21          2: machine
22               //stop activity
23               asml_running := false
24               returnValue.value := getAttrValueFrom(EG, thisObject,
                                       classAttributeLabel2)
25             step
26               skip
27     step
26       return returnValue
```

Die Funktion *string_Product_setType*, die der Methode *setType* der modifizierten Klasse *Product* entspricht (Abbildung 4.36):

```
01 void_Product_setType(EG as ExtGraph, S as SIGraph, thisObject as
                        ExtNode, t as Value) =
02   var asml_nextStep as Integer = 0
03   var asml_running as Boolean = true
04   var returnValue = new Value(undef)
05   let params = new Parameters({“t”|-> t})
06   var asml_GrrResult as Boolean = false
07   var boundObjects = new BoundObjects({ |-> })
08   while asml_running do
```

```
09     machine
10       match asml_nextStep with
11          0: machine
12               asml_nextStep := 1
13             step
14               skip
15          1: machine
16               asml_GrrResult := runGrr(setType_36, EG, S, thisObject,
                                      boundObjects, returnValue, params)
17             step
18               asml_nextStep := 2
19             step
20               skip
21          2: machine
22               //stop activity
23               asml_running := false
24             step
25               skip
26      step
27        skip
```

Die Funktion *void_AssignShuttle_execute*, die der Methode *execute* der Klasse *AssignShuttle* entspricht (Abbildung 4.37):

```
01 void_AssignShuttle_execute(EG as ExtGraph, S as SIGraph, thisObject as
                               ExtNode) =
02   var asml_nextStep as Integer = 0
03   var asml_running as Boolean = true
04   var returnValue = new Value(undef)
05   let params = new Parameters({|->})
06   var asml_GrrResult as Boolean = false
07   var boundObjects = new BoundObjects({ |-> })
08   while asml_running do
09     machine
10       match asml_nextStep with
11        0: machine
12             asml_nextStep := 1
13           step
14             skip
15        1: machine
16             asml_GrrResult := runGrr(executeassignshuttle_26, EG, S,
                          thisObject, boundObjects, returnValue, params)
17             step
18               void_Shuttle_assignShuttle(EG, S, boundObjects.elements(
                                        executeassignshuttle_26LGNode1))
19             step
20               asml_nextStep := 2
21             step
22               skip
23        2: machine
24             //stop activity
25             asml_running := false
26             step
27               skip
28     step
29       skip
```

Die Funktion *void_Scheduler_executeEvent*, die der Methode *executeEvent* der Klasse *Scheduler* entspricht (Abbildung 4.22):

```
01 void_Scheduler_executeEvent(EG as ExtGraph, S as SIGraph, thisObject as
                               ExtNode) =
02   var asml_nextStep as Integer = 0
03   var asml_running as Boolean = true
04   var returnValue = new Value(undef)
05   let params = new Parameters({|->})
06   var asml_GrrResult as Boolean = false
07   var boundObjects = new BoundObjects({ |-> })
08   while asml_running do
09     machine
10       match asml_nextStep with
11         0: machine
12              asml_nextStep := 1
13            step
14              skip
15         1: machine
16              asml_GrrResult := runGrr(executeevent1_0, EG, S,
                                thisObject, boundObjects, returnValue, params)
17            step
18              //polymorphic method
19              let extNl = getNl(EG)
20              let n = extNl.elements( boundObjects.elements(
                                        executeevent1_0LGNode1) )
21              //class Event
22              if (n = classNode1) then
23                void_Event_execute( EG, S, boundObjects.elements(
                                       executeevent1_0LGNode1) )
24              //class AssignShuttle
25              if (n = classNode7) then
26                void_AssignShuttle_execute( EG, S, boundObjects.elements(
                                              executeevent1_0LGNode1) )
27              //class LoadShuttle
28              if (n = classNode8) then
29                void_LoadShuttle_execute( EG, S, boundObjects.elements(
                                             executeevent1_0LGNode1) )
30              //class UnloadShuttle
31              if (n = classNode9) then
32                void_UnloadShuttle_execute( EG, S, boundObjects.elements(
                                              executeevent1_0LGNode1) )
33              //class MoveToNextStation
34              if (n = classNode18) then
35                void_MoveToNextStation_execute( EG, S,
                          boundObjects.elements( executeevent1_0LGNode1) )
36              //class ShuttleArrived
37              if (n = classNode19) then
38                void_ShuttleArrived_execute( EG, S,
                          boundObjects.elements( executeevent1_0LGNode1) )
39              //class Produce
40              if (n = classNode10) then
41                void_Produce_execute( EG, S, boundObjects.elements(
                                        executeevent1_0LGNode1) )
42              asml_nextStep := 2
43            step
44              skip
45         2: machine
46              asml_GrrResult := runGrr(executeevent2_1, EG, S,
                                thisObject, boundObjects, returnValue, params)
47            step
48              asml_nextStep := 3
49            step
50              skip
```

```
51          3: machine
52               //stop activity
53               asml_running := false
54             step
55               skip
56      step
57        skip
```