Verknüpfung von formaler Verifikation und modellgetriebener Entwicklung

Zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

des Fachbereichs Mathematik/Informatik der Universität Osnabrück vorgelegte

Dissertation

von Dipl.-Inform. Christian Ammann

aus Hildesheim

Datum:	10.04.2015
Erste Gutachterin:	Prof. Dr. Elke Pulvermüller
Zweiter Gutachter:	Prof. Dr. Stephan Kleuker

Inhaltsverzeichnis

Abbildungsverzeichnis

Danksagung

Eine wissenschaftliche Arbeit ist nie das Werk einer einzelnen Person. Deshalb möchte ich mich an dieser Stelle bei allen Beteiligten bedanken. Dabei sind zunächst meine beiden Betreuer, Prof. Dr. Elke Pulvermüller und Prof. Dr. Stephan Kleuker zu nennen. Sie haben diese Dissertation im Rahmen der vielen Gespräche entscheidend und im positiven Sinne geprägt. Es darf auch nicht unerwähnt bleiben, dass jede E-Mail innerhalb sehr kurzer Zeit ausführlich beantwortet wurde. Auch nach fünf Jahren haben sie jede Leseprobe genauestens durchgearbeitet und detailliertes Feedback gegeben.

Des Weiteren hat mich auch Familie immer unterstützt. Dies beinhaltet nicht nur fortwährendes Interesse an meiner Arbeit: Meine Geschwister Claudia und Mathias Ammann haben durch gemeinsame Konzertbesuche für die notwendige Kurzweil gesorgt. An den Vorbereitungen für die Verteidigung der Dissertation hat sich meine Mutter Angelika Ammann aktiv beteiligt. Mein Vater Guido Ammann hat mich immer motiviert, sofern dies notwendig war. Als ich ihm beispielsweise zu Beginn meines Informatikstudiums sagte, dass es vermutlich nichts für mich sei, weil die Mathematikvorlesungen in einem enormen Tempo vorgetragen würden und ich kein Wort verstünde, wies mich darauf hin: „Das ist normal, das muss so sein“.

Parallel zu meiner eigenen Doktorarbeit haben auch ein paar Freunde mit ihrer Dissertation begonnen. Hierzu gehören Martin Heinig und Sabrina Rutter. Bedanken möchte ich mich für die vielen produktiven Gespräche, in denen man gelernt hat, dass andere Doktoranden vor ähnlichen Problemen bzw. Herausforderungen stehen.

Mit Felix Grehl, Nils Hülsmann und Björn Rippe treffe ich mich seit dem Studium in unregelmäßigen Abständen, um gemeinsam Karten zu spielen. Im Rahmen dieser Begegnungen haben sie immer wieder Interesse an meiner Dissertation gezeigt. Durch kritische Rückfragen wurde sie positiv beeinflusst.

Der Model Checker Spin leistet einen wesentlichen Beitrag zu meiner Arbeit. Ich kenne ihn bereits durch meine Tätigkeit als studentische Hilfskraft im Oldenburger OFFIS. Deshalb möchte ich mich an dieser Stelle bei meinen Betreuern Tobe Toben, Bernd Westphal und Jan Rakow bedanken, die mein Interesse an der formalen Verifikation geweckt haben.

Ein großer Teil dieser Arbeit ist an der Hochschule Osnabrück entstanden. Mit Frank Thiesing blicke ich auf sehr angenehme, gemeinsame Konferenzbesuche zurück. Alfons Mönnich und Thomas Fründ haben immer die passende IT-Infrastruktur, wie beispielweise SVN-Server, zur Verfügung gestellt. Ich möchte mich auch bei Ralf Koller bedan-

ken, der mich an meinem ersten Arbeitstag in meinem Büro besuchte, um mit mir in die Mensa zu gehen bzw. mir die Hochschule zu zeigen.

Auch meine Arbeitskollegen in Nürnberg am Fraunhofer IIS haben diese Arbeit beeinflusst. Obwohl ich kein „Bayer“, sondern eher ein „Preuße“ bin, wurde ich von ihnen gut aufgenommen. Thomas Hauenstein hat die Folien meines Abschlussvortrags durchgesehen. Ich möchte mich auch bei meiner ehemaligen Büronachbarin Nina Holzer für die vielen fachlichen Diskussion, Messen und unterhaltsame Dienstreisen bedanken.

1 Einleitung

Die Bedeutung von Software hat in den letzten Jahren immer mehr zugenommen. So sind beispielsweise in der Automobilindustrie nicht nur die Technik im produzierten Fahrzeug, sondern auch die Fahrzeugproduktion und betriebliche Abläufe des Unternehmens von Software gesteuert [15]. Damit einhergehend sind auch der Funktionsumfang und somit die Häufigkeit von Fehlern in Software-Lösungen gestiegen. Deshalb ist es wichtig Methoden zu entwickeln, die die Komplexität von Software kontrollierbarer machen und die Anzahl von Fehlern in einem System reduzieren. Dazu gehört das automatische Generieren von ausführbarem Quellcode aus formalen Modellen heraus [134]. Obwohl durch diesen Ansatz die Anzahl der Fehler in einer Software-Lösung reduziert wird, können jedoch immer noch die Modelle selbst fehlerhaft sein. Um zu verifizieren, dass die entsprechenden Modelle keine Fehler enthalten und alle an sie gestellten Anforderungen erfüllen, kann ein Model Checker [31] Verwendung finden. Daraus leitet sich das Ziel dieser Arbeit ab, nämlich die Integration eines Model Checkers in den Software-Entwicklungsprozess, um:

1. Modelle und Anforderungen zu beschreiben.

2. Automatisiert mit einem Model Checker zu verifizieren, dass die beschriebenen Modelle alle Anforderungen erfüllen.

3. Die Modelle, wenn sie keine Fehler mehr enthalten, automatisiert in ausführbare Software zu transformieren.

Die drei Schritte werden im Folgenden genauer ausgeführt und das Ziel dieser Arbeit beschrieben.

Ein Ansatz um die Qualität von Software zu erhöhen, ist das *Model-Driven Development* bzw *Model-Driven Engineering* (MDD oder MDE) [134] [103]. MDD ist eine Vorgehensweise aus dem Bereich des *Software Engineering*, bei der Quellcode nicht von Hand geschrieben, sondern automatisiert aus Modellen heraus erzeugt wird. Dies geschieht beispielsweise in stark eingeschränkter Form in konventionellen Software-Projekten, wenn UML-Klassendiagramme [111] in Quellcode transformiert werden. Der Aufbau des MDD-Ansatzes ist schematisch in Abbildung 1.1 dargestellt. Dabei wird ein Software-Projekt zunächst in seine unterschiedlichen Domänen, wie beispielsweise die Datenhaltung mit Zugriff auf eine Datenbank oder die Benutzeroberfläche, unterteilt. Anschließend wird für

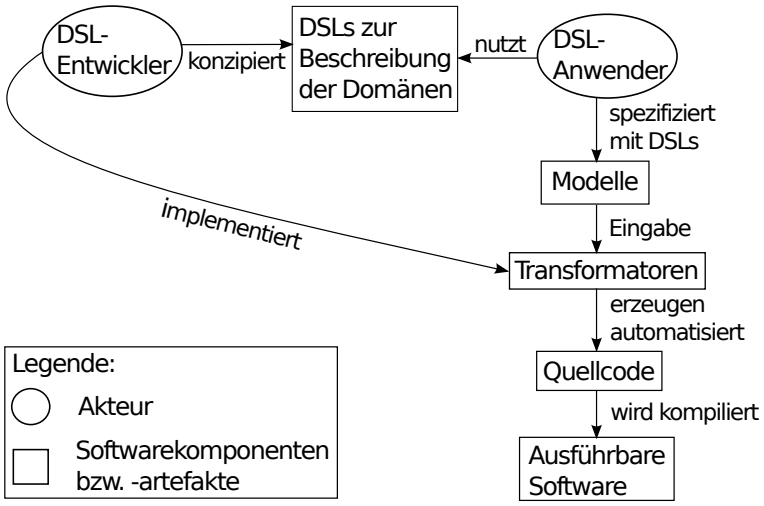

Abbildung 1.1: Model-Driven Development

jede Domäne eine domänenspezifische Sprache (DSL) [69] entwickelt, um damit Modelle beschreiben zu können.

Die folgende Analogie beschreibt, wie sich domänenspezifische Sprachen von Hochsprachen abgrenzen: Das Kerngehäuse eines Apfels kann sowohl mit einem Taschenmesser, als auch mit einem speziellen Apfelentkerner entfernt werden. Das Taschenmesser bietet den Vorteil, dass es für eine breite Menge von Aufgaben eingesetzt werden kann, jedoch nicht optimal für die Entfernung des Kerngehäuses geeignet ist. Der Apfelentkerner kann nur eine spezielle Aufgabe durchführen, erfüllt diese jedoch besonders gut. DSLs entsprechen im Rahmen dieser Analogie dem Apfelentkerner und Hochsprachen dem Taschenmesser. Somit werden DSLs im Gegensatz zu Hochsprachen nur für einen bestimmten Zweck, wie beispielsweise das Umsetzen einer graphischen Benutzeroberfläche, eingesetzt.

Durch die Nutzung von domänenspezifischen Sprachen ergeben sich im Rahmen eines Software-Projekts zwei Rollen, die in Abbildung 1.1 mit einem Kreis dargestellt sind:

- Der DSL-Entwickler konzipiert bzw. implementiert die DSLs und die dazugehörigen Transformatoren. Anschließend stellt er sie dem DSL-Anwender zur Verfügung. Der DSL-Entwickler ist somit für das Beschreiben der Metamodelle verantwortlich.

- Der DSL-Anwender nutzt die DSLs zum Erzeugen der Modelle und überführt sie automatisiert mit den Transformatoren in Quellcode.

Nach der Modelltransformation wird der generierte Quellcode kompiliert und so ausführbare Software erzeugt. Dieses Vorgehen führt bei der erstmaligen Nutzung in einem Software-Projekt zu einem erhöhtem Aufwand, bietet aber eine Reihe von Vorteilen:

- Domänenexperten, die keine Kenntnisse von modernen Programmiersprachen haben, können an der Entwicklung partizipieren und die DSLs zum Implementieren der Software nutzen.

- Durch die automatische Generierung von Quellcode werden Fehler reduziert und somit die Qualität der Software erhöht.

- Domänenspezifische Sprachen und die dazugehörigen Modelltransformatoren können in anderen Software-Projekten wiederverwendet und so langfristig Entwicklungszeit bzw. Entwicklungskosten reduziert werden [66].

Zu beachten ist, dass der MDD-Ansatz nicht vorschreibt, dass das gesamte Software-Projekt mit DSLs umgesetzt werden muss. Ein MDD-Projekt kann sowohl Modelle zur automatischen Übersetzung, als auch von Hand geschriebenen Quellcode enthalten.

Durch die automatische Generierung von Quellcode wird die Anzahl von Fehlern reduziert, die Modelle selbst können aber trotzdem noch Fehler enthalten bzw. die Realität nicht korrekt abbilden. Deshalb ist im Rahmen des MDD-Ansatzes die Verwendung eines *Model Checkers* [31][80] sinnvoll. Bei einem Model Checker handelt es sich um ein Programm, das automatisiert überprüft, ob eine Spezifikation bzw. ein Modell bestimmte Anforderungen erfüllt. Die formale Verifikation stammt aus der theoretischen Informatik und grenzt sich vom klassischen Testen dadurch ab, dass ein Model Checker garantiert 100% des Zustandsraums untersucht, während Testwerkzeuge nur eine Teilmenge des Zustandsraums erfassen. Die Erweiterung des MDD-Ansatzes durch einen Model Checker bietet den Vorteil, dass durch die formale Verifikation Fehler in den Modellen entdeckt bzw. behoben werden können, die sich ansonsten auch auf die automatisch generierte Software auswirken würden.

Das Prinzip des *Model Checking* kann anhand des folgenden Beispiels verdeutlicht werden: Gegeben sei als Spezifikation das bekannte Philosophenproblem [26]. Es ist schematisch in Abbildung 1.2 dargestellt. Dabei handelt es sich um fünf Philosophen, die an einem Tisch sitzen und speisen. Jeder Philosoph verfügt über einen Teller. Er muss sich jedoch das Besteck mit seinen beiden Nachbarn teilen, da es nur fünf Gabeln gibt. Wenn ein Philosoph Hunger bekommt und essen will, wartet er so lange, bis die linke Gabel verfügbar ist und greift sie. Anschließend wartet er, bis die Gabel rechts neben ihm frei wird und nimmt auch sie auf. Danach isst er und legt beide Gabeln wieder zurück an ihren Platz. Wenn ein Philosoph essen will und das Besteck bereits von seinen Sitznachbarn in Verwendung ist, muss er warten bis diese das Besteck wieder freigeben. Eine Anforderung an das Philosophenproblem ist, dass es zu keinen *Deadlocks* kommen darf, da die Philosophen ansonsten verhungern.

Damit ein Model Checker dieses Problem verifizieren kann, müssen die Spezifikation und die Anforderungen formalisiert bzw. in eine *Model Checker-Eingabesprache* übersetzt werden. Anschließend überprüft der Model Checker systematisch alle Zustände, die das

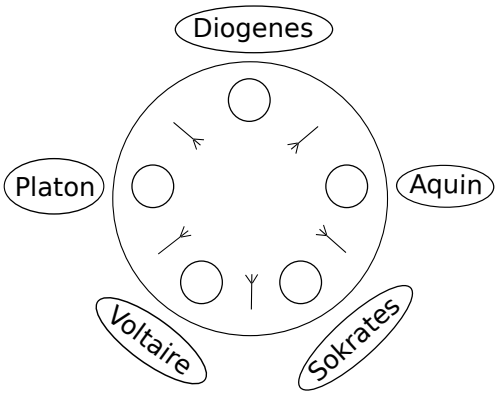

Abbildung 1.2: Das Philosophenproblem

System annehmen kann und erzeugt typischerweise einen Fehlerpfad, falls eine Anforderung in einem der Zustände nicht erfüllt ist. Im Fall des Philosophenproblems wird der *Model Checker* einen Pfad finden, bei dem alle Philosophen ihre linke Gabel greifen und anschließend warten, bis die rechte Gabel frei wird. Dies geschieht jedoch nicht, da jeder Philosoph auf seine Nachbarn wartet und so ein *Deadlock* entsteht. Daher muss der Anwender sein Modell modifizieren (im Falle der Philosophen also den Algorithmus zum Greifen der Gabeln) und anschließend erneut vom Model Checker verifizieren lassen. Dies wird wiederholt, bis alle Anforderungen erfüllt sind und der Model Checker keinen weiteren Fehler meldet.

1.1 Ziel

Diese Dissertation untersucht, wie die Methoden der formalen Verifikation und des Model-Driven Development miteinander kombiniert werden können. Dazu müssen domänenspezifische Sprachen entwickelt und automatisiert in sowohl eine Hochsprache, als auch in eine Model Checker-Eingabesprache transformiert werden. Deshalb wird der MDD-Ansatz im Rahmen dieser Arbeit folgermaßen erweitert: Neben den Transformatoren, die eine DSL in ausführbare Software überführen, werden auch Transformatoren entwickelt werden, die das Modell in eine Model Checker-Eingabesprache übersetzen. Des Weiteren wird jedes Modell mit Anforderungen angereichert, damit der Model Checker verifizieren kann, ob das Modell korrekt ist. Diese Vorgehensweise ist in Abbildung 1.3 schematisch als UML-Aktivitätsdiagramm dargestellt: Ein Transformator überführt ein Modell, das mit einer DSL umgesetzt wurde, zusammen mit den dazugehörigen Anforderungen automatisiert in eine Model Checker-Eingabesprache. Anschließend überprüft der Model-Checker, ob das Modell alle Anforderungen erfüllt. Falls Anforderungen nicht

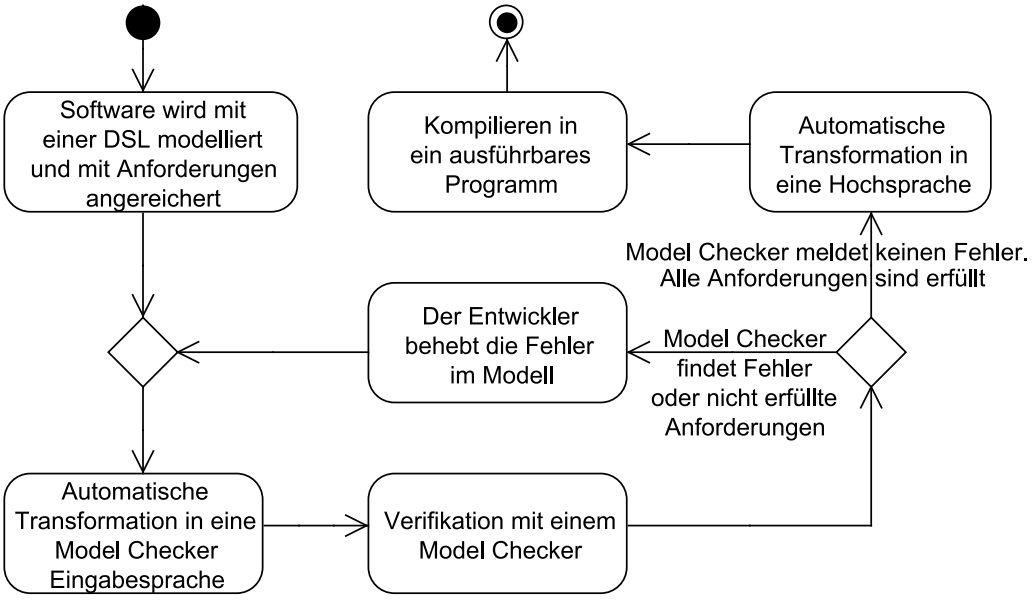

Abbildung 1.3: Verknüpfung von MDD und formaler Verifikation (Aktivitätsdiagramm)

erfüllt sind, wird ein Fehler gemeldet, das Modell entsprechend modifiziert und erneut verifiziert. Wenn das Modell alle Anforderungen erfüllt und der Model Checker keine weiteren Fehler anzeigt, kann das Modell in ausführbare Software übersetzt werden. Ein besonderer Schwerpunkt dieser Dissertation liegt auf der Untersuchung, wie die Kombination von MDD und formaler Verifikation nicht nur für spezielle Anwendungsfälle und domänenspezifische Sprachen, sondern möglichst generisch in beliebigen Softwareprojekten umgesetzt werden kann.

1.2 Problemanalyse

Damit Model-Driven Development und formale Verifikation, wie in Abbildung 1.3 beschrieben, kombiniert werden können, müssen die folgenden Problemstellungen betrachtet werden:

- DSLs müssen entwickelt und Modelltransformatoren geschrieben werden.
- Neben der Transformation in eine Hochsprache, muss die entsprechende DSL auch in eine Model Checker-Eingabesprache überführt werden.

Diese beiden Aspekte werden in den nächsten Abschnitten genauer analysiert und daraus die für diese Arbeit relevanten Teilproblemstellungen abgeleitet.

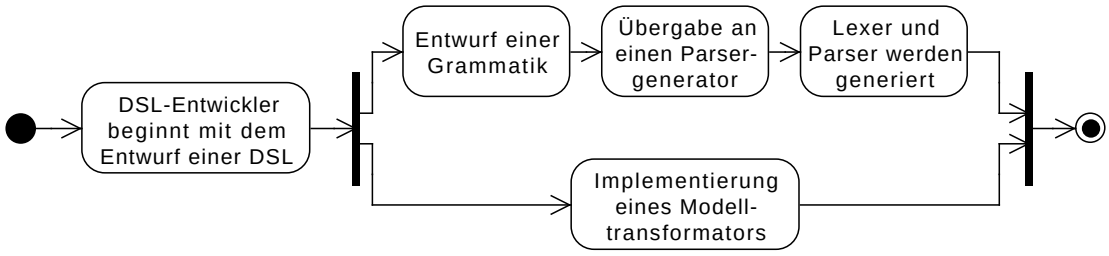

Abbildung 1.4: Erzeugen von Parser und Transformator (Aktivitätsdiagramm)

1.2.1 Entwicklung und Transformation von DSLs

Die Entwicklung von DSLs macht den Einsatz von Werkzeugen und Techniken notwendig, wie sie aus dem Compilerbau [1] bekannt sind. Die dafür erforderlichen Schritte sind schematisch in Abbildung 1.4 als UML-Aktivitätsdiagramm dargestellt. Für den DSL-Entwickler ergeben sich daraus zwei Aufgaben:

- Es muss zunächst für jede domänenspezifische Sprache eine Grammatik geschrieben werden, die als Metamodell fungiert. Die Grammatik besteht aus einer Menge von Produktionsregeln. Sie wird einem Generator übergeben, der daraus einen Lexer und Parser erzeugt.

- Es muss ein Transformator entwickelt werden, der den generierten Parser nutzt, um Modelle, die mit der entsprechenden DSL beschrieben wurden, in eine Zielsprache wie beispielsweise Java zu übersetzen.

Beide Schritte werden in Abbildung 1.4 nicht nacheinander, sondern parallel durchgeführt. Dies begründet sich durch die Tatsache, dass während der Umsetzung des Transformators auch noch Anpassungen an der Grammatik vorgenommen werden können und umgekehrt.

Nach dem Generieren des Parsers bzw. Implementieren des Transformators kann der DSL-Anwender beide Komponenten nutzen, um damit Modelle zu beschreiben und automatisiert in eine Zielsprache zu überführen. Dieses Vorgehen ist in Abbildung 1.5 als UML-Aktivitätsdiagramm skizziert. Der DSL-Anwender beschreibt zunächst mit der entsprechenden domänenspezifischen Sprache ein Modell. Das Modell wird dem Lexer bzw. Parser übergeben. Wenn es keine syntaktischen Fehler enthält, erzeugt der Parser aus dem Modell einen Abstract Syntax Tree (AST). Zum Abschluss liest der Transformator den Abstract Syntax Tree ein und übersetzt ihn in die entsprechende Zielsprache.

Um das in diesem Abschnitt beschriebene Verfahren besser zu veranschaulichen, wird nun ein Beispiel vorgestellt, das auf dem Philosophenproblem basiert. Dafür muss zunächst eine Grammatik geschrieben werden. Die Syntax der darin enthaltenen Produk-

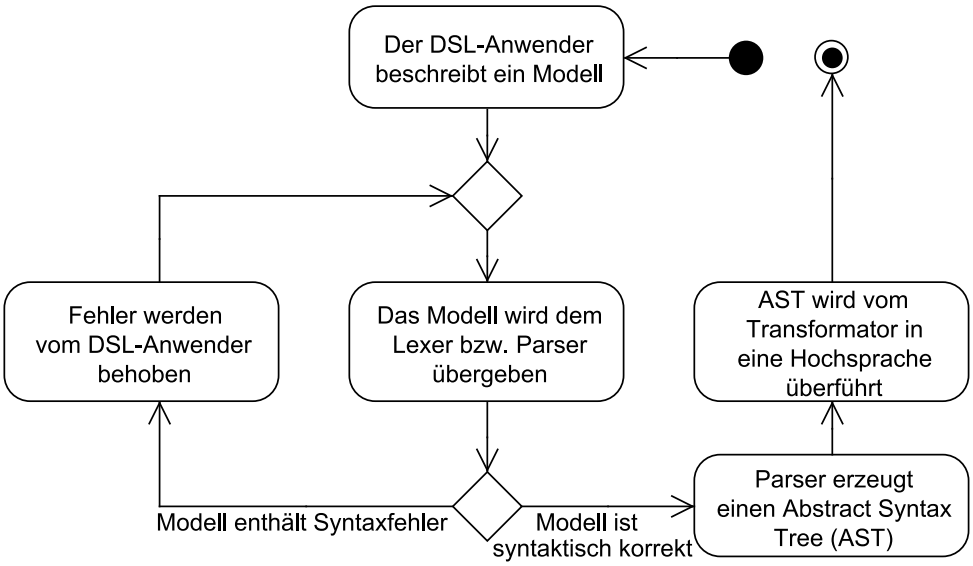

Abbildung 1.5: Verwendung von Parser und Transformator (Aktivitätsdiagramm)

tionsregeln ist abhängig von dem verwendeten Parsergenerator. Deshalb wird an dieser Stelle ein Beispiel in *Pseudocode* gegeben:

```
1 Initial -> Philosophen
2
3 Philosophen -> Philosoph Philosophen
4 Philosophen -> Philosoph
5
6 Philosoph -> philosoph Label ;
7
8 Label -> [a..z..A..Z]*
```

Listing 1.1: Grammatik zum Deklarieren einer Menge von Philosophen

Listing 1.1 repräsentiert die Grammatik bzw. das Metamodell einer Sprache, die an das Philosophenproblem aus Abbildung 1.2 angelehnt ist und das Deklarieren einer Menge von Philosophen erlaubt. Die Grammatik besteht aus fünf Produktionsregeln. Terminale sind unterstrichen hervorgehoben. Die Startregel *Initial* wird zunächst nach *Philosophen* abgeleitet. Anschließend kann *Philosophen* entweder nach *Philosoph* oder *Philosoph Philosophen* transformiert werden. Die mehrmalige Anwendung der Regel in Zeile 3 erzeugt eine Liste von Philosoph-Objekten. Abschließend wird jedes Philosoph-Objekt in das Format *philosoph <Bezeichner> ;* überführt. Das folgende Modell demonstriert die daraus resultierende Sprache:

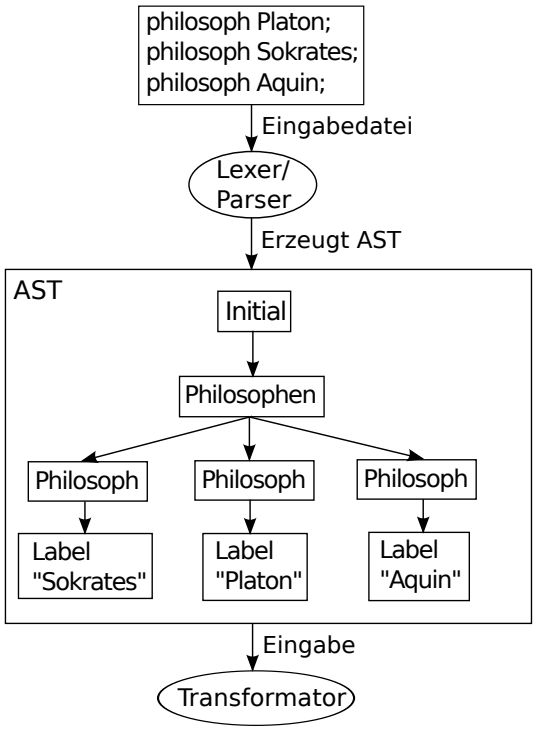

Abbildung 1.6: Erzeugung des AST

```
1 philosoph Platon;
2 philosoph Sokrates;
3 philosoph Aquin;
```

Listing 1.2: Deklaration von drei Philosophen

Damit das Modell aus Listing 1.2 in eine Zielsprache übersetzt werden kann, muss es von einem Parser eingelesen und validiert werden. Das Generieren eines Parsers erfolgt automatisiert mit einem speziellen Werkzeug wie beispielsweise ANTLR [116]. Es liest die zuvor beschriebene Grammatik ein und erzeugt daraus Quellcode, der den Parser repräsentiert. Zur Vereinfachung wird in diesem Abschnitt davon ausgegangen, dass der Parsergenerator Java-Quellcode erzeugt. Auch die Produktionsregeln der Grammatik werden vom Parsergenerator in Klassen übersetzt. Im Fall von Listing 1.1 würden vom Parsergenerator die folgenden Klassen generiert werden: *Initial*, *Philosoph*, *Philosophen* und *Label*. Nach dem Generieren des Parsers kann dieser kompiliert und verwendet werden. Der Parser erhält das Modell aus Listing 1.1 als Eingabedatei und überprüft es auf syntaktische Fehler. Anschließend werden die zuvor aus den Produktionsregeln erzeugten Klassen instanziiert (*Initial*, *Philosophen*, usw.) und zu einem Abstract Syntax Tree zusammengesetzt.

In Abbildung 1.6 ist schematisch gezeigt, wie das Philosophenbeispiel aus Listing 1.2 von einem Parser in einen AST transformiert wird. Das Objekt *Initial* ist das Wurzelelement des AST und enthält einen Zeiger auf ein Philosophen-Objekt. Das Philosphen-Objekt enthält eine Liste von Philosoph-Objekten. Jedem Philosoph-Objekt ist ein *Label* zugeordnet, das als Container für eine Zeichenkette dient und den Namen des jeweiligen Philosophen enthält.

Nach dem Überführen in einen AST kann das Modell in die Zielsprache übersetzt werden. Zu diesem Zweck bekommt der Transformator den Abstract Syntax Tree übergeben. Er greift auf die darin enthaltenen Elemente zu und erzeugt die jeweilige Zielsprache. Das folgende Beispiel zeigt einen möglichen Transformator, der mit Java implementiert ist. Er iteriert über die Elemente des AST und generiert für jeden Philosophen eine gleichnamige Klasse:

```
1 void translate(Initial initial){
2    List<Philosoph> philosophen = initial.getPhilospohen();
3    for(Philosoph p : philosophen){
4      writeToFile("class " + p.getLabel() "{ }\n");
5    }
6 }
```

Listing 1.3: Parsen des AST

Die Methode *translate()* aus Listing 1.3 bekommt das Wurzelobjekt des AST als Parameter übergeben. Anschließend wird über die Philosophen-Liste iteriert und jedes Element in eine Java-Klassendeklaration überführt. Dabei wird davon ausgegangen, dass es eine Methode namens *writeToFile()* gibt, die eine Zeichenkette in die Ausgabedatei des Transformators schreibt.

Auch in der vorliegenden Arbeit werden Modelle mit domänenspezifischen Sprachen abgebildet und dafür Transformatoren implementiert. Daraus lassen sich die folgenden Teilproblemstellungen ableiten:

- Einfache Grammatiken wie in Listing 1.1 lassen sich mit relativ geringem Aufwand implementieren. Die Komplexität steigt jedoch deutlich, wenn Elemente zum Modellieren von Verhalten, wie beispielsweise *Expressions* und *Statements*, in einer DSL genutzt werden sollen. Statements und Expressions werden in Hochsprachen eingesetzt, um Verhalten und mathematische Ausdrücke zu modellieren, wie beispielsweise $inta = (b * sqrt(2)) | c;$. Die steigende Komplexität wird beim Betrachten der Java-Grammatik ersichtlich [115], in der allein 25 Ableitungsregeln mit deutlich erhöhtem Umfang für die Umsetzung von Expressions enthalten sind.

- Durch den erhöhten Umfang der Produktionsregeln steigt auch die Komplexität des zu implementierenden Transformators.

- Bei steigender Komplexität einer Sprache erhöht sich nicht nur der Umfang der

Produktionsregeln, es ist auch zusätzliches Expertenwissen im Bereich Compilerbau notwendig, um beispielsweise Linksrekursion oder Shift-Reduce-Konflikte [1] zu lösen.

Die genannten Problemstellungen müssen gelöst werden, um eine Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation in Software-Projekten zu ermöglichen.

1.2.2 Verifikation von DSLs

Der vorherige Abschnitt erläutert, wie DSLs entwickelt und in eine Zielsprache transformiert werden können. Für die erfolgreiche formale Verifikation von domänenspezifischen Sprachen müssen diese mit Anforderungen angereichert und in eine Model Checker-Eingabesprache übersetzt werden. Der Model Checker verarbeitet die Eingabesprache und untersucht automatisiert, ob der Zustandsraum des Modells nicht-erfüllte Anforderungen enthält. In Abbildung 1.7 ist eine Teilmenge vom Zustandsraum des Philoso-

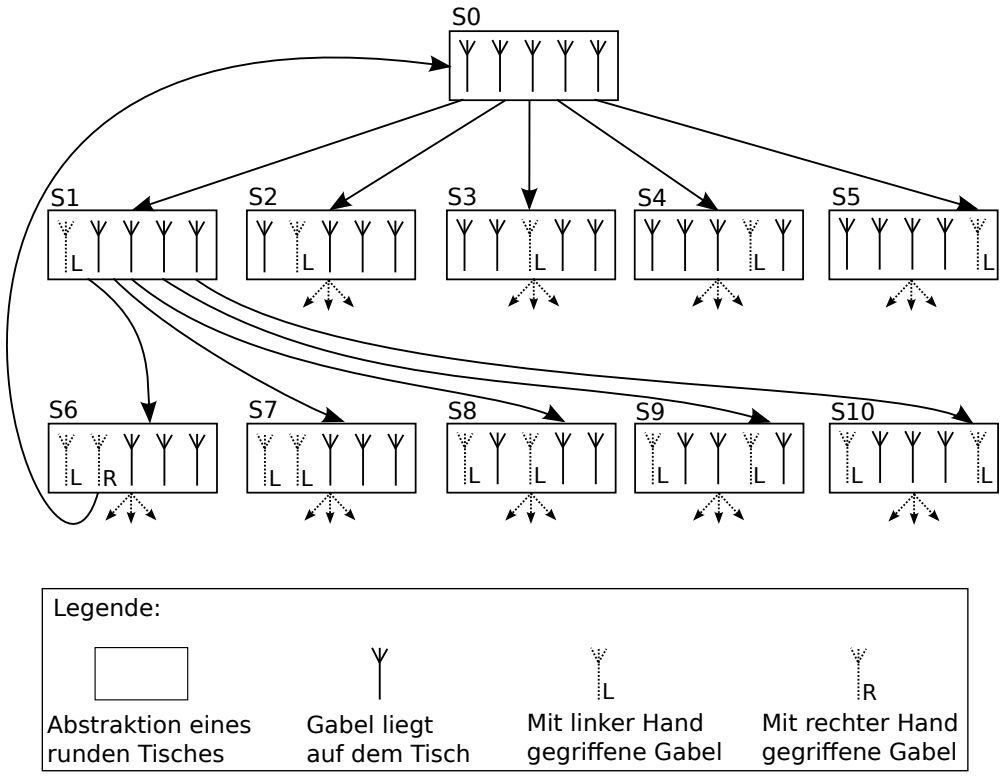

Abbildung 1.7: Teilzustandsraum des Philosophenproblems

phenproblems schematisch dargestellt. Jeder Zustand enthält fünf Gabeln, die entweder auf dem Tisch liegen oder mit linken bzw. rechten Hand eines Philosophen gegriffen wurden. Da es zu keinen Deadlocks kommen darf, muss der Model Checker den gesamten Zustandsraum untersuchen und sicherstellen, dass jeder Zustand mindestens eine ausgehende Transition besitzt.

Zu Beginn des Verifikationsprozesses in S_0 liegen alle Gabeln auf dem Tisch. Anschließend kann eine Transition in einen der fünf möglichen Folgezustände S_1, S_2, S_3, S_4 oder S_5 ausgeführt werden, bei der jeweils einer der Philosophen seine linke Gabel greift. Zur Vereinfachung sind die ausgehenden Transitionen der Zustände S_2 bis S_5 nicht in der Abbildung enthalten und nur mit gestrichelten Linien angedeutet. Ausgehend von Zustand S_1 kann entweder der erste Philosoph die rechte Gabel aufnehmen, oder einer der anderen Philosophen seine linke Gabel. Nachdem der erste Philosoph beide Gabeln aufgenommen hat und sich das System in Zustand S_6 befindet, kann er essen und sie wieder zurück auf den Tisch legen. Danach befindet sich das System wieder im Zustand S_0. Die weiteren ausgehenden Transitionen der Zustände S_6, S_7, S_8, S_9 und S_{10} sind in Abbildung 1.7 nur angedeutet.

Um Zyklen zu erkennen, muss der Zustandsraum eines Modells vom Model Checker in einem Speichermedium, wie beispielsweise dem Hauptspeicher oder der Festplatte, gehalten werden. Die zunehmende Größe eines Modells kann zu einem exponentiellen Wachstum des Zustandsraums führen. Dieses Problem wird auch *State Space Explosion* [29] genannt. Es bewirkt, dass Modelle nicht mehr komplett verifiziert werden können, da der *Model Checker* mehr Speicher benötigt als verfügbar ist.

Um der *State Space Explosion* zu begegnen, können zwei Arten von Optimierungen im Rahmen des *Model Checking* angewendet werden:

- Optimierungen auf *Model Checker-Ebene*.
- Optimierungen auf Modellebene.

Bei den Optimierungen auf *Model Checker-Ebene* handelt es sich um Techniken, die vom *Model Checker* automatisiert durchgeführt werden. Dazu gehört beispielsweise die Kompression des Zustandsraums im Hauptspeicher mit einem Komprimierungsalgorithmus [107]. Bei den Optimierungen auf Modellebene handelt es sich um besondere Modellierungstechniken seitens des Anwenders, die mit der Model Checker-Eingabesprache umgesetzt werden. Das folgende Beispiel verdeutlicht diesen Ansatz:

Gegeben sei eine Model Checker-Eingabesprache, die das Deklarieren von Variablen erlaubt. Variablen können vom Typ *Byte* oder *Integer* sein. Jede Integer-Variable habe eine Größe von vier Bytes. Mit der Eingabesprache soll das Philosophenproblem beschrieben und anschließend verifiziert werden. Die fünf Gabeln werden mit einem Integer-Array implementiert, wobei jedes Element des Arrays eine Gabel repräsentiert und einen Wert von 0, 1 oder 2 annehmen kann:

- 0: Die Gabel liegt auf dem Tisch
- 1: Die Gabel wurde mit der rechten Hand gegriffen.
- 2: DIe Gabel wurde mit der linken Hand gegriffen.

Während der Verifikation legt der Model Checker bereits besuchte Zustände im Speicher ab, um so eventuelle Zyklen im Zustandsraum zu erkennen. Da die aktuelle Belegung des Gabel-Arrays bekannt sein muss, wird pro Zustand ein Integer-Array der Größe fünf gespeichert. Somit hat jeder Zustand eine Größe von 20 Bytes. Der Model Checker würde deshalb 2000 Bytes Hauptspeicher benötigen, um 100 Zustände zu verifizieren. Eine Optimierung auf Modellebene ist in diesem Fall das Ersetzen des Integer-Arrays durch ein Byte-Array. Dadurch reduziert sich die Zustandsgröße von 20 auf 5 Bytes und der Speicherverbrauch des Model Checkers beträgt nur noch 500 Bytes.

Für diese Dissertation lassen sich aus den beschriebenen Aspekten der formalen Verifikation die folgenden Teilprobleme ableiten:

- Bei der automatischen Übersetzung in eine *Model Checker-Eingabesprache* müssen Optimierungstechniken auf Modellebene angewendet werden, die den Zustandsraum eines Modells verkleinern. Dadurch ist sichergestellt, dass der *State Space Explosion* entgegen gewirkt wird und der *Model Checker* auch größere Modelle verifizieren kann. Dies erfordert entsprechendes Expertenwissen.
- Model Checker-Eingabesprachen, wie beispielsweise Promela [68], besitzen Eigenschaften, die von den Konzepten moderner Hochsprachen abweichen. Die Übersetzung einer DSL in eine Model Checker-Eingabesprache erfordert deshalb Expertenwissen. Wenn Software- und DSL-Entwickler keine entsprechenden Kenntnisse vorweisen können, ist die Verwendung eines Model Checkers problematisch. Die Notwendigkeit von Expertenwissen im Bereich der formalen Verifikation muss deshalb reduziert werden, damit Model Checking auch in Projekten eingesetzt werden kann, in denen der DSL-Entwickler über keine entsprechenden Kenntnisse verfügt.

Ein Ziel dieser Arbeit ist die Lösung der genannten Problemstellungen, um eine Verknüpfung von formaler Verifikation und modellgetriebener Entwicklung in beliebigen Software-Projekten zu erreichen.

1.3 Methodik und Lösungsansatz

Das Ziel dieser Arbeit ist es, die formale Verifikation in die modellgetriebene Entwicklung zu integrieren. Des Weiteren soll das zu entwickelnde Verfahren in möglichst vielen Softwareprojekten einsetzbar sein. Die vorangegangenen Abschnitte 1.2.1 und 1.2.2 haben jedoch gezeigt, dass dafür die Berücksichtigung von verschiedenen Aspekten, wie das

Beschreiben von Produktionsregeln, Optimierungen auf Modellebene, usw. notwendig ist.

Es ist wünschenswert, dass die sich daraus ergebenen Problemstellungen nicht in jedem Software-Projekt, das MDD und formale Verifikation miteinander kombiniert, erneut bearbeitet werden müssen. Deshalb wird im Rahmen dieser Arbeit ein Framework vorgestellt, das Softwareentwickler bei der Verknüpfung von modellgetriebener Entwicklung formaler Verifikation unterstützt. Dieses Framework trägt den Namen *DSL Verification Framework* bzw. DVF [6]. Es wird im weiteren Verlauf der Arbeit sowohl entworfen, als auch prototypisch realisiert. Zu beachten ist, dass dieser Abschnitt lediglich die Konzepte des DVF vorstellt. Eine detaillierte Beschreibung erfolgt nach Vorstellung der Grundlagen und einer Analyse der verwandten Arbeiten in Abschnitt 4.

Die Problemanalysen aus den vorangegangenen Abschnitten haben gezeigt, dass für die Umsetzung einer domänenspezifischen Sprache und der anschließenden Transformation in eine Hoch- bzw. Model Checker-Eingabesprache die folgenden Schritte notwendig sind:

1. Es muss eine Grammatik geschrieben werden, die aus einer Menge von Produktionsregeln besteht. Ein Parsergenerator erzeugt daraus einen Parser.

2. Es muss ein Transformator geschrieben werden, der den Parser nutzt und die mit der DSL umgesetzten Modelle in eine Model Checker-Eingabesprache übersetzt.

3. Es muss ein Transformator geschrieben werden, der den Parser nutzt und die mit der DSL umgesetzten Modelle in eine Hochsprache übersetzt.

Um diesen Prozess zu vereinfachen und die Notwendigkeit von Expertenwissen zu reduzieren, besteht das DSL Verification Framework aus zwei zentralen Komponenten:

- Einer vorgefertigten Menge von Produktionsregeln.

- Vorgefertigten Transformatoren, die die Elemente einer DSL in eine Hoch- bzw. Model Checker-Eingabesprache übersetzen.

Somit implementiert ein DSL-Entwickler, wenn er eine domänenspezifische Sprache mit dem DVF umsetzt, nicht alle Produktionsregeln und Transformatoren manuell. Stattdessen entnimmt er, wie aus einem Bausatz, die für ihn relevanten Elemente und integriert sie in sein DSL-Projekt. Somit müssen lediglich die Produktionsregeln und Komponenten des Transformators manuell umgesetzt werden, die nicht Teil des DVF sind.

Bevor die Konzepte des DSL Verification Frameworks genauer ausgeführt werden, verdeutlicht ein Beispiel die Problemstellung. Dabei handelt es sich um eine Erweiterung der Philosophen-DSL aus Listing 1.1. Mit ihr werden im weiteren Verlauf des Kapitels die Komponenten und Mechanismen des DVF veranschaulicht. Sie basiert auf der folgenden Grammatik:

```
1 Initial -> Variablen
2 Variablen -> Variable Variablen
3 Variablen -> Philosophen
4
5 Philosophen -> Philosoph Philosophen
6 Philosophen -> Philosoph
7 Philosoph -> philosoph Label { Statements }
8 Label -> [a..z..A..Z]*
```

Listing 1.4: Erweiterte Grammatik zum Deklarieren einer Menge von Philosophen

Die Grammatik zeigt, dass jedes Modell aus einer Menge von globalen Variablen und einer Menge von Philosophen besteht. Ein Philosoph enthält Statements, die ausgeführt werden, wenn der Philosoph Hunger bekommt. Zur besseren Übersicht sind die Produktionsregeln *Variable* und *Statements* nicht in Listing 1.4 enthalten. Das nächste Beispiel verdeutlicht die Nutzung der Philosophen-DSL mit einem konkreten Modell:

```
1 int[5] gabeln;
2
3 Philosoph Platon{
4   wait(gabeln[0] == 0){ gabeln[0] = 1; }
5   wait(gabeln[1] == 0){ gabeln[1] = 2; }
6   //Essen ...
7   gabeln[0] = 0;
8   gabeln[1] = 0;
9 }
10
11 //...
```

Listing 1.5: Modell der erweiterten DSL für Philosophen

Das Modell in Listing 1.5 besteht aus einem Array und einem Philosophen. Die Semantik des Gabel-Arrays ist so wie in Abschnitt 1.2.2 beschrieben umgesetzt: Ein Wert von 0 bedeutet, dass die entsprechende Gabel auf dem Tisch liegt. Wenn eine Gabel mit der linken Hand gegriffen wurde, nimmt das entsprechende Element im Array den Wert 1 an. Sollte ein Philosoph hingegen die Gabel mit seiner rechten Hand aufnehmen, enthält das Element im Array eine 2.

Im Philosoph Platon sind vier Statements enthalten, die ausgeführt werden, wenn er Hunger bekommt. Nach dem Erreichen eines Wait-Blocks pausiert der Philosoph so lange, bis die darin enthaltene Expression den Wert *true* annimmt. Im Fall von Listing 1.5 wird somit der Philosph Platon, wenn er Hunger bekommt, zunächst überprüfen, ob die Gabeln auf dem Teller liegen, sie aufnehmen und anschließend wieder zurücklegen.

Ausgehend von der Philosophen-DSL wird im weiteren Verlauf dieses Abschnitts

das DSL Verification Framework vorgestellt. Es enthält zur Unterstützung von DSL-Entwicklern eine Menge von vorgefertigten Produktionsregeln. Diese Produktionsregeln können bei Bedarf in die Grammatik einer domänenspezifischen Sprache eingefügt werden (für eine vollständige Auflistung der Produktionsregeln siehe Abschnitt 4.3). Dazu gehören beispielsweise Regeln für Statements, Expressions, Variablen oder Message-Queues. Im Fall der Grammatik aus Listing 1.4 führt dies zu dem folgenden Ergebnis:

- Ein DSL-Entwickler muss lediglich die Produktionsregeln für *Philosoph*, *Philosophen* und *Label* manuell implementieren.
- Die Produktionsregeln für *Statements* und *Variablen* entnimmt er dem DVF und fügt sie in sein DSL-Projekt bzw. seine Grammatik ein.

Somit muss ein DSL-Entwickler, wenn er das DVF nutzt, nur noch einen Teil der Produktionsregeln manuell implementieren.

Aus der Grammatik wird ein Parser generiert. Er liest Modelle ein und erzeugt einen Abstract Syntax Tree. Der AST wird von einem Transformator entgegengenommen, und in die entsprechende Zielsprache überführt. Deshalb muss der DSL-Entwickler nicht nur eine Grammatik, sondern auch einen Transformator implementieren. Um auch hier den Aufwand zu reduzieren, beinhaltet das DSL Verification Framework zwei vorgefertigte Transformatoren. Sie überführen automatisiert alle Objekte des Abstract Syntax Trees, die auf den Produktionsregeln des DVF basieren, in eine Hoch- und eine Model Checker-Eingabesprache. Dadurch ergeben sich die folgenden Vorteile:

- Der DSL-Entwickler muss keine Transformatoren für die Produktionsregeln implementieren, die er aus dem DVF entnommen hat. Stattdessen stellt das DSL Verification Framework bereits vorgefertigte Transformatoren zur Verfügung.
- Der DSL-Entwickler benötigt keine Kenntnisse im Bereich Model Checking, da der Transformator des DVF auch eine Übersetzung in eine Model Checker-Eingabesprache vornimmt und automatisiert die bereits angesprochenen Optimierungen auf Modellebene anwendet.

Obwohl das DVF vorgefertigte Produktionsregeln und Transformatoren enthält, führt der beschriebene Ansatz zu dem folgenden Problem: Modelle können nicht automatisiert in die Zielsprachen überführt werden. Dies liegt an genau den Elementen der DSL, die nicht Teil des DVF sind. Im Fall der Grammatik aus Listing 1.4 wird beispielsweise die Produktionsregel *Philosoph* manuell vom DSL-Entwickler umgesetzt. Das bedeutet, die Transformatoren des DVF können zwar Statements und Variablen verarbeiten, eine Übersetzung von Philosoph-Elementen ist jedoch nicht möglich. Ein DSL-Entwickler müsste, um dieses Problem zu lösen, die Transformatoren des DVF manuell so anpassen, dass sie das Element *Philosoph* in einen *Thread* oder etwas Vergleichbares überführen.

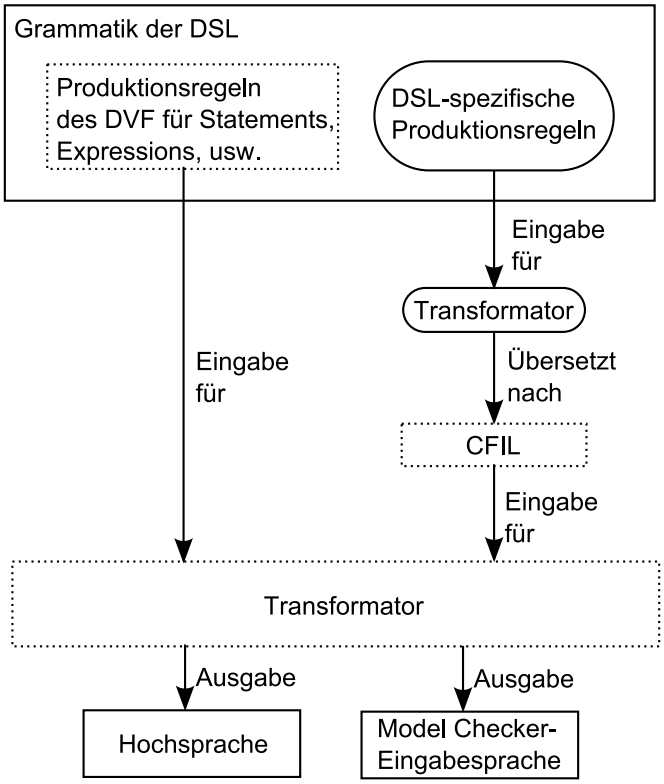

Abbildung 1.8: Komponenten des DSL Verification Frameworks

Das manuelle Implementieren bzw. Modifizieren eines Transformators ist jedoch nicht wünschenswert, da hierfür wieder Expertenwissen im Bereich der jeweiligen Zielsprachen notwendig ist. Da die Transformatoren des DVF auch eine Übersetzung in eine Model Checker-Eingabesprache durchführen, müssten somit für eine entsprechende Modifikation auch tiefergehende Kenntnisse im Bereich der formalen Verifikation vorhanden sein. Durch eine Modifikation des Transformators würde deshalb ein wichtiger Vorteil des DVF verloren gehen.

Um diesem Problem zu begegnen, wird das DVF folgendermaßen erweitert: Neben Produktionsregeln und Transformatoren für Statements, Expressions, usw. sind auch plattformunabhängige Sprachkonstrukte zum Steuern des Kontrollflusses Teil des DVF. Sie werden im weiteren Verlauf der Arbeit *Control Flow Intermediate Language* bzw. CFIL genannt. Der sich daraus ergebende Aufbau des DVF ist in Abbildung 1.8 zu sehen. Dabei sind die Elemente einer DSL, die der DSL-Entwickler implementieren muss, mit abgerundeten Ecken hervorgehoben. Alle Elemente, die das DVF zur Verfügung stellt, sind mit gestrichelten Linien markiert. Die Nutzung des DVF gestaltet sich fol-

gendermaßen: Ein DSL-Entwickler schreibt für seine DSL eine Grammatik und somit Produktionsregeln. Dabei implementiert er Produktionsregeln für Statements, Expressions, usw. nicht manuell, sondern verwendet die vorgefertigten Elemente des DVF. Lediglich für die DSL-spezifischen Sprachkonstrukte, wie beispielsweise das Element *Philosoph* aus Listing 1.5, müssen vom DSL-Entwickler noch die entsprechenden Produkionsregeln „von Hand“ geschrieben werden. Damit der DSL-Entwickler keinen Transformator entwickeln bzw. anpassen muss, der die DSL-spezifischen Elemente in die Model Checker-Eingabesprache überführt, wird vom DVF die *Control Flow Intermediate Language* zur Verfügung gestellt. Der DSL-Entwickler schreibt deshalb einen Transformator, der die DSL-spezifischen Elemente in die CFIL überführt. Die CFIL kann anschließend vom DVF in die beiden Zielsprachen übersetzt werden. Der Vorteil dieser zusätzlichen Abstraktionsebene ist, dass ein DSL-Entwickler die Elemente seiner DSL, die nicht Teil des DVF sind, lediglich in die plattformunabhängige CFIL übersetzen muss. Die CFIL wird anschließend vom DVF automatisiert in die Hoch- bzw. Model Checker Eingabesprache transformiert. Der DSL-Entwickler benötigt somit keine Kenntnisse im Bereich formalen Verifikation, wenn er vollständig auf die CFIL-Komponenten zurückgreift.

1.4 Gliederung der Arbeit

Diese Arbeit hat den folgenden Aufbau: Das DSL Verification Framework basiert auf einer Reihe unterschiedlicher Technologien. Aus diesem Grund erfolgt zunächst in Abschnitt 2 eine Vorstellung der für das Verständnis dieser Dissertation notwendigen Grundlagen. Dazu gehört in Kapitel 2.1 die Auswahl eines geeigneten Model Checkers und das Beschreiben seiner Eingabesprache. Anschließend evaluiert Abschnitt 2.2, welcher Parsergenerator für die Umsetzung des DVF geeignet ist. Der Model Checker und der Parsergenerator werden in den darauf folgenden Kapiteln genutzt, um den Prototypen des DVF zu implementieren. Weitere wichtige Grundlagen sind UML-Statecharts und Web-Applikationen, da sie im Rahmen der Fallstudien Verwendung finden. Entsprechende Einführungen sind in Abschnitt 2.3 und 2.4 enthalten.

Nach der Vorstellung aller Grundlagen wird in Kapitel 3 der Stand der Technik beschrieben. Von besonderem Interesse ist, wie sich die vorliegende Dissertation von ihm unterscheidet bzw. welche Vorteile das DSL Verification Framework bietet.

Nach einer Diskussion des Stands der Forschung wird in Kapitel 4 das DSL Verification Framework (DVF) vorgestellt. Das Ziel des DVF ist es, die Entwicklung und die automatische Transformation von DSLs in sowohl Hoch-, als auch Model Checker-Eingabesprachen zu unterstützen. Zu diesem Zweck stellt es eine Menge von Produktionsregeln bereit, die DSL-Entwickler in ihren domänenspezifischen Sprachen einsetzen können. Die Produktionsregeln werden in Abschnitt 4.3 genauer beschrieben. Die mit dem DVF umgesetzten DSLs werden vom DSL-Anwender zum Beschreiben von Modellen genutzt. Kapitel 4.5 zeigt, wie das DSL Verification Framework die entsprechenden

Modelle validiert. Im Anschluss werden in 4.6 und 4.7 die Transformatoren vorgestellt, die Modelle automatisch in eine Hoch- und Model Checker-Eingabesprache überführen.

Um zu evaluieren, ob und mit welchem Gewinn das DVF auch in der Praxis eingesetzt werden kann, beinhaltet diese Arbeit zwei Industriefallstudien. Die erste Fallstudie in Abschnitt 5.1 ist eine DSL für die Implementierung eines Systems namens *assyControl*, das zur Überwachung von Handmontageprozessen mittels Ultraschall dient. Die Präsentation der zweiten Fallstudie erfolgt in Abschnitt 5.2. Dabei handelt es sich um ein Software-System namens *InTune*, das das Ausführen von Tests in verteilten Systemen ermöglicht. Sowohl *assyControl*, als auch *inTune*, werden von der *soft2tec GmbH* [132] entwickelt. Zum Abschluss erfolgt in Kapitel 6 sowohl eine Zusammenfassung der gesamten Dissertation, als auch ein Ausblick auf weiterführende Arbeiten.

Diese Arbeit ist Teil des Forschungsprojektes *„Korrekte verteilte Java-Applikationen“* (KoverJa)[79], das vom *Bundesministerium für Bildung und Forschung* gefördert wird. Bei allen Abbildungen vom Typ Aktivitäts-, Klassen- oder Zustandsdiagramm handelt es sich um die entsprechenden Diagrammtypen aus der UML-Spezifikation Version 2.0 [111].

2 Grundlagen

Das Ziel dieser Arbeit ist die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation. Dazu wird in Kapitel 1 die Problemstellung in Teilprobleme aufgeteilt und das *DSL Verification Framework* (DVF) als Lösungsansatz vorgestellt. Obwohl die Konzepte des DVF möglichst unabhängig von konkreten Werkzeugen gehalten sind, werden für dessen Implementierung bestehende Technologien eingesetzt.

Diese Technologien leiten sich aus dem Aufbau einer domänenspezifischen Sprache ab, der in Abbildung 2.1 zu sehen ist und nun genauer vorgestellt wird. Eine DSL, bei deren Umsetzung das DVF Verwendung findet, setzt sich aus drei Ebenen zusammen. Auf der obersten Ebene befindet sich eine domänenspezifische Sprache, die ein DSL-Entwickler umgesetzt hat. Sie besteht aus DSL-spezifischen Produktionsregeln und einem Transformator, der die entsprechenden Elemente der DSL in die Control Flow Intermediate Language (CFIL) überführt. In der zweiten Ebene ist das DSL Verification Framework angesiedelt. Es stellt generisch einsetzbare Produktionsregeln zur Verfügung, die ein DSL-Entwickler in seine Grammatik integriert. Des Weiteren gehört ein Transformator zum DVF, der ein Modell, das in der CFIL vorliegt, in eine Hoch- bzw. Model Checker-Eingabesprache überführt. Die dritte Ebene besteht aus einem Model Checker und einem Parsergenerator.

- Der Parsergenerator erzeugt aus den Produktionsregeln des DSL-Entwicklers einen Parser, der die ihm übergebenen Modelle in einen Abstract Syntax Tree (AST) überführt. Anschließend kann der AST von einem Transformator eingelesen und weiterverareitet werden.

- Der Model Checker stellt eine Eingabesprache zur Verfügung, in die die CFIL-Modelle vom DVF überführt werden. Dadurch kann der Model Checker den Zustandsraum untersuchen und sicherstellen, dass alle Anforderungen erfüllt sind.

Für die prototypische Implementierung des DVF, die in Kapitel 4 vorgestellt wird, müssen somit aus bestehenden Werkzeugen ein geeigneter Model Checker und Parsergenerator ausgewählt werden. Deshalb stellt dieses Kapitel im Rahmen von Abschnitt 2.1 verschiedene Model Checker vor und kategorisiert sie. Anhand der unterschiedlichen Stärken und Schwächen wird der Model Checker Spin [68] ausgewählt. Er nutzt die Eingabesprache Promela. Eine Einführung in die Grundlagen von Spin bzw. Promela, die für das Verständnis dieser Arbeit notwendig sind, erfolgt in den Abschnitten 2.1.1 und 2.1.2.

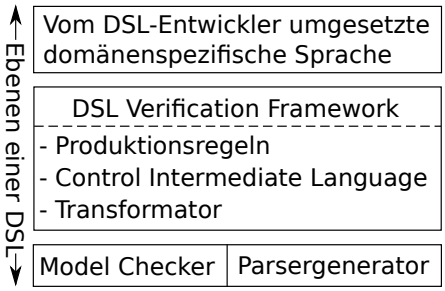

Abbildung 2.1: Architektur des DSL Verification Frameworks

Ein weiterer wichtiger Aspekt im DVF ist die modellgetriebene Entwicklung. Für die Umsetzung einer domänenspezifischen Sprache im Rahmen des *Model-Driven Development* (MDD) ist ein Parsergenerator notwendig. Neben reinen Parsergeneratoren gibt es auch sogenannte DSL-Frameworks. Diese enthalten zusätzliche Komponenten, wie beispielsweise eine automatische Validierung, um den DSL-Entwickler und -Anwender bei ihrer Arbeit zu unterstützen. Das DVF nutzt deshalb keinen reinen Parsergenerator, sondern ein DSL-Framework namens Xtext. Xtext verwendet den ANTLR-Parsergenerator [116] und stellt eigene Beschreibungssprachen für Produktionsregeln bzw. Modelltransformatoren zur Verfügung. Deshalb werden in Abschnitt 2.2.1 die für das Verständnis dieser Arbeit notwendigen Xtext-Grundlagen beschrieben.

Ein weiterer wichtiger Aspekt dieser Arbeit sind zwei industrielle Fallstudien, um das DVF zu evaluieren. Die erste Fallstudie verwendet eine domänenspezifische Sprache zur Beschreibung von endlichen Automaten. Daher wird in Abschnitt 2.3 eine Einführung in UML-Statecharts [111] gegeben. Die zweite Fallstudie implementiert mit dem *Google Web Toolkit* (GWT) eine Web-Anwendung. In Kapitel 2.4 wird daher das GWT genauer vorgestellt.

2.1 Formale Verifikation

Dieser Abschnitt gibt eine Einführung in das Konzept der formalen Verifikation und stellt verschiedene Model Checker vor. Von den bestehenden Model Checkern wird Spin ausgewählt, um im Rahmen des DSL Verification Frameworks genutzt zu werden. Das bedeutet gemäß Abbildung 1.8: Der Transformator des DVF, der ein Modell in eine Model Checker-Eingabesprache überführt, wird in der prototypischen Implementierung in Abschnitt 4 eine Übersetzung nach Promela vornehmen.

Der Begriff formale Verifikation steht unter anderem für einen Ansatz, bei dem ein Model Checker automatisiert überprüft, ob das Modell M die Spezifikation ϕ vollständig erfüllt ($M \models \phi$) [31][80]. Damit das Modell und die Spezifikation von einem Model

Checker verifiziert werden können, müssen sie zunächst mit einer *Model Checker-Eingabesprache* formalisiert werden. Nach der Formalisierung werden sie dem Model Checker als Eingabe übergeben und der Verifikationsprozess beginnt. Dafür gibt es verschiedene Vorgehensweisen [31]. So kann ein Model Checker die Eingabe in einen endlichen Automaten überführen. Beim Verifizieren wird ein Graph erzeugt, der dem Zustandsraum des Modells entspricht und für jeden Zustand untersucht, ob er die entsprechenden Anforderungen erfüllt.

Es gibt Model Checker mit verschiedenen Stärken und Schwächen für unterschiedliche Anwendungsfälle. Dies wird beispielsweise in der Arbeit von Frappier et al. [51] deutlich, die verschiedene Model Checker-Werkzeuge miteinander vergleicht. Daher muss für die formale Verifikation von Softwaremodellen in Form von domänenspezifischen Sprachen zunächst der passende Model Checker gefunden werden [3].

Wichtige Kriterien bei der Auswahl sind, neben dem eigentlichen Anwendungsschwerpunkt: die Verbreitung, aktive Weiterentwicklung und Performanz. Ein weiterer Aspekt ist gemäß Abschnitt 1.3 die Möglichkeit, die State Space Explosion durch automatische Optimierungen auf Model Checker- und Modellebene zu vermeiden. Diesbezüglich gibt es die folgenden Ansätze: Modelle können in Binary Decision Diagrams (BDD) [45] überführt und mit optimierten Suchverfahren verifiziert werden [95]. Des weiteren kann der Zustandsraum auf Kosten der Laufzeit komprimiert werden [107], um so den Speicherverbrauch des Model Checkers zu reduzieren. Im Rahmen der *Partial Order Reduction* [68][57] wird der Zustandsraum vom Model Checker analysiert und die Zustände nebenläufiger Prozesse in Äquivalenzklassen zusammengefasst. Das *Bitstate Hashing* [67] ähnelt der Kompression des Zustandsraums. Es werden allerdings keine komprimierten Zustände, sondern stattdessen Hash-Werte gespeichert. Eine weitere Optimierung ist das Ausnutzen symmetrischer Eigenschaften in einem Modell. Symmetrie [109][41] ist in ein Spezialfall, der zum Teil automatisiert und zum Teil vom Anwender auf Modellebene eingesetzt werden kann. Deshalb wird Symmetrie im Zusammenhang mit dieser Arbeit in Abschnitt 2.1.4 gesondert vorgestellt.

Bekannte und verbreitete Model Checker sind beispielsweise VIS [21] und SMV [28]. Der Anwendungsschwerpunkt von VIS/SMV ist die Verifikation von Hardwarebeschreibungssprachen, wie beispielsweise VHDL [71]. Die Eingabesprache von SMV ist an die Konzepte von VHDL angelehnt, die sich deutlich von Programmiersprachen wie beispielsweise Java abgrenzen. Für die Verifikation von DSLs ist deshalb die Verwendung eines Model Checkers wünschenswert, der nicht auf Hardware, sondern auf die Verifikation von Software spezialisiert ist.

Verbreitete Model Checker zur Software-Verifikation sind der *Java Pathfinder* [148] und *Blast* [17], die direkt Java- bzw. C-Quellcode einlesen können. Sie sind im Vergleich zu VIS/SMV besser für die formale Verifikation von DSLs geeignet. Beide haben den Vorteil, dass keine zusätzliche Transformation in eine Model Checker-Eingabesprache notwendig ist, da sie Hochsprachen direkt verifizieren können. Ein Nachteil ist, dass bei-

de Model Checker nicht den vollen Sprachumfang von C bzw. Java unterstützen und somit der Quellcode der Modelle entsprechend angepasst werden muss. Des Weiteren geht durch die Nutzung von Blast bzw. dem Java Pathfinder die hohe Abstraktionsebene von DSLs verloren. Dies ist ein Nachteil, da Modelle auf hoher Abstraktionsebene, wenn sie von einem Model Checker untersucht werden, typischerweise einen kleineren Zustandsraum haben. Der Model Checker benötigt dadurch weniger Hauptspeicher und dem Problem der State Space Explosion wird entgegengewirkt.

Für die Verifikation einer DSL ist deshalb ein Model Checker wünschenswert, der eine Eingabesprache zur Verfügung stellt, in die DSLs transformiert werden können, ohne ihren hohen Abstraktionsgrad zu verlieren. In dieser Arbeit wird daher mit dem *System Software Award* [12] ausgezeichneten *Spin Model Checker* [68] gearbeitet, um Modelle zu verifizieren. Spin verfügt über einen offenen Quellcode, ist weit verbreitet und wird aktiv weiterentwickelt. Der Schwerpunkt von Spin liegt auf Analyse verteilter, kommunizierender Systeme. Spins Eingabesprache Promela ist an die Hochsprache C angelehnt und erlaubt sowohl detailliertere Implementierungen, als auch das Erstellen von Modellen auf einem höherem Abstraktionsgrad durch beispielsweise spezielle Datentypen zur Umsetzung von Message-Queues für synchrone oder asynchrone Kommunikation. Des Weiteren wird von Promela auch das Modellieren von Nicht-Determinismus unterstützt. Um der bereits angesprochenen State Space Explosion zu begegnen, beherrscht Spin eine Reihe von Optimierungstechniken auf Model Checker-Ebene, wie beispielsweise die Kompression des Zustandsraums, *Partial Order Reduction* [68] oder *Bitstate Hashing* [67].

Zum besseren Verständnis dieser Dissertation wird in Abschnitt 2.1.1 eine Einführung in den Model Checker Spin gegeben. Eine genaue Beschreibung der Syntax bzw. Semantik von Promela-Modellen und den entsprechenden Anforderungen folgt in Abschnitt 2.1.2. Abschließend wird in Abschnitt 2.1.3 ein konkretes Beispiel mit Promela implementiert, verifiziert und auch der Begriff *Zustandsraum* im Zusammenhang mit Spin genauer erläutert. Um den Speicherverbrauch des Verifikationsprozesses zu minimieren, können symmetrische Eigenschaften in Modellen ausgenutzt werden [109]. Zum Abschluss wird deshalb in Abschnitt 2.1.4 beschrieben, wie Symmetrie mit Spin und Promela genutzt werden kann. Zu beachten ist, dass in diesem Kapitel keine vollständige Einführung in Spin bzw. Promela gegeben wird, sondern lediglich die für diese Arbeit relevanten Sprachkonstrukte vorgestellt werden.

2.1.1 Spin

Dieser Abschnitt stellt den Model Checker Spin vor. Er wird benötigt, um den vom DVF generierten Promela-Quellcode zu verifizieren. Wenn Spin keine nicht-erfüllten Anforderungen findet, kann das Modell abschließend vom DVF in eine Hochsprache transformiert werden und in dem entsprechenden Softwareprojekt Verwendung finden.

Der von Gerard J. Holzmann entwickelte Model Checker Spin dient zum Verifizieren von Softwaresystemen und Kommunikationsprotokollen. Spin wird mit offenem Quellcode ausgeliefert und kann deshalb auf unterschiedlichen Plattformen, beispielsweise Windows [100], Linux [144] oder FreeBSD [139] kompiliert und ausgeführt werden. Die

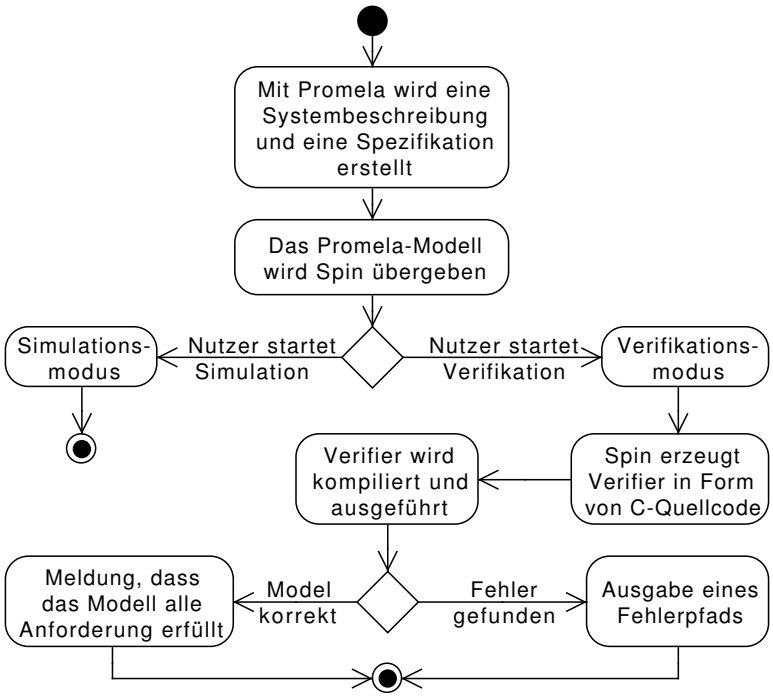

Abbildung 2.2: Spin-Workflow (Aktivitätsdiagramm)

Nutzung und der Aufbau von Spin sind in Abbildung 2.2 schematisch als UML-Aktivitätsdiagramm dargestellt: Ein Entwickler benutzt die Eingabesprache Promela, um das System und die Anforderungen zu beschreiben. Das so erzeugte Promela-Modell wird Spin anschließend per Kommandozeile übergeben. Spin hat zwei Betriebsmodi:

- Im Simulationsmodus kann der Anwender interaktiv das System ausführen und so beispielsweise Implementierungsfehler finden. Dieser Modus wird unter anderem zum *Debuggen* eines Modells eingesetzt.
- Im Verifikationsmodus wird aus Modell und Anforderungen ein sogenannter Verifier generiert, der automatisiert den gesamten Zustandsraum eines Modells untersucht.

Spin generiert den Verifier in Form von C-Quellcode. Um ihn ausführen zu können, muss er zunächst kompiliert werden. Nach dem Starten untersucht der Verifier automatisiert

alle Zustände, die ein System annehmen kann und meldet einen Fehler, falls in einem der Zustände eine Anforderung nicht erfüllt ist. Wenn ein Fehler gefunden wird, kann Spin optional einen Fehlerpfad ausgeben.

2.1.2 Promela

In diesem Abschnitt erfolgt eine Einführung in die Konzepte von Promela. Eine Beschreibung von Promela ist für das Verständnis dieser Dissertation notwendig, da der Transformator in Kapitel 4 Modelle in die Spin-Eingabesprache überführt. Promelas Semantik weicht von klassischen Hochsprachen wie Java ab, orientiert sich stattdessen an Dijkstras *Guarded Command Language* [40] und erlaubt das Modellieren von nebenläufigen Prozessen, die synchron oder asynchron kommunizieren.

Das folgende Beispiel demonstriert, wie eine einfache *Hello World-Applikation* mit Promela beschrieben werden kann:

```
1 active proctype Main(){
2   printf("Hello World\n");
3 }
```

Listing 2.1: Hello World in Promela

In Listing 2.1 wird der Prozess *Main* deklariert. Das Schlüsselwort *active* bewirkt, dass der Prozess nicht nur deklariert, sondern auch eine Instanz des Prozesses gestartet wird. *Printf()* wird zum Anzeigen von Zeichenketten genutzt und gibt die Zeichenkette *Hello World* aus. Zu beachten ist, dass Anweisungen zur Ausgabe von Text hilfreich sind, um die Analyse von Fehlerpfaden übersichtlicher zu gestalten. Printf-Anweisungen sind jedoch kein zentraler Bestandteil vom Promela, da sie das Model Checker-Ergebnis nicht beeinflussen.

Im vorherigem Beispiel wird lediglich ein Prozess instanziiert. Das nächste Beispiel zeigt, wie in Promela mehrere Instanzen eines Prozesses gestartet und ihnen Parameter übergeben werden können. Diese Spracheigenschaft ist für das DVF relevant, da auch die CFIL ein Sprachkonstrukt für nebenläufige Prozesse enthält, das vom Transformator des DVF nach Promela überführt wird.

```
1 proctype PrintNumber(byte printMe){
2   printf("Byte: %d", printMe);
3 }
4 init{
5   atomic{
6     run PrintNumber(1);
7     run PrintNumber(2);
8   }}
```

Listing 2.2: Instanziieren mehrerer Prozesse

In Listing 2.2 werden die Prozesse *PrintNumber* und *init* deklariert. Das Sprachkonstrukt *init* ist ein besonderes Element in Promela und entspricht einem Prozess, der zum Initialisieren des Gesamtsystems dient. Die Anweisung *atomic* fasst mehrere *Statements* zu einem *Statement* zusammen. Dies beeinflusst die Semantik von Zeile 6 und 7 wie folgt:

- Die beiden Run-Anweisungen werden zum Zeitpunkt t_x ausgeführt.
- Ohne das Atomic-Statement werden Zeile 6 zum Zeitpunkt t_x und Zeile 7 zum Zeitpunkt t_{x+1} ausgeführt.

Der Prozess *PrintNumber* bekommt ein *Byte* als Parameter übergeben und gibt den Wert der Parametervariable anschließend mit der bereits bekannten *printf()* Funktion aus.

Jeder Prozess hat eine eindeutige Identifikationsnummer, die ihm zum Zeitpunkt der Instanziierung zugewiesen wird. Die Vergabe dieser Prozess-IDs ist im Rahmen der vorliegenden Arbeit für das Umsetzen und Transformieren von sogenannten symmetrischen Arrays relevant, bei denen es sich um eine Optimierung auf Modellebene handelt und die in Abschnitt 2.1.4 genauer vorgestellt werden. Der Prozess, der als erstes gestartet wird, erhält die Prozess-ID 0, der zweite die Prozess-ID 1, usw. Wenn ein Prozess terminiert, kann seine Identifikationsnummer neu vergeben werden. Das Promela-Schlüsselwort *_pid* erlaubt die Referenzierung der jeweiligen Prozess-ID. Da es sich bei dem Init-Block auch um einen Prozess handelt und dieser immer als erstes in einem Modell instanziiert wird, erhält er die Null als eindeutige Identifikationsnummer.

In Listing 2.2 ist lediglich eine Variable beim Starten eines Prozesses als Parameter übergeben worden. Das DVF beinhaltet jedoch eine Produktionsregel, um Variablen mit verschiedenen Datentypen, Arrays, usw. deklarieren zu können. Da der Transformator des DVF diese Produktionsregel nach Promela überführt, erklärt das nächste Beispiel genauer, wie Variablen mit Promela deklariert werden können:

```
1 mtype = {e1, e2}
2 mtype global_enum;
3 byte global_byte;
4
5 active proctype PrintNumber(){
6   int local_int;
7   global_enum = e2;
8   global_byte = 4;
9   local_int = global_byte / 2;
10  printf("local_int: %d\n", local_int);
11 }
```

Listing 2.3: Lokale und globale Variablen

In Listing 2.3 wird der Prozess *PrintNumber* deklariert, der auf lokale und globale Variablen zugreift. Die lokale Variable *local_int* kann nur innerhalb des Prozesses *PrintNumber* referenziert werden, während jeder Prozess auf die globalen Variablen *global_enum* und *global_byte* zugreifen kann. Die verwendeten Datentypen, wie beispielsweise *byte* und *int*, haben die selbe Syntax und Semantik wie in Java oder C. Die einzige Ausnahme stellen Mtypes dar, bei denen es sich um Aufzählungen handelt. Der Zugriff auf Variablen erfolgt mit Operatoren für Bitmanipulation, Arithmetik, usw.

In den bisher gezeigten Promela-Beispielen werden Prozesse erzeugt, Variablen deklariert und Berechnungen durchgeführt. Das DVF enthält jedoch auch Produktionsregeln zur Umsetzung von Statements und Expressions. Ein DSL-Entwicker kann diese Produktionsregeln in seine Grammatik integrieren, um die domänenspezifische Sprache mit Sprachkonstrukten wie If-Blöcken oder Schleifen zu ergänzen. Um in Abschnitt 4.6.11 zu verstehen, wie der DVF-Transformator derartige Sprachkonstrukte in die jeweilige Zielsprache übersetzt, wird mit den folgenden Beispielen die Umsetzung von Schleifen, bedingter Ausführung, usw. in Promela gezeigt:

```
1 byte a = 1;
2
3 active proctype Main(){
4   if
5   :: a == 1 -> printf("a == 1\n");
6   :: a == 2 -> printf("a == 2\n");
7   fi;
8 }
```

Listing 2.4: If-Anweisung in Promela

Ein If-Statement besteht aus einer Menge von Expressions. Jeder Expression sind ein oder mehrere Statements mit dem Pfeil-Operator zugeordnet, die ausgeführt werden, wenn die entsprechende Expression den Wert *true* annimmt. In Listing 2.4 wird in Abhängigkeit von der Variablen *a* entweder *a == 1* oder *a == 2* ausgegeben. Dabei ist zu beachten, dass ein If-Statement in Promela blockierend sein kann und somit von der Semantik anderer Hochsprachen abweicht. Wird beispielsweise Listing 2.4 modifiziert und *a* mit dem Wert 3 initialisiert, so würde der Prozess *Main* nach Erreichen des If-Statement so lange blockieren, bis *a* den Wert 1 oder 2 annimmt und eine der beiden Verzweigungen ausgeführt werden kann. Um das Blockieren des If-Statements zu unterbinden, muss ein Else-Block eingefügt werden:

```
1 byte a = 1;
2
3 active proctype Main(){
4   if
5   :: a == 1 -> printf("a == 1\n");
```

```
6     :: a == 2 -> printf("a == 2\n");
7     :: else -> skip;
8     fi;
9 }
```

Listing 2.5: If-Anweisung mit einer Else-Verzweigung

Die Else-Verzweigung in Listing 2.5 wird ausgeführt, wenn keine Expression in dem If-Block den Wert *true* annimmt. Die Skip-Anweisung ist funktional äquivalent zum Nop-Opcode der x86-Architektur [36]. Die Verwendung der Skip-Anweisung ist notwendig, da in einem If-Block pro Verzweigung mindestens ein Statement enthalten sein muss.

Im Gegensatz zu Hochsprachen wie Java oder C/C++ ist der If-Block in Promela nicht nur blockierend, sondern auch nicht-deterministisch. Dies verdeutlicht das folgende Beispiel:

```
1 mtype = {yes, no};
2
3 active proctype Main(){
4   mtype user_input;
5   if
6   :: true -> user_input = yes;
7   :: true -> user_input = no;
8   fi;
9 }
```

Listing 2.6: Nicht-Deterministischer If-Block

In Listing 2.6 wird ein If-Block genutzt, um eine Benutzereingabe zu implementieren, bei der ein Anwender in einer Eingabemaske entweder *yes* oder *no* auswählt. Die Eingabe wird mit einem If-Block umgesetzt und in der Variable *user_input* gespeichert. Der If-Block besteht aus zwei Verzweigungen, die beide von der Bedingung *true* abhängig sind. Zur Laufzeit sind also beide Verzweigungen ausführbar. In diesem Fall wählt Spin im Simulationsmodus nicht-derministisch eine der beiden Verzweigungen aus. Im Rahmen einer interaktiven Simulation kann die Auswahl auch vom Benutzer getroffen werden. Da Spin während der Verifikation den gesamten Zustandsraum eines Modells betrachtet, wird im Verifikationsmodus automatisch untersucht, wie sich das Modell verhält, wenn entweder die erste, oder die zweite Verzweigung des If-Blocks ausgeführt wird.

Neben dem If-Statement, das für die bedingte Ausführung von Statements innerhalb eines Modells verwendet wird, gibt es den Do-Block zur Implementierung von Schleifen:

```
1 active proctype Main(){
2   byte a = 1;
3   do
4   :: a != 10 -> a++;
```

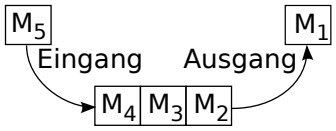

Abbildung 2.3: Funktionsweise einer FIFO-Queue

```
5     :: a == 10 -> break;
6   od;
7 }
```

Listing 2.7: Do-Schleife

In Listing 2.7 wird mittels einer Schleife die Variable a inkrementiert. Die Schleife terminiert, wenn a den Wert 10 annimmt. Die Do-Schleife besteht aus einer Menge von Expressions, denen jeweils mit dem Operator „->“ ein oder mehrere Statements zugeordnet sind. Mit jedem Schleifendurchlauf wird eine Verzweigung ausgewählt, deren Expression den Wert *true* annimmt und anschließend ausgeführt. Der Schleifendurchlauf findet so lange statt, bis die Schleife entweder mit einem Goto-Statement [38] verlassen wird, oder ein Break-Statement die Ausführung der Schleife beendet. Do-Schleifen können, genau wie If-Blöcke, blockieren oder sich nicht-deterministisch verhalten.

Das DSL Verification Framework stellt neben Statements und Expressions auch Produktionsregeln zur Verfügung, die die Kommunikation von nebenläufigen Prozessen in einer domänenspezifischen Sprache ermöglichen. Deshalb zeigt das nächste Beispiel, wie die Kommunikation von Promela-Prozessen über sogenannte *Message-Channel* umgesetzt werden kann. Bei einem Message-Channel handelt es sich um eine spezielle Variable, deren Deklaration über den Datentyp *chan* erfolgt. Message-Channel können als FIFO-Queues (First In, First Out) genutzt werden, um Nachrichten zu versenden oder zu empfangen. Die Funktionsweise einer FIFO-Queue ist in Abbildung 2.3 schematisch dargestellt. Die Abbildung zeigt eine Queue, auf der die Nachrichten M_2, M_3 und M_4 gespeichert sind. Die Nachricht M_1 wird abgerufen während M_5 als neue Nachricht auf der Queue gespeichert wird.

Das folgende Beispiel demonstriert, wie zwei Prozesse in Promela über einen Message Channel miteinander kommunizieren:

```
1 chan queue = [2] of {byte, bool};
2
3 active proctype Send(){
4   queue!1,true;
5 }
6
7 active proctype Receive(){
```

```
8   byte recv;
9   if
10  :: queue?receive,true ->
11          printf("received: %d, true", recv);
12  :: queue?receive,false ->
13          printf("received: %d, false", recv);
14  fi;
15 }
```

Listing 2.8: Kommunikation von Prozessen

In Listing 2.8 wird zunächst ein Message Channel mit dem Bezeichner *queue* deklariert. Jeder Message Channel bekommt bei seiner Deklaration zwei Parameter übergeben:

- Die Größe des Message Channel und somit die maximale Anzahl von Nachrichten, die in ihm gespeichert werden können.
- Eine Menge von Datentypen, die den Inhalt der Nachricht repräsentieren.

In Listing 2.8 wird ein asynchroner Channel deklariert, der zwei Nachrichten speichern kann. Jede Nachricht besteht aus jeweils einem *Byte* und einem *Bool*. Neben dem Message Channel werden auch die beiden Prozesse *Send* und *Receive* deklariert bzw. gestartet. Der Prozess *Send* benutzt den Operator „!“ um die Nachricht {*1,true*} an den Message-Channel *queue* zu senden. Dabei ist zu beachten, dass die Ausführung des Operators „*!*“ in zwei Fällen blockierend sein kann:

- Die Kommunikation über einen Message Channel ist synchron, wenn die Channel-Größe dem Wert 0 entspricht. In diesem Fall blockiert „!“ so lange, bis ein anderer Prozess die gesendete Nachricht abruft.
- Jeder Message Channel kann eine maximale Anzahl von n Nachrichten aufnehmen. Der Operator ! blockiert, wenn auf einem Message Channel bereits n Nachrichten gespeichert sind und noch eine weitere gesendet werden soll. Die Blockierung wird aufgehoben, wenn ein anderer Prozess eine Nachricht von dem entsprechenden Message Channel abruft und so wieder Speicherplatz zur Verfügung steht. Diese Semantik kann per Kommandozeile so angepasst werden, dass volle Channel nicht mehr blockieren und die entsprechende Nachricht stattdessen verworfen wird.

Der Prozess *Receive* empfängt die von *Send* übertragene Nachricht und gibt den Inhalt mittels *printf()* aus. Der Empfangsoperator „?“ blockiert, falls ein Channel leer ist und somit keine Nachricht abgerufen werden kann. Der If-Block in dem Prozess *Receive* in Listing 2.8 wird deshalb erst dann ausgeführt, wenn *Send* die Nachricht übertragen hat. Der Operator „?“ beinhaltet den Bezeichner des Message-Channels, von dem die Nachricht abgerufen werden soll und eine Menge von Nachrichtenfeldern:

- Wenn ein Nachrichtenfeld den Bezeichner einer Variable enthält, wird der Wert der Nachricht in der entsprechenden Variable gespeichert.
- Wenn ein Nachrichtenfeld einen konkreten Wert enthält (beispielsweise *true*, $false$, 1, usw.), kann die Empfangsoperation nur ausgeführt werden, wenn der in der Nachricht gespeicherte Wert äquivalent ist.

Der If-Block in Listing 2.8 enthält zwei Verzweigungen, die jeweils ausgeführt werden, wenn das zweite Feld der Nachricht dem Wert *true* oder $false$ entspricht. In beiden Fällen wird das erste Feld der Nachricht in der Variablen *receive* gespeichert und anschließend ausgegeben.

Damit ein Modell, das mit einer domänenspezifischen Sprache beschrieben ist, verifiziert werden kann, muss der DSL-Anwender entsprechende Anforderungen formulieren. Auch Promela ermöglicht das Spezifizieren von Anforderungen. Dafür können unter anderem Assert-Statements genutzt werden:

```
1 #define COUNTER_MAX 255
2
3 active proctype Main(){
4   byte counter = 1;
5   do
6   :: counter <= COUNTER_MAX ->
7     printf("%d\n", counter);
8     counter++;
9     assert(counter > counter - 1);
10  :: else -> break;
11  od;
12 }
```

Listing 2.9: Assert-Statements

Das Modell in Listing 2.9 deklariert die Konstante *COUNTER_MAX* (Promela erlaubt die Verwendung des C-Präprozessors [74]) und den Prozess *Main*. Der Prozess *Main* beinhaltet eine Schleife, die die Variable *counter* mit jedem Durchlauf inkrementiert und den Zählerwert mittels *printf()* ausgibt. Die Schleife terminiert, wenn die Zählervariable den Wert *COUNTER_MAX* erreicht. Da die Zählervariable einem *Byte* entspricht und *COUNTER_MAX* einen Wert von 255 hat, ist die Gefahr eines Variablenüberlaufs gegeben. Deshalb muss nach jedem Schleifendurchlauf gelten: $counter > counter - 1$. Um dies sicherzustellen, ist ein Assert-Statement eingefügt.

Spin untersucht im Verifikationsmodus den Zustandsraum eines Modells und wertet die Expressions in den Assert-Statements aus. Wenn eine Expression den Wert *false* annimmt, wird von Spin ein Fehler gemeldet und bei Bedarf ein Fehlerpfad angezeigt. In Listing 2.9 führt ein Implementierungsfehler zu einem Überlauf der Variable *counter*,

nachdem sie den Wert 255 angenommen hat. Durch den Überlauf nimmt der Ausdruck $counter > counter - 1$ den Wert $false$ an, Spin meldet einen Fehler und der Anwender muss sein Modell entsprechend korrigieren.

Darüber hinaus können Anforderungen in Promela-Modellen mit linear-temporaler Logik (LTL) ausgedrückt werden. Bei linear-temporaler Logik handelt es sich um ein Konzept von Pnueli et al. [122]. Sie ermöglicht, im Gegensatz zu reinen *boolschen* Ausdrücken das Formalisieren einer zeitlichen Komponononente, so dass Anforderungen wie beispielsweise „x gilt immer und irgendwann gilt y“ umgesetzt werden können:

```
1 ltl counterproperty { always (counter > counter - 1) }
2 #define COUNTER_MAX 255
3
4 active proctype Main(){
5   byte counter = 1;
6   do
7   :: counter <= COUNTER_MAX ->
8     printf("%d\n", counter);
9     counter++;
10  :: else -> break;
11  od;
12 }
```

Listing 2.10: LTL-Formel in Promela

Listing 2.10 beinhaltet die Schleifenimplementierung aus Listung 2.9. Die Anforderung an das Modell wird aber nicht mit einem Assert-Statement, sondern einer LTL-Formel umgesetzt. LTL-Formeln werden in Promela mit dem Schlüsselwort *ltl* eingeleitet und bestehen aus einem Bezeichner und der eigentlichen Formel (in geschweiften Klammern). In der Formel können die Schlüsselwörter *eventually*, *always*, *until*, *implies*, *equivalent*, *weakuntil*, *stronguntil*, *release* und boolsche Operatoren genutzt werden, um das zeitliche Verhalten zu spezifizieren.

2.1.3 Beispiel

Das vorangegangene Unterkapitel stellt alle Elemente der Eingabesprache Promela vor, die für diese Dissertation relevant sind. Dieser Abschnitt demonstriert das Zusammenspiel von Promela und Spin. Dazu wird das Philosophenproblem mit Promela implementiert und anschließend von Spin verifiziert. Der grobe Aufbau des mit Promela beschriebenen Philosophen-Modells gestaltet sich folgendermaßen:

```
1 mtype{frei, links, rechts}
2
3 mtype gabel[5];
```

```
4
5 /* Essender Philosoph */
6 proctype Philosoph(){ /* ... */ }
7
8 /* Initialisiere das Gesamtsystem */
9 init{ /* ... */ }
```

Listing 2.11: Implementierung des Philosophenproblems mit Promela

Das Modell besteht aus den beiden Prozessdeklarationen *init* und *Philosoph*. Die Gabeln sind mit einem Array vom Typ *mtype* implementiert:

- Wenn eine Gabel auf dem Tisch liegt, wird dem entsprechenden Array-Element der Wert *frei* zugewiesen.
- Wenn ein Philosoph eine Gabel mit der linken Hand greift, wird dem entsprechenden Array-Element der Wert *links* zugewiesen.
- Wenn ein Philosoph eine Gabel mit der rechten Hand greift, wird dem entsprechenden Array-Element der Wert *rechts* zugewiesen.

Der Prozess *Init* hat den folgenden Aufbau:

```
1 init{
2   gabel[0] = frei;
3   gabel[1] = frei;
4   gabel[2] = frei;
5   gabel[3] = frei;
6   gabel[4] = frei;
7
8   atomic{
9     run Philosoph();
10    run Philosoph();
11    run Philosoph();
12    run Philosoph();
13    run Philosoph();
14  }
15 }
```

Listing 2.12: Initialisierung des Philosophenproblems

Das Modell im obigen Listing 2.12 initialisiert das Gabel-Array und startet im Anschluss die Philosoph-Prozesse. Die genaue Umsetzung der Philosophen wird mit dem nächsten Listing verdeutlicht:

```
1  proctype Philosoph(){
2    byte lplatz;
3    byte rplatz;
4    if
5    :: _pid == 1 -> lplatz = 4; rplatz = 0;
6    :: else -> lplatz = _pid - 2; rplatz = _pid - 1;
7    fi;
8
9    do
10   :: true ->
11     if
12     :: gabel[lplatz] == frei ->
13       gabel[lplatz] = links;
14       printf("Philosoph %d, linke gabel genommen\n", _pid);
15     fi;
16     if
17     :: gabel[rplatz] == frei ->
18       gabel[rplatz] = rechts;
19       printf("Philosoph %d, rechte gabel genommen\n", _pid);
20     fi;
21     printf("Philosoph %d hat gegessen\n", _pid);
22     gabel[lplatz] = frei;
23     gabel[rplatz] = frei;
24   od;
25 }
```

Listing 2.13: Ein Philosoph als Promela-Prozess

Der Prozess *Philosoph* besteht aus zwei Abschnitten: Anhand der Prozess-ID wird in einem If-Block berechnet, welche Elemente des Gabel-Arrays die linke und rechte Gabel repräsentieren. Das Ergebnis wird in den Index-Variablen *lplatz* und *rplatz* gespeichert. Anschließend wird in einer Endlosschleife überprüft, ob die beiden Gabeln verfügbar sind. Im Fall der Verfügbarkeit werden die Gabeln aufgenommen und im Anschluss wieder zurück auf den Platz gelegt. Da ein Philosoph auf die Verfügbarkeit seines jeweiligen Bestecks wartet, wird das Gabel-Array mit blockierenden If-Blöcken abgefragt. Um das Modell besser im Simulator untersuchen zu können, werden Textnachrichten mit Printf-Anweisungen ausgegeben. Die Textnachrichten geben Auskunft darüber, was der jeweilige Philosoph gerade tut.

Auffällig ist, dass das Modell keine Anforderungen enthält, die mit Assert-Statements oder LTL-Formeln implementiert sind. Dies liegt daran, dass Spin die folgenden Aspekte automatisch verifiziert:

- Zugriffe auf ungültige Speicherbereiche, wie das Lesen und Schreiben über Array-Grenzen hinweg (vergleichbar mit der *ArrayOutOfBoundsException* in Java).

- Jedes Statement in einem Promela-Modell kann durch Voranstellen eines Labels, das mit dem Prefix *end* beginnt, als gültiger Endzustand markiert werden. Wenn ein Prozess in einem blockierendem Statement endet, das nicht als gültiger Endzustand markiert wird, meldet der Model Checker einen Fehler bzw. einen *Deadlock*. Das manuelle Spezifizieren der Anforderung „Es darf keine Deadlocks geben“ ist also nicht notwendig.

Das Modell kann nach der Implementierung und vor der Verifikation mit dem Simulator ausgeführt und getestet werden. Ein Ausschnitt der Ausgabe des Simulators ist in dem folgenden Listing zu sehen:

```
1          Philosoph 2, linke gabel genommen
2                              Philosoph 5, linke gabel genommen
3          Philosoph 2, rechte gabel genommen
4          Philosoph 2 hat gegessen
5              Philosoph 3, linke gabel genommen
6      Philosoph 1, linke gabel genommen
7              Philosoph 3, rechte gabel genommen
```

Listing 2.14: Simulatorausgabe

Es zeigt, dass der Simulator zufällig Prozesse auswählt und sie ausführt. Die Prozesse greifen auf das Gabel-Array zu, nehmen Besteck auf und legen es wieder zurück. Zu beachten ist, dass der Simulator bei jeder Printf-Anweisung Tabulatoren in Abhängigkeit der jeweiligen Prozess-ID einfügt, um deutlich zu machen, welcher Prozess gerade eine Zeichenkette ausgibt.

Zum Starten der Verifikation wird Spin mit dem Befehl *„spin -a philosophen.pml“* aufgerufen und erzeugt so eine Ausgabedatei namens *pan.c*, bei der es sich um den Verifier handelt. Dieser Prozess ist vergleichbar mit dem Ansatz der modellgetriebenen Entwicklung: Spin transformiert das (Promela-)Modell in ausführbaren Quellcode. Ein Compiler, wie beispielsweise der *GCC* (GNU C Compiler) [140], kann den Verifier in ein ausführbares Programm übersetzen. Der kompilierte Verifier durchsucht, wenn er ausgeführt wird, den gesamtem Zustandsraum des Modells. Der Begriff *Zustandsraum* wird in Abbildung 1.7 erläutert. Der Zustandsraum eines Promela-Modells ist wie folgt definiert: Wenn Spin eine Promela-Beschreibung in einen Verifier übersetzt, wird jeder Prozess in einen endlichen Automaten überführt. Der Automat, der den Prozess *Philosoph* repräsentiert, ist schematisch in Abbildung 2.4 dargestellt. In der Abbildung stehen Kreise für Zustände und Pfeile für Transitionen. Jeder Zustand enthält einen, von Spin generierten Bezeichner, wie beispielsweise *S7*, oder *S2*.

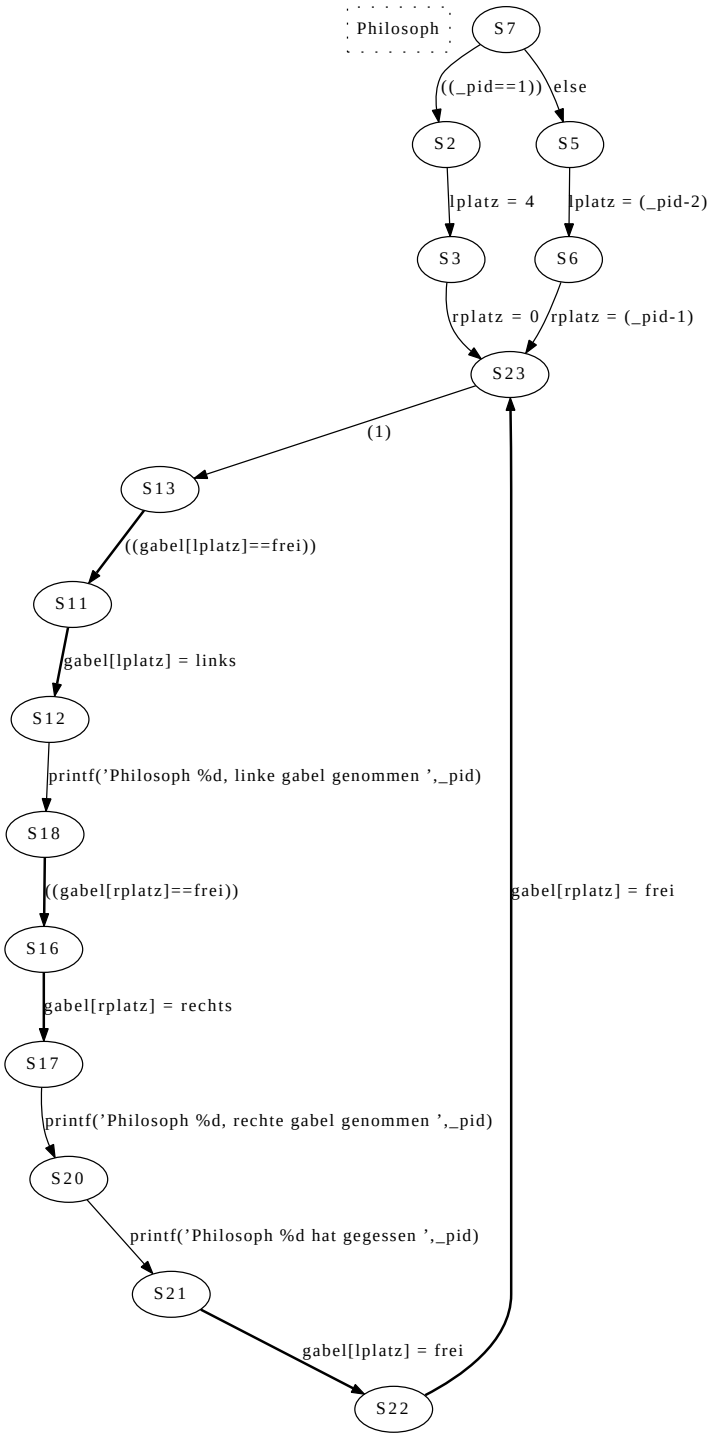

Abbildung 2.4: Philosoph-Prozess als endlicher Automat

Grundsätzlich wird jedes Statement in eine Transition übersetzt, die ausführbar ist, wenn das Statement nicht blockiert. If-Verzweigungen werden in Transitionen übersetzt, die ausgeführt werden, wenn der entsprechende Ausdruck den Wert *true* annimmt. Die Zustände S_2 bis S_7 in Abbildung 2.4 initialisieren die Variablen *lplatz* und *rplatz*. Im Zustand S_{23} beginnt die Ausführung der Endlosschleife zum Greifen und Zurücklegen der Gabeln.

Das gesamte Modell besteht zur Laufzeit nach dem Terminieren des Init-Prozesses aus fünf Automaten, die parallel ausgeführt werden und jeweils einen aktiven Zustand haben. Der Zustand des Gesamtsystems bzw. der Systemzustand setzt sich somit aus den folgenden Komponenten zusammen:

- Die aktiven Zustände der laufenden Prozesse.
- Die Belegung der lokalen Variablen.
- Die Belegung der globalen Variablen.

Der Zustandsraum besteht aus der Summe der Systemzustände. Daraus wird deutlich, dass die Größe von Arrays, Variablen oder die Anzahl von Statements in einem Prozess die Größe des Zustandsraums direkt beeinflussen und sich somit entsprechend auf den Speicherverbrauch bzw. die Laufzeit des Verifiers auswirken.

Nach dem Kompilieren und Starten des Verifiers wird der Zustandsraum des Philosophenproblems untersucht und unter anderem die folgende Meldung ausgegeben:

```
1 pan:1: invalid end state (at depth 114)
2 pan: wrote philosophen.pml.trail
```

Listing 2.15: Verifierausgabe

In Listing 2.15 wird vom Verifier gemeldet, dass das Modell einen ungültigen Endzustand erreicht hat. Die Datei *philosophen.pml.trail* enthält den genauen Fehlerpfad, der von Spin entweder textuell oder graphisch visualisiert werden kann. Das folgende Listing zeigt die textuelle Auswertung des Fehlerpfads:

```
1                                   Philosoph 5, linke gabel genommen
2                              Philosoph 4, linke gabel genommen
3                         Philosoph 3, linke gabel genommen
4                    Philosoph 2, linke gabel genommen
5               Philosoph 1, linke gabel genommen
6 spin: trail ends after 39 steps
```

Listing 2.16: Fehlerpfad

Der Fehlerpfad zeigt, dass es zu einem Deadlock kommt, wenn die Philosophen der Reihe nach ihre linke Gabel aufnehmen.

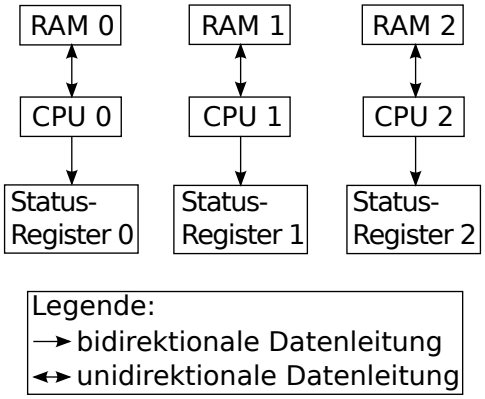

Abbildung 2.5: Beispiel für Symmetrie

2.1.4 Symmetrie

In der Einleitung dieser Arbeit wird das Problem der State Space Explosion beschrieben und wie wichtig es daher ist, den Zustandsraum eines Modells zu minimieren. Eine Möglichkeit, um den Zustandsraum eines Modells zu reduzieren, wird von Norris et al. [109] vorgestellt und basiert auf dem Ausnutzen von symmetrischen Eigenschaften. Symmetrie bedeutet im Bereich der formalen Verifikation, dass die Permutationen der Elemente eines Arrays für den Model Checker nicht relevant sind. Auch das DSL Verification Framework beinhaltet Produktionsregeln, mit denen symmetrische Arrays in eine domänenspezifische Sprache integriert werden können. Deshalb erklärt dieser Abschnitt zunächst anhand eines Beispiels den Begriff Symmetrie. Im Anschluss wird gezeigt, wie symmetrische Arrays mit Spin bzw. Promela modelliert werden können, um den Zustandsraum eines Modells zu verkleinern. Dies ist für das Verständnis von Kapitel 4 wichtig, da der Transformator des DVF symmetrische Arrays als Teil einer DSL nach Promela überführt.

Das folgende Beispiel verdeutlicht den Begriff Symmetrie. Es besteht aus drei baugleichen CPUs, die alle das selbe Programm ausführen und ist in Abbildung 2.5 schematisch dargestellt. Jede CPU ist eine Von-Neumann-Architektur [149] und hat einen eigenen Hauptspeicher, in dem sich sowohl ausführbarer Code, als auch Daten befinden. Des Weiteren reagiert jede CPU über Sensoren auf externe Stimuli und schreibt ihren aktuellen Status in ein Status-Register. Eine mögliche Anforderung an den Model Checker ist es, zu verifizieren, dass niemals zwei oder mehr CPUs gleichzeitig einen Fehler im Status-Register melden. Da nicht relevant ist, ob der Fehler von CPU_0 und CPU_1, CPU_0 und CPU_2, usw. hervorgerufen wird, handelt es sich bei diesem Beispiel um Symmetrie.

Symmetrie kann in Modellen mit oder ohne Nebenläufigkeit auftreten. Modelle mit

Nebenläufigkeit haben den Vorteil, dass die symmetrischen Eigenschaften mit speziellen Algorithmen automatisiert erkannt werden können [42]. Bei Modellen ohne Nebenläufigkeiten muss hingegen vom Anwender manuell markiert werden, wo die Permutation eines Arrays die Semantik eines Modells nicht beeinflusst.

Es gibt Model Checker, die das Modellieren von Symmetrie ermöglichen. Norris et al. stellen beispielsweise den Model Checker Murphi als Referenzimplementierung vor [109][41], dessen Eingabesprache einen speziellen Datentyp namens *Scalarset* zum Deklarieren symmetrischer Arrays beinhaltet und die auf *Misra and Chandy's Unity Formalism* [25] basiert. Die Begriffe „symmetrisches Array" und „Scalarset" werden im weiteren Verlauf dieser Arbeit als Synonyme genutzt.

Auch der Model Checker Spin, der in dieser Arbeit verwendet wird, kann Symmetrie zur Reduktion des Zustandsraums ausnutzen. Hierfür entwickeln Donaldson et al. [43][42] eine Erweiterung namens TopSpin, die das Ausnutzen von symmetrischen Eigenschaften in Promela-Modellen ermöglicht und somit das Risiko der State Space Explosion reduziert. TopSpin erkennt Symmetrie in Modellen mit Nebenläufigkeit automatisch und arbeitet nach dem folgenden Prinzip:

1. Der Promela-Quellcode wird analysiert und überprüft, ob durch den Zugriff von nebenläufigen Prozessen auf globale Arrays Symmetrie vorliegt.
2. Spin wird aufgerufen und generiert den Verifier in Form von C-Quellcode.
3. TopSpin modifiziert den Quellcode des Verifiers, so dass Symmetrie in den entsprechenden Arrays beim Verifizieren berücksichtigt wird.

Durch eine bestimmte Modellierungstechnik kann mit TopSpin auch Symmetrie in Modellen ohne Nebenläufigkeit umgesetzt werden. Dies verdeutlichen die folgenden beiden Beispiele:

```
1  byte a[3];
2
3  proctype main(){
4    //code here
5    //...
6  }
7
8  init{
9    run main();
10 }
```

Listing 2.17: Promela-Modell mit einem Array

Im obigen Listing ist ein Promela-Modell zu sehen, das den Prozess *main* und das Array *a* enthält. Der von *main* ausgeführte Promela-Code entfällt, um das Modell übersichtlicher

zu gestalten. Das Array *a* soll von TopSpin als symmetrisches Array bzw. Scalarset detektiert werden. Dazu wird das Modell wie folgt erweitert:

```
1  byte a[4];
2  pid a_index;
3
4  proctype main(){
5    a_index = 1;
6    a[a_index] = 5;
7    //...
8  }
9
10 proctype dummy ( ) {
11 end:
12   do :: false
13   od
14 }
15
16 init{
17   atomic{
18     run dummy();
19     run dummy();
20     run dummy();
21     run main();
22   }
23 }
```

Listing 2.18: Symmetrisches Array in Promela ohne Nebenläufigkeit

Damit TopSpin ein Scalarset erkennt, darf es nicht mehr direkt, sondern nur noch von Index-Variablen vom Typ *pid* indiziert werden. Das Schlüsselwort *_pid* erlaubt die Referenzierung einer Prozess-ID und entspricht somit dem Wertebereich eines *Bytes*. Daher wird im obigen Listing die Variable *a_index* eingefügt. Mit ihr erfolgt der Zugriff auf Array-Elemente. Für jedes Element im Scalarset *a* wird ein Dummy-Prozess gestartet. Der Dummy-Prozess führt keinen Code aus, besteht lediglich aus einem blockierendem Statement und kann nicht terminieren. Damit Spin beim Verifizieren keinen Deadlock meldet, ist der Endlosschleife das Label *end* vorangestellt. Es markiert die Schleife als validen Endzustand.

Da das symmetrische Array in Listing 2.17 eine Größe von 3 hat, werden drei Dummy-Prozesse instanziiert. Der Init-Prozess bekommt die Prozess-ID 0 zugewiesen. Somit erhalten die drei Dummy-Prozesse die IDs 1, 2 und 3. Die Prozess-IDs der drei Dummy-Prozesse müssen nach Donaldson et al. [42] zum Indizieren des Scalarsets genutzt werden.

Nur durch diese Modellierung detektiert TopSpin nebenläufige Zugriffe auf die Elemente des Arrays und erkennt *a* als Scalarset. Dementsprechend darf *a_index* nur Werte zwischen 1 und 3 annehmen, da dies den Prozess-IDs von *Dummy* entspricht. Die Index-Variable *a_index* darf somit nicht das Element *a[0]* referenzieren. Aus diesem Grund muss die Größe des Scalarsets *a* von 3 auf 4 erweitert werden (vgl. Listing 2.18 Zeile 1). Der Zugriff auf das erste Element des symmetrischen Arrays ist exemplarisch in Listing 2.18 im Prozess *main* zu sehen. Dabei wird *a_index* zunächst eine 1 zugewiesen, da es sich dabei um die Prozess-ID des ersten Dummy-Prozesses handelt.

2.2 Model-Driven Development

Der vorangegangene Abschnitt erläutert das Konzept der formalen Verifikation, den Model Checker Spin und dessen Eingabesprache Promela. Bevor ein Modell vom DVF in eine Model Checker-Eingabesprache wie Promela überführt werden kann, müssen Grammatiken umgesetzt, ein Parser erzeugt und Modelltransformatoren implementiert werden. Gemäß Abbildung 2.1 ist somit die Integration eines Parsergenerators in die prototypische Implementierung des DVF notwendig.

Im Bereich der modellgetriebenen Entwicklung werden dazu nicht die klassischen Ansätze wie *Lex* und *Yacc* [91] genutzt, da sich eigene *MDD-Frameworks* etabliert haben. Der in dieser Arbeit vorgestellte Ansatz, nämlich die Verknüpfung von MDD und formaler Verifikation, ist generisch und kann daher mit einem beliebigen DSL-Framework umgesetzt werden. Um aber in Abschnitt 4 eine Referenzimplementierung vorzustellen, muss eines der vorhandenen Werkzeuge ausgewählt werden. Aufgrund der weiten Verbreitung, kontinuierlichen Weiterentwicklung und des erfolgreichen Einsatzes in anderen Projekten wie beispielsweise [62], wird im Rahmen dieser Arbeit das Xtext-Framework genutzt. Das Xtext-Framework besteht aus den Komponenten Xtext bzw. Xpand und dient zum Umsetzen von domänenspezifischen Sprachen in Softwareprojekten. Beide werden von der Itemis GmbH als *Open Source Projekte* entwickelt, sind Plugins für die Eclipse-IDE und können so sehr einfach in den Softwareentwicklungsprozess integriert werden. Xtext nutzt den ANTLR-Parsergenerator [116] und beinhaltet eine Beschreibungssprache für die Produktionsregeln von Grammatiken. Xpand dient zum Erzeugen von Transformatoren einer mit Xtext entwickelten DSL. Zu beachten ist, dass die Begriffe Transformator und Übersetzungsschablone im Verlauf dieser Arbeit synonym genutzt genutzt werden. Vorteile von Xtext/Xpand sind beispielsweise das automatische Generieren eines Editors mit Syntax-Highlighting und Validierung. Auch *Scoping*, also die Gültigkeit von Sprachkonstrukten in einem bestimmten Kontext, wird nativ unterstützt. Des Weiteren existieren für Xtext Erweiterungen, wie beispielsweise [16], um die Validierung zu vereinfachen.

Für das bessere Verständnis dieser Arbeit wird in den folgenden Unterkapiteln das Xtext-Framework genauer vorgestellt: In Abschnitt 2.2.1 wird gezeigt, wie mit Xtext die

Grammatik einer domänenspezifischen Sprache beschrieben werden kann. Anschließend wird Xpand in Kapitel 2.2.2 vorgestellt. Ein wichtiger Aspekt des Xtext-Frameworks ist die automatische Validierung von Modellen. In Abschnitt 2.2.3 wird deshalb der Validierungsmechanismus genauer beschrieben.

2.2.1 Xtext

Dieser Abschnitt stellt die Xtext-Komponente genauer vor und beschreibt ihre Rolle im DSL Verification Framework. Der Ansatz zur Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation, den auch das DSL Verification Framework umsetzt, ist in Abbildung 1.3 zu sehen. Darin wird deutlich, dass der DSL-Anwender ein Modell mit einer domänenspezifischen Sprache beschreibt und es im Anschluss von einem Transformator in eine Model Checker-Eingabesprache überführt wird. Dabei wird im Rahmen dieser Arbeit für die prototypische Realisierung des DVF der Model Checker Spin verwendet und somit das Modell vom Transformator in Spins Eingabesprache Promela überführt.

Gemäß Abbildung 1.8 sind für die automatische Übersetzung in die Zielsprachen zwei Transformatoren notwendig. Damit der DSL-Entwickler einen Transformator implementieren kann, findet ein entsprechender Parser Verwendung. Dieser wird von einem Parsergenerator auf Basis der Grammatik bzw. der darin enthaltenen Produktionsregeln erzeugt. Bei diesem Schritt unterstützt die Xtext-Komponente den DSL-Entwickler: Xtext ist als Plugin direkt in die Eclipse-IDE integriert und beinhaltet eine Beschreibungssprache für Grammatiken bzw. Produktionsregeln. Des Weiteren kann Xtext aus den entsprechenden Produktionsregeln einen Parser generieren. Ein Vorteil von Xtext ist, dass neben dem Parser auch ein graphischer Editor für die spezifizierte Sprache erzeugt wird, der beispielsweise Syntax-Highlighting beherrscht. Xtext nutzt den ANTLR-Parsergenerator, um aus der Grammatik einen Parser zu erzeugen. Der Parser kann von Xpand genutzt werden, um einen Transformator zu implementieren. Abbildung 2.6 zeigt den Aufbau eines Xtext-Projekts in der Eclipse-IDE. Jedes Xtext-Projekt besteht aus mindestens vier Paketen:

- *diss.exampledsl* enthält die Xtext-Grammatik und eine MWE2-Datei. Die MWE2-Datei ähnelt einem *Makefile* und startet den Parsergenerator.

- *formatting* enthält Java-Quellcode, der die automatische Formatierung im generierten Editor steuert.

- *scoping* enthält Java-Quellcode, mit dem der Anwender die unterschiedlichen Scopes in seinem Modell beeinflussen kann. Ein Beispiel für *scoping* findet sich in der Sprache Promela, bei der lokale Variablen im Gegensatz zu globalen Variablen nur im Kontext des entsprechenden Prozesses referenziert werden können.

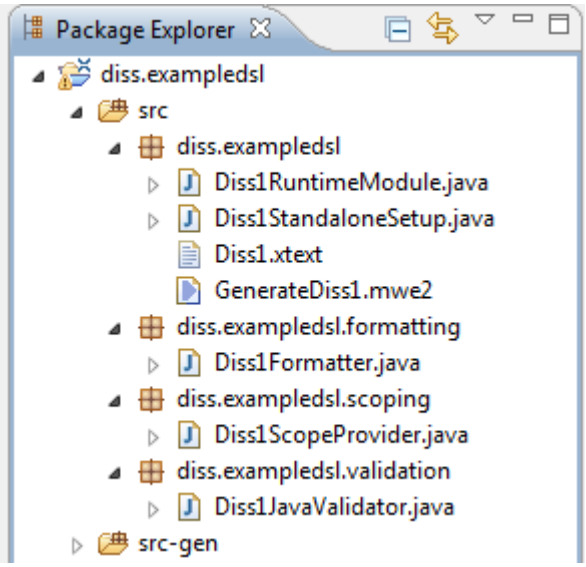

Abbildung 2.6: Aufbau eines Xtext-Projekts

- *validation* enthält Java-Quellcode, der zum Validieren des Modells aufgerufen wird. Die Validierung kann beispielsweise genutzt werden, um in einer DSL sicherzustellen, dass ein Bezeichner immer mit einem Großbuchstaben beginnt.

Jede Grammatik, die mit Xtext umgesetzt wird, besteht aus einer Menge von Produktionsregeln. Das folgende Beispiel zeigt eine einfache Xtext-Grammatik:

```
1 Model:
2   philosophen+=Philosoph*
3 ;
4
5 Philosoph:
6   'Philosoph' name=ID ';'
7 ;
```

Listing 2.19: Auflistung von Philosophen

Die spezifizierte Sprache ist äquivalent zur Philosophen-DSL, die Listing 1.1 mittels Pseudocode beschreibt. Dementsprechend dient sie zum Aufzählen von Philosophen. In Xtext hat jede Produktionsregel den folgenden Aufbau: <Bezeichner>: <Inhalt>. Somit besteht die Grammatik in Abbildung 2.19 aus den Produktionsregeln *Model* und *Philosoph*. Die Startregel wird nicht in der Grammatik, sondern in den Xtext-Konfigurationsdateien spezifiziert. Im weiteren Verlauf dieses Abschnitts wird davon

ausgegangen, dass jede Grammatik eine Startregel namens *Model* enthält. *Model* hat den folgenden Aufbau: <Ableitungsbezeichner>
<Quantitätsoperator1> <Zielregel> <Quantitätsoperator2>. Die genannten Elemente haben die folgende Semantik:

- Der Ableitungsbezeichner *philosophen* wirkt sich auf den generierten Parser aus und wird später genauer erläutert.
- Der Quantitätsoperator1 gibt an, ob die Zielregel einmal oder mehrmals abgeleitet werden darf. Dabei steht $+=$ für eine Mehrfachableitung und $=$ für eine Einfachableitung.
- *Philosoph* ist der Name der Zielregel, in die abgeleitet werden soll.
- Der Quantitätsoperator2 gibt an, ob es 0 bis n oder 1 bis n Ableitungen der Zielregel geben muss. Dabei steht $*$ für 0 bis n und $+$ für 1 bis n. Zu beachten ist, dass der Quantitätsoperator2 entfällt, wenn es sich um eine einfache Ableitung handelt und der Quantitätsoperator1 den Wert $=$ annimmt. Bei einer mehrfachen Ableitung muss der Quantitätsoperator1 auf $+=$ und der Quantitätsoperator2 auf $+$ oder $*$ gesetzt werden.

Die Produktionsregel *Model* kann somit in 0 bis n Philosoph-Objekte abgeleitet werden. Die Regel Philosoph wird abgeleitet nach: Philosoph <Bezeichner> ;. Die Zielregel ID ist eine Regel, die in Xtext bereits vordefiniert ist und eine Zeichenkette repräsentiert. Dabei ist zu beachten, dass ID im Gegensatz zu String (auch eine vordefinierte Regel) einzigartig sein muss und der Parser bei mehrfacher Verwendung der selben Zeichenkette einen Syntaxfehler meldet.

Nach dem Beschreiben der Grammatik kann über den MWE2-Workflow ANTLR aufgerufen und ein Parser generiert werden. Jede Produktionsregel wird von ANTLR in eine Java-Klasse übersetzt. Das bedeutet, dass im Fall von Listing 2.19 die Klassen *Model* und *Philosoph* generiert werden. Die so erzeugten Klassen werden vom Parser eingesetzt, um den *Abstract Syntax Tree* aufzubauen. Wenn beispielsweise Listing 1.2 dem generierten Parser übergeben wird, wird ein AST erzeugt, der in Abbildung 2.7 schematisch dargestellt ist. Das Wurzel-Objekt *Model* enthält eine Liste von Philosoph-Objekten. Dabei ist *EList* ein besonderer Listentyp des *Eclipse Modeling Frameworks* (EMF) [138], der ähnlich aufgebaut ist wie die Vector-Java-Klasse. An dieser Stelle wird die Bedeutung des Ableitungsbezeichners deutlich: Die Menge der Philosoph-Objekte wird in der Liste *philosophen* gespeichert, der Name des jeweiligen Philosophen in der Variable *name*. Aus den Ableitungsbezeichnern werden auch die entsprechenden Get- und Set-Methoden generiert. Ein Transformator, der auf den AST zugreift, kann die Get-Methoden aufrufen, um die Kinder eines Knotens zu erhalten und so über den AST zu iterieren.

Das nächste Beispiel demonstriert das Setzen von Querverweisen innerhalb einer Grammatik:

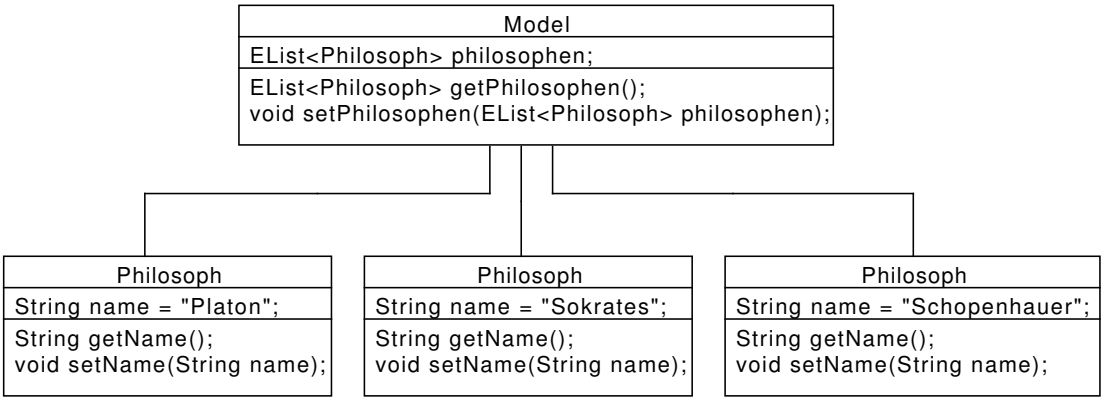

Abbildung 2.7: Mit Xtext erzeugter AST (Objektdiagramm)

```
1  Model:
2          philosophen+=Philosoph*
3          teller+=Teller*
4  ;
5
6  Philosoph:
7          'Philosoph' name=ID ';'
8  ;
9
10 Teller:
11         'Teller von' besitzer=[Philosoph] ';'
12 ;
```

Listing 2.20: Philosophen mit Tellern

Das obige Listing erweitert die Philosophen-DSL. Neben dem Deklarieren von Philosophen ist es auch möglich, Teller zu spezifizieren. Jeder Teller hat genau einen Philosophen als Besitzer. Der Besitzer des Tellers könnte in Xtext mittels *besitzer=ID* implementiert werden. Dies würde jedoch zu dem Problem führen, dass in einem Modell eine beliebige Zeichenkette als Tellerbesitzer eingesetzt werden kann, obwohl nur die Namen bereits deklarierter Philosophen zulässig sind. Um diesem Problem zu begegnen, erlaubt Xtext das Setzen von Querverweisen (vgl. Zeile 11 in Listing 2.20). Der Querverweis bewirkt, dass lediglich eine Zeichenkette gültig ist, die bereits in einem Philosoph-Objekt als Name genutzt wird. Somit erweitern Querverweise eine kontextfreie Grammatik mit kontextsensitiven Anteilen. Das folgende Modell verdeutlicht den Ansatz:

```
1 Philosoph Platon;
2 Philosoph Sokrates;
3
4 Teller von Platon;
5 Teller von Christian;
```

Listing 2.21: Fehlerhaftes Modell

Das obige Listing 2.21 beschreibt ein Modell mit der Philosophen-DSL. Durch die Nutzung der Querverweise meldet der Parser einen Fehler in Zeile 5, da kein Philosph mit dem Bezeichner *Christian* existiert.

Die Grammatik in Listing 2.20 spezifiziert eine domänenspezifische Sprache, in der erst Philosophen und anschließend Teller deklariert werden. Diese strikte Trennung kann mit dem Oder-Operator aufgehoben werden:

```
1 Model:
2         containers+=Container*
3 ;
4
5 Container:
6         philosoph=Philosoph | teller=Teller
7 ;
8
9 Philosoph:
10        'Philosoph' name=ID ';'
11 ;
12
13 Teller:
14        'Teller von' besitzer=[Philosoph] ';'
15 ;
```

Listing 2.22: Oder-Operator in Xtext

Mit der Grammatik wird eine DSL beschrieben, die fast äquivalent zur DSL aus Listing 2.20 ist. Die einzige Ausnahme besteht darin, dass Philosophen und Teller in beliebiger Reihenfolge deklariert werden dürfen. Um die Grammatik zu erweitern, wird die Ableitungsregel *Container* eingefügt, die einen Oder-Operator enthält. Der Oder-Operator bewirkt, dass *Container* entweder nach *Philosoph*, oder nach *Teller* abgeleitet werden darf. Neben dem Oder-Operator können mit Xtext auch DSLs umgesetzt werden, die unterschiedliche Scopes aufweisen. Bei Scopes handelt es sich um Bereiche, in denen bestimmte Elemente referenziert werden können. Dies veranschaulicht die folgende domänenspezifische Sprache:

```
Model:
        tupel+=Tupel*
        instanzen+=Instanz*
        zuweisungen+=Zuweisung*
;

Tupel:
        "tupel" name=ID "{" variablen+=Variable+ "}"
;

Variable: "int" name=ID ";" ;

Instanz: t=[Tupel] name=ID ";";

Zuweisung: tupel=[Tupel] "." instanz=[Instanz]
        "=" wert=INT ";"
;
```

Listing 2.23: Xtext-Grammatik für Tupel-DSL

In der obigen Xtext-Grammatik wird eine domänenspezifische Sprache für zusammengesetzte Datentypen beschrieben. Sie ermöglicht das Deklarieren von sogenannten Tupeln. Jedes Tupel enthält eine Menge von Integer-Variablen. Tupel können nicht nur deklariert, sondern auch instanziiert werden. Auf die Menge der Instanzen folgt eine Menge von Zuweisungen. Das folgende Modell ist mit der Tupel-DSL umgesetzt und demonstriert ihre Anwendung:

```
tupel A{ int a; int b; }
tupel B{ int c; int d; }

A i1;
B i2;

i1.a = 5;
```

Listing 2.24: Mit der Tupel-DSL umgesetztes Modell

In dem Modell werden die beiden Tupel A und B deklariert. Von jedem Tupel gibt es eine Instanz i_1 bzw. i_2. Des Weiteren ist ein Statement Teil des Modells, das dem Integer a im Tupel i_1 den Wert 5 zuweist. Bei der Zuweisungs-Operation ist zu beachten, dass das Element a nur in dem Scope von A gültig ist. Somit ist beispielsweise die Anweisung $i_2.a = 5$ ungültig, da sich im Scope von i_2 nur die Variablen c bzw. d befinden. Um Scopes zu implementieren, sind die Querverweise in der Grammatik 2.23 nicht ausreichend.

Daher muss das Paket *scoping* im Xtext-Projekt mit der folgenden Methode ergänzt werden:

```
1 public IScope scope_Zuweisung_instanz
2   (Zuweisung zuweisung, EReference ref) {
3   Model m = (Model) zuweisung.eContainer();
4   for(Tupel tupel : m.getTupel()){
5     if(tupel.getName().equals(zuweisung.getTupel().getName()))
6       return Scopes.scopeFor(tupel.getVariablen());
7   }
8
9   BasicEList<Variable> empty = new BasicEList<Variable>();
10  return Scopes.scopeFor(empty);
11 }
```

Listing 2.25: Scoping der Tisch-DSL

Die Methode im obigen Listing wird für jedes Element im AST aufgerufen, das vom Typ *Zuweisung* ist. Sie überprüft, ob sich die referenzierte *Variable* im Scope des entsprechenden *Tupels* befindet. Im Fall von Listing 2.25 wird somit verifiziert, dass das Objekt *instanz* ein Teil des Scopes von *tupel* ist.

Der Rückgabewert, die Parameter und auch der Name der Methode sind vorgegeben. Der Name setzt sich aus den folgenden Komponenten zusammen: scope_<Regelname>_<Elementname>. Die Komponenten haben die folgende Semantik:

- Der *Regelname* ist der Name der Produktionsregel, in der sich das Element befindet, dessen Scope ermittelt werden soll.

- Der *Elementname* ist der Name des Elements, dessen Scope ermittelt werden soll.

Der erste Parameter der Methode muss äquivalent zum *Regelnamen* sein. Der AST ist bidirektional und der Elternknoten eines Objekts wird von der Methode *eContainer()* zurückgegeben. Somit wird zunächst in Zeile 3 das Wurzelelement des AST ermittelt. Darauf folgt in Zeile 4 eine Iteration über alle Tupel, die Teil des Modells sind. Für jedes Tupel-Objekt wird überprüft, ob es sich um das referenzierte Element in *zuweisung* handelt. Wenn es sich um das entsprechende Element handelt, wird in Zeile 5 ein sogenanntes IScope-Objekt zurückgegeben, das alle Variablen des Tupels enthält. Im Fall von Listing 2.24 bedeutet dies beispielsweise für die Zuweisung $i_1.a = 5$, dass das IScope-Objekt die Variablen a und b enthält. Xtext untersucht im Anschluss automatisiert, ob die *instanz* in *zuweisung* ein Element des IScopes ist und meldet gegebenenfalls einen Fehler.

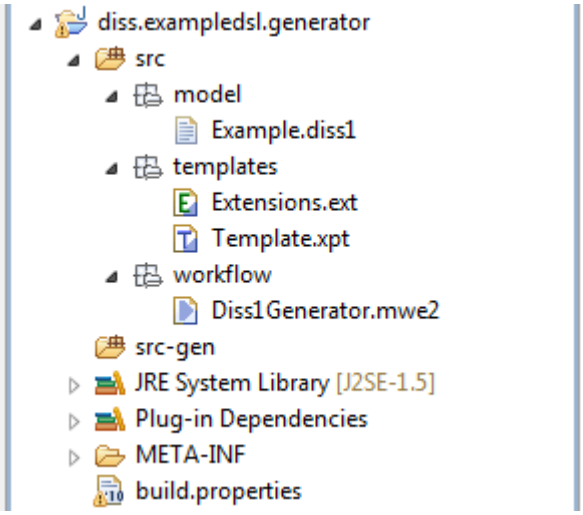

Abbildung 2.8: Aufbau eines Xpand-Projekts

2.2.2 Xpand

Mit Xpand können Modelltransformatoren entwickelt werden, die DSLs in eine Zielsprache überführen. Dies wirkt sich auf die vorliegende Arbeit wie folgt aus:

- Der DSL-Entwickler spezifiziert seine Grammatik mit Xtext und generiert daraus automatisiert einen Parser sowie einen Editor mit Syntaxhighlighting.
- Die Transformatoren des DVF und des DSL-Entwicklers, die eine Übersetzung in eine Hoch-, Model Checker-Eingabesprache und CFIL vornehmen, werden mit Xpand implementiert.

Dementsprechend stellt dieser Abschnitt alle Eigenschaften von Xpand vor, die für das Verständnis dieser Arbeit notwendig sind.

Xpand ist ein Eclipse-Plugin. Wenn in der Eclipse-IDE ein Xtext-Projekt angelegt wird, kann zusätzlich auch ein passendes Xpand-Projekt generiert werden. Der Aufbau eines Xpand-Projekts ist in Abbildung 2.8 gezeigt. Es besteht aus drei Komponenten:

- Im Ordner *model* befinden sich die Modelle, die mit der DSL implementiert sind und in die Zielsprache übersetzt werden sollen.
- Im Ordner *templates* befindet sich der Transformator. Der Transformator wird mit Xpand- (xpt) und Xtend-Dateien (ext) implementiert. Bei Xtend handelt es sich um eine Erweiterung für Xpand, die im Verlauf dieses Abschnitts noch genauer vorgestellt wird.

- Im Ordner *workflow* befindet sich eine MWE2-Datei, die den Transformator startet und gegebenenfalls vom Entwickler modifiziert werden muss. Der MWE2-Workflow ruft eine Main-Methode auf, die sich in einer der Xpand-Dateien im Ordner *templates* befindet und startet so den Transformationsprozess. Da das Xpand-Projekt in Abbildung 2.8 nur eine Xpand-Datei enthält, ist die Main-Methode somit Teil von *Template.xpt*.

Im weiteren Verlauf dieses Abschnitts wird davon ausgegangen, dass die domänenspezifische Sprache aus Listing 2.23 mit Xpand nach Java transformiert werden soll. Der Transformationsprozess wird über die MWE2-Datei gestartet. Das folgende Listing zeigt einen Ausschnitt des MWE2-Workflows:

```
1 component = org.eclipse.xpand2.Generator {
2   expand = "templates::Template::main FOREACH tupel"
3   outlet = {
4     path = "c:\output"
5   }
6 }
```

Listing 2.26: MWE2-Workflow eines Xpand-Projekts

In diesem Teil des MWE2-Workflows wird über die Variable *output* das Verzeichnis festgelegt, in dem der Transformator seine Ausgabedateien speichern soll. Die Variable *expand* gibt Aufschluss darüber, welcher Transformator über welchen Teil des AST iterieren soll. Der Inhalt von *expand* hat die folgende Struktur: <Transformatorverzeichnis> :: <Transformatorname> :: <Main-Methode> FOREACH <AST-Element>.

- Im *Transformatorverzeichnis* befinden sich die Xpand- und Xtend-Dateien.
- Der *Transformatorname* ist der Dateiname des Transformators, der aufgerufen werden soll. Im Fall von Listing 2.26 erfolgt somit der Aufruf der Datei *template.xpt*.
- Jede Xpand-Datei enthält eine Menge von Methoden. Die *Main-Methode* ist die Methode, die zu Beginn des Transformationsprozesses ausgeführt werden soll. Somit wird in Listing 2.26 zu Beginn des Transformationsprozess die Xpand-Methode *main* in der Datei *template.xpt* aufgerufen.
- Das *AST-Element* wird der *Main-Methode* als Parameter übergeben.
- Das Sprachkonstrukt *FOREACH* ist erforderlich, wenn das *AST-Element* eine Liste ist. Es bewirkt, dass die *Main-Methode* für jedes Element, das in der Liste enthalten ist, aufgerufen wird. Somit wird beispielsweise für das Modell in Listing 2.24 die *Main-Methode* genau zweimal aufgerufen und ihr die Elemente *A* bzw. *B* als Parameter übergeben. Wenn die *Main-Methode* nicht auf einer Liste, sondern

nur auf einem speziellen Element operieren soll, kann alternativ das Schlüsselwort *FOR* genutzt werden.

Nach dem Starten der MWE2-Datei aus Listing 2.26 wird der von Xtext erzeugte Parser aufgerufen, die Eingabedatei im Ordner *model* eingelesen und ein AST erzeugt. Anschließend iteriert das Xpand-Framework über den AST und ruft für jedes Tupel-Objekt die Methode *main* auf. Die folgenden Beispiele zeigen verschiedene Implementierungen der Main-Methode, um die unterschiedlichen Sprachkonstrukte von Xpand zu demonstrieren:

- Das Implementieren von Xpand-Methoden und das Schreiben in eine Ausgabedatei.
- Die bedingte Ausführung mit If-Blöcken.
- Das Umsetzen von Schleifen.

Das erste Beispiel zeigt, wie Tupel-Objekte mit Xpand in Java-Klassen überführt und das Ergebnis in eine Ausgabedatei geschrieben werden kann:

```
1 «DEFINE main FOR Tupel-»
2   «FILE name+".java"-»
3   class «name-» { }
4   «ENDFILE-»
5 «ENDDEFINE»
```

Listing 2.27: Xpand-Mainmethode

Xpand benutzt zum Umsetzen der Xpand-Befehle französische Anführungszeichen. Methoden werden mit dem Schlüsselwort *DEFINE* deklariert. Der Aufruf einer Methode erfolgt durch *EXPAND*. Das Schlüsselwort *FOR* markiert den Beginn des Methodenparameters. Die Methode *main* nutzt den Xpand-Befehl *FILE*, um eine Ausgabedatei zu öffnen. Dabei wird alles, was sich zwischen *FILE* und *ENDFILE* befindet, in die Ausgabedatei geschrieben. Der Name der Ausgabedatei setzt sich aus dem Bezeichner des Tupels und dem Suffix *.java* zusammen. In jede Ausgabedatei wird die Zeichenkette „class { <Name des Tupels> }“ geschrieben. Das bedeutet, im Fall des Modells aus Listing 2.24 würde der Transformator zwei Java-Klasse generieren und sie in den Dateien *A.java* bzw. *B.java* speichern.

In Beispiel 2.27 werden zwar alle Tupel in eine Klasse übersetzt, es fehlen jedoch die darin enthaltenen Variablen. Deshalb nutzt das nächste Listing Schleifen, um über alle Variablen eines Tupels zu iterieren und sie so nach Java zu überführen:

```
1 «DEFINE main FOR Tupel-»
2   «FILE name+".java"-»
```

```
3   class «name-» {
4     «FOREACH variablen AS s SEPARATOR "; "»
5       int «s.name-»
6     «ENDFOREACH»
7   }
8   «ENDFILE-»
9 «ENDDEFINE»
```

Listing 2.28: Schleifen mit Xpand

Die FOREACH-Schleife hat den folgenden Aufbau: FOREACH <Liste> AS <aktuelles Element>. Das optionale Schlüsselwort *SEPARATOR* bewirkt, dass die aufgezählten Elemente mit einem „;“ getrennt werden. Somit überführt der Transformator aus Listing 2.28 nicht nur alle Tupel in entsprechende Java-Klassen, sondern ergänzt sie auch um die jeweiligen Integer-Variablen. Neben Schleifen können mit Xpand auch If-Blöcke implementiert werden:

```
1 «DEFINE main FOR Tupel-»
2   «FILE "analysis.txt"-»
3     «IF name.length<=2»
4       «name» hat eine laenge von <= 2
5     «ELSEIF name.length<=4»
6       «name» hat eine laenge von 3 oder 4
7     «ELSE»
8       «name» ist groesser als 4
9     «ENDIF»
10  «ENDFILE-»
11 «ENDDEFINE»
```

Listing 2.29: Bedingte Ausführung mit Xpand

Das obige Listing setzt eine Main-Methode um, die keine direkte Transformation vornimmt, sondern die Länge der Tupelnamen analysiert und das Ergebnis in eine Textdatei schreibt. In Abhängigkeit der Länge eines Bezeichners wird eine von drei Verzweigungen gewählt. Eine Xpand-Verzweigung besteht aus mindestens einem IF-Block. Optional können mehrere ELSEIF-Blöcke, sowie ein ELSE-Block hinzugefügt werden. Der Aufbau der Blöcke ist äquivalent zu anderen Hochsprachen: IF <Bedingung> <Programm-Code>.

Mit den bisherigen Sprachkonstrukten können Methoden, Schleifen und If-Blöcke deklariert werden. Dies ist für einfache Transformatoren ausreichend. Bei einer komplexeren Übersetzung ist es notwendig, bestimmte Werte zwischenzuspeichern und Variablen deklarieren zu können. Auch ein Zugriff auf die Java-Klassenbibliothek zum Erzeugen von Listen, Rechenoperationen, etc. ist wünschenswert. Zu diesem Zweck kann mit Xtend

auf Java zugegriffen werden:

```
1 String getFirstChar(String param) :
2    JAVA MyPackage.getFirstChar(java.lang.String)
3 ;
```

Listing 2.30: Java-Schnittstelle mit Xtend

Im obigen Beispiel wird mit Xtend eine Methode namens *getFirstChar()* deklariert, die eine Zeichenkette zurückgibt und einen String als Parameter erhält. Die deklarierte Methode kann direkt in Xpand aufgerufen werden. In der Methode wird die statische Java-Funktion *getFirstChar()* aufgerufen, die sich in dem Paket *MyPackage* befindet. Bei dem Parameter ist zu beachten, dass der komplette Klassenpfad angegeben werden muss. Der Rückgabewert der Java-Funktion wird automatisiert der Xtend-Funktion übergeben. Als Einschränkung gilt: Es können nur statische Java-Methoden aufgerufen werden können.

2.2.3 Validierung

Die vorangegangenen Unterkapitel zeigen, wie mit dem Xtext-Framework Grammatiken beschrieben und Transformatoren implementiert werden können. Ein weiterer, wichtiger Aspekt von Xtext ist das automatische Erkennen von Fehlern in einer DSL. Dafür stehen verschiedene Validierungsmechanismen zur Verfügung, die dieser Abschnitt genauer vorstellt. Es wird zwischen zwei Problemklassen unterschieden:

- Fehler in einem Modell, die der Parser erkennt.
- Fehler in einem Modell, die nicht gegen die spezifizierte Grammatik verstoßen und die somit nicht vom Parser abgefangen werden können.

Das folgende Beispiel verdeutlicht die erste Problemklasse. Gegeben ist eine DSL, die das Deklarieren von Variablen ermöglicht. Dabei kann eine Variable vom Typ Integer oder vom Typ Byte sein. Des Weiteren kann optional ein Initialisierungswert spezifiziert werden. Die dafür notwendige Xtext-Grammatik besteht aus zwei Produktionsregeln und ist wie folgt definiert:

```
1 VariableDeclare:
2   type=VariableType name=ID
3   ("=" init=INT)? ";"
4 ;
5
6 VariableType:
7   "byte" | "int"
8 ;
```

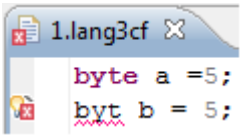

Abbildung 2.9: Syntax-Fehler durch falschen Variablentyp

Listing 2.31: Xtext-Grammatik für Variablen

Zum Umsetzen des Initialisierungswerts in *VariableDeclare* wird die Produktionsregel *INT* genutzt. *INT* ist bereits im Xtext-Framework vordefiniert und steht für einen vorzeichenlosen, ganzzahligen Wert mit einer Datenwortbreite von 32-Bit. *INT* ist in Xtext die einzige vordefinierte Produktionsregel für numerische Werte. Es gibt keine Alternativen wie *BYTE*, *FLOAT*, etc.

Der vorangegangene Abschnitt erläutert, dass Xtext nicht nur einen Parser, sondern auch einen Editor generiert. Abbildung 2.9 zeigt ein Modell der Sprache aus Listing 2.31 und nutzt dafür den erzeugten Editor. Das Modell besteht aus zwei Variablen vom Typ *byte*. Aufgrund eines Tippfehlers handelt es sich jedoch bei *b* um keine gültige Syntax. Der Parser wird in diesem Fall feststellen, dass sich die Regel *VariableType* nicht ableiten lässt und einen Fehler melden, den der Editor automatisch anzeigt.

Es gibt jedoch auch Fehler in Modellen, die der Parser nicht detektiert. Ein DSL-Entwickler kann beispielsweise die Anforderung stellen, dass Bytes nur mit 8-Bit Werten zwischen 0 und 255 initialisiert werden können. Analog dazu sollen Integer einen Initialwert zwischen 0 und 4294967295 aufweisen, was 32-Bit entspricht. Diese Anforderung ist in der Grammatik der Sprache nicht enthalten. Somit wird das folgende Beispiel vom Parser für gültig befunden, obwohl es einen Syntaxfehler enthält:

```
1 byte a = 300;
2 int b = 0;
```

Listing 2.32: Fehlerhafte Variableninitialisierung

Der Variable *a* wird ein Wert zugewiesen, der größer als 255 ist. Da es sich bei 300 jedoch um einen zulässigen Wert im Sinne der Grammatik aus Listing 2.31 handelt, kann der der Parser keinen Fehler erkennen und dem Anwender melden. Trotzdem ist Listing 2.32 kein gültiges Modell.

Um derartige Fehler zu erkennen, wird von Xtext ein Mechanismus namens *Validierung* [50] zur Verfügung gestellt. Das dafür notwendige Paket ist in Abbildung 2.6 zu sehen und enthält die Klasse *Diss1JavaValidator.java*. Der Name der Validierungsklasse setzt sich aus dem DSL-Namen und dem Suffix *JavaValidator* zusammen. Die Klasse kann vom Anwender mit Methoden ergänzt werden, die Fehler wie in Listing 2.32 erkennen und dem DSL-Anwender melden. Das folgende Beispiel zeigt eine Beispielimplementierung

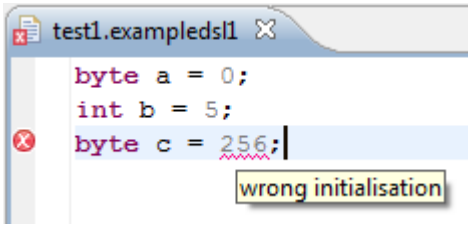

Abbildung 2.10: Syntax-Fehler durch falschen Initialisierungswert

der Klasse *STMJavaValidator.java* für die DSL aus Listing 2.31:

```
1 public class STMJavaValidator extends AbstractSTMJavaValidator {
2   @Check
3   public void checkInitialisation(VariableDeclare var){
4     if(var.getType().equals("byte") &&
5       (var.getInit()>255 || var.getInit()<0))
6       error("wrong initialisation",0);
7   }
8 }
```

Listing 2.33: Warnung bei fehlender Initialisierung einer Variable

Im obigen Listing wird die Validator-Klasse so implementiert, dass sie dem Benutzer einen Fehler meldet, wenn in einem Modell eine Byte-Variable einen falschen Initialisierungswert aufweist. Der Validierungsalgorithmus geht nach dem folgenden Schema vor: Nachdem der Parser den AST erfolgreich und ohne Syntaxfehler aufgebaut hat, wird der AST dem Validierungsframework übergeben. Methoden, denen eine Check-Annotation vorangestellt ist (Zeile 2 in Listing 2.33), sind Validierungsmethoden und müssen vom DSL-Entwickler spezifiziert werden. Jede Validierungsmethode hat im Methodenkopf genau einen Parameter, dessen Typ ein Element des AST ist. Das Validierungsframework iteriert über alle Objekte des *Abstract Syntax Tree* und ruft für jedes Objekt, sofern es den passenden Typ hat, die entsprechende Validierungsmethode auf.

Im Fall von Listing 2.32 enthält der AST nach dem Parsen unter anderem zwei Objekte vom Typ *VariableDeclare*. Das Validierungsframework [61] iteriert über jedes Element des *Abstract Syntax Tree* und ruft zweimal die Methode *checkInitialisation()*, mit den beiden VariableDeclare-Objekten als Parameter, auf. Jedes Objekt vom Typ *VariableDeclare* enthält eine Variable mit dem Bezeichner *init*, die den Initialisierungswert repräsentiert. Die Methode *checkInitialisation()* überprüft, ob das übergebene Objekt vom Typ Byte ist und der Initialisierungswert den korrekten Definitionsbereich hat.

Da die Variable *b* in Listing 2.32 nicht korrekt initialisiert wird, meldet das Validierungsframework einen Fehler. Die Fehlermeldung wird mit der Methode *error()* ausge-

geben, die Teil der Klasse *AbstractSTMJavaValidator* ist. Da das Xtext-Framework ein Plugin für Eclipse generiert, das als Editor für die entsprechende DSL arbeitet, erfolgt die Fehlerausgabe in der Eclipse-IDE. Syntaxfehler werden rot hervorgehoben und zeigen ein *Tooltip* mit detaillierteren Informationen an. Ein Beispiel dafür ist in Abbildung 2.10 zu sehen.

2.3 UML-Statecharts

Die vorherigen Abschnitte haben Spin und Xtext vorgestellt, die die Grundlagen des DSL Verification Frameworks bilden. Dieses Unterkapitel beschreibt UML-Statecharts [111]. Sie spielen im DVF zwar keine direkte Rolle, sind jedoch für das Verständnis der ersten Fallstudie in Abschnitt 5.1 wichtig. Die Einführung in UML-Statecharts gliedert sich wie folgt: Zunächst werden die Grundlagen der UML-Statecharts beschrieben. Im Anschluss erfolgt in Unterkapitel 2.3.1 die Vorstellung eines Beispiels. Da einige semantische Aspekte in der UML-Spezifikation nicht präzise definiert sind, werden diese Lücken in Abschnitt 2.3.2 geschlossen. Zu beachten ist, dass nicht alle Eigenschaften der UML-Statecharts in Kapitel 5.1 benötigt werden. Deshalb beschreibt dieser Abschnitt lediglich die Statechart-Grundlagen, die für das Verständnis der Fallstudie erforderlich sind.

UML-Statecharts sind endliche Automaten. Sie ermöglichen das Modellieren von Verhalten und setzen sich aus Zuständen bzw. Transitionen zusammen. Zustandswechsel erfolgen durch den Erhalt einer bestimmten Eingabe. Endliche Automaten erlauben somit die kompakte Darstellung reaktiver Systeme. Bei den UML-Statecharts handelt es sich um Harel-Statecharts [65] in einem objektorientierten Kontext. Sie haben den Vorteil einer weiten Verbreitung [8] und werden von bestehenden Werkzeugen, wie beispielsweise IBM-Rhapsody [72], unterstützt. Mit ihnen ist es unter anderem möglich, das Verhalten von Klassen und Objekten zu modellieren. Klassen, die einen Automaten enthalten, werden als *aktiv* bezeichnet. Aktiv bedeutet im Kontext der UML die nebenläufige Ausführung des Objekts als Thread bzw. Prozess.

Jeder Automat innerhalb einer Klasse besteht aus einer Menge von Zuständen, Transitionen und reagiert auf externe Stimuli (sogenannte *Events*). Die Ausführung des Automaten beginnt in einem Startzustand und terminiert in einem Endzustand. Daneben gibt es die folgenden beiden Zustandstypen:

- Simple-States: Einfache Zustände ohne spezielle Eigenschaften.

- Composite-States: Zusammengesetzte Zustände, die einen Startzustand, Simple States, Endzustände und auch weitere Composite-States beinhalten können. Mit Composite-States können hierarchische Automaten implementiert werden. Sie können auch nebenläufige Regionen beinhalten. Automaten mit Composite-States können auch in Automaten ohne Composite-States transformiert werden. Das Entfer-

nen der Composite-States und das Umwandeln in einen Automaten ohne Hierarchie wird als *flatten* bezeichnet [150] [152].

Zustände können, wenn sie betreten oder verlassen werden, Programm-Code ausführen. Der Programm-Code wird entweder mit einer Hochsprache formuliert [72], oder mit einer speziell für Statecharts angepassten Action-Language. Zur Kapselung des Programm-Codes stehen in jedem Zustand die folgenden Komponenten zur Verfügung:

- *Entry-Action*: Wird ausgeführt, wenn der Zustand betreten wird und kann nicht unterbrochen werden.
- *Exit-Action*: Wird beim Verlassen des Zustands ausgeführt und kann nicht unterbrochen werden.

Zustandswechsel erfolgen über Transitionen. Jede Transition kann drei verschiedene Elemente enthalten:

- *Guard*: Die Transition kann nur ausgeführt werden, wenn die Bedingung, die im Guard-Element enthalten ist, den Wert *true* annimmt.
- *Trigger*: Die Transition kann nur ausgeführt werden, wenn das im Trigger spezifizierte Event vorliegt.
- *Effect*: Programm-Code wird im Effect-Block deklariert und zusammen mit der Transition ausgeführt.

Automaten reagieren auf Events. Um synchrone und asynchrone Kommunikation umzusetzen, werden Events in verschiedene Kategorien unterteilt:

- *Signal-Events* werden für asynchrone Kommunikation eingesetzt. Ein Automat kann ein Signal-Event versenden und anschließend mit der Ausführung fortfahren. Der Sender kann dem Event Parameter hinzufügen, die an den Empfänger übertragen werden.
- *Call-Events* werden für synchrone Kommunikation eingesetzt. Der Sender wird erst weiter ausgeführt, wenn der Empfänger das gesendete Call-Event verarbeitet hat. Der Sender kann dem Event Parameter hinzufügen, die an den Empfänger übertragen werden.

In jedem Statechart wird ein sogenannter Run-To-Completion-Step als Endlosschleife ausgeführt, der die einkommenden Events bearbeitet, auswählt welche Transition ausgeführt werden kann und den Zustandswechsel bewirkt. Der Run-To-Completion-Step arbeitet nach dem folgenden Algorithmus:

1. Wähle ein Signal aus und entferne es von der Queue oder dem entsprechenden Speichermedium.

2. Evaluiere, welche der Transitionen ausgeführt werden können. Wenn keine Transition ausführbar ist, springe zu 1.

3. Wähle die Transition aus, die ausgeführt werden soll.

4. Führe die Exit-Action des aktiven Zustands aus.

5. Führe den Effect der Transition aus und markiere den Zielzustand der Transition als aktiv.

6. Führe die Entry-Action des neuen, aktiven Zustands aus.

7. Springe zu 1.

Um die beschriebenen Statechart-Elemente besser zu veranschaulichen, wird nun ein Statechart-Beispiel vorgestellt und beschrieben.

2.3.1 Beispiel

In Abbildung 2.11 ist ein einfaches Statechart-Diagramm zu sehen. Die Klasse *Statechart* enthält sowohl die lokalen Variablen a und $result$, als auch einen endlichen Automaten. Zu Beginn seiner Ausführung wechselt der Automat vom Initialzustand in den Zustand $Wait$. Der Zustand $Wait$ besitzt eine ausgehende Transition zum Zustand *Calculate*, die wiederum einen Trigger und einen Effect enthält. Die Transition kann nur ausgeführt werden, wenn zuvor das Signal ON empfangen wurde. Das Signal ON wird zusammen mit einem Byte als Parameter übertragen. Der Programm-Code in dem Effect-Block bewirkt, dass der Byte-Parameter in der Variablen a gespeichert wird.

Beim Betreten des Zustands *Calculate* wird eine Entry-Action ausgeführt, die das Ergebnis von $a\%2$ in der Variable $result$ speichert. Nach dem Ausführen der Entry-Action können zwei verschiedene Transitionen ausgeführt werden, die keinen Trigger, sondern lediglich einen Guard enthalten. Wenn die Variable $result$ den Wert 0 enthält, wird zurück in den Zustand $Wait$ gewechselt. Wenn $result$ hingegen den Wert 1 angenommen hat, wird die Transition in den Zustand *Complex* ausgeführt. Bei *Complex* handelt es sich um einen Composite-State, der zwei nebenläufige Regionen enthält. Jede Region hat eigene Initial- und Endzustände. Nachdem beide Regionen terminiert haben, wird ein Completion-Event erzeugt und der Automat wechselt wieder in den Zustand *Standby*. In diesem Zustand wartet er auf das Signal OFF, um abschliessend den Zustand $Wait$ zu betreten.

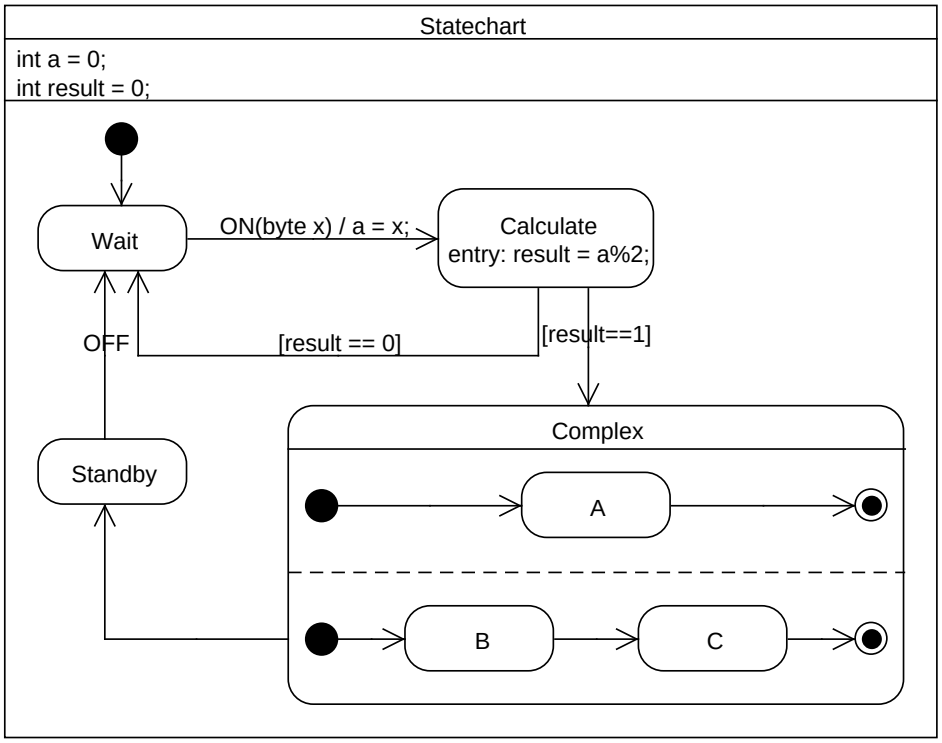

Abbildung 2.11: Einfacher endlicher Automat (Zustandsdiagramm)

2.3.2 Definition der Semantik

Die beiden vorangegangenen Abschnitte stellen die für diese Arbeit relevanten Eigenschaften der UML-Statecharts vor und verdeutlichen sie anhand eines Beispiels. Einige Aspekte sind in der UML-Spezifikation [111] nur informell beschrieben. Dies ist problematisch, wenn Statecharts, wie in dieser Arbeit, mit einer domänenspezifischen Sprache umgesetzt werden sollen, da für die Implementierung der Modelltransformatoren eine vollständige Definition der Semantik erforderlich ist. Deshalb ist es das Ziel dieses Abschnitts, diese Lücken zu schließen und für die in 2.3.1 vorgestellten Elemente eine gültige Semantik zu beschreiben.

Ein wichtiger Aspekt in Statechart-Modellen sind Signale, mit denen Kommunikation beschrieben werden kann. Da endliche Automaten in dieser Arbeit genutzt werden, um das Verhalten von aktiven Klassen zu implementieren, benötigen die instanziierten Objekte ein Speichermedium, auf dem die empfangenen Signale abgelegt und vom Run-To-Completion-Step verarbeitet werden können. Der UML-Standard schreibt nicht vor, wie

die konkrete Implementierung des Mediums, auf dem die Signale vom Automaten empfangen werden, umgesetzt werden muss. Dies ist für eine reine Spezifikation von endlichen Automaten ausreichend. Im Rahmen der vorliegenden Arbeit werden jedoch Transformatoren implementiert, die Statecharts in eine Hoch- bzw. Model Checker-Eingabesprache überführen. Für die Umsetzung der Transformatoren muss die genaue Semantik des Speichermediums definiert werden. Dies wird auch im Rahmen der verwandten Arbeiten in Abschnitt 3.3.1 deutlich, die unterschiedliche Medien zur Speicherung von Signalen vorschlagen. Das folgende Listing zeigt eine mögliche Implementierung mit Promela, die jedoch zu Problemen führt und somit demonstriert, wie wichtig die geeignete Auswahl des Signal-Speichermediums ist:

```
1 #define ON  0
2 #define OFF 1
3
4 bool events[2];
```

Listing 2.34: Events als Arrays

Gegeben sei die Statemachine aus Abbildung 2.11, die auf die Events ON und OFF sensitiv ist. Wenn der Automat ein Event empfängt, wird das entsprechende Element in dem Array *events* auf *true* gesetzt. Beim Empfangen von ON wird also der folgende Programm-Code ausgeführt: *events[ON] = true*. Eine derartige Implementierung führt jedoch zu den folgenden Problemen:

- Es kann immer nur ein *ON*- und ein *OFF-Signal* empfangen werden. Wenn beispielsweise mehrere ON-Signale empfangen werden, verfallen diese automatisch.
- Wenn mehrere Werte in dem Array den Wert *true* annehmen, muss der Run-To-Completion-Step so erweitert werden, dass ein Algorithmus die unterschiedlichen Prioritäten der Events auswertet und das Event mit der höchsten Priorität auswählt. Die unterschiedlichen Prioritäten von Events sind eine Fehlerquelle, da ein DSL-Anwender mit den Prioritäten eventuell nicht vertraut ist und dementsprechend fehlerhafte Modelle erzeugt.
- Wenn ein Modell um eine Programmlogik erweitert wird, die die Prioritäten von Events auswertet, vergrößert diese den Zustandsraum des Modells und kann somit zur State Space Explosion führen.

In dieser Arbeit wird deshalb die Semantik der UML-Statecharts so erweitert, dass Signale auf einer FIFO-Queue gespeichert werden, deren Funktionsweise in Abbildung 2.3 dargestellt ist. Die Implementierung als FIFO-Queue bietet die folgenden Vorteile:

- Im Gegensatz zur Array-Implementierung können mehrere Nachrichten vom selben Typ empfangen werden.

- Die Priorität der Events ergibt sich aus der Reihenfolge, in der sie empfangen werden. Eine Erweiterung des Run-To-Completion-Steps ist somit nicht notwendig.

Damit Automaten Nachrichten abrufen und versenden können, gilt im Rahmen dieser Arbeit: Jede Instanz einer Klasse, deren Verhalten mit einem Statechart beschrieben ist, verfügt über eine endliche FIFO-Queue. Von ihr können Signale abgerufen oder gespeichert werden. Wenn ein Abruf erfolgt, die Queue aber keine Elemente enthält, blockiert der Aufruf, bis ein Signal vorliegt. Analog dazu ist das Verhalten definiert, wenn eine Queue keine weiteren Elemente aufnehmen kann, aber ein Schreibzugriff durchgeführt wird: Der schreibende Prozess blockiert, bis Speicherplatz auf der Queue zur Verfügung steht. Die Nutzung anderer Medien zum Speichern von Events, wie zum Beispiel die Beschreibung in Listing 2.34, verbleibt für zukünftige Arbeiten.

Auch die Semantik der Call- und Completion-Events muss genauer betrachtet werden. Call-Events ermöglichen die Umsetzung von synchroner Kommunikation und werden in UML mit unterschiedlichen Diagrammtypen genutzt. In Sequenz-Diagrammen dienen sie beispielsweise zur Modellierung von Methodenaufrufen. Auch in Zustandsdiagrammen können Call-Events genutzt werden, um Funktionsaufrufe abzubilden. Bei UML-Statecharts sind einige Aspekte bezüglich der Verwendung von Call-Events nicht genau definiert bzw. problematisch. Gegeben seien beispielsweise die zwei Objekte A und B, deren Verhalten mit einem endlichen Automaten umgesetzt ist. A sendet an B ein synchrones Call-Event namens $sync_1$. Dabei ergeben sich die folgenden Probleme, deren Lösung in der UML-Spezifikation nicht behandelt wird:

- Nach dem Empfang und der Bearbeitung von $sync_1$, muss B ein Signal an A senden, damit A die Ausführung fortsetzt. Syntax und Semantik dieses besonderen Rückgabe-Signals sind jedoch nicht Teil der UML-Spezifikation.

- UML-Statecharts können Signal-Events, die im aktuellen Zustand irrelevant sind, verwerfen. Wenn das Call-Event $sync_1$ verworfen wird, kann ein Deadlock in A entstehen. Die UML-Spezifikation gibt keine Auskunft, ob auch Call-Events verworfen werden können und dieses Problem auf Modell-Ebene gelöst werden muss.

Das Problem der Call-Events wird in dieser Arbeit mit der folgenden Semantik gelöst: Call-Events können nur genutzt werden, wenn ein Automat die Funktion einer Klasse aufruft, deren Verhalten mit Methoden beschrieben ist. In diesem Fall setzt der Automat die Ausführung erst dann fort, wenn die aufgerufene Funktion terminiert.

Für die Umsetzung der synchronen Kommunikation zwischen zwei endlichen Automaten sind jedoch keine Call-Events zulässig. Stattdessen müssen Signal-Events genutzt werden. Dies veranschaulicht das Zustandsdiagramm in Abbildung 2.12. Gegeben sind die zwei Objekte a und b, deren Verhalten mit einem endlichen Automaten beschrieben ist. Der Automat a sendet im Zustand $S1$ an b das Signal sig_1, das synchron verarbeitet

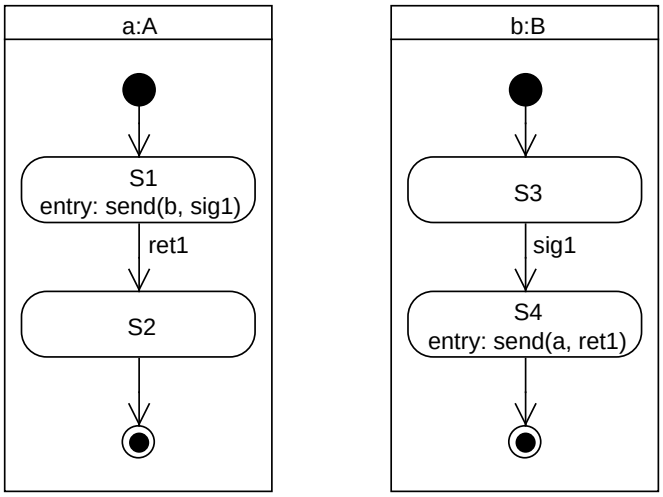

Abbildung 2.12: Synchrone Kommunikation (Zustandsdiagramm)

werden soll. Zu diesem Zweck wird ein zusätzliches Signal namens ret_1 in das Modell eingefügt, das von b an a gesendet wird, nachdem sig_1 empfangen und darauf entsprechend reagiert worden ist.

Neben den Call-Events müssen auch die sogenannten Completion-Transitions genauer betrachtet werden. Eine Completion-Transition ist eine Transition, die keinen Trigger besitzt und ausführbar ist, wenn ein Completion-Event vorliegt. Completion-Events werden nicht explizit von einem Automaten auf einer Queue abgelegt, sondern in bestimmten Situationen automatisch generiert:

- Completion-Events werden in einem Simple-State erzeugt, wenn alle darin enthaltenen Actions terminiert haben.
- In Composite-States wird ein Composite-Event generiert, wenn einer der darin enthaltenen Endzustände erreicht wird.

Zur Veranschaulichung dient das Beispiel in Abbildung 2.13. Gegeben sei die Automaten-Klasse A, die die lokale Variable b enthält und sensitiv auf das Signal sig_1 ist. Die Transition von $S1$ nach $S2$ ist eine Completion-Transition. Wenn der Automat den Zustand S1 erreicht, das Signal sig_1 vorliegt und b den Wert $true$ annimmt, sind auf den ersten Blick sowohl die Transition nach S2, als auch die Transition nach S3 ausführbar. Dieses nicht-deterministische Verhalten wird in der Statechart-Spezifikation [111] wie folgt gelöst: Completion-Events haben eine höhere Priorität als Call- oder Signal-Events.

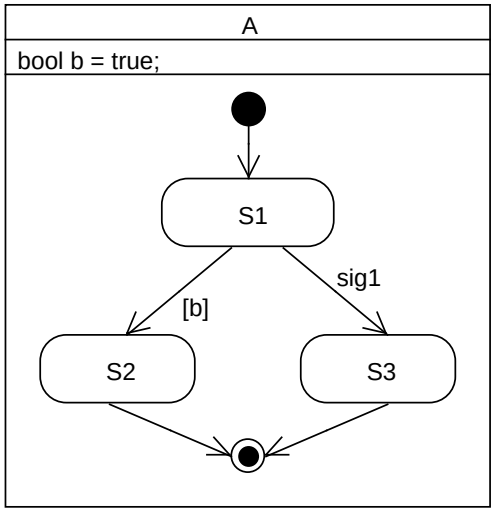

Abbildung 2.13: Automat mit einem Completion-Event (Zustandsdiagramm)

Somit wird in dem Automaten in Abbildung 2.13 die Completion-Transition nach $S2$ ausgeführt.

Im Rahmen dieser Arbeit und der domänenspezifischen Sprache in Abschnitt 5.1 werden Completion-Events bzw. Transitions wie folgt umgesetzt: Completion-Events werden nicht wie Signal-Events erzeugt und auf einer Message-Queue gespeichert. Stattdessen gilt: Wenn ein Zustand eine Menge von ausgehenden Transitionen hat, so wird bei der Ausführung des Automaten zunächst vom Run-To-Completion-Step überprüft, ob darin Completion-Transitions enthalten sind. Wenn der Zustand eine ausführbare Completion-Transition enthält, so hat diese eine höhere Priorität, als Transitionen mit einem Trigger. Da die UML-Statecharts deterministisch sind, muss beim Modellieren sichergestellt werden, dass bei mehreren Completion-Transitions immer nur eine Guard-Expression den Wert *true* annehmen kann. Da auch ein Model Checker verifizieren kann, ob ein endlicher Automat deterministisch ist, wird hier bereits ein möglicher Anwendungsfall für die formale Verifikation deutlich.

Mit UML-Statecharts kann auch Nebenläufigkeit umgesetzt werden. Einer der Hauptgründe für die nebenläufige Ausführung von Teilen eines Programms, ist die Reduzierung der Laufzeit und der dadurch entstehende Geschwindigkeitszuwachs. Nebenläufigkeit kann in UML-Statecharts mit zwei unterschiedlichen Methoden modelliert werden:

- Mehrere Instanzen von aktiven Klassen, deren Verhalten mit einem endlichen Automaten modelliert ist, werden nebenläufig ausgeführt.

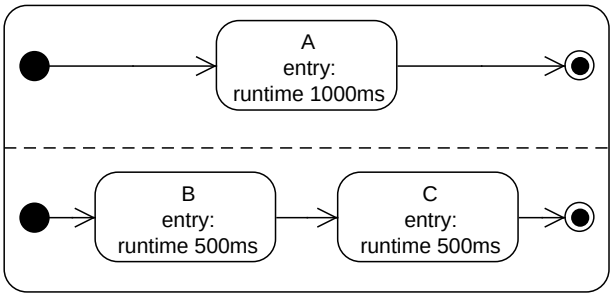

Abbildung 2.14: Nebenläufige Region (Zustandsdiagramm)

- In Composite-States können Regionen enthalten sein, die nebenläufig ausgeführt werden.

In Abbildung 2.14 ist ein Composite-State zu sehen, der zwei nebenläufige Regionen enthält. Darin werden der Zustand A und die Zustände {B, C} parallel ausgeführt. Der Zustand A enthält eine Entry-Action, die aus mehreren nicht näher spezifizierten Statements besteht. Es wird angenommen, dass diese Statements eine konstante Laufzeit von 1000ms aufweisen. Analog dazu enthalten auch die Entry-Actions von B bzw. C eine Menge von Statements mit einer konstanten Laufzeit von 500ms. Bei einer parallelen Ausführung der beiden Regionen wird somit eine Gesamtlaufzeit von 1000ms benötigt.

Die UML-Spezifikation sieht jedoch vor, dass nebenläufige Regionen vom Run-To-Completion-Step synchron ausgeführt werden. Das bedeutet, Zustandswechsel finden erst dann statt, wenn alle Entry-Actions der aktiven Zustände terminiert haben. Abbildung 2.14 wird deshalb folgendermaßen ausgeführt: Zunächst wechseln beide Regionen in die Zustände A und B. Die Entry-Action in B terminiert nach 500ms. B wechselt jedoch erst in den Zustand C, wenn die Entry-Action in A nach 1000ms terminiert hat. Die Gesamtlaufzeit des Systems beträgt somit 1500ms. Da die Vorteile von Parallelität in nebenläufige Regionen nicht optimal genutzt werden und Nebenläufigkeit auch mit dem Instanziieren von mehreren aktiven Automaten-Klassen implementiert werden kann, unterstützt diese Arbeit keine nebenläufige Regionen.

2.4 Google Web Toolkit

In diesem Abschnitt wird das Google Web Toolkit (GWT) [59] genauer vorgestellt. Es ist ein Framework zur Umsetzung von Web-Anwendungen und findet im Rahmen der zweiten Fallstudie in Abschnitt 5.2 Verwendung. Die Beschreibung des GWT gliedert sich wie folgt: Zunächst wird der Begriff *Web-Applikation* anhand eines Beispiels genauer

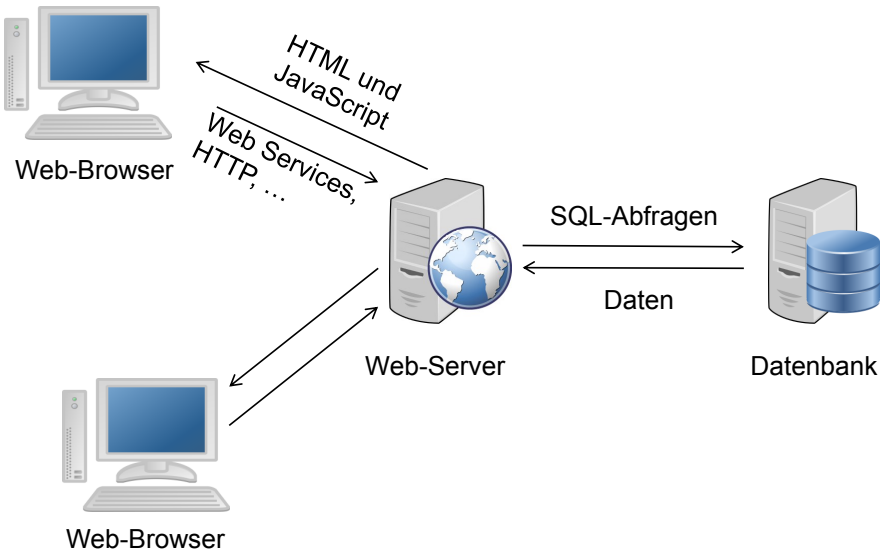

Abbildung 2.15: Aufbau einer Web-Anwendung

beschrieben. Da Web-Anwendungen aus einer Client- und einer Server-Seite bestehen, folgt darauf die Umsetzung des Clients. Zum Abschluss wird beschrieben, wie der Web-Server mit GWT implementiert werden kann.

Web-Applikationen, wie beispielsweise der *Google Calendar* oder *Redmine* [85], werden in einem Web-Browser ausgeführt und typischerweise mit Javascript [49] sowie der *HyperText Markup Language* (HTML) [155] entwickelt. Der exemplarische Aufbau einer Web-Anwendung ist in Abbildung 2.15 zu sehen. Die Client-Seite führt einen Browser aus, empfängt vom Web-Server HTML-Dokumente und stellt sie entsprechend dar. Der Anwender kann über Hyperlinks [155] weitere HTML-Dokumente anfordern. Alternativ ist es auch möglich, Daten via POST, GET, oder Web-Services [157] an den Server zu übertragen.

Der Server kann eine Anbindung an eine Datenbank [113] besitzen. Sie enthält beispielsweise eine Liste gültiger Benutzernamen und Passwörter. Im Fall eines Logins gibt ein Benutzer die entsprechenden Daten in ein HTML-Formular ein. Der Client überträgt sie an den Server. Dieser überprüft mittels einer SQL-Abfrage, ob Benutzername und Kennwort gültig sind und sendet die Antwort zurück an den Client bzw. den Browser.

Aus Abbildung 2.15 wird deutlich, dass Web-Applikationen Nebenläufigkeit beinhalten und bei der Implementierung verschiedene Technologien, wie HTML oder SQL, Verwendung finden. Dementsprechend ist es wichtig, Methoden zu entwickeln, um die Komplexität des Entwicklungsprozesses beherrschbar zu machen und somit auch die Anzahl von Fehlern zu reduzieren. Ein möglicher Ansatz ist die Verwendung eines Frameworks wie

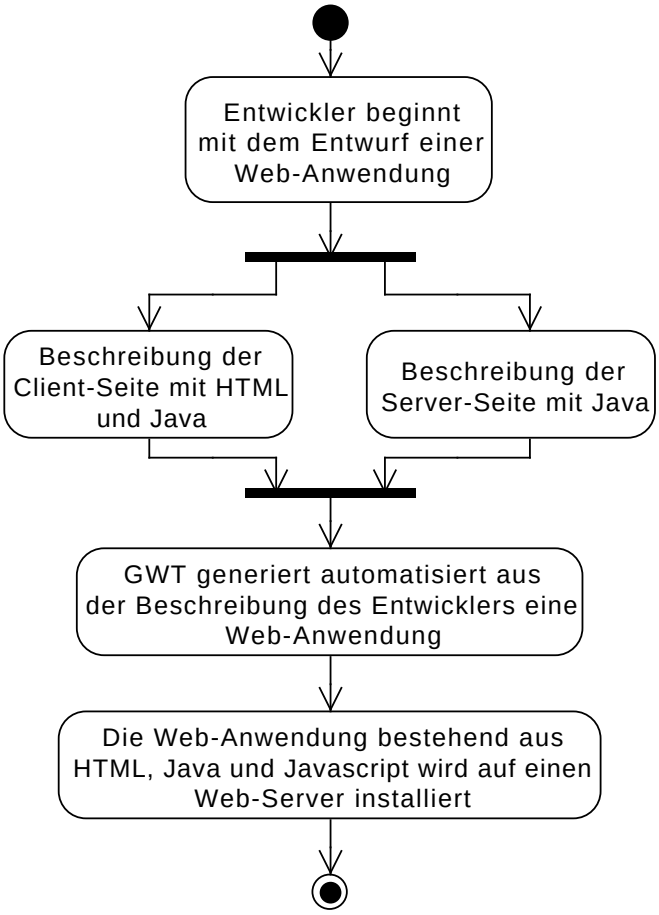

Abbildung 2.16: GWT-Workflow (Aktivitätsdiagramm)

das GWT.

Das Google Web Toolkit ist ein *Open Source Framework* und unterstützt Entwickler bei der Umsetzung von Web-Anwendungen. Der Workflow des GWT ist in Abbildung 2.16 zu sehen und gestaltet sich wie folgt: Ein Entwickler beschreibt das Verhalten der Web-Applikation sowohl server- als auch client-seitig mit Java. Die Umsetzung der strukturellen Aspekte erfolgt mit HTML. Das GWT erzeugt aus dem Java- bzw. HTML-Code automatisiert eine Web-Anwendung. Die generierte Client-Seite besteht aus Javascript bzw. HTML und wird in einem Web-Browser ausgeführt. Die Server-Seite ist Teil des Web-Servers und führt Javacode in Form von sogenannten Servlets [60] aus.

Dieses Vorgehen führt zu verschiedenen Vorteilen: Ein Entwickler muss nicht mehr dafür Sorge tragen, dass seine Web-Applikation mit den verschiedenen Browsern kom-

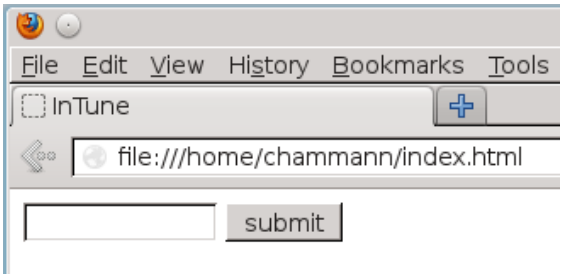

Abbildung 2.17: Beispiel für eine Web-Anwendung

patibel ist. Stattdessen erfolgt die Generierung von Javascript, das auf Firefox [104], Chrome [58], usw. lauffähig ist. Des Weiteren werden große Teile des Clients und des Servers einheitlich mit Java beschrieben und nicht unterschiedliche Sprachen, wie Javascript und PHP (PHP: Hypertext Preprocessor) [141], verwendet. Ein weiterer Vorteil des GWT ist, dass nicht nur eine Transformation von Java nach Javascript erfolgt. GWT ordnet dem generierten Javascript auch die ursprünglichen Java-Statements zu. Dadurch kann der Java-Quellcode in einer Entwicklungsumgebung wie Eclipse [137] auf Fehler hin untersucht werden, während gleichzeitig die Ausführung im Browser erfolgt.

Die Verwendung des GWT wird im weiteren Verlauf des Kapitels anhand eines Beispiels beschrieben. Es ist in Abbildung 2.17 zu sehen. Dabei handelt es sich um eine einfache Web-Anwendung, die aus einer Eingabefläche und einem Schalter besteht. Der Nutzer kann Zeichenketten in die Eingabefläche eintragen. Sobald er den Submit-Schalter betätigt, wird die entsprechende Zeichenkette über einen Web-Service an einen Web-Server übertragen und in einer Datenbank gespeichert.

Für eine Implementierung mit dem GWT muss zunächst ein GWT-Projekt angelegt werden. Das Google Web Toolkit ist als Plugin in die Eclipse IDE integriert. Dementsprechend gibt es einen Assistenten, der nach Übergabe eines Bezeichners ein entsprechendes Eclipse-Projekt generiert. Das Projekt besteht aus:

- Einem HTML-Dokument, das die Struktur der Webapplikation abbildet und vom Anwender beliebig erweitert werden kann.
- Java-Quellcode, der das Verhalten von Client und Server repräsentiert und den der Entwickler seinen Bedürfnissen entsprechend anpasst.

Das folgende Beispiel zeigt den Aufbau eines einfachen HTML-Dokuments, das Teil eines GWT-Projekts ist:

```
1 <html>
2   <head>
3     <title>InTune</title>
```

```
4     <script type="text/javascript" language="javascript"
5     src="intune/intune.nocache.js"></script>
6   </head>
7
8   <body>
9     <h1>InTune</h1>
10    <div id="inTune"></div>
11  </body>
12 </html>
```

Listing 2.35: HTML in einem GWT-Projekt

In dem obigen Beispiel ist eine HTML-Seite zu sehen, die ein Teil des GWT-Projekts *InTune* ist. Der Header des Dokuments wird automatisch generiert. Der Body kann vom Entwickler mit eigenen Komponenten, wie Links, Grafiken, usw. ergänzt werden. In Zeile 10 ist eine Div-Anweisung mit der ID *inTune* zu sehen. Der Entwickler kann die ID in seinem Java-Quellcode referenzieren. Dies bietet den folgenden Vorteil: Der Entwickler spezifiziert in seinem Java-Quellcode das Verhalten von verschiedenen Komponenten, wie beispielsweise einer Tabelle, Texteingabefeldern oder Buttons. Unter Angabe der ID werden diese Komponenten, nachdem das GWT Java und HTML in eine Web-Anwendung überführt hat, an der entsprechenden Stelle in das HTML-Dokument und somit die Webseite eingebettet. Das folgende Beispiel verdeutlicht den Ansatz:

```
1 package com.google.gwt.sample.intune.client;
2 //...
3
4 public class InTune implements EntryPoint {
5   private TextBox textBox = new TextBox();
6   private Button submitButton = new Button("Submit");
7   private HorizontalPanel   panel = new HorizontalPanel();
8
9   public void onModuleLoad() {
10    panel.add(textBox);
11    panel.add(submitButton);
12    RootPanel.get("inTune").add(panel);
13    //...
14  }
15 }
```

Listing 2.36: Beschreibung von Widgets mit GWT und Java

Das obige Listing zeigt Java-Quellcode, der einen Schalter und ein Textfeld in eine GWT-Anwendung einfügt. Dazu wird die Klasse *InTune* generiert, die das Interface *EntryPoint*

umsetzt. *EntryPoint* erfordert das Implementieren der Methode *onModuleLoad()*, die beim Laden der Seite aufgerufen wird. Die Klassen *TextBox*, *Button* und *HorizontalPanel* werden vom GWT zur Verfügung gestellt. *TextBox* und *Button* repräsentieren Elemente, mit denen eine graphische Benutzerschnittstelle umgesetzt werden kann. *Panels* dienen zum Anordnen von Elementen. So werden alle Komponenten in einem *HorizontalPanel* horizontal angeordnet, während in einem *VerticalPanel* die Anordnung vertikal erfolgt. Die Get-Methode im *RootPanel* bekommt den Namen der Div-Umgebung aus Listing 2.35 übergeben und liefert ein weiteres Panel-Objekt zurück. Elemente, die in dieses Panel eingefügt werden, erscheinen an der betreffenden Stelle im HTML-Dokument.

Listing 2.36 zeigt, wie die Komponenten mit Java spezifiziert und im Anschluss in das HTML-Dokument integriert werden können. Das nächste Beispiel demonstriert die Umsetzung des Verhaltens der entsprechenden Komponenten:

```
1 submitButton.addClickHandler(new ClickHandler() {
2   public void onClick(ClickEvent event) {
3     String text = textBox.getText();
4     sendToServer(text);
5   }
6 });
```

Listing 2.37: Verhalten in einem PushButton

Der obige Java-Quellcode ergänzt den *submitButton* aus Listing 2.36 mit einem *EventHandler*, der auf *ClickEvents* reagiert. Das Konzept des EventHandlings ist aus dem Swing-Framework [125] bekannt und wird an dieser Stelle nicht im Detail vorgestellt. Der *submitButton* aus Listing 2.37 liest den Inhalt des Textfeldes aus und übergibt ihn der Methode *sendToServer()*:

```
1 //...
2
3 void sendToServer(String text){
4   AsyncCallback<Boolean> callback =
5   new AsyncCallback<boolean>() {
6     public void onFailure(Throwable caught) {
7       //catch errors
8     }
9
10    public void onSuccess(boolean result) {
11      //...
12    }
13  };
14  server.sendText(text, callback);}
```

Listing 2.38: Aufrufen von Webservices

SendToServer() überträgt die Zeichenkette zum Web-Server, der sie in einer Datenbank speichert. Zu diesem Zweck ruft der Client einen Web-Service auf. Web-Services werden im GWT-Framework auch als *Remote Procedure Calls* (RPCs) bezeichnet und asynchron ausgeführt. Die Deklaration von RPCs auf der Server-Seite entspricht dem Deklarieren von Java-Methoden. Der Client kann die Methoden jedoch nicht direkt aufrufen. Stattdessen wird ein sogenanntes Callback-Objekt erzeugt, das zwei Methoden enthält:

- onFailure: Die Methode wird aufgerufen, wenn es einen Fehler bei der Ausführung des RPCs gibt.
- onSuccess: Die Methode wird aufgerufen, wenn der RPC erfolgreich ausgeführt wurde und terminiert. Der Parameter *result* ist der Rückgabewert des RPCs.

Jeder Web-Service wird vom Client wie eine reguläre Java-Methode aufgerufen. Dabei wird immer als zusätzlicher Parameter das entsprechende Callback-Objekt übergeben. Da RPCs asynchron sind, wird die Ausführung des Clients sofort nach dem RPC-Aufruf fortgesetzt.

3 Stand der Forschung

Das Ziel dieser Disertation ist es, die Methoden der formalen Verifikation und der modellgetriebenen Entwicklung miteinander zu verknüpfen, um so die Qualität von Software zu erhöhen. Dafür muss der Stand der verwandten Arbeiten analysiert werden. Deshalb gliedert sich dieses Kapitel wie folgt: Zunächst werden aus den Eigenschaften des DSL Verification Frameworks Anforderungen abgeleitet. Aus den Anforderungen lassen sich die Themenkomplexe der verwandten Arbeit extrahieren, die im weiteren Verlauf dieses Kapitels untersucht werden. Aus dem Ergebnis dieser Analyse können abschließend die Schwerpunkte dieser Dissertation erkannt und bei der Umsetzung des DSL Verification Frameworks in Abschnitt 4 berücksichtigt werden.

Das genaue Vorgehen zur Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation ist in der Einleitung in Abbildung 1.3 zu sehen: Im Rahmen der modellgetriebenen Entwicklung werden Teile eines Softwareprojekts nicht mit einer Hoch-, sondern einer domänenspezifischen Sprache (DSL) umgesetzt. Des Weiteren spezifiziert der Entwickler Anforderungen, die die beschriebenen Modelle erfüllen müssen. Im Anschluss werden sowohl das Modell, als auch die Anforderungen in eine Model Checker-Eingabesprache transformiert. Dadurch kann ein Model Checker automatisiert verifizieren, dass das Modell alle Anforderungen erfüllt. Falls dies nicht der Fall ist, muss der Entwickler das mit der DSL beschriebene Modell modifizieren und erneut verifizieren. Wenn der Model Checker keine weiteren Fehler findet, wird das Modell in eine Hochsprache überführt und so in lauffähige Software transformiert.

Aus der Beschreibung des DSL Verification Framework leiten sich die thematischen Schwerpunkte der verwandten Arbeiten ab, die in diesem Kapitel untersucht werden müssen. Sie sind in Abbildung 3.1 zu sehen und ihr Gesamtzusammenhang ist wie folgt: Der große Themenkomplex, der in Abbildung 3.1 in weitere Teilbereiche aufgeteilt wird, ist die Verknüpfung der formalen Verifikation mit der modellgetriebenen Entwicklung. Abschnitt 1.2 stellt ein wichtiges Problem im Bereich der formalen Verifikation vor: Damit auch größere Modelle verifiziert werden können, müssen Optimierungen auf Model Checker- und Modellebene angewendet werden. Da bei dem in dieser Arbeit vorgestellten Ansatz möglichst wenig Expertenwissen vorausgesetzt wird, gilt eine wichtige Einschränkung: Die Optimierungen sollen möglichst automatisiert anwendbar sein. Deshalb wird in Abschnitt 3.1 ein wichtiges Themengebiet der verwandten Arbeiten, nämlich die Reduktion des Zustandsraums auf Modellebene ohne die Notwendigkeit von Expertenwissen, diskutiert.

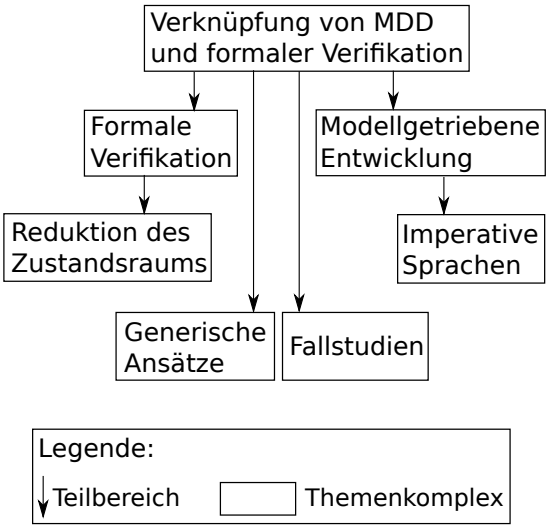

Abbildung 3.1: Gliederung der verwandten Arbeiten

Ein weiteres wichtiges Themengebiet, das für diese Arbeit relevant ist, ist gemäß Abbildung 3.1 MDD [134] bzw. die modellgetriebene Entwicklung. Im Rahmen des MDD-Ansatzes werden Modelle mit domänenspezifischen Sprachen beschrieben und in ausführbare Software transformiert. Domänenspezifische Sprachen enthalten nicht nur die strukturellen Aspekte eines Modells, sondern auch Verhalten. Zum Umsetzen von Verhalten können sogenannte imperative Sprachkonstrukte eingesetzt werden. Hierzu gehören beispielsweise Zuweisungen oder boolsche Ausdrücke. Derartige Elemente sollen auch vom DSL Verification Framework zur Verfügung gestellt werden. Deshalb untersucht Abschnitt 3.2 den Stand der Forschung im Bereich der imperativen Sprachen und evaluiert, welche Elemente in das DSL Verification Framework integriert werden müssen.

Das Ziel des DSL Verification Framework ist die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation. Daher sind besonders verwandte Arbeiten von Interesse, in denen untersucht wird, wie Modelle in eine Hochsprache und eine Model Checker-Eingabesprache transformiert werden können. Eine Übersicht über derartige Arbeiten wird in Abschnitt 3.3.1 gegeben. Neben spezifischen Fallstudien, wie beispielsweise dem Modellieren und Verifizieren von Statecharts oder von Webapplikationen, sind generische Ansätze von besonderem Interesse. Deshalb untersucht Abschnitt 3.3.2 verwandte Arbeiten, die das Beschreiben, Transformieren und Verifizieren von beliebigen Modellen ermöglichen.

Der Stand der Forschung lässt sich somit in zwei Kategorien einteilen: Die Arbeiten über Optimierungen im Bereich der formalen Verifikation und imperative Sprachen (vgl.

Abschnitt 3.1 und 3.2) fließen als nutzbare Grundlagen in die vorliegende Dissertation ein. Bei den Fallstudien und generischen Ansätzen (vgl. Abschnitt 3.3.1 und 3.3.2) handelt es sich hingegen um echte verwandte Arbeiten. Sie verfolgen ein ähnliches Ziel, nämlich das Erstellen und Transformieren von Modellen in unterschiedliche Zielsprachen. Dieses Kapitel evaluiert ihre Stärken bzw. Schwächen und erläutert die Unterschiede zum DSL Verification Framework.

3.1 Optimierungen im Bereich der formalen Verifikation

Da Model Checker aufgrund der State Space Explosion [29] eine sehr hohe Laufzeit und einen sehr hohen Speicherverbrauch aufweisen können, ist es wichtig, beim Verifizieren verschiedene Optimierungen zu berücksichtigen, die die Größe des Zustandsraums reduzieren. In der Einleitung in Abschnitt 1.2.2 werden die Optimierungen in zwei verschiedene Kategorien unterteilt: Model Checker- und Modellebene. Die Optimierungen auf Model Checker-Ebene werden vom jeweiligen Model Checker automatisch durchgeführt oder können vom Anwender aktiviert bzw. deaktiviert werden. Die Optimierungen auf Modellebene müssen hingegen beim Umsetzen des Modells mit der Model Checker-Eingabesprache berücksichtigt werden. Abbildung 3.2 zeigt dafür exemplarisch einen entsprechenden Workflow. Ein Anwender mit Expertenwissen im Bereich Model Checking beschreibt ein Modell mit einer Model Checker-Eingabesprache. Im Anschluss nutzt er Optimierungen auf Modellebene, um den Zustandsraum des Modells zu verkleinern. Zum Abschluss wird das Modell vom Model Checker verifiziert, der automatisiert Optimierungen auf Model Checker-Ebene ausführt.

Auch bei der Verknüpfung von MDD und formaler Verifikation sollen Optimierungen auf Modell- bzw. Model Checker-Ebene genutzt werden, um das Risiko der State Space Explosion zu reduzieren. Der Workflow aus Abbildung 3.2 kann jedoch im Rahmen dieser Arbeit nicht umgesetzt werden, da das DSL Verification Framework (DVF) die Modelle automatisiert in die Model Checker-Eingabesprache überführt und kein Anwender mit Expertenwissen vorhanden ist. Demnach sind zwar trotzdem die Optimierungen auf Model Checker-Ebene möglich, die Optimierungen auf Modellebene müssen jedoch vom DVF durchgeführt werden. Alternativ können die Optimierungsmöglichkeiten dem DSL-Anwender auch in abstrakter Form zur Verfügung stehen, so dass er sie in Modellen nutzen kann, ohne vertiefte Kenntnisse im Bereich der formalen Verifikation zu haben. Aus dem Grund werden in diesem Abschnitt verwandte Arbeiten und ihre Verwendung im DVF wie folgt diskutiert:

- Welche Möglichkeiten der Optimierung auf Modellebene gibt es?
- Wie können sie im DVF eingesetzt werden?
- Wie grenzen sie sich vom DVF ab?

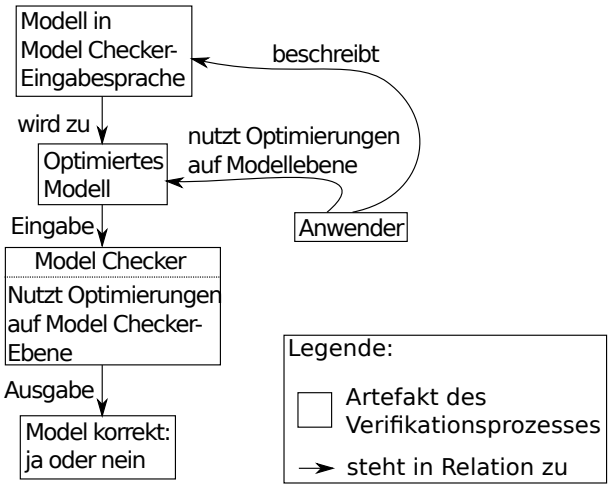

Abbildung 3.2: Umsetzung eines Modells

Theodorus Ruijs behandelt die Reduktion des Zustandsraums von Promela-Modellen in seiner Arbeit *Towards Effective Model Checking* [127]. Dort wird zunächst auf das Aufsetzen, Strukturieren und Warten von *Model Checking Projekten* eingegangen. Hierfür wird das Werkzeug *XSpin/Project* vorgestellt, das zum Verwalten von Verifikationsprojekten dient. Anschließend werden verschiedene Optimierungstechniken auf Modellebene evaluiert, die den Zustandsraum eines Modells verkleinern und so Speicherverbrauch und Laufzeit von Spin reduzieren:

- Das Ersetzen von Boolean-Arrays durch Integer-Variablen.
- Das Generieren von Zufallswerten in einem Promela-Modell.
- Die Reduzierung des Definitionsbereichs von Variablen.

Die Ergebnisse der Arbeit werden im Rahmen von verschiedenen industriellen Fallstudien evaluiert. In seiner Arbeit *Partial Model Checking* [11] reduziert Henrik Anderson den Zustandsraum von Modellen, die mehrere nebenläufige Prozesse enthalten. Dazu werden nicht mehr alle Prozesse gleichzeitig verifiziert, sondern jeder Prozess einzeln durch Abstraktion seiner Kommunikationsschnittstelle. Eine ähnliche Möglichkeit wird in [145] beschrieben: Gegeben sei eine Menge von n Prozessen. Jeder Prozess führt k Instruktionen aus. Der Zustandsraum kann reduziert werden, indem bei jedem Prozess alle Instruktionen abstrahiert und zu einer atomaren Operation zusammengefasst werden. Auf diese Weise sinkt die Größe des Zustandsraums von $(k+1)^n$ auf 2^n. In [146] wird vorgeschlagen, dass eine eingeschränkte Menge von Anforderungen auch verifiziert werden kann, wenn die Ausführungsreihenfolge der Prozesse feststeht und so die Größe

des Zustandsraum auf $n * k + 1$ reduziert. Clark et al. [30] stellen einen Ansatz vor, bei dem der aktuelle Zustand eines Systems durch das Kreuzprodukt der Belegung seiner Variablen definiert wird. Durch die Abstraktion des Wertebereichs der Variablen, wird die Größe des Zustandsraums reduziert und somit auch der State Space Explosion entgegen gewirkt. Die Arbeit wird anhand einer Fallstudie, die eine *Arithmetic Logic Unit* (ALU) mit 64 verschiedenen Registern verifiziert, evaluiert.

Die in diesem Abschnitt vorgestellten verwandten Arbeiten [11] [30] [127] [145] [146] sind ein wichtiger Ansatz, um den Zustandsraum eines Modells zu reduzieren und so auch größere Modelle verifizierbar zu machen. Deshalb sollen sie, sofern dies möglich ist, auch im DVF Verwendung finden. Die Arbeit von Theodorus Ruijs [127] wird daher wie folgt in der vorliegenden Dissertation genutzt:

Das DSL Verification Framework soll die in Modellen enthaltenen Boolean-Arrays automatisiert durch Integer-Variablen ersetzen. Hierfür muss das DVF eine Ableitungsregel enthalten, mit der ein DSL-Entwickler Arrays in seiner DSL modellieren kann. Wenn anschließend ein DSL-Anwender mit der entsprechenden domänenspezifischen Sprache ein Modell umsetzt und nach Promela transformiert, wird vom DVF überprüft, ob es Boolean-Arrays enthält. Sofern dies der Fall ist, werden die Arrays in Abhängigkeit ihrer Größe, in eine oder mehrere Integer-Variablen überführt. Die Zugriffe auf die einzelnen Bits der Variablen, die den Elementen des ursprünglichen Arrays entsprechen, werden mit Makro-Instruktionen umgesetzt.

Auch das von Theodorus Ruijs [127] entwickelte Verfahren zum Generieren von Zufallswerten kann im DVF angewendet werden. Dazu wird das DSL Verification Framework mit Sprachkonstrukten ergänzt, die das Erzeugen von Zufallszahlen ermöglichen. Diese werden vom DVF, sofern ein DSL-Entwickler sie in seiner domänenspezifischen Sprache nutzt, gemäß des Transformationsalgorithmus von Ruijs nach Promela transformiert.

Ein weiterer Ansatz von Theodorus Ruijs [127] zur Verkleinerung des Zustandsraums eines Modells ist die Reduktion des Definitionsbereichs von Variablen. Dieses Verfahren kann jedoch im Rahmen der vorliegenden Arbeit nicht automatisiert genutzt werden, da der komplette Definitionsbereich einer Variable erst nach Absuchen des gesamten Zustandsraums und somit auch erst nach dem Verifikationsprozess ersichtlich wird.

Ein weiterer wichtiger Aspekt der verwandten Arbeiten [11] [30] [145] [146] ist das Ausnutzen von Abstraktion. Diese Technik ist aus zwei Gründen für das DVF interessant:

- Durch Abstraktion kann der Zustandsraum eines Modells reduziert werden.
- Im Rahmen der modellgetriebenen Entwicklung werden oft nur Teile eines Softwaresystems mit einer domänenspezifischen Sprache beschrieben. Somit müssen die Elemente eines Softwareprojekts, die mit einer Hochsprache umgesetzt werden sollen, in der DSL in abstrakter Form abgebildet werden.

Deshalb soll auch das DVF die eigenschaftserhaltende Abstraktion in domänenspezifischen Sprachen ermöglichen. Zu diesem Zweck wird es entsprechende Sprachkonstrukte

enthalten, die ein DSL-Entwickler in seine DSL integrieren kann. Der DSL-Anwender kann im Anschluss entscheiden, welche Bereiche seines Modells nur in abstrakter Form umgesetzt werden sollen. Das entsprechende Modell wird vom DVF so nach Promela überführt, dass Spin die eigenschaftserhaltende Abstraktion berücksichtigt und so den Zustandsraum minimiert.

3.2 Imperative Sprachen

Der vorangegangene Abschnitt 3.1 zeigt Optimierungen, die beim Transformieren einer DSL in eine Model Checker-Eingabesprache berücksichtigt werden müssen, um den Zustandsraum zu verkleinern und so der State Space Explosion entgegen zu wirken. Das Ziel des DVF ist es jedoch nicht nur die Transformation einer DSL zu ermöglichen, sondern auch das Beschreiben der domänenspezifischen Sprache zu vereinfachen. Dazu enthält es eine Menge von vorgefertigten Produktionsregeln. Dementsprechend ist bei der Umsetzung des DVF darauf zu achten, möglichst die Produktionsregeln ins DSL Verification Framework zu integrieren, die in vielen DSL-Projekten benötigt werden (vgl. Abschnitt 3.3.1 für Beispiel-DSLs). Zu diesem Zweck ist es hilfreich, die Sprachkonstrukte von DSLs in die folgenden beiden Kategorien einzuteilen und genauer zu untersuchen:

- *Strukturelle Sprachkonstrukte* werden von Klassendiagrammen umgesetzt, um beispielsweise die Anzahl und Relation der Klassen untereinander definieren.

- *Sprachkonstrukte für Verhalten* sind beispielsweise in Zustandsdiagrammen zu finden, um das konkrete Verhalten einer Klasse zu definieren. Hierzu gehören beispielsweise Zuweisungen oder logische Ausdrücke.

Struktur und Verhalten können mit sogenannten imperativen Sprachen beschrieben werden. Deshalb untersucht dieser Abschnitt den Stand der Forschung im Bereich der imperativen Sprachen. Das Ziel ist es, geeignete Sprachkonstrukte zu finden und in das DSL Verification Framework zu integrieren. Dadurch kann der DSL-Entwickler eine Menge von Sprachkonstrukten auswählen, sie zu seiner DSL hinzufügen und so beispielsweise das Modellieren von Zuweisungen oder mathematischen Ausdrücken ermöglichen.

Imperative Sprachen sind in vielen wissenschaftlichen Arbeiten von Bedeutung, die sich mit der Umsetzung von modellgetriebener Entwicklung oder formaler Verifikation befassen: So werden beispielsweise in [87] oder [102] endliche Automaten mit einer DSL modelliert und in eine Model Checker-Eingabesprache transformiert. Die entsprechenden Automaten können Entry- oder Exit-Actions enthalten, die beim Betreten oder Verlassen eines Zustands ausgeführt werden. Beide Arbeiten beschreiben zwar, dass Entry- oder Exit-Actions Zuweisungen, arithmetische Ausdrücke, usw. ausführen, eine genaue Definition von Syntax und Semantik bleibt aber aus. Die dafür notwendigen Statements

werden nicht genauer spezifiziert oder orientieren sich an der Eingabesprache des verwendeten Model Checkers.

Ähnlich verhält es sich bei Castro et al. [23], die die automatische Generierung von Informationssystemen durch MDD untersuchen. Dafür werden zwei Metamodelle beschrieben, mit denen sich Geschäftsprozesse und Informationssysteme umsetzen lassen. Auch hier fehlt die konkrete Umsetzung einer imperativen Sprache.

In [143] wird modellgetriebene Entwicklung eingesetzt, um Web-Anwendungen zu entwickeln. Dabei gibt es verschiedene Modelltypen, um die Datenhaltung, die Oberfläche einer Webseite und die Kommunikationen mit externen Quellen zu beschreiben. Singh et al. [131] nutzen ein System zur Patientenerfassung im medizinischen Bereich als Fallstudie, um die Konzepte von MDD zu verdeutlichen. In beiden Ansätzen gibt es keine konkrete Definition der dafür notwendigen imperativen DSL.

Die Vernachlässigung einer imperativen Sprache ist auch in XMI [110] erkennbar, das beispielsweise von [87] oder [102] als Eingabe für die Modelltransformationen genutzt wird: Die XMI-Spezifikation der UML-Statecharts beschreibt den Inhalt von Entry- oder Exit-Actions lediglich als Zeichenketten. Werkzeuge wie IBM-Rhapsody [73], die XMI nutzen und Modelle in eine Hochsprache überführen, ermöglichen es deshalb dem Anwender die Statements in Entry- oder Exit-Actions mit der Sprache umzusetzen, in die die Modelle anschließend überführt werden.

Werkzeuge wie Hugo/RT [98] [128] stellen eine Ausnahme dar: Hugo hat eine eigene, proprietäre Eingabesprache, die, ähnlich wie Java, das Deklarieren von Klassen und Variablen ermöglicht. Das Verhalten einer Klasse muss mit einem endlichen Automaten umgesetzt werden. In Entry-Actions, Exit-Actions, Guard- oder Effect-Blöcken können Statements bzw. Expressions spezifiziert werden, die arithmetische Ausdrücke oder Zuweisungen ausführen. Die Klassen sind aktiv, wodurch die entsprechenden Instanzen nebenläufig ausgeführt werden.

Die imperative Sprache von Hugo/RT führt zu dem folgenden Problem: Die Nutzung von aktiven Klassen und die nebenläufige Ausführung der entsprechenden Objekte ist nicht immer wünschenswert [5]. Gegeben sei das Beispiel aus Abbildung 3.3. Das Modell besteht aus zwei Elementen, nämlich dem endlichen Automaten FSM und einem Stack. Der endliche Automat instanziiert ein Objekt vom Typ Stack und speichert darauf einen Byte-Wert. Obwohl das Modell trivial ist, kann es nicht mit Hugo/RT umgesetzt werden. Der Grund ist, dass Hugo/RT nur Klassen ermöglicht, deren Verhalten mit einem Automaten beschrieben sind. Um Abbildung 3.3 mit Hugo/RT umzusetzen, müsste somit die Klasse Stack als nebenläufiger, endlicher Automat beschrieben werden, der auf Push- und Pop-Signale reagiert. Eine derartige Implementierung hat den Nachteil, dass durch die Nebenläufigkeit der Zustandsraum des Modells vergrößert wird und somit auch das Risiko der *State Space Explosion* wächst.

Imperative Sprachkonstrukte sind auch in den Hochsprachen B [84], C [78] oder Java [60] enthalten, um Software zu beschreiben. Dazu gehören beispielsweise Klassen,

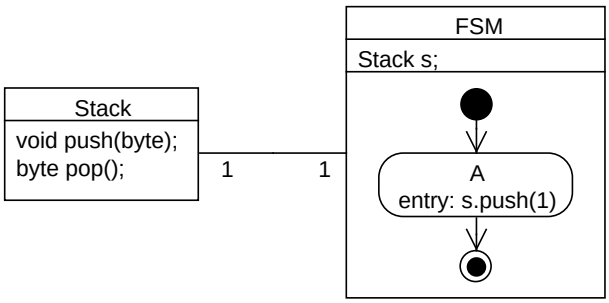

Abbildung 3.3: Stack als abstrakter Datentyp

Methoden, Zuweisungen, Variablendeklarationen. Sie kommen aber auch in domänenspezifischen Modellierungssprachen wie [135] oder [130] zum Einsatz.

Aus dem Stand der Forschung lassen sich für das DSL Verification Framework die folgenden Erkenntnisse ableiten: Imperative Sprachkonstrukte sind in vielen verwandten Arbeiten wichtig, werden jedoch nur oberflächlich betrachtet, da die Schwerpunkte auf anderen Bereichen liegen, wie der graphischen Oberfläche oder der Kommunikation von nebenläufigen Prozessen. Des Weiteren wird am Beispiel von Hugo/RT [98] [128] deutlich, dass sich eine imperative Sprache mit einer reduzierten Menge von Sprachkonstrukten negativ auf die Größe des Zustandsraums auswirken kann.

Das DVF soll zum Beschreiben von Verhalten möglichst viele verschiedene Elemente in Form von Ableitungsregeln enthalten, die ein DSL-Entwickler benötigt. Dazu gehören die aus Hochsprachen C [78] oder Java [60] bekannten Elemente:

- Deklaration von Variablen.
- Deklaration und Aufruf von Methoden.
- Deklaration und Instanziierung von Klassen.
- Zuweisungen.
- Arithmetische Ausdrücke.

Um eine gewisse Sprachmächtigkeit zu garantieren, sollen die Ausdrücke des DVF die gängigen Operatoren für Addition, Multiplikation, Subtraktion, Division, Bitmanipulation und logische Operationen enthalten. Des Weiteren sollen Schleifen und bedingte Ausführung von Ausdrücken möglich sein, um den Kontrollfluss zu steuern.

Daneben müssen auch Elemente verfügbar sein, mit denen sich Anforderungen an das Modell, wie beispielsweise *Assertions*, formulieren lassen. Außerdem sollte das DVF

gemäß der Analyse der verwandten Arbeiten in Abschnitt 3.1 spezielle Sprachkonstrukte für eigenschaftserhaltende Abstraktion und Array-Symmetrie beinhalten. Bei der Implementierung des DVF ist darauf zu achten, dass diese Transformatoren und Produktionsregeln bei Bedarf mit neuen Elementen erweitert werden können. Eine genaue Definition der imperativen Sprachkonstrukte des DSL Verification Frameworks folgt in Abschnitt 4.

3.3 Verknüpfung von MDD und formaler Verifikation

Damit Modelle von einem Model Checker verifiziert werden können, müssen sie in eine Model Checker-Eingabesprache überführt werden. Abschnitt 3.1 zeigt verschiedene Techniken, die den Zustandsraum eines Modells verkleinern, wenn sie bei der Transformation in die Model Checker-Eingabesprache berücksichtigt werden. In dieser Arbeit sollen Modelle jedoch nicht nur in eine Model Checker-Eingabesprache, sondern auch in eine Hochsprache überführt werden. Dazu gibt es eine Reihe von verwandten Arbeiten, die sich mit einem der beiden Themenkomplexe befassen: So wird von Iyenghar et al. [75] untersucht, wie UML-Modelle in Software für eingebettete Systeme überführt werden können. Der Schwerpunkt liegt dabei auf der Generierung und Integration von Testfällen für das jeweilige Zielsystem. Auf der anderen Seite wird von Pulvermüller et al. [124] untersucht, wie Modelle graphisch umgesetzt und von einem Model Checker verifiziert werden können.

In diesem Abschnitt soll jedoch explizit der Stand der Forschung betrachtet werden, der Modelle sowohl verifiziert, als auch in lauffähige Software überführt. Von besonderem Interesse sind dabei die folgenden Aspekte:

- Wie werden die entsprechenden Modelle umgesetzt?
- Wie ist der Aufbau der Transformatoren?
- In welche Zielsprachen werden die Modelle überführt?

Zu diesem Zweck werden zunächst in Kapitel 3.3.1 verwandte Arbeiten betrachtet, die Modelle und Transformatoren von spezifischen Anwendungsfällen implementieren. Da in dieser Dissertation im Rahmen von zwei Industriefallstudien domänenspezifische Sprachen für endliche Automaten und Webapplikationen implementiert werden, liegt der Schwerpunkt der Untersuchung auf Statecharts und Web-Anwendungen.

Das in dieser Dissertation vorgestellte DSL Verification Framework soll generisch einsetzbar sein. Deshalb sind auch verwandte Arbeiten von Interesse, die unabhängig von einer speziellen Fallstudie operieren. Aus diesem Grund untersucht Abschnitt 3.3.2 die generischen Ansätze zur Verknüpfung von MDD und formaler Verifikation.

3.3.1 Fallstudien

In diesem Abschnitt werden Fallstudien vorgestellt bzw. diskutiert, die modellgetriebene Entwicklung und formale Verifikation miteinander verknüpfen. Das bedeutet, dass jede Arbeit eine bestimmte Domäne mit Modellen umsetzt. Die Modelle werden anschließend verifiziert und, sofern alle Anforderungen erfüllt sind, in eine Hochsprache transformiert. Nach der Beschreibung der verwandten Arbeiten erfolgt eine Diskussion der Anforderungen, die sich daraus für das DSL Verification Framework ableiten.

Matougui et al. [47] stellen eine domänenspezifische Sprache für die *COSMOS Middleware* (COntext entitieS coMpositiOn and Sharing) vor [33]. COSMOS wird mit einer Sprache namens *FRACTAL ADL* programmiert, deren Nutzung komplex und aufwendig sein kann. Daher entwickeln Matougui et al. in ihrer Arbeit die sogenannte *COSMOS DSL*. Sie vereinfacht die Nutzung von COSMOS und wird automatisiert in die FRACTAL ADL transformiert. Des Weiteren erfolgt eine Übersetzung in Petrinetze [119], damit die Modelle der COSMOS DSL von einem Model Checker verifiziert werden können.

James Peterson [53] stellt mehrere graphische DSLs vor, um damit eingebettete Systeme zu modellieren. Dazu gehört beispielsweise die *Object DSL*, zur Umsetzung von Objekten, oder die *Actors DSL* zur Spezifikation von Anforderungen. Die domänenspezifischen Sprachen werden in die Eingabesprache des *PHAVer Model Checkers* [53] transformiert und im Anschluss verifiziert. Die Arbeit enthält eine Fallstudie zur Steuerung eines Bahnübergangs.

Cao et al. [22] entwickeln eine domänenspezifische Sprache namens *DSL-CBI*. Die DSL ist graphisch und dient zum Modellieren von *Computer Based Interlocking Systems* (CBI). Dabei handelt es sich um Systeme zur automatischen Steuerung von Bahnhöfen. Ein Transformator überführt die DSL-Modelle und die an sie gestellten Anforderungen in die Eingabesprache des SMV [28] Model Checkers, der sie verifiziert.

Da in Kapitel 5.1 im Rahmen der ersten Fallstudie UML-Statecharts mit einer DSL umgesetzt werden, erfolgt im weiteren Verlauf dieses Abschnitts eine Untersuchung der verwandten Arbeiten, die sich mit endlichen Automaten befassen. Damit UML-Statecharts erfolgreich verifiziert werden können, muss zunächst die Semantik vervollständigt bzw. formalisiert werden. Dies ist notwendig, da die UML die Semantik von Statecharts nur informell spezifiziert. Deshalb stellen beispielsweise Latella et al. in [88] eine Formalisierung für endliche Automaten vor, die auch eine vollständige Semantik für Statecharts beinhaltet. Es wird jedoch nur eine Teilmenge der Statecharts-Features betrachtet. Dazu gehören hierarchische Zustände, Parallelität in hierarchischen Zuständen sowie *Call-* und *Signal Events*. Objektorientierung, Variablen, Expressions, Statements und Signalparameter, die Teil dieser Arbeit und der Semantik aus Abschnitt 2.3 sind, werden hingegen nicht berücksichtigt. Ein besonderer Schwerpunkt von Latella et al. liegt auf:

- Der Beschreibung des *Dispatchers*, der auswählt, welches *Event* von einem Au-

tomaten bearbeitet werden soll, falls mehrere Signale gleichzeitig vorliegen (vgl. Abschnitt 2.3).

- Der Priorisierung von Transitionen, falls mehrere Transitionen gleichzeitig ausführbar sind.

Darauf aufbauend wird in [87] eine Transformation von UML-Statecharts nach Promela vorgestellt. Jeder Automat besteht aus einer Message-Queue, einem *Array* von Zuständen und einem aktiven Prozess. Das Zustands-Array besteht aus Bits und dient zum Kodieren des aktuellen, aktiven Zustands. Es können mehrere Bits gleichzeitig gesetzt sein, da im Falle von nebenläufigen Regionen auch mehrere Zustände gleichzeitig aktiv sein können. Die Message-Queue wird in Abhängigkeit von der Implementierung des *Dispatchers* entweder als Promela-Channel oder auch als Bit-Array umgesetzt.

Auch Dong et al. [44] definieren eine eigene Semantik für Statecharts. Des Weiteren stellen sie ein Konzept für endliche Automaten namens *Extended Hierarchical Automaton* (EHA) vor. Um UML-Statecharts verifizieren zu können, werden sie in EHAs transformiert. Im Anschluss verifiziert ein Model Checker, dass die EHAs alle an sie gestellten Anforderungen, in Form von LTL-Formeln [122], erfüllen. Andrzej Wasowski synthetisiert endliche Automaten für die Programmierung von Mikrocontrollern [151]. Der vorgestellte Ansatz unterstützt unter anderem Do-Actions, einen Ringpuffer [114] zum Speichern von Signalen und transformiert die Automaten in die Programmiersprache C [74]. Alexandre et al. [35] erweitern Syntax und Semantik von UML-Statecharts um Eigenschaften wie beispielsweise Taktgeber oder Echtzeitanforderungen. Die Verifikation der endlichen Automaten erfolgt mit dem Model Checker UPPAAL [86].

Das Werkzeug Hugo/RT [98][128] ist eine weitere Arbeit auf dem Gebiet der Verifikation von UML-Statecharts. Es übersetzt UML-Statecharts und Anforderungen, die ein Modell erfüllen muss, sowohl nach Promela, als auch nach Java. Der Übersetzungsalgorithmus nach Promela entspricht dem von Latella et al.[87]. Hugo/RT erlaubt als Eingabesprache zur Beschreibung der UML-Statecharts entweder XMI [110] oder eine proprietäre, domänenspezifische Sprache namens UTE (Textual UML Format). Der Benutzer kann aktive Klassen spezifizieren, die jeweils einen Zustandsautomaten enthalten. Die Kommunikation zwischen zwei aktiven Klassen wird durch das Senden und Empfangen von Signalen realisiert.

Holzmann et al. [102] stellen ein Werkzeug zur Verifikation von Zustandsautomaten vor, die mit *I-Logix Statemate* implementiert wurden (*I-Logix Statemate* war bis 2007 eine eigenständige Software und ist jetzt Teil der *IBM Rational* Entwicklungsumgebung [73]). Die Zustandsautomaten von *Statemate* orientieren sich an Harel-Statecharts. Für die erfolgreiche Verifikation stellen Holzmann et al. zunächst ein Zwischenformat namens *Extended Hierarchical Automata* (EHA) vor, das eine Untermenge von *Statemate* umsetzt. Anschließend wird ein Übersetzungsalgorithmus von den *Extended Hierarchical Automata* nach Promela gezeigt. Holzmann et al. verfolgen dabei den Ansatz, dass das

von ihnen entwickelte Werkzeug zunächst die graphische Notation eines mit *Statemate* erzeugten Automaten in einen *Extended Hierarchical Automata* überführt und diesen abschließend nach Promela übersetzt. Beim Erzeugen des Promela-Quellcodes werden verschiedene Optimierungen genutzt, um den Zustandsraums zu verkleinern und so der *State Space Explosion* entgegen zu wirken.

In der Arbeit *Modeling and Analyzing Software Behavior in UML* [123] [94] geht es, neben der Analyse und Modellierung von UML-Diagrammen, auch um die Verifikation von UML-Statecharts. Dazu werden zunächst Syntax und Semantik vollständig formalisiert und anschließend ein Übersetzungsalgorithmus nach Promela vorgestellt. Dieser Algorithmus wird von dem Werkzeug *vUML* umgesetzt, das automatisiert eine UML-Spezifikation nach Promela übersetzt, mit Spin verifiziert und anschließend Fehlerpfade visualisiert.

Wuwei et al. [129] definieren für UML-Statecharts eine Semantik, die äquivalent zu der von *Abstract State Machines* (ATS) [19] ist. Des Weiteren wird ein Transformationswerkzeug vorgestellt, das Statecharts, die mit XMI beschrieben sind, in AST-Modelle übersetzt. Die AST-Modelle werden im Anschluss in eine Hochsprache überführt oder mit einem eigens dafür entwickelten Model Checker namens TABU (*Tool for the active Behaviour of UML*) verifiziert [14].

Neben endlichen Automaten werden in dieser Arbeit im Rahmen der zweiten Fallstudie auch Web-Applikationen mit einer DSL umgesetzt, in eine Hochsprache überführt und von einem Model Checker verifiziert. Aus diesem Grund erfolgt im weiteren Verlauf dieses Abschnitts eine Untersuchung der verwandten Arbeiten, die eine Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation für Web-Anwendungen umsetzen. Die folgenden Aspekte sind von besonderem Interesse:

- Wie werden Beschreibungssprachen für Web-Applikationen umgesetzt?
- Wie werden die entsprechenden Modelle in Hoch- bzw. Model Checker-Eingabesprachen überführt?

Eine Möglichkeit, um Web-Anwendungen zu beschreiben, ist *MARIA XML*, bei der es sich um eine auf XML [20] basierende Sprache von Paterno et al. in [117] handelt. Der Schwerpunkt von *MARIA XML* liegt auf der Umsetzung von GUIs zur Nutzung von *Web Services*. Der dabei verfolgte Ansatz ist, dass aus der Spezifikation der genutzten *Web Services* automatisiert eine *MARIA XML Beschreibung* der Benutzeroberfläche generiert wird. Dazu werden die Parameter der Methoden analysiert, um beispielsweise im Falle eines *Strings* ein Textfeld für die Benutzereingabe zu erzeugen. Des Weiteren können automatisiert GUIs für verschiedene Zielplattformen erstellt werden (beispielsweise für Desktop-Anwendungen oder mobile Endgeräte). Das gesamte Konzept wird anhand einer Fallstudie zur Fernsteuerung eines Hauses demonstriert.

Freudenstein et al. [54] stellen ein Framework vor, das eine DSL für die Beschreibung von Webapplikationen zur Verfügung stellt. Für die domänenspezifische Sprache gibt

es mehrere graphische Editoren, um Modelle mit verschiedenen Diagrammtypen, wie beispielsweise Geschäftsprozessen [48], umzusetzen. Die DSL basiert auf der *Extensible Markup Language* (XML) [20] und wird nicht in eine Zielsprache übersetzt, sondern zur Laufzeit interpretiert bzw. ausgeführt. Ähnliche Ansätze, die auch auf Basis von Geschäftsprozessen Web-Anwendungen generieren, sind [83] und [126].

O'Hear et al. [112] stellen zwei domänenspezifische Sprachen vor, mit denen graphische Oberflächen für ERP-Software (Enterprise Resource Planning) generiert werden können. Aus den Modellen werden Web-Anwendungen generiert, die auf AJAX (Asynchronous JavaScript and XML) [56] basieren und dynamisch erzeugte Berichte in Form von PDF-Dateien anzeigen können.

Im Bereich der Qualitätssicherung von Web-Anwendungen wird von Alalfi et al. [2] ein Überblick über den Stand der Forschung gegeben. Dazu gehört, neben dem klassischem Testen, auch die formale Verifikation: Die Verifikation der Interaktion von Browser und Webserver wird in [93] von Licata et al. untersucht. Um die Bedeutung von formaler Verifikation im Bereich von Web-Anwendungen zu demonstrieren, wird ein Beispiel gezeigt, das zu einer Fehlbuchung in einem Onlinesystem zur Reservierung von Flügen führt. Um derartige Fehler mittels *Model Checking* zu finden, modellieren Licata et al. Webapplikationen mit dem *FLAVERS Toolkit* [32] als *Web Kontrollfluss Graphen*. Das *FLAVERS Toolkit* enthält einen *Model Checker*, der anschließend für die formale Verifikation der Web-Anwendungen genutzt wird. Bordbar et al. [18][10] nutzen Model-Driven Architecture, um Web-Anwendungen mit Klassendiagrammen zu modellieren. Die Diagramme werden in die Eingabesprache des *Alloy Model Checkers* [76] überführt und die Kommunikation zwischen Browser und Geschäftslogik verifiziert. Han et al. [64] modellieren die Navigation auf Webseiten mit endlichen Automaten. Die Statecharts werden automatisiert in die Eingabesprache des *SMV Model Checkers* überführt und verifiziert. Knapp et al. [81] nutzen UML-Statecharts und UML-Klassendiagramme, um Web-Anwendungen zu beschreiben. Die so erzeugten Modelle werden dem bereits vorgestellten Werkzeug Hugo/RT [128] übergeben und in die Eingabesprachen der Model Checker Spin oder UPPAAL transformiert.

Nakajima et al. [106] untersuchen die Verifikation von *Web Service Flows*. Ein *Web Service Flow* ist ein gerichteter Graph und besteht aus Knoten, die dem Aufruf eines *Web Services* entsprechen. Die Kanten des Graphs repräsentieren den Daten- bzw. Kontrollfluss. Zum Beschreiben der *Web Service Flows* wird von Nakajima et al. eine an die WSFL (Web Services Flow Language) [92] angelehnte Sprache genutzt. Nakajima et al. nutzen für die Verifikation den *Spin Model Checker*. Dazu werden die Knoten des *Web Service Flow Graphen* in Promela-Prozesse und die Kanten in *Message Channel* übersetzt. Der Vorteil dieses Ansatzes ist, dass im Gegensatz zum klassischem Testen kein Datenverkehr entsteht und so auch das Netzwerk nicht überlastet werden kann. Das gesamte Vorgehen wird abschließend mit einer Reisebüro-Fallstudie demonstriert, bei der pro Buchung unterschiedliche *Web Services* von Geschäftspartnern wie Fluglini-

en, Hotels, usw. aufgerufen werden.

Leung et al. [90] beschreiben eine Methode für die Modellierung und Verifikation von Webseiten, die auf Harel-Statecharts basiert. Dazu unterteilen sie zunächst Webseiten in eine Menge von *Links* und eine Menge von *Pages*. Anschließend wird eine Abbildung von Webseiten auf Statecharts vorgestellt. Dabei werden *Pages* als Zustände und *Links* als Transitionen umgesetzt. Nach der erfolgreichen Transformation einer Webseite in einen endlichen Automaten kann abschließend Spin für die formale Verifikation genutzt werden.

Deutsch et al. [37] stellen eine Beschreibungssprache für Webapplikationen vor, die auf WebML [24] basiert. Mit der Beschreibungssprache können Webapplikationen bestehend aus Webseiten, einer Datenbankanbindung und Benutzerinteraktionen implementiert werden. Jede Web-Anwendung besteht mindestens aus zwei Webseiten: Einer Startseite und einer Seite zum Anzeigen von Fehlermeldungen. Nach dem Modellieren der Webapplikation wird sie in eine *Abstract State Machine* (ASM) übersetzt, mit Anforderungen in Form von LTL-Formeln angereichert und einem *Model Checker* für *ASM Transducer* [133] verifiziert.

Der Stand der Forschung zeigt, dass die Modellierung und automatischen Transformation von verschiedenen Modelltypen in Hoch- bzw. Model Checker-Eingabesprachen im Rahmen von vielen Arbeiten untersucht worden ist. Die Umsetzung der Modelle erfolgt entweder mit proprietären DSLs, oder Beschreibungssprachen wie XMI. Für die Verifikation kommen eigene Model Checker, oder bestehende Werkzeuge, wie beispielsweise Spin, zum Einsatz.

Bei jeder der vorgestellten Arbeiten handelt es sich um eine Speziallösung, bei der nachträgliche Anpassungsmaßnahmen nicht vorgesehen sind. Dies ist aus dem folgenden Grund, insbesondere für UML-Statecharts, ein Nachteil: Die Arbeiten über endliche Automaten setzen, in Abhängigkeit von ihrem Schwerpunkt und bedingt durch die nicht vollständig beschriebene Semantik in der UML-Spezifikation, verschiedene Eigenschaften der Statecharts unterschiedlich um. So nutzen beispielsweise Latella et al. [87] einen FIFO-Queue, während Wasowski et al. [151] einen Ringpuffer verwenden. Diese unterschiedlichen Implementierungen ergeben sich aus verschiedenen Anforderungen in den entsprechenden Fallstudien. Wenn eine der verwandten Arbeiten im Rahmen eines Softwareprojekts eingesetzt werden soll, führt dies, bei einer nachträglich gewünschten Ergänzung mit beispielsweise Taktgebern aus [35], zu Problemen und erhöhtem Aufwand.

Hier wird der Vorteil eines generisch einsetzbaren Frameworks zur Verknüpfung von formaler Verifikation und modellgetriebener Entwicklung erkennbar: Das in dieser Arbeit entwickelte DVF stellt dem DSL-Entwickler Ableitungsregeln für verschiedene Sprachеigenschaften und Transformatoren in Hoch- bzw. Model Checker-Eingabesprache zur Verfügung. Dadurch muss nur noch ein Teil der jeweiligen DSL manuell implementiert werden. Die Komplexität des DSL-Projekts sinkt, wodurch Wartbarkeit und Wieder-

verwertbarkeit erhöht werden. Da das DVF keine tiefergehenden Kenntnisse im Bereich Model Checking voraussetzt, kann eine DSL auch im Rahmen von Projekten erweitert werden, in denen kein enstprechendes Expertenwissen vorhanden ist. Gleichzeitig wird durch den Stand der Forschung im Bereich UML-Statecharts auch eine Anforderung an das DVF deutlich: Das DSL Verification Framework muss nachträglich erweiterbar sein, um bei Bedarf zusätzliche Elemente, wie beispielsweise den Ringpuffer aus [151] zu integrieren.

Des Weiteren konzentriert sich der Stand der Forschung im Bereich von UML-Statecharts auf die Transformation der jeweiligen Statechart-Elemente, wie Zustände und Transitionen. Die endlichen Automaten der UML beinhalten jedoch auch Verhalten, das beispielsweise beim Betreten und Verlassen von Zuständen ausgeführt wird. Daraus ergibt sich die Notwendigkeit, in Abschnitt 3.2 auch den Stand der Forschung im Bereich der imperativen Sprachen zu untersuchen. Anhand der Ergebnisse wird evaluiert, wie Verhalten mit dem DVF, und somit auch in unterschiedlichen domänenspezifischen Sprachen, umgesetzt werden kann.

Der Stand der Forschung im Bereich der Web-Anwendungen zeigt, dass es viele Arbeiten gibt, die sich mit dem Modellieren und Verifizieren befassen. Dabei fällt auf, dass immer nur einzelne Aspekte untersucht werden:

- Der Aufruf von Webservices.
- Die Navigation in Webseiten.

Bei einer domänenspezifischen Sprache kann es jedoch wünscheswert sein, dass zwar nur ein bestimmter Aspekt mit der jeweiligen DSL beschrieben wird, trotzdem aber auch Kommunikationsschnittstellen zu anderen Domänen vorhanden sind. Dies verdeutlicht das folgende Beispiel: Gegeben sei eine DSL, mit der die Oberfläche einer Webseite beschrieben werden kann. Die Webseite kann Elemente, wie beispielsweise Schaltflächen, enthalten, die einen Webservice aufrufen. Die Webservices sind nicht Teil der DSL und werden direkt mit einer Hochsprache umgesetzt. Für die Verifikation muss jedoch auch in der DSL eine abstrakte Form des aufgerufenen Webservices vorhanden sein, die zumindest Aufschluss über den zu erwartenden Rückgabewert gibt. Daraus lässt sich an das DVF die folgende Anforderung ableiten: Für die Verifikation von Schnittstellen (und auch für die Reduktion des Zustandsraums) muss das DVF dem DSL-Entwickler die Möglichkeit geben, eigenschaftserhaltene Abstraktion in seiner domänenspezifischen Sprache zu verankern.

Des Weiteren werden in keiner der verwandten Arbeiten im Gebiet der UML-Statecharts und Web-Applikationen, bei der Transformation in eine Model Checker-Eingabesprache, Optimierungen auf Modellebene zur Reduktion des Zustandsraums berücksichtigt. Derartige Optimierungen sollen in die Entwicklung des DVF einfließen, um auch den Zustandsraum größerer Modelle verifizieren zu können.

3.3.2 Generische Ansätze

Das vorherige Kapitel 3.3.1 zeigt spezifische Fallstudien, die Modelle einer bestimmten Domäne erzeugen, verifizieren und in ausführbare Software überführen. Zu diesem Zweck werden domänenspezifische Sprachen entworfen und Transformatoren implementiert. Bei den Ansätzen aus Abschnitt 3.3.1 zeigt sich die folgende Schwierigkeit: Diese verwandten Arbeiten sind statische Lösungen, die keine nachträgliche Anpassungen vorsehen. Probleme ergeben sich, wenn eine DSL im Rahmen eines Softwareprojekts mehrmals modifiziert wird, dadurch DSL-Interpreter, DSL-Übersetzer usw. geändert werden müssen und es so zu Inkonsistenzen kommen kann. Dies wird beispielsweise bei UML-Statecharts deutlich, die viele verschiedene Eigenschaften und semantische Ergänzungen aufweisen. Dieser Abschnitt untersucht deshalb, welche generische Frameworks es gibt, deren Ziel die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation ist. Die verwandten Arbeiten lassen sich in zwei Kategorien unterteilen:

1. Frameworks, die dem DSL-Entwickler ein Zwischenformat zur Verfügung stellen, in das die Modelle überführt werden.

2. Frameworks, die dem DSL-Entwickler semantische Konstrukte zur Verfügung stellen, mit denen er das Metamodell ergänzen kann. Die semantischen Informationen werden vom Framework bei der Übersetzung in die verschiedenen Zielsprachen ausgewertet und berücksichtigt.

Die folgenden Arbeiten gehören zur ersten Kategorie und arbeiten mit Zwischenformaten: Amstel et al. [147] untersuchen die Anpassungen von domänenspezifischen Sprachen, die durch neue Anforderungen oder Erkenntnisse im Laufe eines Softwareprojekts notwendig werden. Durch das Verändern von Syntax und Semantik einer DSL müssen auch die Modelltransformatoren angepasst werden. Im Rahmen einer Fallstudie wird zunächst eine DSL namens *Simple Language of Communicating Objects* (SLCO) vorgestellt, die zur Beschreibung von dynamischen Kommunikationssystemen dient. Ein Kommunikationssystem besteht aus einer Menge von Objekten, wobei die Semantik jedes Objekts mit einem endlichen Automaten beschrieben ist. Objekte können untereinander über sogenannte Message-Channel kommunizieren. Für die SLCO werden drei Modelltransformatoren vorgestellt:

- Übersetzung in eine Sprache zum Ausführen des Modells.
- Übersetzung in eine Sprache für die Simulation des Modells.
- Übersetzung in eine *Model Checker Eingabesprache* für die Verifikation des Modells.

Bei der Model Checker-Eingabesprache handelt es sich in der Arbeit von Amstel et al. um Promela. Pro Zielsprache (Ausführung, Simulation und Verifikation) ist die Übersetzung

auf mehrere Transformatoren aufgeteilt, die entweder Elemente der DSL verändern oder neue Elemente hinzufügen. Bei einer Transformation wird das SLCO-Modell zunächst automatisiert in ein weiteres SLCO-Modell überführt (um beispielsweise synchrone in asynchrone Message-Channel umzuwandeln), bevor eine Übersetzung in eine Zielsprache wie Promela stattfindet. Durch die Modularität der Modelltransformation können semantische Änderungen in der DSL schnell implementiert werden, da lediglich einer der Transformatoren angepasst werden muss.

Thibault et al. [142] stellen ein Framework zur Generierung von DSLs und anschließenden Transformation in verschiedene Zielsprachen vor. Alternativ kann auch, anstelle eines Transformators, ein Interpreter generiert werden. Um die DSL-Entwicklung zu vereinfachen, muss der DSL-Entwickler die Modelle nicht direkt in die Zielsprache, sondern stattdessen in endliche Automaten überführen. Dieses Zwischenformat wird vom Framework entweder interpretiert, oder in die entsprechenden Zielsprachen übersetzt. Die Umsetzung der Arbeit wird mit einer DSL zur Implementierung von Grafikkartentreibern demonstriert.

Auch Pedro at al. [118] verifizieren Modelle und generieren aus ihnen im Anschluss ausführbare Prototypen. Dazu sieht ihr Lösungsansatz vor, dass Modelle nicht direkt in die Zielsprache, sondern Petri-Netze [119] überführt werden. Die Petri-Netze können anschließend mit den Methoden der formalen Verifikation untersucht, oder automatisiert in Java-Quellcode transformiert werden.

Der bisher in diesem Abschnitt vorgestellte Stand der Forschung gehört zur ersten Kategorie und arbeitet mit Zwischenformaten. Die folgenden verwandten Arbeiten setzen die zweite Kategorie um und bereichern Metamodelle mit semantischen Informationen: Certan et al. stellen das Werkzeug VIATRA (Visual Automated Transformations) [34] vor. Der DSL-Entwickler spezifiziert mit XMI (XML Metadata Interchange) [110] Metamodell und Transformationsschablonen. VIATRA generiert daraus die entsprechenden Modelltransformatoren. Als Fallstudie werden UML-Statecharts nach Promela überführt. Ein weiterer, wichtiger Aspekt ist, dass neben den Modellen, auch die Transformatoren validiert werden.

Lukman et al. [97] stellen einen Ansatz vor, der aus dem Metamodell einer DSL automatisiert Werkzeuge, wie beispielsweise einen Interpreter oder einen Debugger, generiert. Dies geschieht aus der Motivation heraus, dass DSLs häufig von Domänenexperten ohne oder mit wenig Programmiererfahrung genutzt werden, die die entsprechenden Werkzeuge nicht selbst implementieren können. Zu diesem Zweck wird das Metamodell bzw. die Grammatik einer Sprache mit semantischen Regeln und Attributen aus einer proprietären DSL erweitert. Anschließend wird ein Algorithmus vorgestellt, der anhand des Metamodells automatisiert den *Abstract Syntax Tree* eines Modells auswertet und die dazugehörigen semantischen Regeln ausführt.

Chen et al. [27] stellen einen ähnlichen Ansatz namens *Semantic Anchoring* vor, um die Syntax einer domänenspezifischen Sprache mit einer Semantik zu verknüpfen und

so die DSL von einem Interpreter ausführbar bzw. einem *Model Checker* verifizierbar zu machen. Dazu werden Syntax und Semantik einer domänenspezifischen Sprache auf die *Abstract State Machine Language* (AsmL [101]) abgebildet. Anschließend wird mit der *Unified Model Transformation Language* (UMT) [77] ein Transformator erzeugt, der automatisiert Modelle der DSL in die AsmL überführt.

Der Ansatz wird mit einer DSL namens FSM (*Finite State Machines*) demonstriert, die aus Zuständen, Transitionen, Events und endlichen Automaten besteht. Zustände, Transitionen und *Events* werden auf AsmL-Datentypen abgebildet. Endlichen Automaten werden mit AsmL als Prozesse implementiert, die auf *Events* reagieren und Transitionen ausführen. Mittels UMT wird ein Transformator geschrieben, der den *Abstract Syntax Tree* einer FSM-Beschreibung nach AsmL überführt und mit den bestehenden AsmL-Werkzeugen ausführbar bzw. verifizierbar macht.

Die Analyse der verwandten Arbeiten zeigt, dass es bereits generische Frameworks gibt, um Modelle zu beschreiben und in Zielsprachen zu überführen. Der erste Ansatz, der Zwischenformate verwendet, führt zu den folgenden Vorteilen:

- Ein Anwender überführt sein Modell lediglich in die Zwischensprache. Im Anschluss kann ein Transformator, der Teil des Frameworks ist, die Übersetzung in die Zielsprache durchführen. Der Vorteil im Bereich der formalen Verifikation ist, dass der DSL-Entwickler nur wenig Expertenwissen in der jeweiligen Model Checker-Eingabesprache benötigt. Er muss lediglich mit dem Zwischenformat vertraut sein.

- Auch wenn ein Modell in mehrere Zielsprachen übersetzt werden soll, muss der Anwender nur einen Transformator implementieren, der das das Modell in die Zwischensprache überführt. Die Übersetzung von der Zwischensprache in die jeweiligen Zielsprachen übernehmen Transformatoren, die Teil des entsprechenden Frameworks sind.

Von Nachteil sind fehlende Erweiterungsmöglichkeiten, die bei der ersten Kategorie der verwandten Arbeiten nicht vorgesehen sind. Dies ist problematisch, wenn das Zwischenformat nicht ausreichend ist, um ein bestimmtes DSL-Projekt umzusetzen. Ein weiterer Nachteil ist, dass der DSL-Entwickler für sein Modell einen vollständigen Transformator in die Zwischensprache implementieren muss. Dies kann, in Abhängigkeit der Komplexität der Zwischensprache, sehr aufwändig sein.

Das Problem des erhöhten Arbeitsaufwands wird von den verwandten Arbeiten, die zur zweiten Kategorie gehören, wie folgt gelöst: Der DSL-Entwickler bekommt vorgefertigte, semantische Aspekte zur Verfügung gestellt und kann diese, je nach Bedarf, in sein Metamodell integrieren. Zusätzlich besitzt das jeweilige Framework Transformatoren, die die semantischen Aspekte bei der Überführung in verschiedene Zielsprachen berücksichtigen. Dadurch wird der Arbeitsaufwand des DSL-Entwicklers reduziert und die Umsetzung der DSL bzw. des Transformators beschleunigt. Problematisch ist bei

den verwandten Arbeiten, dass immer davon ausgegangen wird, dass das zu transformierende Modell korrekt ist und alle Anforderungen erfüllt. Die formale Verifikation mit einem Model Checker ist nicht vorgesehen.

Da beide Ansätze aus dem Stand der Forschung vorteilhaft sind, wird das in dieser Arbeit vorgestellte DSL Verification Framework sie miteinander kombinieren. Zum Einen ist die *Control Flow Intermediate Language* (CFIL) als Zwischenformat ein Teil des DVF. Ein DSL-Entwickler muss deshalb nur einen Transformator, zur Überführung in das CFIL-Format entwickeln und benötigt keine tiefergehenden Kenntnisse in den Zielsprachen, wie beispielsweise Promela. Des Weiteren stellt das DVF dem DSL-Entwickler verschiedene Produktionsregeln zur Verfügung, die er in seine Grammatik integriert. Ihre Semantik ist genau definiert und die Transformatoren des DVF überführen die Sprachkonstrukte automatisiert in eine Hoch- und eine Model Checker-Eingabesprache. Dazu gehören beispielsweise Regeln für Statements oder Expressions. Verglichen mit dem Stand des Forschung bietet das DVF jedoch noch weitere Vorteile: Da die formale Verifikation ein Teil des DVF ist, beinhaltet es Sprachkonstrukte, wie beispielsweise symmetrische Arrays, die den Zustandsraum des Modells verkleinern. So wird dem Problem der State Space Explosion entgegengewirkt. Ein weiterer Vorteil ist, dass das DVF Erweiterungsmöglichkeiten bietet. Dadurch kann ein DSL-Entwickler neue Sprachkonstrukte einfügen, falls der Funktionsumfang des DVF für ein DSL-Projekt nicht ausreichend ist.

4 Verknüpfung von MDD und formaler Verifikation

In diesem Kapitel werden die Konzepte und die Implementierung des DSL Verification Frameworks (DVF) vorgestellt. Das Ziel des DVF ist es, die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation zu ermöglichen. Die Analyse der Problemstellungen, die bei der Umsetzung des DVF berücksichtigt werden müssen, findet sich in Kapitel 1. Dabei wird auch deutlich, dass für das DVF ein Model Checker und ein Parsergenerator notwendig sind. Kapitel 2 stellt deshalb zwei geeignete Werkzeuge vor:

- Das Xtext-Framework für die Entwicklung und Transformation von domänenspezifischen Sprachen [46].
- Den Model Checker Spin [68], in dessen Eingabesprache Promela die Modelle überführt und verifiziert werden.

Bei Xtext und Spin ist zu beachten, dass diese Dissertation primär das DVF als Konzept vorstellt. Deshalb werden Xtext und Spin nur benötigt, um einen Prototypen zu entwickeln und die Machbarkeit des DVF nachzuweisen. Für die konkrete Implementierung des DVF können auch alternative Werkzeuge verwendet werden.

Um die Funktionsweise des DVF zu verstehen, muss die Umsetzung der modellgetriebenen Softwareentwicklung in Abbildung 1.1 betrachtet werden. Dabei wird deutlich, dass für den Einsatz der modellgetriebenen Entwicklung die folgenden Schritte notwendig sind:

1. Der DSL-Entwickler beschreibt die Grammatik seiner domänenspezifischen Sprache mit Produktionsregeln.
2. Der DSL-Entwickler implementiert einen Modelltransformator in eine Hochsprache.
3. Der DSL-Anwender nutzt die Sprache, um damit seine Modelle zu beschreiben und diese automatisiert in die entsprechende Zielsprache zu überführen.

Die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation erfordert eine Erweiterung von Punkt 2: Der DSL-Entwickler muss nicht nur einen Transformator

für die Überführung in eine Hoch-, sondern auch in eine Model Checker-Eingabesprache implementieren. Des Weiteren ist Expertenwissen im Bereich der Model Checker-Eingabesprache notwendig.

Der zentrale Ansatzpunkt des DSL Verification Frameworks zur Unterstützung des DSL-Entwicklers ist wie folgt: Das DVF enthält vorgefertigte Produktionsregeln und vorgefertigte Transformatoren. Die Nutzung des DSL Verification Frameworks kann mit einer Produktionslinie in der industriellen Fertigung verglichen werden: Ein herzustellendes Produkt besteht zu einem Teil aus vorgefertigten Standardkomponenten. Für einige Aspekte existieren jedoch keine vorgefertigten Elemente. Deshalb müssen die entsprechenden Komponenten speziell angepasst oder hergestellt werden. Analog dazu ist auch die Funktionsweise des DSL Verification Frameworks: Ein DSL-Entwickler, der eine bestimmte domänenspezifische Sprache umsetzen will, entnimmt einen Teil der Produktionsregeln und Transformatoren aus dem DVF. Lediglich die Elemente, die DSL-spezifisch sind, werden manuell implementiert. Somit führt die Nutzung des DSL Verification Frameworks zu den folgenden Vorteilen:

- Der generelle Aufwand, der bei einer Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation durch die Implementierung der beiden Transformatoren bzw. Produktionsregeln entsteht, wird durch die Verwendung vorgefertigter Komponenten reduziert. Somit wird die Entwicklung von domänenspezifischen Sprachen beschleunigt.

- Das DVF enthält vorgefertigte Produktionsregeln zur Beschreibung von logischen Ausdrücken und Zuweisungen. Für die manuelle Umsetzung derartiger Produktionsregeln sind tiefergehende Kenntnnisse im Bereich Compilerbau notwendig, um beispielsweise Links-Rekursion in der entsprechenden Grammatik aufzulösen [1]. Somit reduziert das DVF durch vorgefertigte Produktionsregeln die Notwendigkeit von Expertenwissen im Bereich der modellgetriebenen Entwicklung.

- Das DVF stellt dem DSL-Entwickler einen Transformator in eine Model Checker-Eingabesprache zur Verfügung. Durch den vorgefertigten Transformator wird die Notwendigkeit von Expertenwissen im Bereich der formalen Verifikation reduziert.

- Das DVF berücksichtigt bei der automatischen Transformation in die Model Checker Eingabesprache die in Abschnitt 3.1 vorgestellten Optimierungen auf Modellebene, um den Zustandsraum des Modells zu verkleinern und so der State Space Explosion entgegen zu wirken.

Im weiteren Verlauf dieses Abschnitts wird das DSL Verification Framework genauer vorgestellt. Die Beschreibung erfolgt zunächst in Abschnitt 4.1 auf hoher Abstraktionsebene. Danach verdeutlicht Unterkapitel 4.2 die genaue Funktionsweise des DVF mit

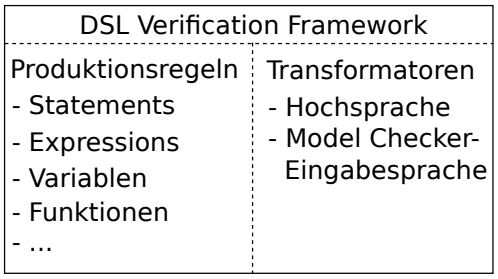

Abbildung 4.1: Aufbau des DVF auf hoher Abstraktionsebene

einem konkreten Beispiel. Zum Abschluss werden die detaillierten Konzepte und Referenzimplementierungen der einzelnen DVF-Komponenten in den Abschnitten 4.3 bis 4.8 präsentiert.

4.1 Konzept

Abbildung 4.1 zeigt den Aufbau des DSL Verification Frameworks. Sie macht deutlich, dass das DVF aus zwei Komponenten besteht: Vorgefertigte Produktionsregeln und vorgefertigte Transformatoren. Zu den Produktionsregeln gehören beispielsweise *Statements*, wie sie in C [89] oder Java [115] Verwendung finden. Mit Statements können Methoden aufgerufen oder Variablen Werte zugewiesen werden. Dementsprechend gehören auch Produktionsregeln zum DVF, die das Deklarieren von Variablen und Methoden ermöglichen. Konkret bedeutet das für einen DSL-Entwickler: Wenn er eine domänenspezifische Sprache beschreiben will, die beispielsweise das Deklarieren von Variablen ermöglicht, dann setzt er die dafür notwendigen Produktionsregeln nicht manuell um, sondern entnimmt sie dem DVF. Eine genaue Beschreibung der im DVF enthaltenen Produktionsregeln sowie eine Begründung für den ausgewählten Sprachumfang erfolgt in Abschnitt 4.3.

Die zweite zentrale Komponente des DVF, die anhand von Abbildung 4.1 deutlich wird, sind Transformatoren. Der erste Transformator überführt Modelle in eine Hochsprache. Der zweite Transformator nimmt eine Übersetzung in eine Model Checker-Eingabesprache vor. Beide Transformatoren sind modular aufgebaut und unterstützen genau die Sprachkonstrukte, die auch als Produktionsregeln Teil des DVF sind. Das bedeutet für einen DSL-Entwickler: Wenn er eine Grammatik umgesetzt hat und dafür beispielsweise die DVF-Produktionsregeln zum Beschreiben von Variablen verwendet, dann stellt ihm das DSL Verification Framework auch passende Transformatoren zur Verfügung, die Variablen automatisiert in eine Hoch- bzw. Model Checker-Eingabesprache überführen. Beide Transformatoren werden im Detail in den Abschnitten 4.7 und 4.6 vor-

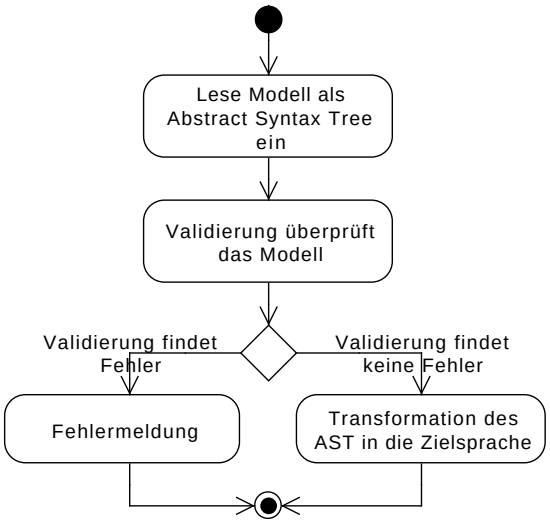

Abbildung 4.2: Funktionsweise der Transformatoren (Aktivitätsdiagramm)

gestellt. Die Übersetzung der einzelnen DVF-Elemente gliedert sich wie folgt: Zunächst wird die Semantik des jeweiligen Sprachkonstrukts beschrieben. Im Anschluss erfolgt die Diskussion möglicher Transformationsansätze, um daraus einen geeigneten Algorithmus auszuwählen. Für den Nachweis einer korrekten bzw. vollständigen Übersetzung muss bewiesen werden, dass die Sprachkonstrukte des DVF und der jeweiligen Zielsprache semantisch äquivalent sind. Hierfür bietet sich die Nutzung einer operationellen Semantik an [108][121]. Dies übersteigt jedoch den Rahmen der vorliegenden Dissertation. Deshalb wird die Korrektheit der Übersetzung lediglich auf informeller Ebene gezeigt und das Bilden einer operationellen Semantik im Rahmen eines formalen Beweises verbleibt für zukünftige Arbeiten.

Abbildung 4.2 zeigt die Funktionsweise der DVF-Transformatoren mit einem UML-Aktivitätsdiagramm. Jeder Transformator bekommt als Eingabe den vom Parser erzeugten *Abstract Syntax Tree* (AST) übergeben. In einem ersten Schritt erfolgt die Validierung (vgl. Abschnitt 2.2.3). Sie verifiziert verschiedene Aspekte des Eingabemodells, wie beispielsweise Typsicherheit. Der genaue Funktionsumfang der Validierung wird im Detail in Abschnitt 4.5 beschrieben. Wenn die Validierung im Eingabemodell keine Fehler findet, erfolgt die Transformation in die Zielsprache.

Neben der Funktionsweise der Transformatoren ist auch von Interesse, wie sie ein DSL-Entwickler in sein DSL-Projekt integriert. Daher wird im weiteren Verlauf dieses

Abschnitts der für die Nutzung des DSL Verification Frameworks notwendige Arbeitsablauf beschrieben:

1. Nachdem der DSL-Entwickler die domänenspezifische Sprache geplant hat, entnimmt er die notwendigen Produktionsregeln aus dem DVF und integriert sie in seine Grammatik.
2. Für alle DSL-spezifischen Sprachkonstrukte, die nicht im DVF enthalten sind, implementiert der DSL-Entwickler eigene Produktionsregeln.
3. Der DSL-Entwickler erzeugt aus der Grammatik einen Parser.
4. Der DSL-Entwickler integriert die beiden DVF-Transformatoren in sein Projekt. Sie ermöglichen eine automatische Übersetzung des CFIL-Formats nach Java und Promela.
5. Der DSL-Anwender beschreibt mit der domänenspezifischen Sprache Modelle und Anforderungen. Diese werden anschließend mittels Parser bzw. Transformatoren in eine Hoch- und eine Model Checker-Eingabesprache überführt. Der Model Checker verifiziert automatisch, dass das Modell alle Anforderungen erfüllt. Sofern ein Fehler gefunden wird, modifiziert der DSL-Anwender sein Modell und lässt es erneut in eine Model Checker-Eingabesprache transformieren. Sobald der Model Checker keine verletzten Anforderungen mehr findet, kann die Generierung des Hochsprachen-Quellcodes erfolgen.

Dieses Vorgehen führt zu einem Problem. Der DSL-Entwickler muss, wie bereits beschrieben, einen Teil der Produktionsregeln selbst implementieren. Somit erzeugt der Parser einen AST, dessen Elemente in zwei Klassen eingeteilt werden können: Sprachkonstrukte, die der DSL-Entwickler beschrieben hat und Sprachkonstrukte, die Teil des DVF sind. Die Transformatoren des DVF können jedoch nur die Elemente des AST verarbeiten, die auf Produktionsregeln des DSL Verification Frameworks basieren. Die Semantik der Sprachkonstrukte, die der DSL-Entwickler manuell implementiert hat, ist den Transformatoren des DVF hingegen nicht bekannt. Somit können sie auch nicht automatisiert in eine Hoch- bzw. Model Checker-Eingabesprache übersetzt werden.

Das bedeutet, der DSL-Entwickler muss die Transformatoren des DVF so erweitern, dass sie auch die vom ihm beschriebenen Sprachkonstrukte unterstützen. Dies ist jedoch nicht wünschenswert, da für eine Anpassung der Transformatoren Expertenwissen im Bereich der Zielsprachen, zu denen auch eine Model Checker-Eingabesprache gehört, notwendig ist. Das DSL Verification Framework soll jedoch auch in Projekten genutzt werden können, ohne dass Expertenwissen im Bereich der formalen Verifikation seitens DSL-Entwickler bzw. DSL-Anwender vorhanden ist.

Die Analyse der verwandten Arbeit in Abschnitt 3.3.2 zeigt, dass dieses Problem durch die Transformation in ein Zwischenformat gelöst werden kann. Deshalb stellt das DSL

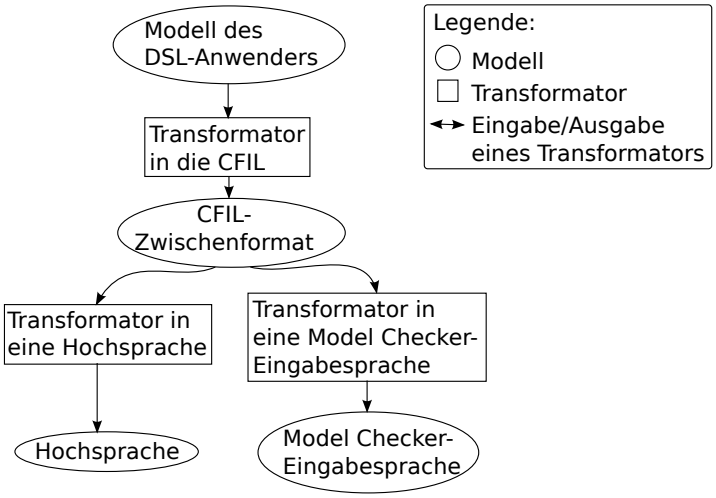

Abbildung 4.3: Bedeutung der Control Flow Intermediate Language

Verification Framework dem DSL-Entwickler die sogenannte *Control Flow Intermediate Language* zur Verfügung. Die Bedeutung der CFIL wird anhand von Abbildung 4.3 deutlich. Sie zeigt, wie ein Modell vom DVF mittels der CFIL in eine Hoch- bzw. Model Checker-Eingabesprache transformiert werden kann. Der Abstract Syntax Tree des Modells besteht aus Sprachkonstrukten, die Teil des DVF sind, oder die vom DSL-Entwickler umgesetzt werden. Der DSL-Entwickler schreibt deshalb für alle Elemente seiner Sprache, die nicht Teil des DVF sind, einen Transformator, der die entsprechenden Sprachkonstrukte in Elemente der CFIL überführt. Die Transformatoren des DVF, die im Anschluss die Übersetzung in eine Hoch- bzw. Model Checker-Eingabesprache durchführen, unterstützen die Sprachkonstrukte der CFIL. Das bedeutet, der DSL-Entwickler muss nicht, wie zuvor beschrieben, die beiden bestehenden Transformatoren erweitern, sondern stattdessen einen Transformator in die plattformunabhängige CFIL implementieren. Dies führt zu dem Vorteil, dass keine Kenntnisse im Bereich der formalen Verifikation notwendig sind. Des Weiteren kann es auch nicht zu Inkonsistenzen kommen, wenn zwei voneinander unabhängige Transformatoren modifiziert werden müssen. Durch die Integration der CFIL ergibt sich für den DSL-Entwickler der folgende Arbeitsablauf:

1. Der DSL-Entwickler entnimmt die Produktionsregeln aus dem DVF, die er für sein DSL-Projekt benötigt. Für die DSL-spezifischen Elemente schreibt er eigene Produktionsregeln.

2. Aus der daraus resultierenden Grammatik wird automatisiert ein Parser generiert.

3. Der DSL-Entwickler schreibt einen Transformator, der alle Sprachkonstrukte, die

auf seinen eigenen Produktionsregeln basieren, in die CFIL übersetzt. Sprachkonstrukte, die ein Teil des DVF sind, bleiben hingegen unverändert.

4. Der DSL-Anwender nutzt die so erzeugte domänenspezifische Sprache zum Beschreiben von Modellen. Die Modelle werden vom Transformator des DSL-Entwicklers automatisiert in die CFIL überführt. Danach können sie von den Transformatoren des DVF in eine Hoch- bzw. Model Checker-Eingabesprache übersetzt werden.

Die in der CFIL enthaltenen Sprachkonstrukte werden im Detail in Abschnitt 4.4 vorgestellt.

Das DSL Verification Framework beinhaltet Transformatoren und Produktionsregeln, die ein DSL-Entwickler zur Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation in seine DSL-Projekte integrieren kann. Im Rahmen eines DSL-Projekts kann es notwendig sein, das DVF zu erweitern. Dazu gehört beispielsweise das Hinzufügen von Produktionsregeln, die ein DSL-Entwickler in späteren Projekten wiederverwendet. Alternativ kann es auch notwendig sein, die CFIL mit neuen Sprachkonstrukten zu ergänzen. Deshalb zeigt Abschnitt 4.8, wie das DSL Verification Framework mit neuen Elementen und Transformatoren erweitert werden kann.

4.2 Anwendungsbeispiel

In diesem Abschnitt wird das DSL Verification Framework anhand eines konkreten Beispiels erläutert. Das Ziel ist es zu demonstrieren, wie ein Entwickler vorgehen muss, der mit dem DVF eine DSL entwirft. Verglichen mit dem vorangegangenen Unterkapitel werden dabei mehr Details ersichtlich und die Grundlage für eine genauere Beschreibung der einzelnen DVF-Komponenten in den darauf folgenden Abschnitten 4.3 bis 4.7 gelegt.

Gegeben sei ein DSL-Entwickler, der mit dem DVF eine domänenspezifische Sprache für endliche Automaten umsetzen möchte. Um derartige Automaten-Modelle in Zielsprachen zu überführen, muss der DSL-Entwickler mit dem DVF wie folgt vorgehen:

1. Er entwirft ein Konzept, in dem die Elemente der DSL und ihre Semantik enthalten sind.

2. Er beschreibt eine Grammatik.

3. Wenn es vorgefertigte Produktionsregeln im DVF gibt, integriert er sie in seine Grammatik.

4. Wenn das DVF keine passenden Produktionsregeln enthält, beschreibt er sie manuell.

5. Er implementiert einen Transformator, der alle Sprachkonstrukte, die auf manuell beschriebenen Produktionsregeln basieren, in die CFIL übersetzt.

6. Abschließend kann er mit der DSL Modelle beschreiben und sie autmatisiert in eine Hoch- bzw. Model Checker-Eingabesprache überführen.

Die Schritte 1-5 werden im weiteren Verlauf dieses Unterkapitels genauer vorgestellt.

Im Rahmen des ersten Arbeitsschritts muss sich der DSL-Entwickler überlegen, welche Sprachkonstrukte in der domänenspezfischen Sprache enthalten sind. Da es sich im Rahmen des Anwendungsbeispiels um eine DSL für endliche Automaten handelt, besteht sie aus drei Komponenten:

- Einer Menge von Variablen, auf die der Automat lesend und schreibend zugreifen kann.
- Eine Menge von Zuständen.
- Eine Menge von Transitionen.

Jede Transition kann eine sogenannte Guard-Expression enthalten. Das bedeutet, sie ist nur ausführbar, wenn die darin enthaltene Expression den Wert *true* annimmt. Das folgende Beispiel zeigt eine mögliche DSL, die die genannten Eigenschaften umsetzt:

```
1 byte a = 3;
2
3 state A;
4 state B;
5
6 transition{
7   A -> B,
8   guard{a==3}
9 }
```

Listing 4.1: Einfacher endlicher Automat

Das obige Listing 4.1 besteht aus einer Variablen, zwei Zuständen und einer Transition. Die Transition ist ausführbar, wenn die Guard-Expression $a == 3$ den Wert *true* annimmt.

Nach der Konzeptionierung entwirft der DSL-Entwickler eine Grammatik (vgl. Schritt 2-4 zu Beginn dieses Abschnitts). Sie muss Produktionsregeln enthalten, um Variablen, Zustände, Transitionen und Expressions abzubilden:

```
1 Model:
2   variables+=DVFVariableDeclare*
3   states+=State*
```

```
4   transitions+=Transition*
5 ;
6
7 State:
8   'state' name=ID ';'
9 ;
10
11 Transition:
12   'transition' '{' source=[State]
13   '->' destination=[State] 'guard'
14   '{' (expr=DVFExpression)? '}' '}'
15 ;
```

Listing 4.2: Grammatik für endliche Automaten

Die Grammatik im obigen Listing ist mit Xtext umgesetzt. Dies ist durch die Tatsache begründet, dass auch die Produktionsregeln in der prototypischen Umsetzung des DSL Verification Frameworks auf Xtext basieren. Die Syntax der Xtext-Spezifikationssprache wird in Abschnitt 2.2.1 genauer vorgestellt.

Der DSL-Entwicker hat in Listing 4.2 für seine domänenespezifische Sprache drei Produktionsregeln deklariert. *Model* ist das Wurzelelement und wird in Variablen, Zustände und Transitionen abgeleitet. Das DVF enthält eine vorgefertigte Produktionsregel zum Beschreiben von Variablen namens *DVFVariableDeclare*. Der DSL-Entwickler muss deshalb keine eigene Regel für Variablen umsetzen, sondern nutzt stattdessen *DVFVariableDeclare* aus dem DVF. Die prototypische Umsetzung von *DVFVariableDeclare* kann in Abschnitt 4.3.2 eingesehen werden.

Im Gegensatz zu Variablen sind die Produktionsregeln für Zustände und Transitionen DSL-spezifisch. Daher werden sie manuell vom DSL-Entwickler umgesetzt. Jeder Zustand besteht aus dem Schlüsselwort *state* und einem eindeutigen Bezeichner. Transitionen werden durch das Schlüsselwort *transition* eingeleitet und bestehen aus einem Ausgangszustand, einem Zielzustand und optional einer sogenannten *guard*. Die jeweilige Transition ist nur dann ausführbar, wenn die in *guard* enthaltene Expression den Wert *true* annimmt. Das DVF stellt eine Produktionsregel namens *DVFExpression* zum Beschreiben von Expressions zur Verfügung. Der DSL-Entwickler muss somit die entsprechende Produktionsregel nicht manuell implementieren, sondern kann sie dem DVF entnehmen. Eine prototypische Umsetzung von *DVFExpression* ist in Abschnitt 4.3.3 zu sehen.

Nach der Umsetzung der Grammatik erfolgt die Implementierung eines Transformators. Sprachkonstrukte, die auf *DVFExpression* und *DVFVariableDeclare* basieren, können direkt von den Transformatoren des DVF verarbeitet werden. *Transition* und *State* müssen hingegen in die Control Flow Intermediate Language (CFIL) überführt

werden. Der Transformations-Algorithmus verwendet dafür aus der CFIL die folgenden Elemente:

- Thread: Der Inhalt eines Thread-Blocks wird als eigener Prozess ausgeführt. Somit führen mehrere Thread-Blöcke zu Nebenläufigkeit. Ein Thread beinhaltet eine Menge von Statement-Blöcken.
- Statement-Block: Jeder Statement-Block kapselt eine Menge von Statements. Statement-Blöcke werden in einem Thread sequentiell ausführt.

Des Weiteren wird vom Transformator ein sogenannter Wait-Block genutzt, der Teil des DVF ist, jedoch nicht zur Control Flow Intermediate gehört. Der Aufbau eines Wait-Blocks ähnelt der Switch-Anweisung aus Java oder C [60]. Der Unterschied ist, dass ein Wait-Block erst dann die Ausführung fortsetzt, wenn eine der enthaltenen Expressions den Wert *true* annimmt. Wait-Blöcke werden im Detail in Abschnitt 4.3.4 vorgestellt. Die genaue Beschreibung von Threads und Statement-Blöcken erfolgt in Unterkapitel 4.4.

Nach der Einführung von Wait-Blöcken, Statement-Blöcken und Threads kann im weiteren Verlauf dieses Abschnitts näher auf den Transformator eingegangen werden. Dieser verwendet den folgenden Algorithmus, um endliche Automaten, die auf der Grammatik aus Listing 4.2 basieren, in die CFIL zu überführen:

1. Es wird ein Thread mit dem Namen *FSM* erzeugt. Dieser Thread repräsentiert den endlichen Automaten.
2. Jeder Zustand des Automaten wird in einen Statement-Block überführt.
3. Für jede ausgehende Transition, die Guard-Expressions enthält, wird ein Wait-Block in den Statement-Block eingefügt. Der Wait-Block führt genau dann eine Goto-Anweisung in den Folgezustand aus, wenn die Guard-Expression den Wert *true* annimmt.
4. Transitionen ohne Guard-Expressions werden direkt in Goto-Anweisungen überführt.
5. Wenn ein Zustand keine ausgehenden Transitionen enthält, wird eine Exit-Anweisung eingefügt, die die Ausführung des Threads und somit des endlichen Automaten beendet.

Basierend auf diesem Algorithmus wird der endliche Automat aus Listing 4.1 in das folgende CFIL-Modell überführt:

```
1 thread FSM entry A{
2   byte a;
3
```

```
4    statementblock A{
5      wait(a==3){
6          goto B;
7      }
8    }
9
10   statementblock B{
11     exit;
12   }
13 }
```

Listing 4.3: Transformation eines Automaten in die CFIL

Das obige Listing 4.3 enthält einen Thread mit dem Bezeichner FSM. Bei der Deklaration eines Threads muss ein Statement-Block als Eintrittspunkt angegeben werden. Das Modell aus Listing 4.3 beginnt daher mit der Ausführung von Statement-Block A. In A wird durch die Wait-Anweisung gewartet, bis die Variable a den Wert 3 annimmt. Abschließend terminiert der Thread in Statement-Block B durch Ausführung der Exit-Anweisung. Das Verhalten des Modells entspricht also genau dem Automaten aus Listing 4.1. Nach der Implementierung eines entsprechenden Transformators kann der DSL-Anwender mit der domänenspezifischen Sprache für endliche Automaten Modelle beschreiben. Diese werden vom Transformator des DSL-Entwicklers und den Transformatoren des DVF automatisiert in eine Hoch- bzw. Model Checker-Eingabesprache überführt.

Dieser Abschnitt hat die Verwendung des DSL Verification Frameworks anhand eines konkreten Beispiels demonstriert. Darauf aufbauend wird in den folgenden Unterkapiteln detailliert auf die einzelnen Komponenten des DVF eingegangen.

4.3 Vordefinierte DVF-Produktionsregeln

In diesem Abschnitt werden die Produktionsregeln vorgestellt, die Teil des DVFs sind. Die Regeln sind mit Xtext beschrieben und können in DSL-Projekte eingefügt werden. Das Beispiel im letzten Kapitel demonstriert dieses Vorgehen, indem die Regeln *DVFExpression* und *DVFVariableDeclare* in die Grammatik einer domänenspezifischen Sprache integriert werden.

Die Produktionsregeln des DVF werden unter den folgenden Gesichtspunkten ausgewählt: Die Analyse der verwandten Arbeiten in den Abschnitten 3.2 und 3.3.1 hat gezeigt, dass domänenenspezifische Sprachen sowohl DSL-spezische Elemente, als auch generische Sprachkonstrukte enthalten. DSL-spezifische Elemente sind auf die Bedürfnisse der entsprechenden domänenspezifische Sprache zugeschnitten. Dazu gehören beispielsweise in einer DSL, die das Modellieren von Bahnhöfen [22] ermöglicht, Elemente zur

Beschreibung einer Bahnschranke. In domänenspezifischen Sprachen für endliche Automaten [44] [87] [98] [128] entsprechen genau die Sprachkonstrukte den DSL-spezifischen Elementen, mit denen beispielsweise Zustände oder Transitionen umgesetzt werden können. Bei domänenspezifischen Sprachen für graphischen Oberflächen [20] [54] [112] sind Elemente zur Beschreibung von Texteingabefeldern oder Dialogboxen DSL-spezifisch.

Im Gegensatz zu den DSL-spezifischen Elementen können auch generische Sprachkonstrukte in domänenspezifischen Sprachen enthalten sein. Dazu gehören Elemente zum Modellieren von Verhalten, wie beispielsweise Variablen oder Statements. Sie werden im weiteren Verlauf dieser Arbeit gemäß Abschnitt 3.2 als imperative Sprachkonstrukte bezeichnet. Sie finden unter anderem in [98][128] Verwendung, um das Verhalten eines Modells beim Betreten oder Verlassen eines Zustands zu beschreiben. Konkret bedeutet dies, dass beispielsweise beim Betreten eines Zustands Methoden aufgerufen, arithmetische Operationen durchgeführt oder Variablen Werte zugewiesen werden können. Analog dazu können imperative Sprachkonstrukte auch in domänenspezifischen Sprachen für Web-Anwendungen oder graphischen Oberflächen [10] [18] [106] Verwendung finden, um beispielsweise das Verhalten der verschiedenen GUI-Elemente zu beschreiben.

Die Analyse der verwandten Arbeiten in Abschnitt 3.2 hat gezeigt, dass imperative Sprachkonstrukte in unterschiedlichen Modellierungssprachen genutzt werden. Beim Betrachten der zugrunde liegenden Grammatiken wird jedoch deutlich: Der Umfang und die Komplexität der Produktionsregeln bzw. des Transformators steigt deutlich an, wenn Statements, Expressions, Methodenaufrufe, usw. Teil einer Sprache sind [78] [115]. Aus dem Grund stellt das DSL Verification Framework dem DSL-Entwickler verschiedene Produktionsregeln zur Verfügung, mit denen er die imperative Sprachkonstrukte in seine Grammatik und somit seine DSL integrieren kann.

Die Auswahl der Produktionsregeln, die das DSL Verification Framework zur Verfügung stellt, orientiert sich an bestehenden Sprachen. Dazu gehören die Hochsprachen C [78] und Java [60]. Auch die imperativen Sprachkonstrukte des Werkzeugs Hugo/RT [98] [128] werden evaluiert. Neben Sprachen, die erst in eine Zielsprache transformiert werden müssen, gibt es auch Sprachen, die ein Interpreter zur Laufzeit auswertet. Dazu gehören die Script-Sprachen Python [120] und Perl [63]. Als Vertreter von funktionalen Programmiersprachen werden die Sprachkonstrukte von Haskell [70] untersucht. Des Weiteren erfolgt eine Evaluierung von Go [105], das als Sprache zur Systemprogrammierung konzipiert ist. Daraus ergeben sich für das DSL Verification Framework Produktionsregeln, die in die folgenden Kategorien unterteilt werden können:

- Eine Beschreibung von Sprachkonstrukten, mit denen eine DSL zur besseren Strukturierung auf mehrere Quellcode-Dateien aufgeteilt werden kann, erfolgt in Abschnitt 4.3.1.
- Sprachkonstrukte zum Deklarieren von Variablen werden in Abschnitt 4.3.2 vorgestellt.

- Abschnitt 4.3.3 beschreibt Produktionsregeln für Expressions, mit denen in einer DSL mathematische Ausdrücke oder Bitmanipulationen umgesetzt werden können.
- Abschnitt 4.3.4 zeigt Produktionsregeln für Statements, mit denen in einer domänenspezifischen Sprache beispielsweise Schleifen, If-Blöcke oder Zuweisungen beschrieben werden können.
- Abschnitt 4.3.5 stellt Sprachkonstrukte vor, mit denen in einer DSL Methoden deklariert werden können.
- Unterkapitel 4.3.6 stellt Produktionsregeln zur Umsetzung von objektorientierten Aspekten vor.

Die einzelnen Abschnitte sind wie folgt strukturiert: Zunächst werden die Sprachkonstrukte vorgestellt und ihre Syntax mit Xtext-Produktionsregeln beschrieben. Danach erfolgt, anhand von Beispielen, eine informelle Beschreibung der Semantik. Die genaue Definition der Semantik ergibt sich aus der Transformation nach Java bzw. Promela, die in den Abschnitten 4.6 bzw. 4.7 vorgestellt wird. Falls die im DVF enthaltenen Sprachkonstrukte in einem DSL-Projekt nicht ausreichend sind, zeigt Abschnitt 4.8, wie das DSL Verification Framework erweitert werden kann.

4.3.1 Include

Wenn der Quellcode eines Modells, das mit einer DSL beschrieben ist, wächst, kann es wünschenswert sein, ihn zur besseren Lesbarkeit auf mehrere Dateien aufzuteilen. Deshalb stellt das DVF eine Produktionsregel namens *DVFInclude* zur Verfügung, die sich an der Syntax von C [78] orientiert. Wenn ein DSL-Entwickler *DVFInclude* in seine Grammatik integriert, steht in der domänenspezifischen Sprache das Sprachkonstrukt *include* zur Verfügung, mit dem weitere Quellcode-Dateien in das Modell eingebunden werden können. Abbildung 4.4 zeigt exemplarisch die Verwendung der Include-Anweisung. Gegeben sei eine DSL, in der unter anderem auf Variablen zugegriffen werden kann. Mit der domänenspezifischen Sprache wird ein Modell beschrieben, das die Dateien *a.dsl* und *b.dsl* inkludiert. *A.dsl* deklariert die Variable *a*, *b.dsl* enthält die Variable *b*. Durch die Include-Anweisung können *a* und *b* im Modell referenziert werden, um beispielsweise eine Addition durchzuführen. Die dazugehörige Produktionsregel *DVFInclude* hat den folgenden Aufbau:

```
1 DVFInclude:
2   "include" importURI=STRING ";"
3 ;
```

Listing 4.4: Inkludieren von Dateien

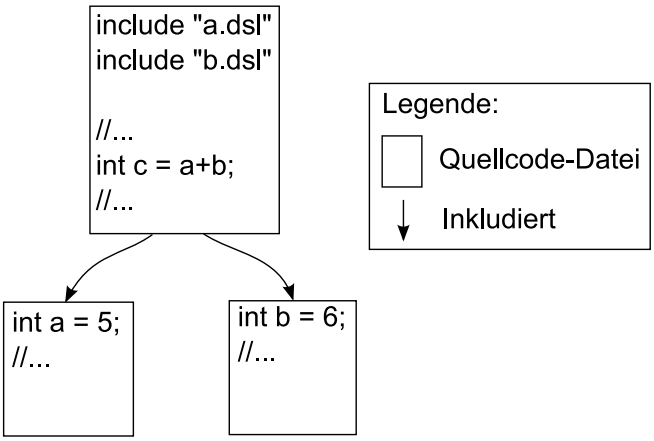

Abbildung 4.4: Anwendung der Include-Anweisung

Der Include-Mechanismus wird nativ von Xtext unterstützt. Daher muss der DSL-Entwickler den Mechanismus, der eine inkludierte Datei beim Parsen in den AST einbindet, nicht selber implementieren. Stattdessen ist das Schlüsselwort *importURI* in Xtext für das Inkludieren von Dateien reserviert und das Einbinden der externen Dateien wird automatisiert, ohne das Zutun des DSL-Entwicklers, durchgeführt.

4.3.2 Variablen

Es kann notwendig sein, eine DSL zu entwerfen, die das Deklarieren von Variablen ermöglicht, um darin Werte zwischenzuspeichern. Zu diesem Zweck enthält das DSL Verification Framework Produktionsregeln, mit denen die folgenden Elemente in domänenspezifischen Sprachen modelliert werden können:

- *DVFEnumDeclare* ermöglicht das Beschreiben von Aufzählungen. Wenn ein DSL-Entwickler *DVFEnumDeclare* in seine Grammatik integriert, können in der domänenspezifischen Sprache Aufzählungen deklariert werden, wie sie beispielsweise aus C [78] bekannt sind.

- *DVFVariableDeclare* ermöglicht das Beschreiben von Variablen, wie sie beispielsweise in C [78], Python [120] oder Perl [63] Verwendung finden. Wenn ein DSL-Entwickler *DVFVariableDeclare* in seine Grammatik integriert, können in der domänenspezifischen Sprache Variablen mit verschiedenen Datentypen und optional einem Initialisierungswert deklariert werden.

- *DVFScalarDeclare* ermöglicht das Beschreiben symmetrischer Arrays. Wenn ein DSL-Entwickler *DVFScalarDeclare* in eine Grammatik integriert, kann der DSL-

Anwender symmetrische Arrays zur Reduktion des Zustandsraums, wie sie in Abschnitt 2.1.4 vorgestellt werden, deklarieren.

- *DVFSignalDeclare* ist ein spezieller Datentyp für die asynchrone Kommunikation. Wenn die Produktionsregel *DVFSignalDeclare* in eine Grammatik integriert wird, können in der entsprechenden domänenspezifischen Sprache Message-Queues deklariert und asynchrone Signale versendet werden. Derartige Sprachkonstrukte finden beispielsweise in Java [115] oder Promela Verwendung, damit nebenläufige Prozesse miteinander kommunizieren [68] können.

Die Xtext-Beschreibungen dieser Elemente werden im weiteren Verlauf dieses Unterkapitels genauer vorgestellt.

DVFEnumDeclare

Wenn ein DSL-Entwickler die Produktionsregel *DVFEnumDeclare* in die Grammatik seiner domänenspezifischen Sprache integriert, können Aufzählungen deklariert werden. Aufzählungen haben den Vorteil, dass sie im Gegensatz zu Integer-Werten aussagekräftige Bezeichner aufweisen und somit die Lesbarkeit des Quellcodes eines Modells verbessern. Das folgende Listing zeigt die Umsetzung der Produktionsregel *DVFEnumDeclare*, damit im Anschluss die Syntax des Sprachkonstrukts beschrieben werden kann:

```
1 DVFEnumDeclare:
2   "enum" name=ID
3   "{" first=DVFEnumElement ("," next+=DVFEnumElement)* "}"
4 ;
5
6 DVFEnumElement:
7   name=ID
8 ;
```

Listing 4.5: Xtext-Grammatik für Enumerations

Jede Aufzählung besteht aus dem Schlüsselwort *enum*, einem Bezeichner und einer Menge von Aufzählungs-Elementen. Die Menge der Elemente steht in geschweiften Klammern und muss eine Mächtigkeit von mindestens 1 haben. Das folgende Beispiel demonstriert die Anwendung des Sprachkonstrukts:

```
1 enum Essen = {Pizza, Spaghetti, Lasagne}
2 enum Wetter = {Sonnenschein, Regen}
```

Listing 4.6: Deklaration von Aufzählungen

Das obige Listing 4.6 enthält die Aufzählungen *Essen* und *Wetter*. *Essen* setzt sich aus den Elementen *Pizza*, *Spaghetti*, *Lasagne* zusammen. Das *Wetter* kann entweder *Sonnenschein* oder *Regen* entsprechen.

Deklaration von Variablen

Das DVF beinhaltet eine Produktionsregel namens *DVFVariableDeclare*. Wenn diese Produktionsregel in eine Grammatik eingefügt wird, können im Rahmen der entsprechenden domänenspezifischen Sprache Variablen deklariert werden. Das nächste Listing zeigt die Umsetzung der Produktionsregel *DVFVariableDeclare*, damit im Anschluss die genaue Syntax des Sprachkonstrukts vorgestellt werden kann:

```
1  DVFVariableDeclare:
2    (
3      stattype=DVFVariableType
4      |
5      dyntype=[DVFIdentifier]
6    )
7    name=ID (bracket="[" size=INT "]")?
8    (op="=" init=DVFExpression)? ";"
9  ;
10
11 DVFVariableType:
12   bytetype="byte" | inttype="int" | booltype="bool" |
13   stringtype="string"
14 ;
15
16 DVFIdentifier:
17   DVFClass | DVFEnumDeclare | DVFEnumElement |
18   DVFScalarDeclare | DVFVariableDeclare | DVFFunctionDeclare |
19   DVFSignalDeclare
20 ;
```

Listing 4.7: Xtext-Grammatik für Variablen

Die Variablendeklaration basiert auf drei Produktionsregeln: Mit *DVFVariableDeclare* wird die eigentliche Variable spezifiziert. *DVFVariableType* wird für Variablen mit statischem Typ, wie beispielsweise Byte, benötigt. Nicht-statische Variablentypen, wie beispielsweise eine Klasse, werden mittels *DVFIdentifier* umgesetzt.

In jeder Variablendeklaration können eckige Klammern genutzt werden, um ein Array zu erzeugen. Die Array-Größe ist statisch und muss zusammen mit den eckigen Klammern angegeben werden. Die Initialisierung einer Variable ist optional und besteht aus einer Zuweisung und einer Expression. Die Regel *DVFVariableType* kann einen der vier Werte *byte*, *int*, *bool* oder *string* annehmen. Des Weiteren kann durch eine Ableitung nach *DVFIdentifier* auch eine Klasse oder eine Aufzählung als Datentyp angegeben werden.

Das folgende Beispiel zeigt zur Veranschaulichung eine domänenspezifische Sprache, deren Grammatik die Produktionsregeln *DVFEnumDeclare* und *DVFVariableDeclare* enthält:

```
1 enum Essen = {Pizza, Spaghetti, Lasagne}
2 Essen favorite = Essen.Spaghetti;
3 byte a[5] = 5;
```

Listing 4.8: Deklaration von Variablen

Das obige Listing 4.8 deklariert zwei Variablen und eine Aufzählung. Die Aufzählung *Essen* enthält die Elemente Pizza, Spaghetti und Lasagne. Im Anschluss wird die Variable *favorite* deklariert und ihr der Wert *Spaghetti* zugewiesen. Neben Aufzählungen unterstützt das DVF die statischen Datentypen *bool*, *byte*, *int* und *string*. Jede Variable kann entweder einzeln oder als Array deklariert werden. Dies geschieht beispielsweise in Listing 4.8 in Zeile 3. Wenn beim Anlegen eines Arrays ein Initialisierungswert übergeben wird, werden alle Elemente des Arrays mit dem entsprechenden Wert beschrieben. So enthalten beispielsweise in Listing 4.8 alle Elemente des Arrays *a* den Wert 5. Eine direkte Initialisierung mit verschiedenen Werten ist nicht möglich und ein Teil von zukünftigen Arbeiten. Das DVF definiert für die numerischen Datentypen *bool*, *byte* und *int* die folgenden Definitionsbereiche:

Datentyp	Minimaler Wert	Maximaler Wert
bool	true	false
byte	0	255
int	-2.147.483.648	2.147.483.647

Die Validierung des DSL Verification Frameworks stellt sicher, dass bei Zuweisungen und Initialisierungen die Wertebreiche der entsprechenden Variablentypen eingehalten werden.

Deklaration von symmetrischen Arrays

Neben Enumerations und Variablen können mit dem DVF in domänenspezifischen Sprachen auch symmetrische Arrays genutzt werden. Dafür muss der DSL-Entwickler die Produktionsregel *DVFScalarDeclare* in seine Grammatik integrieren. Bei den symmetrischen Arrays bzw. Scalarsets handelt es sich um einen besonderen Variablentyp, der zu den Optimierungen auf Modellebene gehört [41][109] und bereits in 2.1.4 vorgestellt worden ist. Scalarsets können beispielsweise genutzt werden, um in einer hardware-nahen DSL mehrere funktional-äquivalente Register einer CPU zu beschreiben. Ein weiterer Anwendungsfall ist bei Schaefer et al. [13] zu finden, die ein Kommunikationsprotokoll für Autos im Konvoi beschreiben. Die Zustände der Fahrzeuge können in einem symmetrischen Array gespeichert werden, da ihre Permutation für die Semantik des Modells nicht relevant ist.

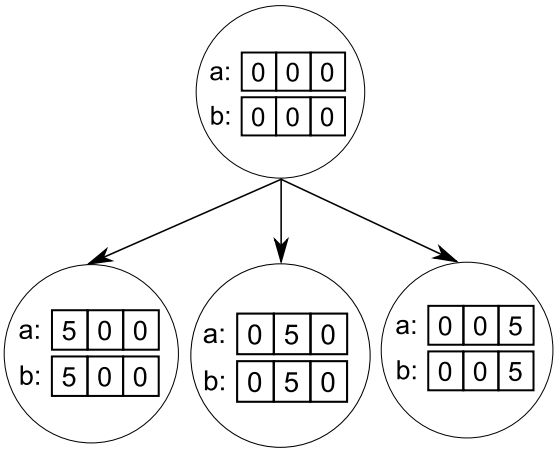

Abbildung 4.5: Zustandsraum bei nicht-symmetrischen Arrays

Die Implementierung der Produktionsregel *DVFScalarDeclare* wird mit dem nächsten Listing verdeutlicht:

```
1 DVFScalarDeclare:
2   "scalarset" name=ID "[" size=INT "]" "{"
3     (variables+=DVFVariableDeclare ";")+
4   "}"
5 ;
```

Listing 4.9: Xtext-Grammatik für Scalarsets

Jedes Scalarset besteht aus dem Schlüsselwort *scalarset*, einem eindeutigen Bezeichner, der Anzahl der Elemente in eckigen Klammern und den eigentlichen Elementen in geschweiften Klammern. Jedes Element entspricht wiederum einer Variablendeklaration, die mit der Produktionsregel *DVFVariableDeclare* umgesetzt wird. Ein Scalarset muss mindestens ein Element enthalten. Scalarsets, die weitere Scalarsets enthalten, sind nicht zulässig. Das folgende Listing zeigt zur Veranschaulichung eine DSL, deren Grammatik die Produktionsregel *DVFScalarDeclare* enthält:

```
1 scalarset scalar1[3]{
2   byte a;
3   byte b;
4 }
```

Listing 4.10: Deklaration von Scalarsets

Im obigen Beispiel wird das Scalarset *scalar1* deklariert, das eine Größe von 3 hat und die Elemente *a* und *b* enthält. Bei Scalarsets handelt es sich um symmetrische Arrays.

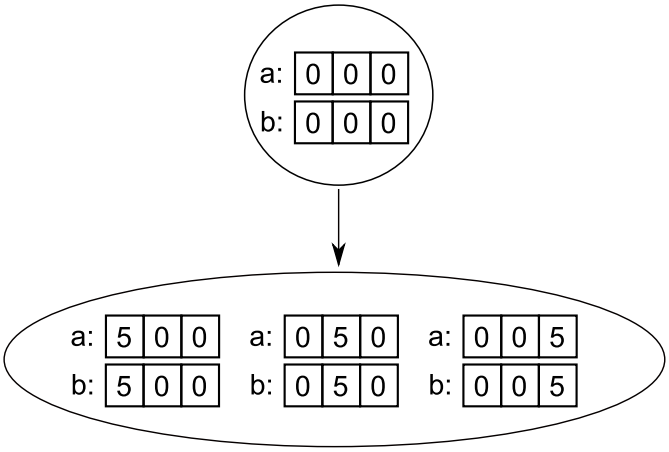

Abbildung 4.6: Zustandsraum bei symmetrischen Arrays

Das bedeutet, es werden die Arrays a und b vom Typ Byte erzeugt, die beide eine Größe von drei haben. Die Besonderheit ist, dass die Reihenfolge bzw. die Permutation der enthaltenen Elemente für den Model Checker nicht relevant ist. Dies verdeutlicht das folgende Beispiel: Gegegeben sei das Scalarset *scalar1* aus Listing 4.10, das mit dem Wert 0 initialisiert wird. Des Weiteren sei eine Operation gegeben, die den Arrays a und b an einer zufälligen Position den Wert 5 zuweist.

Abbildung 4.5 zeigt den Zustandsraum dieses Modells aus Sicht des Model Checkers, wenn a und b nicht als Scalarsets, sondern als reguläre Arrays implementiert werden. Jeder Zustand enthält die Belegung der Arrays a und b. Der Startzustand hat drei Folgezustände, da a und b an drei verschiedenen Indizes den Wert 5 zugewiesen bekommen können. Abbildung 4.5 zeigt den Zustandsraum des selben Modells, allerdings unter Verwendung von Scalarsets und somit symmetrischen Arrays. Da die Permutation der Array-Elemente für den *Model Checker* nicht mehr relevant ist, wird der Zustandsraum von 4 auf 2 reduziert.

Deklaration von Message-Queues

Neben Scalarsets können mit dem DVF in domänenspezifischen Sprachen auch Message-Queues deklariert werden. Um Message-Queues in einer domänenspezifischen Sprache zu nutzen, muss der DSL-Entwickler die Produktionsregel *DVFSignalDeclare* in seine Grammatik integrieren. Message-Queues können beispielsweise in Modellen genutzt werden, in denen nebenläufige Prozesse miteinander kommunizieren. Das folgende Listing zeigt die Umsetzung von *DVFSignalDeclare* mit Xtext:

```
1 DVFSignalDeclare:
```

```
2   "signal" name=ID
3   "{"
4     (
5       sigfirst=DVFVariableDeclare
6       "," signext+=DVFVariableDeclare)*
7     )?
8   "}"
9 ;
```

Listing 4.11: Xtext-Grammatik für Signaldeklarationen

Jede Queue besteht aus einem Bezeichner und einer Parameterliste. Nachrichten werden mit dem FIFO-Prinzip (First In, First Out) gespeichert bzw. abgerufen. Die genaue Funktionsweise der FIFO-Queue wird in Abschnitt 2.1.2 in Abbildung 2.3 verdeutlicht. Das nächste Beispiel zeigt eine domänenspezifische Sprache, die mit dem DVF einen Message-Queue umsetzt:

```
1 signal queue{byte var1, bool var2};
```

Listing 4.12: Deklaration einer Message-Queue

Im obigen Listing 4.8 wird eine Message-Queue namens *queue* angelegt, auf der Nachrichten gespeichert werden, die aus einem Byte und einem Boolean bestehen. Queues können nur eine endliche Menge von Nachrichten speichern. Die Größe der Queues wird den Transformatoren des DVFs per Kommandozeile übergeben. Das Senden und Empfangen von Nachrichten erfolgt mit den Methoden *send()* bzw. *receive()*, deren Syntax und Semantik in Abschnitt 4.3.4 vorgestellt werden.

4.3.3 Expressions

Mit den vorangegangenen Produktionsregeln können Variablen deklariert und darin Werte gespeichert werden. Häufig ist es wünschenswert, einer Variable nicht nur eine Konstante, sondern das Ergebnis einer Berechnung zuzuweisen. Zu diesem Zweck gibt es in Hochsprachen wie Java [60], C [78], Python [120], Perl [63] oder Hugo/RT [98] sogenannte Expressions. Jede Expression beinhaltet ein oder mehrere Literale, deren Verknüpfung über Operatoren erfolgt. Die Expressions des DVF setzen das Konzept der Kurzschlussauswertung um. Sie ermöglicht das vorzeitige Abbrechen einer Auswertung, wenn der Teilausdruck das Ergebnis bereits eindeutig bestimmt. Die folgenden beiden Beispiele mit den drei Variablen a, b und c verdeutlichen die Expressions des DVF:

- $a = b + 5 * (c - 1)$ speichert in a das Ergebnis der mathematischen Expression $b + 5 * (c - 1)$.
- $b = a \mid 25$ verknüpft die Bits von a mit der Ganzzahl 25 und speichert das Ergebnis in b.

Damit Expressions in einer domänenspezifischen Sprache genutzt werden können, beinhaltet das DSL Verification Framework die Produktionsregel *DVFExpression*. Wenn *DVFExpression* vom DSL-Entwickler in eine Grammatik integriert wird, stehen in der domänenspezifischen Sprache mathematische Ausdrücke und verschiedene Operatoren zur Verfügung. Bei der Umsetzung der Produktionsregel *DVFExpression* im Rahmen des DVF müssen die folgenden Problemstellungen beachtet werden:

- In Abhängigkeit des verwendeteten Parsers (LR-Parser oder LL-Parser) darf in den Produktionsregeln entweder keine Links-, oder keine Rechts-Rekursion auftreten. Des Weiteren ist es von Vorteil, wenn sich in dem abgeleiteten Abstract Syntax Tree die Operatorpräzedenz wiederspiegelt [1].
- Neben binären Operatoren müssen auch unäre Operatoren zum Invertieren von Ausdrücken enthalten sein.
- In einem Ausdruck können nicht nur Konstanten und Bezeichner, sondern auch zusammengesetzte Elemente, wie die Methoden innerhalb eines Objekts referenziert werden.

Die folgenden Abschnitte stellen die Produktionsregel *DVFExpression* vor und zeigen, wie die angesprochenen Problemstellungen gelöst werden. Zum Abschluss verdeutlichen Beispiele die Syntax der DVF-Expressions.

Links-Rekursion und Operatorpräzedenz

Das nächste Listing zeigt die Umsetzung von *DVFExpression* mittels Xtext. Da Xtext auf ANTLR und somit einem LR-Parser basiert, ist nur Rechts-Rekursion zulässig [116]. Des Weiteren sind die Produktionsregeln so strukturiert, dass sich die Operatorpräzedenz in einem entsprechenden Abstract Syntax Tree wiederspiegelt:

```
1 DVFExpression:
2   term1=DVFConditionalAndExpression
3   (op1="||" rest1=UDLExpression)?
4 ;
5
6 DVFConditionalAndExpression:
7   term1=DVFInclusiveOrExpression
8   (rest1+=DVFConditionalAndExpressionLoop)*
9 ;
10 DVFConditionalAndExpressionLoop:
11   op1="&&" rest1=DVFInclusiveOrExpression
12 ;
13
```

```
14 DVFInclusiveOrExpression:
15   term1=DVFAndExpression
16   (rest1+=DVFInclusiveOrExpressionLoop)*
17 ;
18 DVFInclusiveOrExpressionLoop:
19   op1="|" rest1=UDLAndExpression
20 ;
21
22 DVFAndExpression: ...
23
24 DVFArithmetikExpressionMult:
25   term1=DVFUnaryExpression
26   (rest1+=DVFArithmetikExpressionMultLoop)*
```

Listing 4.13: Xtext-Grammatik für Expressions ohne Links-Rekursion

Zur besseren Übersicht ist im obigen Listing 4.13 die Grammatik nicht komplett dargestellt. Mit den einzelnen Produktionsregeln werden verschiedene Operatoren innerhalb eines Ausdrucks umgesetzt. Die Ableitungsreihenfolge der Regeln ist aufsteigend nach der Präzendenz der Operatoren sortiert. Dies hat den Vorteil, dass sich die Operatorpräzendenz bereits im AST wiederspiegelt und von einem Transformator leichter ausgewertet werden kann. Somit hat beispielsweise der Operator für das logische Oder eine höhere Präzedenz als der für das bitweise Oder.

Die Initialregel *DVFExpression* dient zum Implementieren des Operators für das logische Oder. Die rechte Seite der Produktionsregel kann rekursiv abgeleitet werden und so mehrere Oder-Operatoren konkatenieren. Die linke Seite von *DVFExpression* wird nach *DVFConditionalAndExpression* überführt. *DVFConditionalAndExpression* kann auf der rechten Seite in mehrere, verkettete logische Und-Operatoren transformiert werden. Die linke Seite wird hingegen nach *DVFInclusiveOrExpression* abgeleitet. Die Funktionalität von *DVFInclusiveOrExpression*, *DVFAndExpression*, usw. gestaltet sich analog zu *DVFConditionalAndExpression*. Alle Ableitungsregeln sind nur auf der rechten Seite rekursiv und können somit von einem LR-Parser verarbeitet werden. Die letzte Produktionsregel in Listing 4.13 heißt *DVFArithmetikExpressionMult* und stellt den binären Operator mit der höchsten Präzedenz dar. Sie wird nach *DVFUnaryExpression* transformiert, um die unären Operatoren umzusetzen.

Unäre Operatoren

Neben den binären Operatoren für Logisches-Oder, Exklusives-Oder, Addition, usw. unterstützt die imperativen Sprachkonstrukte des DVF auch zwei unäre Operatoren:

- „!“ negiert einen boolschen Ausdruck.

- „−“ negiert einen Integer- oder einen Byte-Wert.

Das folgende Listing zeigt, wie diese beiden Operatoren mit einer Xtext-Grammatik spezifiziert werden können:

```
1 DVFUnaryExpression:
2   (op=DVFUnaryOp)? rest2=DVFPrimary)
3 ;
4 DVFUnaryOp: "-" | "!"
5
6 DVFPrimary:
7   term=DVFTerm | expr=DVFParExpression
8 ;
9
10 DVFParExpression:
11   "(" expr=DVFExpression ")"
12 ;
```

Listing 4.14: Xtext-Grammatik für unäre Operatoren und Klammerung

DVFUnaryExpression kann einem zuvor abgeleiteten Ausdruck einen der beiden unären Operatoren als Prefix voranstellen. Im Anschluss erfolgt eine Transformation nach *DVFPrimary*, um in einen geklammerten Ausdruck oder nach *DVFTerm* überführt werden zu können.

Referenzierung von Literalen

Dieser Abschnitt stellt die Produktionsregel *DVFTerm* vor, mit der sich einzelne oder zusammengesetzte Literale in einer Expression ableiten lassen. Dazu wird *DVFTerm* entweder in einen statischen Wert, wie beispielsweise *true*, oder eine Referenz auf eine Variable bzw. Methode abgeleitet:

```
1 DVFTerm:
2   number=INT |
3   characters=STRING |
4   bool_true = "true" |
5   bool_false = "false" |
6   label = DVFLabel |
7   scalarquery = DVFScalarQuery |
8 ;
```

Listing 4.15: Xtext-Grammatik für Literale und Referenzen auf Variablen

Das obige Listing zeigt, dass *DVFTerm* in eine ganze Zahl, *true*, *false*, *DVFLabel* oder *DVFScalarQuery* abgeleitet werden kann. *DVFLabel* entspricht einer Referenz auf eine Variable bzw. Funktion, während *DVFScalarQuery* die Methoden zum Lesen bzw. Schreiben eines Scalarsets umsetzt:

```
1 DVFScalarQuery:
2   (
3     (op="writeOne" | op="writeAll" | op="exist" | op="all")
4     "(" scalar=[DVFScalarDeclare]
5     "." element=[DVFVariableDeclare] ")"
6   )
7   |
8   (ni="nextIndex" "(" scalar=[DVFScalarDeclare] ")" ";")
9 ;
```

Listing 4.16: Xtext-Grammatik für die Abfrage von Scalarsets

Um auf ein Scalarset lesend bzw. schreibend zuzugreifen, werden die Schlüsselwörter *writeOne*, *writeAll*, *all*, *exist* und *nextIndex* genutzt. Sie sind notwendig, da der direkte Zugriff auf Scalarsets die symmetrischen Eigenschaften aufheben kann (vgl. Abschnitt 2.1.4). Jede der Operationen besteht aus einem Scalarset-Bezeichner und dem Namen des Elements, auf das zugegriffen werden soll. Mit *writeOne*, *writeAll* und *nextIndex* erfolgt der Schreibzugriff. Für Leseoperationen stehen *exist* und *all* zur Verfügung. Ihre genaue Verwendung bzw. den Grund ihrer Integration in das DVF zeigt der nächste Abschnitt im Rahmen einiger Beispiele.

Gemäß Listing 4.15 kann die Produktionsregel *DVFTerm* nicht nur nach *DVFScalarQuery*, sondern auch nach *DVFLabel* abgeleitet werden. Die Umsetzung der Produktionsregel *DVFLabel* veranschaulicht das folgende Beispiel:

```
1 DVFLabel:
2   label=[DVFIdentifier] ("[" expr=DVFExpression "]")?
3   ("." label2+=DVFLabelMore)*
4   (openbracket="(" (params=DVFParameter)? ")" )?
5 ;
6
7 DVFParameter:
8   param1=DVFExpression ( "," param2+=DVFExpression)*
9 ;
10 DVFLabelMore:
11   label=[DVFIdentifier] ("[" expr=DVFExpression "]")?
12 ;
```

Listing 4.17: Xtext-Grammatik für Referenzen auf Variablen und Methoden

DVFLabel ermöglicht das Referenzieren von Methodenaufrufen, Variablen, Arrays oder Elementen innerhalb von Objekten. Dabei ist zu beachten, dass sowohl Variablen, als auch Methoden in Objekten enthalten sein können, die wiederum Teil eines weiteren Objekts sind. *DVFLabel* besteht aus einem Label und eckigen Klammern, falls es sich um ein Array handelt. Darauf folgt eine Menge von Ableitungen nach *DVFLabelMore*, um mehrere Punkt-Operatoren zu konkatenieren. Am Ende von *DVFLabel* sind im Falle eines Methodenaufrufs runde Klammern zulässig. Die Klammern enthalten eine Menge von Expressions, die die übergebenen Parameter repräsentieren.

Anwendungsbeispiele

Dieser Abschnitt verdeutlicht die sprachlichen Eigenschaften von *DVFExpression* anhand konkreter Beispiele. Das folgende Listing zeigt ein Modell, das mit einer domänenspezifischen Sprache beschrieben ist, deren Grammatik die DVF-Produktionsregeln *DVFExpresssion* und *DVFVariableDeclare* enthält:

```
1 byte a = 1;
2 byte b = 2;
3 a = a + b * (a/2-(2*2));
```

Listing 4.18: Arithmetische Expression

Im obigen Beispiel werden zwei Variablen deklariert. Danach wird a mit einer arithmetischen Operation ein neuer Wert zugewiesen. Neben +, -, * und / werden mit *DVFExpression* die folgenden Operatoren unterstützt:

- Beliebige Klammerung zur Beeinflussung der Operatorpräzedenz.
- Die Operatoren +,-,* und / für Addition, Substraktion, Multiplikation und Division.
- Logisches Und/Oder mittels && und ||.
- Bitweises Und/Oder mittels & und |.
- Zuweisungen mittels =.
- Vergleich mittels ==, ! =, <, >, <=, >=.
- Negation mittels !.

Mit den Expressions des DVFs können, wie Regel *DVFLabel* zeigt, auch Variablen und Funktionen referenziert werden, die Teil einer Klasse bzw. Aufzählung sind. Dies verdeutlicht das folgende Beispiel. Es zeigt eine DSL, deren Grammatik die Produktionsregeln des DVF zur Umsetzung von Klassen, Statements und Expressions enthält:

```
1 class Buffer10{
2   byte index;
3   byte content[10];
4   bool isEmpty(){ /* ... */ }
5 }
6 Buffer10 b;
7
8 if(b.isEmpty() || b.index <=5) //...
```

Listing 4.19: Referenzieren von Variablen in Objekten

Im obigen Listing ist eine Expression in einem If-Block zu sehen. Die Anwendung von If-Blöcken, Methoden und Klassen wird im Detail im weiteren Verlauf dieser Arbeit in den entsprechenden Unterkapiteln beschrieben. Die Expression zeigt, neben den bereits vorgestellten Operatoren, dass der Operator . eingesetzt werden kann, um auf Methoden oder Attribute zuzugreifen, die Teil eines Objekts sind.

Das nächste Beispiel verdeutlicht die Verwendung von *writeOne()*, *writeAll()*, *exist()*, *all()* und *nextIndex*, die ein Teil der Produktionsregel *DVFScalarQuery* sind. Sie finden Verwendung, um auf Scalarsets zuzugreifen. Scalarsets bieten den Vorteil, dass der Zustandsraum eines Modells unter der Ausnutzung von Symmetrie äußerst effektiv verkleinert werden kann. Scalarsets können jedoch nur verwendet werden, wenn im Modell die Permutation eines Arrays nicht relevant ist. Das bedeutet, der folgende Zugriff auf das Scalarset *scalar1* ist nicht zulässig, da die Reihenfolge der Elemente das Ergebnis beeinflusst:

```
1 scalarset scalar1[3]{
2   byte a;
3 }
4
5 if(scalar1.a[0] > scalar1.a[1]){ ... }
```

Listing 4.20: Ungültiger Zugriff auf ein Scalarset

Das obige Listing 4.20 deklariert ein Scalarset. In Zeile 5 werden zwei Elemente miteinander verglichen. Da somit die Reihenfolge der Array-Elemente die Semantik des Modells beeinflusst, ist der Zugriff nicht zulässig. Deshalb darf im Rahmen der Expressions des DVF nicht direkt auf ein Scalarset zugegriffen werden. Stattdessen stehen verschiedene Operationen zur Verfügung, die allesamt die Symmetrie des Arrays nicht aufheben [41] [109] und mit denen der Inhalt von Scalarsets abgefragt bzw. verändert werden kann:

- *All()* entspricht dem Allquantor [82] und liefert *true* zurück, wenn alle Elemente eines symmetrischen Arrays eine bestimmte Eigenschaft erfüllen.
- *Exist()* entspricht dem Existenzquantor [82] und liefert *true* zurück, wenn mindestens ein Element eines symmetrischen Arrays eine bestimmte Eigenschaft erfüllt.

- *WriteAll()* weist allen Elementen eines symmetrischen Arrays einen bestimmten Wert zu.

- *WriteOne()* weist einem zufälligen Element eines symmetrischen Arrays einen bestimmten Wert zu. Ein mehrmaliger Aufruf von *writeOne()* greift immer auf das selbe Element des Arrays zu. Um ein neues zufälliges Element auszuwählen, muss *nextIndex()* aufgerufen werden.

Die Anwendung der Scalarset-Operationen verdeutlicht das folgende Beispiel:

```
1 scalarset scalar1[2]{
2   byte a;
3 }
4
5 if(all(scalar.a) == 0){
6   writeAll(scalar.a) = 10;
7   writeOne(scalar.a) = 1;
8 }
9 nextIndex(scalar1);
10 writeOne(scalar.a) = 3;
```

Listing 4.21: Zugriff auf Scalarsets

Im obigen Listing wird das symmetrische Array *scalar1* deklariert, das aus zwei Bytes besteht. Direkte Zugriffe sind nicht zulässig, da sie die Symmetrie des Arrays aufheben können. Stattdessen müssen zum Lesen oder Schreiben die Sprachkonstrukte *writeOne()*, *writeAll()*, *exist* bzw. *all()* aufgerufen werden. In Zeile 5 liefert der Ausdruck *all()==0* den Wert *true* zurück, wenn alle Elemente in dem entsprechenden Scalarset den Wert 0 haben. Analog dazu kann *writeAll()* in Zeile 6 genutzt werden, um allen Elementen des symmetrischen Arrays a den Wert 10 zuzuweisen.

Im Gegensatz dazu greift *writeOne()* in Zeile 7 nur auf ein Element des symmetrischen Arrays a zu. Dabei ist anzumerken, dass implizit für jedes Scalarset eine Variable zum Indizieren angelegt wird. Diese Variable wird mit einem Zufallswert initialisiert, ist für den Anwender nicht sichtbar und kann auch nicht direkt gelesen und beschrieben werden. Stattdessen wird die Indexvariable von *writeOne()* genutzt, um auf ein Element des Scalarsets zuzugreifen. Das bedeutet, dass in Listing 4.21 in Zeile 7 einem zufälligen Element des Arrays a der Wert 1 zugewiesen wird. Wenn der Indexvariable ein neuer Zufallswert zugewiesen werden soll, muss die Methode *nextIndex()* aufgerufen werden. Dies geschieht in Listing 4.21 in den Zeilen 9 bis 10. Das Ergebnis ist, dass zwei zufälligen Elementen a_i und a_j mit $i \neq j$ die Werte 1 bzw. 3 zugewiesen werden.

4.3.4 Statements

Neben Variablen und Expressions sind die sogenannten Statements ein wichtiges Sprachkonstrukt, um das Verhalten eines Modells zu beschreiben. Sie sind Teil der Hochsprachen wie Java [60], C [78], Python [120] oder Perl [63]. Statements stellen eine Menge von Anweisungen dar, die während der Abarbeitung eines Programms ausgeführt werden. Das DSL Verification Framework stellt daher die Produktionsregel *DVFStatement* zur Verfügung. Ein DSL-Entwickler sie in seine Xtext-Grammatik integrieren, um die domänenspezifische Sprache mit Statements zu ergänzen. *DVFStatement* ermöglicht in einer DSL das Beschreiben der folgenden Aspekte:

- Steuerung des Kontrollflusses mittels bedingter Ausführung, Schleifen und dem Aufrufen bzw. Verlassen von Funktionen.
- Formulieren von Anforderungen an das Modell.
- Asynchrone Kommunikation von nebenläufigen Prozessen.

Im weiteren Verkauf dieses Abschnitts wird die Umsetzung der Produktionsregel *DVFStatement* genauer vorgestellt:

```
1 DVFStatement:
2   (assign=DVFAssignmentOrCall) | (assert=DVFAssert) |
3   (send=DVFSendSignal) | (read=DVFReadSignal) |
4   (ret=DVFReturn) | (loop=DVFLoop) | (ifelse=DVFIf) |
5   (wait=DVFWait) | (sync=DVFSync)
6 ;
```

Listing 4.22: Xtext-Grammatik für Statements

Das obige Listing 4.22 zeigt, dass *DVFStatement* in verschiedene Produktionsregeln abgeleitet werden kann, von denen jede ein bestimmtes Sprachkonstrukt repräsentiert. Sie haben die folgende Semantik:

- *DVFAssignmentOrCall*: Eine Zuweisung, um in einer Variable einen Wert zu Speichern oder ein Funktionsaufruf, um eine Methode auszuführen.
- *DVFAssert*: Eine Assert-Anweisung, um Anforderungen an das Modell zu formulieren.
- *DVFSendSignal*: Der Versand einer Nachricht über eine Message-Queue.
- *DVFReadSignal*: Das Empfangen einer Nachricht von einer Message-Queue.
- *DVFReturn*: Ein Return-Statement, das das Verlassen einer Methode bewirkt.

- *DVFLoop*: Wiederholung von Anweisungen durch Do-Schleifen.
- *DVFIf*: Bedingte Ausführung durch If-Blöcke.
- *DVFWait*: Blockierende Wait-Anweisungen, die die Ausführung erst dann fortsetzen, wenn eine bestimmte Bedingung erfüllt ist.
- *DVFSync*: Synchronisierte Umgebungen, die nur von einem nebenläufigem Prozess gleichzeitig betreten werden können.

Die entsprechenden Produktionsregeln werden im weiteren Verlauf dieses Abschnitts genauer vorgestellt.

Zuweisungen und Funktionsaufrufe

Jedes Statement kann eine Zuweisung oder ein Funktionsaufruf sein. Zuweisungen sind notwendig, um in einer mit *DVFVariableDeclare* deklarierten Variable das Ergebnis einer Expression zu speichern. Funktionsaufrufe springen in eine Methode und setzen dort die Ausführung des Modells fort. Die Produktionsregel zur Umsetzung dieser beiden Sprachkonstrukte hat den folgenden Aufbau:

```
1 DVFAssignmentOrCall:
2   (term1=DVFLabel (op"=" expr=DVFExpression)? ";")
3 ;
```

Listing 4.23: Xtext-Grammatik für Zuweisungen und Funktionsaufrufe

Jede Zuweisung besteht aus einem Bezeichner, dem Zuweisungsoperator und einer Expression. Die Anwendung in domänenspezifischen Sprachen wird in den Listings 4.18 und 4.21 demonstriert. Die Regel *DVFAssignmentOrCall* kann auch in einen einzelnen Funktionsaufruf abgeleitet werden, der keinen Zuweisungsoperator und keine Expression aufweist. Ein Funktionsaufruf entspricht einer Transformation in die Produktionsregel *DVFLabel*, deren Aufbau im Rahmen von Listing 4.17 erklärt wird.

Anforderungen mit Assert-Statements

Neben Zuweisungen sind Assert-Anweisungen ein wichtiger Aspekt im DVF, da mit ihnen Anforderungen an ein Modell beschrieben werden können. Nach der Transformation in eine Model Checker-Eingabesprache überprüft der Model Checker automatisiert, ob das Modell alle Anforderungen erfüllt. Wenn nicht alle Anforderungen erfüllbar sind, wird typischerweise ein Fehlerpfad erzeugt und der DSL-Anwender muss sein Modell modifizieren. Assert-Statements werden mit der Produktionsregel *DVFAssert* umgesetzt, die den folgenden Aufbau hat:

```
DVFAssert:
    "assert" "(" expr=DVFExpression ")" ";"
;
```

Listing 4.24: Xtext-Grammatik für Assert-Anweisungen

Jede Assert-Anweisung wird durch das Schlüsselwort *assert* eingeleitet und beinhaltet eine Expression. Wenn die Expression den Wert *false* annimmt, meldet der Model Checker einen Fehler. Das folgende Beispiel demonstriert die Verwendung von *assert* im Rahmen einer domänenspezifischen Sprache:

```
byte b = 6;
assert(b <= 5);
```

Listing 4.25: Beispiel für ein Assert-Statement

Im obigen Listing 4.25 wird die Variable b deklariert und mit dem Wert 6 initialisiert. Darauf folgt eine Assert-Anweisung. Wenn der Model Checker ein derartiges Modell verifiziert, wird ein Fehler gemeldet, da $b < 5$ den Wert *false* annimmt.

Senden und Empfangen von asynchronen Signalen

Neben Anforderungen können mit einem *DVFStatement* auch Signale in domänenspezifischen Sprachen genutzt werden. Signale dienen zur Kommunikation von nebenläufigen Prozessen und werden auf sogenannten Message-Queues gespeichert bzw. abgerufen. Die Deklaration einer Message-Queue wird in Listing 4.11 verdeutlicht. Die Umsetzung der Produktionsregel, um auf einer Queue ein Signal abzuspeichern, hat den folgenden Aufbau:

```
DVFSendSignal:
  "send" "("
     dest=[DVFSignalDeclare] ("," param+=DVFExpression)*
  ")" ";"
;
```

Listing 4.26: Xtext-Grammatik für den Versand von Signalen

Jede Sendeoperation wird mit dem Schlüsselwort *send* eingeleitet und ähnelt einem Methodenaufruf. Bei dem ersten übergebenen Parameter handelt es sich um die Ziel-Queue. Darauf folgen Expressions, die die Elemente repräsentieren, aus denen sich das Signal zusammensetzt. Das folgende Beispiel verdeutlicht die genaue Anwendung der Sende-Operation:

```
signal queue{byte var1, bool var2};
send(queue, 1, false);
```

Listing 4.27: Beispiel für den Versand eines Signals

Im obigen Listing ist ein Modell zu sehen, in der eine Message-Queue namens *queue* deklariert wird. Ein Statement ruft *send()* auf und speichert in der Queue die Nachricht $(1, false)$. Sende-Operationen blockieren, wenn auf der Queue keine weitere Nachricht gespeichert werden kann. In diesem Fall wird die Ausführung des Modells erst fortgesetzt, wenn durch den Abruf eines Signals wieder Speicherplatz zur Verfügung steht. Deshalb ist es neben dem Senden von Signalen auch wichtig, Signale abrufen zu können. Die nächste Produktionsregel setzt die dafür erforderliche Receive-Anweisung um:

```
1 DVFReadSignal:
2   "receive" "(" target=DVFLabel ")" ";"
3 ;
```

Listing 4.28: Xtext-Grammatik für das Empfangen von Signalen

Die Receive-Anweisung bekommt den Namen einer Message-Queue übergeben, von der eine Nachricht abgerufen werden soll. Wenn die Queue leer ist, blockiert die Anweisung so lange, bis eine Nachricht vorliegt. Das folgende Beispiel verdeutlicht die Anwendung:

```
1 signal queue{byte var1, bool var2};
2 //...
3 recv(queue);
4 if(queue.var1==1){
5   //...
6 }
```

Listing 4.29: Beispiel für den Empfang eines Signals

Im obigen Listing wird eine Message-Queue deklariert, die asynchrone Signale vom Typ $(byte, bool)$ empfängt. In Zeile 3 wird eine Nachricht abgerufen. Zeile 4 zeigt, dass die Elemente der Nachricht über die in der Signaldeklaration spezifizierten Bezeichner abgerufen werden können.

Verlassen einer Methode

Neben dem Senden und Empfangen von Nachrichten sind auch Funktionen ein elementarer Bestandteil des DSL Verification Frameworks. Das DVF stellt deshalb eine Produktionsregel zur Verfügung, die das Deklarieren von Methoden innerhalb einer domänenspezifischen Sprache ermöglicht und die im Detail in Abschnitt 4.3.5 vorgestellt wird. Analog dazu stellt das DVF auch eine Return-Anweisung zur Verfügung, die die Ausführung einer Methode beendet. Das nächste Listing zeigt den Aufbau der Produktionsregel für die Umsetzung eines Return-Statements:

```
1 DVFReturn:
2   "return" (expr=DVFExpression)? ";";
```

Listing 4.30: Xtext-Grammatik für Return-Statements

Das Return-Statement darf nur von Methoden verwendet werden und kann optional eine Expression enthalten, die den Rückgabewert der Methode enthält.

Schleifen

Das DSL Verification Framework bietet neben Funktionen auch die Möglichkeit, den Kontrollfluss eines Modells mit Schleifen zu steuern. Zu diesem Zweck lassen sich mit DVF-Statements sogenannte While-Schleifen umsetzen. Bei der Betrachtung von Hochsprachen, wie beispielsweise C oder Java, fällt auf, dass es neben While-Schleifen auch For- bzw. Do-While-Schleifen gibt. Für den Sprachumfang des DVF wird die While-Schleife gewählt, da For- bzw. Do-While-Schleifen leicht mit While-Schleifen umgesetzt werden können. Das nächste Listing zeigt, wie While-Schleifen mit einer Xtext-Grammatik im Rahmen des DSL Verification Frameworks umgesetzt sind:

```
1 DVFLoop:
2   "while" "(" expr=DVFExpression ")" "{"
3     (stat+=DVFStatement)*
4   "}"
5 ;
```

Listing 4.31: Xtext-Grammatik für Schleifen

Jede Schleife besteht aus dem Schlüsselwort *while*, einer Expression als Schleifenkopf und einer Menge von Statements im Schleifenkörper. Mit jedem Schleifendurchlauf werden die Statements ausgeführt und anschließend in Abhängigkeit des Rückgabewerts der Expression die Schleife entweder beendet oder der Schleifenkörper erneut ausgeführt.

Das folgende Beispiel demonstriert die Verwendung von While-Schleifen in einer domänenspezifischen Sprache:

```
1 int a = 0;
2
3 while(a<10){
4   a=a+1;
5 }
```

Listing 4.32: Anwendung einer Schleife

Das obige Listing zeigt ein Modell, das aus einer Variable und einer While-Schleife besteht. Die Schleife inkrementiert mit jedem Durchlauf den Zähler und terminiert, wenn a den Wert 10 annimmt.

If-Blöcke

Neben Schleifen können If-Blöcke für die bedingte Ausführung in Modellen Verwendung finden. Deshalb lassen sich mit den Statements des DVF auch If-Blöcke umsetzen. Die

dafür notwendigen Produktionsregeln werden mit dem nächsten Listing vorgestellt:

```
DVFIf:
  "if" "(" expr=DVFExpression ")" "{" (stat+=DVFStatement)+ "}"
  (elzeif+=DVFElseIf)*
  (elze=DVFElse)?
;

DVFElseIf:
  "elseif" "(" expr=DVFExpression ")"
  "{" (stat+=DVFStatement)+ "}"
;

DVFElse:
  "else" "{" (stat+=DVFStatement)+ "}"
;
```

Listing 4.33: Xtext-Grammatik für If-Blöcke

Jede Verzweigung besteht aus mindestens einem If-Block, optional mehreren Else-If-Blöcken und abschließend einem Else-Block. Sowohl If-, als auch Else-If-Blöcke bestehen aus einer Expression und einer Menge von Statements. Die jeweiligen Statements werden ausgeführt, wenn die entsprechende Expression den Wert *true* annimmt. Das folgende Beispiel verdeutlicht die Verwendung von If-Blöcken:

```
int a = 0;

if (a==0){
  a=5;
}
else if (a==-1){
  a = 6;
}
else{
  a=7;
}
```

Listing 4.34: Bedingte Ausführung

Das Modell im obigen Listing deklariert die Variable *a*. Im Anschluss wird *a* in Abhängigkeit des aktuellen Werts entweder 5, 6 oder 7 zugewiesen.

Wait-Blöcke

Neben der bedingten Ausführung durch If-Statements, gibt es im DVF auch die Möglichkeit blockierende Wait-Blöcke umzusetzen. Wait-Blöcke können in Modellen eingesetzt werden, die auf bestimmte Ereignisse reagieren, wie beispielsweise den Erhalt einer asynchronen Nachricht. Das nächste Listing zeigt den Aufbau der Xtext-Produktionsregeln, zur Umsetzung von Wait-Blöcken in DSLs:

```
1 DVFWait:
2   "wait" "(" expr=DVFExpression ")" "{" (stat+=DVFStatement)* "}"
3   (morewait+=DVFMoreWait)*
4 ;
5
6 DVFMoreWait:
7   "elsewait" "(" expr=DVFExpression ")"
8   "{" (stat+=DVFStatement)+ "}"
9 ;
```

Listing 4.35: Xtext-Grammatik für Wait-Blöcke

Wait-Blöcke ähneln If-Blöcken. Der Unterschied ist ein blockierendes Verhalten: Wenn keine Expression den Wert *true* annimmt, wird die Ausführung des Modells unterbrochen und erst dann fortgesetzt, wenn eine der Expressions von $false$ auf *true* wechselt. Ein Wait-Block wird beispielsweise in Listing 4.3 genutzt, um die Ausführung einer Transition in einem endlichen Automaten umzusetzen. Das nächste Listing zeigt die Anwendung von Wait-Blöcken im Rahmen einer domänenspezifischen Sprache:

```
1 int a = 0;
2 int b = 0;
3
4 wait(a==0){
5   b=5;
6 }
7 elsewait(a==1){
8   b=6;
9 }
```

Listing 4.36: Blockierende Statements in einer DSL

Im obigen Beispiel ist eine DSL zu sehen, die mit dem DVF umgesetzt ist und Statements bzw. Variablen nutzt. Durch den Wait-Block wartet das Modell, bis a entweder den Wert 0, oder den Wert 1 annimmt. In dem Fall wird der Variable b der Wert 5 oder 6 zugewiesen und anschließend die Ausführung des Modells fortgesetzt.

Synchronize-Umgebungen

Der letzte Abschnitt hat die blockierenden Wait-Blöcke vorgestellt. Des Weiteren ermöglichen die Statements des DVF auch das Modellieren sogenannter Synchronize-Umgebungen. Sie haben die folgende Semantik: Gegeben sei ein Prozess, von dem zwei nebenläufige Instanzen p_0 und p_1 ausgeführt werden. Beide Instanzen greifen mit dem Statement s_i auf eine globale Variable namens a zu. Um Inkonsistenzen im Modell zu vermeiden, kann s_i in eine Synchronize-Umgebung eingebettet werden. Synchronize-Umgebungen haben die besondere Eigenschaft, dass sie immer nur ein Prozess zu einem beliebigem Zeitpunkt t betreten kann. Das bedeutet, wenn p_0 das Statement s_i ausführt, muss p_1 warten, bis p_0 die Synchronize-Umgebung wieder verlassen hat. Synchronize-Umgebungen können mit Xtext wie folgt beschrieben werden:

```
1 DVFSync:
2   "sync" "{" (stat+=DVFStatement)+ "}"
3 ;
```

Listing 4.37: Xtext-Grammatik für Synchronize-Umgebungen

Das obige Listing 4.37 zeigt, dass Synchronize-Umgebungen durch das Schlüsselwort *sync* eingeleitet werden. Darauf folgt eine Menge von Statements. Das nächste Beispiel verdeutlicht die Verwendung von Synchronize-Umgebungen im Rahmen einer domänenspezifischen Sprache:

```
1 int a = 0;
2
3 //Concurrent Process
4 sync{
5   if(a==0){
6     a=a+1;
7   }
8 }
```

Listing 4.38: Synchronize-Umgebungen in einer DSL

Das obige Listing 4.38 enthält die globale Variable a. Des Weiteren sind in Zeile 4 bis 8 Statements zu sehen, die Teil eines nebenläufigen Prozesses sind. Durch die Synchronize-Umgebung kann immer nur ein Prozess den darin enthaltenen If-Block ausführen.

4.3.5 Methoden

Neben Variablen, Expressions und Statements kann ein DSL-Entwickler auch Funktionen in seine DSL-Projekte integrieren. Zu diesem Zweck stellt das DSL Verification Framework die Ableitungsregel *DVFFunctionDeclare* zur Verfügung. Wenn ein DSL-Entwickler *DVFFunctionDeclare* in die Grammatik seiner DSL integriert, können in der

domänenspezifischen Sprache Methoden deklariert werden. Die Umsetzung mit Xtext gestaltet sich wie folgt:

```
1  DVFFunctionDeclare:
2    (abstraction="abstraction")? (
3      dynret=[DVFEnumDeclare] | statret=DVFVariableType |
4      voidret="void" | numret=DVFRange
5    )
6    name=ID "("
7      ((sigelementfirst=DVFVariableDeclare)
8      ("," sigelement+=DVFVariableDeclare)*)?
9    ")"
10   "{"
11     (statements+=DVFStatement)*
12   "}"
13 ;
14
15 DVFRange:
16   "range" "(" from=INT "," to=INT ")"
17 ;
```

Listing 4.39: Xtext-Grammatik für Methoden

Der Aufbau einer Funktionsdeklaration gestaltet sich ähnlich wie in den Hochsprachen C oder Java und beinhaltet die folgenden Elemente:

- Der Methodenkopf wird in Listing 4.39 in den Zeilen 2 bis 9 beschrieben. Er besteht aus einem Rückgabewert, einem eindeutigen Bezeichner und einer Liste von Parametern.
- Der Methodenkörper wird in Listing 4.39 in den Zeilen 10 bis 12 beschrieben. Er besteht aus einer Liste von Statements, die nach dem Aufruf der Funktion ausgeführt werden.

Als Rückgabewerte sind Aufzählungen und Variablen vom Typ Bool, Byte, Int oder String zulässig. Funktionsparameter können eine Klasse, eine Aufzählung, ein Bool, ein Byte, ein Int oder ein String sein. Die Umsetzung der Funktionsparameter geschieht in Listing 4.39 in den Zeilen 7 und 8 mit der Produktionsregel *DVFVariableDeclare* zur Deklaration von Variablen. Für Funktionsparamater gilt:

- Wenn Bool, Byte, Int oder String einer Funktion als Parameter übergeben werden, gilt Value-Semantik [60].
- Wenn eine Klasse einer Funktion als Parameter übergeben wird, gilt Referenz-Semantik [60].

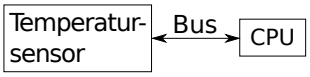

Abbildung 4.7: Kommunikation einer CPU mit einem Sensor

Das folgende Beispiel verdeutlicht die Anwendung der Produktionsregel *DVFFunction-Declare* in einer domänenspezifischen Sprache. Es zeigt ein Modell, in dem durch das DVF Variablen, Methoden und Statements deklariert werden können:

```
byte result;
byte add(byte p1, byte p2){
  return p1 + p2;
}
result = add(1,2);
```

Listing 4.40: Nutzung einer Methode gemäß entsprechend Listing 4.39

Das obige Listing 4.40 deklariert die Methode *add()* und die Variable *result*. Im Anschluss wird *add()* aufgerufen und das Ergebnis der Variable *result* zugewiesen.

Zwei Elemente aus Listing 4.39 sind bisher noch nicht genauer erläutert worden: Das Schlüsselwort *abstraction*, das jeder Methodendeklaration vorangestellt werden kann und der Rückgabetyp *range*. Beide Sprachkonstrukte dienen zur Umsetzung von eigenschaftserhaltender Abstraktion, die in den verwandten Arbeiten [11] [30] [145] [146] als Ansatz zur Optimierung auf Modellebene genutzt wird. Die Anwendung von eigenschaftserhaltender Abstraktion bzw. der beiden Sprachkonstrukte im Rahmen einer domänenspezifischen Sprache wird mit dem folgenden Beispiel verdeutlicht: Abbildung 4.7 zeigt eine CPU, die über einen nicht näher spezifizierten Bus mit einem Temperatursensor kommuniziert. Für die CPU ist nicht die genaue Temperatur relevant, sondern lediglich, ob der Sensor einen Wert über 50° Celsius misst. Die CPU führt ein Programm aus, das mit einer nicht näher spezifizierten DSL umgesetzt ist und eine Methode enthält, die vom Sensor den Temperaturwert abfragt. Das nächste Listing zeigt die Umsetzung des Modells mit einer beispielhaften domänenspezifischen Sprache:

```
enum SensorValue{
  OK, ALERT
}
SensorValue result;

//...
abstraction SensorValue readSensor(){}

//...
```

```
10 result = readSensor();
11 if(result==SensorValue.ALERT){
12   //...
13 }
```

Listing 4.41: Beispiel für eigenschaftserhaltende Abstraktion

Das Modell im obigen Beispiel nutzt Aufzählungen, Variablendeklarationen, Methoden, Statements und Expressions. Zunächst wird eine Aufzählung deklariert, die die Werte OK und ALERT enthält. OK steht für einen Temperaturwert, der bis 50° Celsius reicht. ALERT bedeutet hingegen, dass der Sensor einen Temperaturwert gemessen hat, der 50° Celsius übersteigt. Die Methode *readSensor()* greift über den Bus auf den Sensor zu und liefert in Abhängigkeit der ausgelesenen Temperatur entweder OK oder ALERT zurück.

Sowohl das Bus-System, als auch der Temperatursensor werden nicht mit der domänenspezifischen Sprache beschrieben. Trotzdem soll der Model Checker verifizieren, wie sich das System in Abhängigkeit der Sensorwerte verhält. Deshalb wird die Methode *readSensor()* mit dem Schlüsselwort *abstraction* umgesetzt, was zu eigenschaftserhaltender Abstraktion führt. Diese sogenannte Abstraction-Methode hat nur einen Kopf und keinen Körper. Die Transformation in die Hoch- bzw. Model Checker-Eingabesprache gestaltet sich wie folgt:

- Die Methode *readSensor()* wird mit einem leeren Methodenkörper nach Java überführt. Der Anwender muss die Methode implementieren, damit sie auf den Sensor zugreift.
- Der Transformator nach Promela bewirkt, dass die Methode einen nicht-deterministischen If-Block enthält. Der If-Block gibt entweder den Wert OK oder den Wert ALERT zurück.

Durch diese Art der Modellierung kann der Model Checker das Verhalten des Modells in Abhängigkeit der Temperatur untersuchen, ohne dass eine reale Anbindung an einen Sensor existiert.

Listing 4.41 zeigt eine Abstraction-Methode, deren Rückgabetyp eine Aufzählung ist. In einigen Fällen ist es wünschenswert, neben Aufzählungen auch ganzzahlige Werte zurückzugeben. In diesem Fall kann das Sprachkonstrukt *range* genutzt werden, um den Wertebereich einer Abstraction-Methode für ganzzahlige Werte zu definieren. So gibt beispielsweise eine Abstraction-Methode mit dem Rückgabewert $range(0, 2)$ entweder 0, 1 oder 2 zurück.

Die Nutzung von eigenschaftserhaltender Abstraktion in domänenspezifischen Sprachen kann zu den folgenden Vorteilen führen: Durch Abstraktion einzelner Komponenten wird der Zustandsraum eines Modells reduziert und so dem Risiko der State Space Explosion entgegen gewirkt. Des Weiteren können Schnittstellen zu Elementen spezifiziert

werden, die sich nicht mit der vom DSL-Entwickler entworfenen domänenspezifischen Sprache umsetzen lassen. Die eigenschaftserhaltene Abstraktion ermöglicht es in diesem Fall, das Verhalten des Systems in Abhängigkeit der entsprechenden Komponenten zu verifizieren, ohne dass sie Teil des Modells sind.

4.3.6 Klassen

Das DSL Verification Framework enthält auch eine Produktionsregel, die zur Modellierung von Klassen genutzt werden kann. Dies hat den Vorteil, dass ein DSL-Entwickler auch objektorientierte Konzepte in seiner domänenspezifischen Sprache umsetzen kann. Die Produktionsregel hat den folgenden Aufbau:

```
1 DVFClass:
2   "class" name=ID "{"
3     (variables+=DVFVariableDeclare ";")*
4     (functions+=DVFFunctionDeclare)*
5   "}"
6 ;
```

Listing 4.42: Xtext-Grammatik einer Klasse

Ähnlich wie in Java besteht jede Klasse aus einem eindeutigen Bezeichner und beinhaltet eine Menge von Attributen und Verhalten. Die Variablen und Methoden werden mit den bereits vorgestellten Produktionsregeln *DVFVariableDeclare* und *DVFFunctionDeclare* umgesetzt. Identität und Gleichheit von Objekten sind äquivalent zu Java definiert: Der Vergleichsoperator überprüft die Identität von zwei Objekten. Zum Abfragen von Gleichheit muss eine Methode wie beispielsweise *equals()* implementiert werden. Das nächste Beispiel verdeutlicht die Nutzung von *DVFClass* in einer nicht näher spezifizierten domänenspezifischen Sprache:

```
1 class Counter{
2   int value = 0;
3
4   int inc(){
5     value=value+1;
6     return value;
7   }
8 }
9 Counter counter;
10 int result;
11 //...
12 result = counter.inc();
```

Listing 4.43: Klasse in einer DSL gemäß Grammatik 4.42

Die DSL im obigen Listing ermöglicht das Deklarieren von Klassen, Variablen und Zuweisungen. Die Klasse *Counter* enthält die lokale Variable *value* und die Methode *inc()*. *Inc()* erhöht *value* um eins und gibt den neuen Wert zurück. Ein Statement in Listing 4.43 ruft die Methode *inc()* auf und speichert das Ergebnis in *result*.

4.4 Control Flow Intermediate Language

Der vorangegangenen Abschnitt hat verschiedene Sprachkonstrukte vorgestellt, die dem DSL-Entwickler vom DVF in Form von Produktionsregeln zur Verfügung gestellt werden. Dazu gehören beispielsweise Produktionsregeln zum Deklarieren von Variablen, Methoden oder Expressions. Wenn der DSL-Entwickler eine Grammatik beschreibt und die daraus resultierende domänenspezifische Sprache Variablen beinhalten soll, dann können die dafür notwendigen Produktionsregeln aus dem DVF entnommen werden. Eine manuelle Implementierung entfällt. Durch dieses Vorgehen wird die Entwicklung von domänenspezifischen Sprachen beschleunigt.

Das DSL Verification Framework beinhaltet neben vorgefertigten Produktionsregeln auch zwei vorgefertigte Transformatoren. Der erste Transformator nimmt eine Übersetzung in die Hochsprache Java vor, während der zweite Transformator ein Modell nach Promela überführt. Beide Transformatoren unterstützen als Eingabe alle Sprachkonstrukte, die vom DVF als Produktionsregeln zur Verfügung gestellt werden. Das bedeutet, wenn ein Modell mit einer DSL beschrieben wird, die beispielsweise Statements aus dem DVF verwendet, dann können die beiden Transformatoren die entsprechenden Statements automatisiert nach Java und Promela überführen. Auch dieses Vorgehen hat den Vorteil, dass sich die Entwicklungszeiten von DSLs verkürzen, da die Transformatoren nicht mehr manuell implementiert werden müssen. Des Weiteren ist durch die automatische Transformation nach Promela weniger Expertenwissen im Bereich der formalen Verifikation notwendig.

Das Konzept des DVF führt jedoch zu einem Problem: Eine domänenspezifische Sprache besteht in der Regel nicht nur aus Sprachkonstrukten, die Teil des DVF sind. Auch benutzerspezifische Elemente können ein Teil der DSL sein. Sie werden von den Transformatoren des DVF nicht unterstützt. Um dieses Problem zu lösen, stellt das DSL Verification Framework die sogenannte Control Flow Intermediate Language (CFIL) zur Verfügung (vgl. Unterkapitel 4.1). Das Ziel dieses Abschnitts ist es, das Konzept und den Sprachumfang der CFIL genauer vorzustellen.

Bei der CFIL handelt es sich um Sprachkonstrukte zur Steuerung des Kontrollflusses. Die Transformatoren des DVF sind so konstruiert, dass sie nicht nur die vordefinierten Sprachkonstrukte aus Abschnitt 4.3, sondern auch die CFIL in die jeweiligen Zielsprachen überführen.

Abbildung 4.8 zeigt, wie die CFIL im Rahmen des DSL Verification Frameworks Verwendung findet, um die Verknüpfung von modellgetriebener Entwicklung und formaler

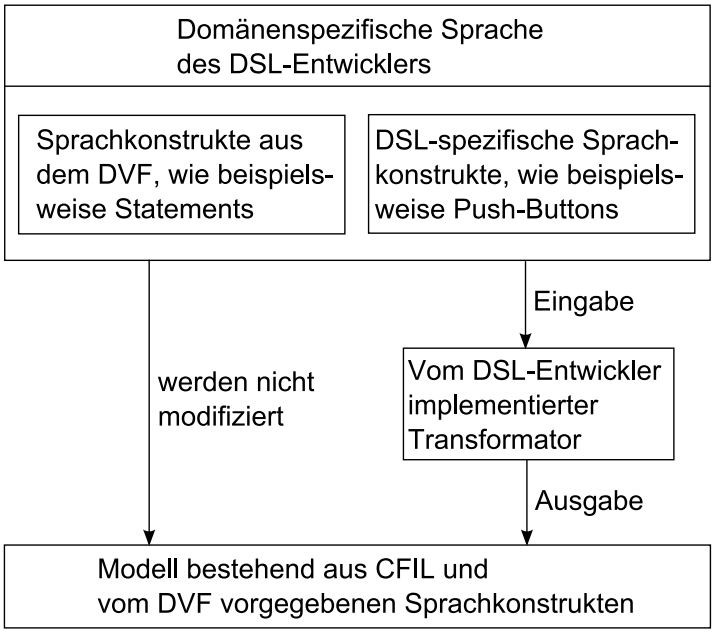

Abbildung 4.8: Funktionsweise der Control Flow Intermediate Language

Verifikation zu ermöglichen. Im oberen Drittel der Abbildung ist der Aufbau einer DSL zu sehen, die aus verschiedenen Sprachkonstrukten bzw. Produktionsregeln besteht. Diese können in zwei Kategorien eingeteilt werden: Sprachkonstrukte aus dem DVF und DSL-spezifische Sprachkonstrukte. Der DSL-Entwickler implementiert einen Transformator, der die DSL-spezifischen Elemente in die CFIL überführt. Alle Sprachkonstrukte, die auf den vordefinierten Produktionsregel basieren, werden hingegen nicht modifiziert.

Nach der Transformation in die CFIL besteht das Modell nur noch aus Sprachkonstrukten, die von den Transformatoren des DVF automatisiert nach Java und Promela überführt werden können. Dieses Vorgehen hat den Vorteil, dass ein DSL-Entwickler keine tiefergehenden Kenntnisse im Bereich der formalen Verifikation haben muss, um einen Transformator für seine DSL zu implementieren. Des Weiteren müssen nicht zwei Transformatoren angepasst werden, was das Risiko von Inkonsistenzen reduziert.

Der Sprachumfang der CFIL hat die folgende Struktur: Die Modelle einer domänenspezifischen Sprache können Nebenläufigkeit beinhalten. Beispiele hierfür sind DSLs zur Beschreibung von Bahnhöfen und Bahnschranken [22]. Auch Kommunikationsprotokolle von im Konvoi fahrenden PKWs [13] oder parallel ausgeführte endlichen Automaten [44] setzen Nebenläufigkeit um. Aus diesem Grund enthält die CFIL Sprachkonstrukte, mit denen nebenläufige Prozesse deklariert, gestartet und gestoppt werden können.

Die nebenläufigen Prozesse führen Statements aus. Daher beinhaltet die Control Flow

Intermediate Language auch Sprachkonstrukte zur Kapselung von Statements. Daraus ergibt sich für die CFIL der folgende Sprachumfang:

- *Threads* ermöglichen das Beschreiben von nebenläufigen Prozessen.
- *Run* startet einen Thread.
- *Exit* beendet einen Thread.
- *Statement-Blöcke* kapseln eine Menge von Instruktionen.
- Mittels *Goto* wird der aktuell ausgeführte Statementblock verlassen und ein neuer Statementblock betreten.

Alle Sprachkonstrukte der CFIL werden im weiteren Verlauf dieses Abschnitts genauer vorgestellt. Jedes Element wird als Xtext Produktionsregel beschrieben, um die Syntax zu verdeutlichen. Dabei ist zu beachten, dass sich die CFIL von den vordefinierten Sprachkonstrukten des DVF abgrenzt und als Zwischensprache für den Transformationsprozess dient. Sie ist nicht dazu gedacht, direkt in die Grammatik einer domänenspezifischen Sprache integriert zu werden.

4.4.1 Nebenläufige Prozesse

Die CFIL beinhaltet Sprachkonstrukte zur Modellierung nebenläufiger Prozesse. Zu diesem Zweck stellt die Control Flow Intermediate Language ein Sprachkonstrukt namens *Thread* zur Verfügung, mit dem nebenläufige Prozesse deklariert werden können. Das Sprachkonstrukt *Thread* hat die folgende Syntax:

```
1 DVFThread:
2   "thread" name=ID "entry" entry=[DVFStatementBlock]
3   ("(" firstparam=DVFVariableDeclare
4     ("," moreparam=DVFVariableDeclare)* ")" )?
5   "{"
6     (variables+=DVFVariableDeclare ";")*
7     (statements+=DVFStatementBlock)+
8   "}"
9 ;
```

Listing 4.44: Xtext-Grammatik des Thread-Blocks

Jede Thread-Deklaration beginnt mit dem Schlüsselwort *thread* und einem eindeutigen Bezeichner. Darauf folgt der Name eines Statement-Blocks, der den Eintrittspunkt darstellt. Statement-Blöcke dienen zum Kapseln von Anweisungen und werden genauer in Abschnitt 4.4.2 vorgestellt.

Jede Thread-Deklaration kann eine Menge von Parametern enthalten, die beim Starten des entsprechenden Prozesses übergeben werden. Auf die Parameterliste folgt in geschweiften Klammern eine Menge von lokalen Variablen und eine Menge von Statement-Blöcken. Das nächste Beispiel verdeutlicht die Anwendung des Sprachkonstrukts *Thread*:

```
1 run Process(0);
2 run Process(1);
3
4 thread Process entry A (byte id){
5   //...
6 }
```

Listing 4.45: Modell mit einem Thread-Block

Das obige Listing 4.45 zeigt ein Modell, das einen Thread-Block enthält. *Process* bekommt ein Byte als Parameter übergeben, das einer eindeutigen Prozess-ID entspricht. Das Beispiel nutzt neben einem Thread-Block auch ein Sprachkonstrukt namens *Run*. Die Run-Direktive hat die folgende Syntax:

```
1 DVFRun:
2   "run" target=[DVFThread]
3   ( "(" firstexpr=DVFExpression
4   ("," moreexpr=DVFExpression)* ")" )?
5   ";"
6 ;
```

Listing 4.46: Xtext-Grammatik für Run

Um einen Thread zu starten, wird das Schlüsselwort *run* genutzt. Darauf folgt der Name des Threads und optional in Klammern eine Liste von Parametern, die dem Thread beim Starten übergeben werden.

4.4.2 Kapselung von Statements

Wenn ein Thread gestartet worden ist, beginnt er mit der Ausführung der darin enthaltenen Statement-Blöcke. Diese haben die folgende Syntax:

```
1 DVFStatementBlock:
2   "statementblock" name=ID "{"
3     (statements+=DVFStatement)*
4   "}"
5 ;
```

Listing 4.47: Xtext-Grammatik des Statementblocks

Jeder Statement-Block besteht aus dem Schlüsselwort *statementblock*, einem Bezeichner und einer Liste von Statements. Für die Statements wird die bereits bekannte Regel *DVFStatement* genutzt. Um einen Statement-Block zu verlassen, muss das Sprachkonstrukt *goto* verwendet werden. Zu beachten ist, dass *goto* lediglich im Rahmen der CFIL-Transformation zulässig ist. Wegen seiner negativen Eigenschaften [39] sollte es nicht in der eigentlichen DSL genutzt werden. Das Goto-Statement hat die folgende Syntax:

```
1 DVFGoto:
2   "goto" destination=[DVFStatementBlock] ";"
3 ;
```

Listing 4.48: Xtext-Grammatik für Goto

Anhand des obigen Listings wird ersichtlich, dass *goto* den Namen eines Statement-Blocks als Sprungziel übergeben bekommt. Neben *Goto* kann die Exit-Direktive verwendet werden, um einen Statement-Block zu verlassen und den entsprechenden Thread zu terminieren. *Exit* hat die folgende Syntax:

```
1 DVFExit:
2   "exit" ";"
3 ;
```

Listing 4.49: Xtext-Grammatik für Exit

Die Produktionsregel *DVFExit* besteht lediglich aus dem Schlüsselwort *exit* (vgl. Listing 4.49).

4.5 Validierung

Das DSL Verification Framework stellt dem DSL-Entwickler verschiedene Produktionsregeln zur Verfügung, die er in die Grammatik seiner domänenspezifischen Sprache integrieren kann. Nach dem Fertigstellen der Grammatik wird daraus ein Parser generiert. Der Parser überführt ein Modell in einen Abstract Syntax Tree. Des Weiteren kann er Syntaxfehler in einem Modell erkennen. Abschnitt 2.2.3 zeigt jedoch, dass für einen Teil der potentiellen Syntaxfehler eine zusätzliche Überprüfung notwendig ist. Zu diesem Zweck enthält Xtext ein Validierungs-Framework. Das Validierungs-Framework wird vom DSL-Entwickler genutzt, um sein DSL-Projekt mit einer zusätzlichen Fehlerüberprüfung für die entsprechenden Produktionsregeln zu ergänzen. Abbildung 4.2 verdeutlicht, dass das Validierungs-Framework auch im DVF Verwendung findet. Deshalb beschreibt dieser Abschnitt die Integration des Validierungs-Frameworks in das DVF.

Das DVF beinhaltet zwei Transformatoren, die Modelle nach Java und Promela überführen. Beide bekommen einen Abstract Syntrax Tree übergeben und nutzen das Validierungs-Framework, um darin enthaltene Fehler zu finden. Wenn der AST keine Fehler

enthält, beginnt der Transformator mit dem eigentlichen Übersetzungsprozess. Falls der Abstract Syntax Tree hingegen nicht korrekt ist, wird ein Fehler gemeldet und der DSL-Anwender muss sein Modell entsprechend modifizieren.

Gemäß Abschnitt 2.2.3 besteht die Validierung aus einer Menge von Methoden. Jede Validierungs-Methode wird automatisiert vom Validierungs-Framework aufgerufen und hat genau einen Parameter, der einem Element des AST entspricht. Dies bedeutet konkret: Gegeben sei eine Validierungs-Methode namens *checkDeclareType()*, die bestimmte Aspekte einer Variablendeklaration verifiziert. Somit ist ihr einziger Parameter ein Objekt vom Typ *DVFVariableDeclare* (vgl. Listing 4.7). Wenn das Validierungs-Framework vom Transformator gestartet wird, iteriert es automatisiert über den gesamten AST und ruft für jedes Element vom Typ *DVFVariableDeclare* die Methode *checkDeclareType()* auf.

Die Umsetzung der Validierungs-Methoden bzw. ihr Funktionsumfang wird in diesem Abschnitt genauer beschrieben. Deshalb zeigt Unterkapitel 4.5.1, wie in domänenspezifischen Sprachen, die die Produktionsregel *DVFVariableDeclare* einbinden, die Typsicherheit innerhalb einer Variablendeklaration sichergestellt werden kann. Die Beschreibung der weiteren Validierungsfunktionalität erfolgt in Abschnitt 4.5.2. Da die gesamte Implementierung sehr komplex ist und es in dieser Arbeit primär darum geht, die Machbarkeit des DVFs nachzuweisen, erfolgt die Beschreibung lediglich auf informeller Basis.

4.5.1 Validierung von Datentypen

Dieser Abschnitt beschreibt exemplarisch eine Validierungs-Methode des DSL Verification Frameworks für das Initialisieren von Variablen. Dabei wird von der Validierung sichergestellt, dass in jeder Variablendeklaration der Datentyp der Initialisierungs-Expression äquivalent zum Datentyp der Variable ist. Das folgende Beispiel verdeutlicht dieses Problem:

```
1 int a;
2 byte b = 7+5;
```

Listing 4.50: Bedeutung von Typvalidierung in domänenspezifischen Sprachen

Das obige Listing 4.50 zeigt eine fiktive DSL, in der unter anderem Variablen deklariert werden können. Dafür findet die im DVF enthaltene Produktionsregel *DVFVariableDeclare* Verwendung. Das Modell aus Listing 4.50 enthält die Variablen *a* und *b*. Bei der Variablendeklaration von *b* wird ein Initialisierungswert spezifiziert. Da *b* vom Typ Byte ist, darf die Initialisierungs-Expression nur ein Byte zurückliefern. Zu beachten ist, dass das DVF nur statische Datentypen unterstützt. Änderungen zur Laufzeit, wie beispielsweise durch *type casts*, sind Teil von zukünftigen Arbeiten. Um die Typsicherheit einer Variablendeklaration zu gewährleisten, müssen somit vom Validierungs-Framework zwei Werte ermittelt und miteinander verglichen werden:

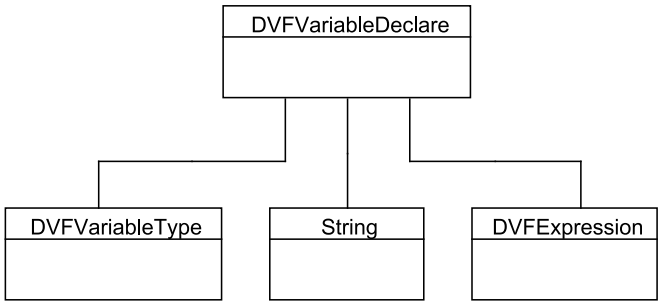

Abbildung 4.9: Variablendeklaration im AST (Objektdiagramm)

- Der Datentyp der deklarierten Variable.
- Der Datentyp der Expression, die zur Initialisierung der Variable verwendet wird.

Listing 4.7 zeigt die Produktionsregel *DVFVariableDeclare*, die vom DVF zum Deklarieren von Variablen zur Verfügung gestellt wird. Daran wird deutlich, dass die Produktionsregel *DVFExpression* genutzt wird, um die entsprechende Variable zu initialisieren. Abbildung 4.9 visualisiert mit einem vereinfachten Objektdiagramm, wie *DVFVariableDeclare* und *DVFExpression* in Relation zueinander stehen, wenn sie Teil eines AST sind. Daran wird deutlich, dass jede Variablendeklaration im Abstract Syntax Tree unter anderem drei Referenzen enthält:

- Das Objekt *DVFVariableType* repräsentiert den Datentyp der Variablen.
- Ein Objekt vom Typ *String* enthält den eindeutigen Variablenbezeichner.
- Das Objekt *DVFExpression* dient zum Initialisieren der Variable.

Aus diesem Aufbau leitet sich die Struktur der Validierungs-Methode ab, die für alle Objekte vom Typ *DVFVariableDeclare* aufgerufen werden muss. Die Methode ermittelt den Datentyp der Variable, der Intialisierungs-Expression, vergleicht beide miteinander und meldet gegebenenfalls einen Fehler. Das folgende Listing stellt die Validierungs-Methode genauer vor:

```
1 public class DVFJavaValidator extends AbstractDVFJavaValidator {
2   @Check
3   public void checkDeclareType(DVFVariableDeclare var){
4     ValDataType variable = getVarType(var);
5     ValDataType init = getExprType(var.getInit());
6     if(!variable.equals(init){
7       String e = variable.toString + "␣!=␣" + init.toString();
```

```
8           error(e,0);
9       }
10    }
11 }
```

Listing 4.51: Validierung der Initialisierung

Gemäß Abschnitt 2.2.3 wird die Validierungs-Methode *checkDeclareType()* in die Klasse *DVFJavaValidator* eingebettet. Sie ist wie folgt umgesetzt:

- In Zeile 4 wird die Methode *getVarType()* aufgerufen, um den Datentyp der Variablendeklaration abzufragen.
- In Zeile 5 wird die Methode *getExprType()* aufgerufen, um den Datentyp der Initialisierungs-Expression abzufragen.
- In den Zeilen 6 bis 9 werden die beiden Datentypen miteinander verglichen und gegebenenfalls ein Fehler gemeldet.

Der Vorteil dieser Implementierung ist, dass die Methoden *getVarType()* und *getExprType()* auch zur Validierung von weiteren Sprachkonstrukten des DVF wiederverwendet werden können. Der weitere Verlauf dieses Abschnitts zeigt die genaue Umsetzung der Methoden *getVarType()* bzw. *getExprType()*, um den Datentyp einer Variablendeklaration bzw. einer Expression zu erhalten.

Ermittlung des Datentyps einer Variablendeklaration

Das folgende Listing stellt die Methode *getVarType()* genauer vor, die den Datentyp einer Variablendeklaration ermittelt:

```
1  ValDataType getVarType(DVFVariableDeclare var){
2    if(var.getStatType() instanceof String){
3      if(var.getStatType().equals("byte"))
4        return new ValDataType(ValDataTypeEnum.Byte, "");
5      else if(var.getStatType().equals("int")
6        return new ValDataType(ValDataTypeEnum.Int, "");
7      //...
8    }
9    else if(var.getDynType() instanceof DVFEnumDeclare)
10     return new ValDataType(ValDataTypeEnum.Enum,
11       var.getDynType().getName());
12   //...
13 }
```

Listing 4.52: Datentyp einer Variablendeklaration

Das obige Listing 4.52 zeigt die Umsetzung der Methode *getVarType()*, die das Objekt einer Variablendeklaration übergeben bekommt und dessen genauen Datentyp ermittelt. Um diese Arbeit übersichtlicher zu gestalten, wird sie nicht vollständig beschrieben. *GetVarType()* führt die folgende Fallunterscheidung durch:

- Die Zeilen 3 bis 4 fragen ab, ob es sich bei der Variable um ein Byte handelt und geben einen entsprechenden Wert zurück.
- Die Zeilen 5 bis 6 fragen ab, ob es sich bei der Variable um ein Integer handelt und geben einen entsprechenden Wert zurück.
- Die Zeilen 9 bis 11 fragen ab, ob es sich bei der Variable um eine Aufzählung handelt und geben einen entsprechenden Wert zurück.

Somit liegt nach Aufruf von *getVarType()* der Datentyp einer Variablendeklaration vor.

Ermittlung des Datentyps einer Expression

In Listing 4.52 wird gezeigt, wie der Datentyp einer Variablendeklaration mit der Methode *getVarType()* ermittelt werden kann. Dieser muss im Anschluss vom Validierungs-Framework mit dem Datentyp der Initialisierungs-Expression verglichen werden. Um den Datentyp einer Initialisierungs-Expression abzufragen, wird die Methode *getExprType()* genutzt (vgl. Listing 4.51). *GetExprType()* erhält als Parameter ein Objekt vom Typ *DVFExpression*. *DVFExpression* referenziert mehrere Objekte, die in einer baumartigen Struktur miteinander verknüpft sind. Dies verdeutlicht das folgende Beispiel:

```
1 byte a = 1;
2 byte b = 2;
3 byte c = a | b;
```

Listing 4.53: Einfache Expression

Im obigen Listing ist ein Modell zu sehen, das mit einer fiktiven DSL und dem DVF Variablen deklariert. Die Variable c wird mit der Expression $a|b$ initialisiert.

Abbildung 4.10 zeigt den AST der Initialisierungs-Expression. Der Abstract Syntax Tree ist zur besseren Übersicht vereinfacht dargestellt und einige Elemente sind lediglich mit „...“ angedeutet. Trotzdem wird ersichtlich, dass der Baum bereits ein hohes Maß an Komplexität erreicht, obwohl es sich um eine sehr einfache Expression handelt. Dies begründet sich durch den Aufbau der Produktionsregel *DVFExpression* aus Listing 4.13. Bei einer alternativen Implementierung könnte die Komplexität auf Kosten von Eigenschaften, wie der im AST enthaltenen Operatorpräzedenz, reduziert werden.

Damit der Datentyp einer Expression ermittelt werden kann, muss *getExprType()* die gesamte baumartige Struktur aus Abbildung 4.10 auswerten. Deshalb ist sie wie folgt umgesetzt: Für jedes Objekt, das Teil einer Expression ist, wird eine Methode aufgerufen,

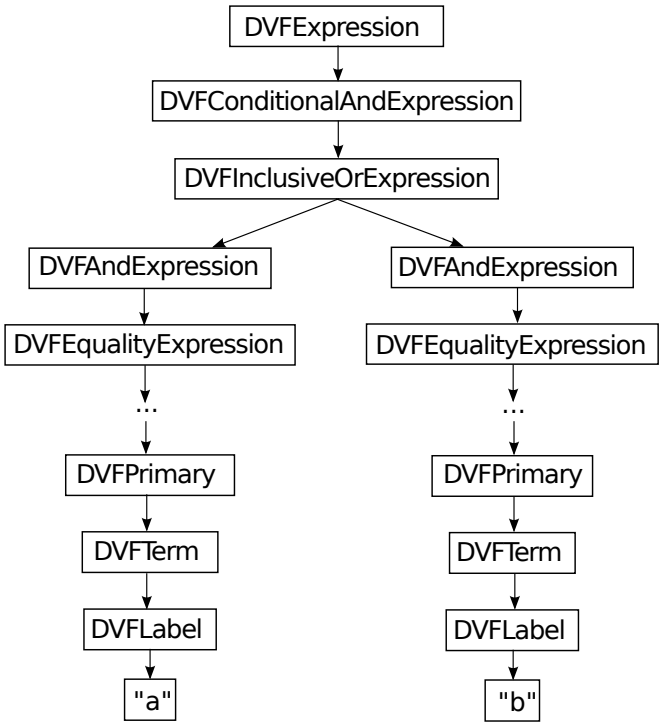

Abbildung 4.10: Abstract Syntax Tree für eine einfache Expression

die den Datentyp auswertet. Dementsprechend gibt es eine Menge von Methoden, denen Objekte vom Typ *DVFAndExpression*, *DVFEqualityExpression*, usw. (vgl Abbildung 4.10) übergeben werden und die deren genauen Datentyp zurückliefern. Dieses Vorgehen verdeutlicht die folgende beispielhafte Implementierung von *getExprType()*:

```
1  ValDataType getExprType(DVFExpression expr){
2    if(expr == null)
3      return null;
4
5    ValDataType term1 =
6      getTypeConditionalAndExpression(expr.getTerm1());
7
8    if(expr.getOp1()==null) return term1;
9    else{
10     ValDataType rest1 = getTypeExpression(expr.getRest1());
11     if(term1.getType().equals(rest1.getType())
12       && term1.getType().equals(UDLDataTypeEnum.Bool))
```

```
13        return term1;
14      else
15        return new ValDataType(ValDataTypeEnum.TypeError,"");
16    }
17  }
```

Listing 4.54: Ermittlung des Datentyps einer Expression mit Java

Die Funktion im obigen Listing ermittelt den Datentyp des Objekts *DVFExpression*. *DVFExpression* enthält ein oder zwei Referenzen auf Objekte vom Typ *DVFConditionalAndExpression* bzw. *DVFExpression*. Diese Unterscheidung hat den folgenden Grund:

- Ein Zeiger: Aufruf der Funktion *getTypeConditionalAndExpression()*, um dessen Datentyp zu ermitteln und ihn zurückzugeben.
- Zwei Zeiger: Beide Objekte sind durch den Operator || miteinander verknüpft. In diesem Fall werden die Datentypen der beiden AST-Elemente durch Aufruf der Funktionen *getTypeConditionalAndExpression()* und *getExprType()* ermittelt. Im Anschluss wird überprüft, ob beide Funktionen den Datentyp *bool* zurückliefern und in Abhängigkeit des Ergebnisses ein Fehler gemeldet. Zu beachten ist, dass dabei lediglich die Datentypen ermittelt werden, um das Modell zu validieren. Eine (Kurzschluss)auswertung der Expression erfolgt erst zur Laufzeit.

Das nächste Listing zeigt die beispielhafte Implementierung der Funktion *getTypeConditionalAndExpression()*, die im obigen Listing 4.54 in Zeile 6 aufgerufen wird:

```
1  ValDataType getTypeConditionalAndExpression
2  (DVFConditionalAndExpression expr){
3    VarDataType previous =
4      getTypeInclusiveOrExpression(ae.getTerm1());
5
6    if(ae.getRest1().isEmpty()) return previous;
7    else{
8      EList<DVFConditionalAndExpressionLoop> ael = ae.getRest1();
9      Iterator<DVFConditionalAndExpressionLoop> it =
10       ael.iterator();
11     ValDataType current = null;
12     while(it.hasNext()){
13       current = getTypeInclusiveOrExpression(
14       it.next().getRest1());
15
16       //expression type is a bool
17       if(previous.getType().equals(current.getType())
```

```
18          && previous.getType().equals(ValDataTypeEnum.Bool)){
19            previous = current;
20          }
21          //expression type is something else
22          else return
23            new ValDataType(ValDataTypeEnum.TypeError,"");
24        }
25        return new ValDataType(ValDataTypeEnum.Bool,"");
26      }
27  }
```

Listing 4.55: Ermittlung des Datentyps von DVFConditionalAndExpression

Im obigen Listing wird der Datentyp der Produktionsregel *DVFConditionalAndExpression* ermittelt. *DVFConditionalAndExpression* besteht entweder aus einem, oder mehreren Zeigern auf Objekte vom Typ *DVFInclusiveOrExpression*. Dabei ist der erste Zeiger in der Variable *term1* gespeichert und alle weiteren, zur besseren Lesbarkeit der Ableitungsregeln, in der Klasse *DVFConditionalAndExpressionLoop* gekapselt.

In Listing 4.55 wird die Methode *getTypeInclusiveOrExpression()* aufgerufen, um den Datentyp der DVFInclusiveOrExpression-Objekte zu ermitteln. Wenn *DVFConditionalAndExpression* nur ein DVFInclusiveOrExpression-Objekt enthält und daher *getRest1()* eine leere Liste zurückliefert, wird der Datentyp des einzigen DVFInclusiveOrExpression-Objekts zurückgegeben. Wenn hingegen mehrere DVFInclusiveOrExpression-Objekte enthalten sind, werden diese in Zeile 8 als Liste vom Typ *EList* gespeichert. Im Anschluss wird über alle Objekte der Liste iteriert und verifiziert, dass alle den Datentyp *bool* zurückliefern. Dies ist notwendig, da alle Objekte durch den Operator && verknüpft, und somit nur Booleans zulässig sind. Die Implementierung der Methode *getTypeInclusiveOrExpression()*, die in Listing 4.55 in Zeile 4 aufgerufen wird, gestaltet sich analog zu der Umsetzung von *getTypeConditionalAndExpression()*.

4.5.2 Informelle Beschreibung der weiteren Validierungs-Methoden

Der vorangegangene Abschnitt zeigt exemplarisch die im DVF enthaltene Validierung, um den Datentyp von Initialisierungs-Expressions in Variablendeklarationen zu verifizieren. Dafür wird eine Validierungs-Methode namens *checkDeclareType()* vorgestellt, die ein Objekt vom Typ *DVFVariableDeclare* als Parameter erhält. Das Validierungs-Framework durchsucht automatisiert den Abstract Syntax Tree eines Modells und ruft für jedes Objekt vom Typ *DVFVariableDeclare* die Methode *checkDeclareType()* auf. Diese vergleicht den Datentyp der Variable mit dem Datentyp der Initialisierungs-Expression und meldet gegebenfalls einen Fehler. Neben den Variablendeklarationen enthält das DVF noch weitere Produktionsregeln, zum Beschreiben von Statements, Expressions, usw. Die dafür notwendige Validierung wird in diesem Abschnitt genauer vorgestellt.

Das Konzept der im DVF enthaltenen Validierung ist wie folgt: Das DSL Verification Framework stellt dem DSL Entwickler verschiedene Sprachkonstrukte in Form von Produktionsregeln zur Verfügung. Der DSL-Entwickler kann diese Produktionsregeln in die Grammatik einer domänenspezifischen Sprache integrieren und sie so beispielsweise mit Variablen, Zuweisungen oder symmetrischen Arrays ergänzen. Die entsprechenden Produktionsregeln werden im Detail in Abschnitt 4.3 vorgestellt. Anhand von Unterkapitel 2.2.1 wird deutlich, dass der Parsergenerator für jede Produktionsregel eine Klasse erzeugt. Aus den Instanzen dieser Klassen setzt sich der Abstract Syntax Tree eines Modells zusammen. Somit ist die Validierung des DVF wie folgt umgesetzt: Für jede vom DVF zur Verfügung gestellte Produktionsregel existiert eine Validierungs-Methode. Dazu gehört beispielsweise *checkDeclareType()* aus Abschnitt 4.5.1, die alle Variablendeklarationen in einem AST verifiziert. Dementsprechend besteht die Validierung des DVF aus Methoden zum Verifizieren von *DVFExpression*, *DVFFunction*, usw. Der Funktionsumfang der Methoden wird im weiteren Verlauf dieses Abschnitts genauer beschrieben und basiert auf den folgenden beiden Aspekten:

- Ein Teil der Funktionalität des Validierungs-Framework orientiert sich an Hochsprachen wie C oder Java. Dazu gehört beispielsweise die Typsicherheit von Expressions.
- Des Weiteren wird die korrekte Verwendung der Sprachkonstrukte sichergestellt, die Optimierungen auf Modellebene umsetzen. Dazu gehört beispielsweise, dass der Range-Datentyp nur als Rückgabewert für Methoden mit eigenschaftserhaltender Abstraktion Verwendung findet (vgl. Listing 4.39).

Im Zuge der DVF-Erweiterbarkeit zeigt Unterkapitel 4.8, wie Anpassungen in den Validierungs-Methoden vorgenommen werden können, falls die in diesem Abschnitt vorgestellte Funktionalität nicht ausreichend ist oder modifiziert werden soll. Die Schnittstelle zur Erweiterung des DSL Verification Framework kann auch genutzt werden, um benutzerspezifische Sprachkonstrukte zu validieren.

Validierung von Variablen

Für die Variablendeklaration wird im DVF die Produktionsregel *DVFVariableDeclare* genutzt. Abschnitt 4.5.1 hat bereits die Validierung der entsprechenden Initialisierungs-Expression vorgestellt. Es gibt jedoch noch weitere Aspekte in Variablendeklarationen, die das DVF validiert und die in diesem Abschnitt genauer vorgestellt werden. Die Deklaration von Variablen erfolgt in Modellen in verschiedenen Bereichen:

- Variablen werden global, in Klassen oder in Scalarsets deklariert.
- Variablen können auch im Rahmen von Message-Queues als Signalparameter genutzt werden.

- Auch die Parameter von Methoden werden mit der Produktionsregel *DVFVariableDeclare* implementiert.

Wie bereits im letzten Abschnitt beschrieben, muss bei den Expressions, die zur Initialisierung eingesetzt werden, darauf geachtet werden, dass ihr Typ dem Datentyp der deklarierten Variable entspricht. Die folgende Besonderheit gilt für Variablen, die Teil eines Scalarsets sind: Sie dürfen nicht als Array mit eckigen Klammern deklariert werden, da jede Variable in einem Scalarset bereits implizit ein Array ist. Des Weiteren stellt die Validierung sicher, dass Variablendeklarationen, wenn sie als Paramater in Signalen oder Methoden genutzt werden, keinen Initialisierungswert aufweisen.

Validierung von Methoden

Neben dem Element *DVFVariableDeclare* muss auch die Produktionsregel *DVFFunction* validiert werden: Wenn es sich um eine Funktion handelt, die eigenschaftserhaltende Abstraktion umsetzt, darf die Methode nur einen leeren Methodenkörper ohne Statements enthalten. Des Weiteren darf sie keine Werte vom Typ Integer zurückgegeben. Integer sind an dieser Stelle nicht zulässig, da der Model Checker beim Verifizieren alle möglichen Rückgabewerte einer Interface-Funktion in Betracht zieht und der Definitionsbereich einer Integer-Variable aufgrund seiner Größe zur *State Space Explosion* führt.

Für Funktionen, die eigenschaftserhaltende Abstraktion umsetzen, kann als Rückgabewert auch das Range-Element spezifiziert werden. Im DSL Verification Framework wird das Range-Element mit der Produktionsregel *DVFRange* umgesetzt. Der Definitionsbereich von *DVFRange* ist in Listing 4.39 mit den beiden Parametern *from* und *to* beschrieben. Die Validierung stellt sicher, dass immer gilt: $from < to$.

Validierung von Statements

Zuweisungen werden im Rahmen der Regel *DVFStatement* in Listing 4.22 umgesetzt. Das DVF validiert für Zuweisungen, die auf *DVFStatement* basieren, die folgenden Aspekte:

- Einer Variable vom Typ x darf nur ein Wert vom selben Typ zugewiesen werden. Daher ist beispielsweise eine Operation, die einer Byte-Variable ein *Bool* zuweist, ungültig. Auch an dieser Stelle muss, wie in Listing 4.54, der Rückgabewert der Zuweisungs-Expression ausgewertet werden.

- Werte dürfen nur Variablen, nicht aber Funktionsaufrufen zugewiesen werden. Das folgende Beispiel ist daher ungültig: $test1() = 5;$

Auch das Return-Statement wird einer zusätzlichen Validierung unterzogen um sicherzustellen, dass es in eine Methode eingebettet ist. Ein weiterer Aspekt von Return-Statements, der vom DVF validiert wird, ist der zurückgegebene Datentyp. So darf

beispielsweise eine Methode mit dem Funktionskopf *byte inc()*{} kein Element vom Typ *Bool* zurückgeben.

Die Regel *DVFSendSignal* ist ein Teil von *DVFStatement* und dient zum Versand von Signalen. Neben dem eigentlichen Signal können zusätzliche Parameter übertragen werden. Anzahl und Datentyp der Parameter werden in der Deklaration des Signals und somit von der Regel *DVFSignalDeclare* umgesetzt. Die Validierung überprüft beim Versand eines Signals die folgenden Aspekte:

- Entspricht die Anzahl der angegebenen Parameter der Anzahl der Parameter in der Signaldeklaration.
- Entsprechen die Datentypen der Parameter den jeweiligen Datentypen in der Signaldeklaration.

Mit der Regel *DVFLabel* werden Bezeichner innerhalb einer Expression umgesetzt. Die Validierung stellt sicher, dass runde Klammern nur bei Bezeichnern verwendet werden, die einer Funktion entsprechen. Des Weiteren dürfen eckige Klammern nur bei Bezeichern genutzt werden, die auf ein Array verweisen.

Bei Funktionsaufrufen werden, genauso wie bei *DVFSignalSend*, Anzahl und Datentyp der jeweiligen Parameter validiert. In Sprachkonstrukten für Schleifen und Verzweigungen sind Expressions enthalten. So bestimmt beispielsweise die Expression im Kopf der While-Schleife, wann die Schleife terminiert. Sowohl bei IF-Blöcken, als auch bei While-Schleifen muss von der Validierung sichergestellt werden, dass die jeweilige Expression einen Wert vom Typ Boolean zurückliefert.

4.6 Übersetzung nach Promela

In den vorangegangenen Abschnitten sind die Produktionsregeln des DVF und die Validierung vorgestellt worden. Mit den Produktionsregeln kann die Grammatik und somit das Metamodell einer DSL umgesetzt werden. Damit das DSL Verification Framework in einem Softwareprojekt zur Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation Verwendung finden kann, sind jedoch auch Transformatoren notwendig. Der Transformationsprozess des DVF ist in Abbildung 4.11 schematisch dargestellt. Zunächst beschreibt der DSL-Anwender mit einer domänenspezifischen Sprache ein Modell. Das Modell wird vom Parser in einen Abstract Syntax Tree (AST) überführt. Der Transformator, den der DSL-Entwickler implementiert hat, übersetzt den AST in die Control Flow Intermediate Language (CFIL). Zum Abschluss wird das CFIL-Modell validiert und in die beiden Zielsprachen transformiert. Dementsprechend sind zwei Transformatoren Teil des DVF:

- Ein Transformator, der das Modell in die Hochsprache Java überführt.

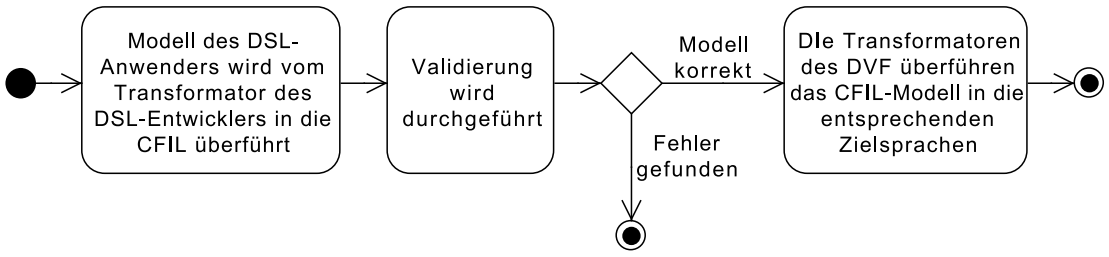

Abbildung 4.11: Transformation in die Zielsprachen (Aktivitätsdiagramm)

- Ein Transformator, der das Modell in die Model Checker-Eingabesprache Promela überführt.

Der Transformationsprozess, der das Modell nach Promela überführt, wird in diesem Abschnitt genauer vorgestellt. Im Anschluss beschreibt Abschnitt 4.7 die Übersetzung in die Hochsprache Java.

Der Transformator nach Promela weist die folgenden Eigenschaften auf: Es werden alle Sprachkonstrukte eines Modells in die Model Checker-Eingabesprache übersetzt, die auf den Produktionsregeln aus Abschnitt 4.3 basieren. Dazu gehören unter anderem Variablen, Expressions, Signale und symmetrische Arrays. Eine besondere Herausforderung ist, dass es für einige Produktionsregeln des DVF, wie beispielsweise Methoden, keine äquivalenten Sprachkonstrukte in Promela gibt. Des Weiteren verarbeitet der Transformator auch alle Elemente, die Teil der CFIL sind. Während des Übersetzungsprozesses werden die Optimierungen auf Modellebene berücksichtigt bzw. angewendet, die Teil des Stands der Forschung aus Abschnitt 3.1 sind, um den Zustandsraum des Modells zu verkleinern. Eine detaillierte Beschreibung dieses Übersetzungskonzepts und einer möglichen Implementierung folgt in den Abschnitten 4.6.1 bis 4.6.12.

Bevor der Transformator mit der Übersetzung in die Zielsprache beginnt, erfolgt eine Analyse. Dabei sind die folgenden Aspekte des Modells von Bedeutung:

- Gemäß Abschnitt 2.1.4 müssen für symmetrische Arrays eine bestimmte Menge von Dummy-Prozessen gestartet werden. Deshalb erfolgt eine Analyse der Scalarsets, um die Anzahl der Dummy-Prozesse zu ermitteln.

- Promela unterstützt keine Methodendeklarationen. Deshalb muss der Transformator Promela-Variablen zum Speichern der Funktionsrückgabewerte und Funktionsparameter generieren. Die Analyse berechnet anhand der Statements in einem Modell, wie viele zusätzliche Variablen angelegt werden müssen.

- Das DVF beinhaltet ein Sprachkonstrukt zum Deklarieren von Message-Queues. Zum Speichern der darauf gespeicherten Signale sind zusätzliche Variablen notwen-

dig. Deshalb werden von der Analyse die Message-Queues untersucht und ermittelt, wie viele lokale Variablen zum Abspeichern der Signalparameter generiert werden müssen.

Nach der Analyse beginnt der eigentliche Transformationsprozess. Dieser übersetzt nacheinander alle Sprachkonstrukte des DVF und der CFIL, unter Berücksichtigung der Optimierungen auf Modellebene, nach Promela. Er hat die folgende Struktur:

- Aufzählungen werden nach Promela transformiert.
- Zeichenketten werden nach Promela transformiert.
- Scalarsets werden nach Promala überführt.
- Globale Variablen werden nach Promela überführt.
- Lokale Variablen in Klassen werden nach Promela überführt.
- Methoden werden nach Promela überführt.
- Signale werden nach Promela überführt.
- Threads und Statementblöcke werden nach Promela überführt.
- Abschließend wird ein Init-Block generiert, der Arrays initialisiert und Prozesse startet.

Die unterschiedlichen Analysen und Transformationen werden in den nächsten Abschnitten genauer vorgestellt.

4.6.1 Analyse der Scalarset-Elemente

Bei der Deklaration eines Scalarsets wird, neben dem Bezeichner und den enthaltenen Elementen, auch die Größe spezifiziert. Gemäß der in Abschnitt 2.1.4 beschriebenen Übersetzung [42] von Scalarsets nach Promela, muss für jedes symmetrische Array eine Menge von Dummy-Prozessen gestartet werden. Die Anzahl der Prozessinstanzen ist äquivalent zur Größe des entsprechenden Scalarsets. Bei einer automatischen Transformation nach Promela ergibt sich somit das folgende Problem: Wenn die Dummy-Prozesse im Promela-Modell instanziiert werden, muss ihre genaue Anzahl bekannt sein. Daher berechnet die in diesem Abschnitt vorgestellte Analyse in einem Modell die Summe aller Scalarset-Größen. Das Ergebnis wird in Abschnitt 4.6.12 genutzt, um Instanzen der Dummy-Prozesse im Init-Block zu starten. Zu beachten ist, dass es sich dabei um eine Problemstellung handelt, die den Model Checker Spin betrifft. Für zukünftige Arbeiten sollte daher evaluiert werden, ob der Einsatz eines alternativen Model Checkers von Vorteil ist, der keine Dummy-Prozesse zur Umsetzung symmetrischer Arrays benötigt.

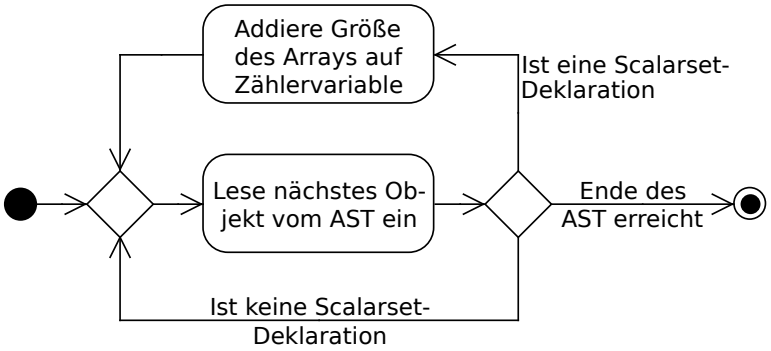

Abbildung 4.12: Analyse der Scalarsets (Aktivitätsdiagramm)

Abbildung 4.12 zeigt das Konzept der in diesem Abschnitt vorgestellten Analyse als Aktivitätsdiagramm. Es zeigt, dass der Transformator über alle Elemente des Abstract Syntax Trees iteriert und überprüft, ob es sich dabei um eine Scalarset-Deklaration handelt. Wenn das entsprechende Objekt kein Scalarset ist, wird das nächste Element des AST abgerufen. Falls es sich um ein Scalarset handelt, addiert der Transformator die Größe des Scalarsets auf eine globale Zählervariable. Der Analysealgorithmus terminiert, sobald das Ende des AST erreicht ist.

Eine beispielhafte Implementierung der Scalarset-Analyse gestaltet sich mit Xpand und Java wie folgt: Gemäß Abschnitt 2.2.2 ist ein sogenannter MWE-Workflow Teil des Transformators. Der Workflow beinhaltet eine Menge von Methoden, die automatisch für bestimmte, im AST enthaltene Objekte aufgerufen werden. Deshalb wird für die Scalarset-Analyse eine Methode namens *analyseScalar()* implementiert und in den MWE-Workflow eingetragen. *AnalyseScalar()* wird automatisch für jedes Objekt vom Typ *DVFVariableDeclare*, das Teil des Abstract Syntax Tree ist, aufgerufen. Die Implementierung mit Java gestaltet sich wie folgt:

```
1 static int scalaramount = 0;
2
3 public static void analyseScalar
4 (DVFScalarDeclare scalar_declare){
5   scalaramount+=scalar_declare.getSize();
6 }
```

Listing 4.56: Java-Methode *analyseScalar()*

Die Methode im obigen Listing bekommt als Parameter ein Objekt vom Typ *DVFScalarDeclare* übergeben und inkrementiert in Abhängigkeit der Scalarsetgröße die statische Variable *scalaramount*. Nach der Ausführung der Scalarset-Analyse ist somit die Summe aller Scalarset-Größen und implizit die Anzahl der zu startenden Dummy-Prozesse in *scalaramount* gespeichert.

4.6.2 Analyse der lokalen Variablen für Parameter und Rückgabewerte

Das DSL Verification Framework beinhaltet Produktionsregeln, mit denen Methoden in domänenspezifischen Sprachen deklariert und aufgerufen werden können. Promela bietet jedoch keine native Unterstützung zum Umsetzen von Methoden. Daher ist es notwendig, dass der Transformator die Elemente einer Methode mit nativen Promela-Sprachkonstrukten nachbildet. Die Anforderungen an einen Methodenaufruf werden anhand der Arbeitsweise eines Hochsprachen-Compilers, wie beispielsweise für C oder Java, deutlich [1]. Dieser übersetzt einen Methodenaufruf wie folgt in Maschinensprache:

- Die Parameter der Methode werden in einer Datenstruktur gespeichert.
- Die Rücksprungadresse wird in einer Datenstruktur gespeichert.
- Die Ausführung der Methode beginnt.
- Die Methode speichert den Rückgabewert in einer Datenstruktur.
- Die Methode terminiert und springt zur Rücksprungadresse.

Daraus ergeben sich für eine Übersetzung von Methodenaufrufen nach Promela die folgenden Anforderungen: Es müssen Variablen zur Verfügung stehen, in denen die Funktionsparameter und Funktionsrückgabewerte gespeichert werden können. Des Weiteren muss nach dem Beenden einer Methode die Ausführung des Modells an der korrekten Rücksprungadresse fortgesetzt werden. Aus diesem Grund führt der Transformator, bevor der eigentliche Übersetzungsprozess beginnt, eine statische Analyse des Modells durch und berechnet, wie viele zusätzliche Variablen für Parameter und Rückgabewerte notwendig sind. Das Ergebnis der Analyse findet im Rahmen der Modelltransformation in den Unterkapiteln 4.6.7 bzw. 4.6.8 Verwendung, um die entsprechenden Variablen zu generieren. Die Analyse wird im weiteren Verlauf dieses Abschnitts genauer vorgestellt.

Abbildung 4.13 zeigt das Konzept des Analyseverfahrens als Aktivitätsdiagramm. Es hat den folgenden Aufbau: Anhand von Abschnitt 4.3.4 wird deutlich, dass die Methodenaufrufe des DSL Verification Frameworks mit Statements umgesetzt werden. Diese sind wiederum ein Teil von sogenannten Threads (vgl. Listing 4.44). Deshalb iteriert der Transformator über alle Elemente des AST und überprüft, ob es sich um einen Thread handelt. Wenn das Objekt ein Thread ist, werden nacheinander alle darin enthaltenen Statements analysiert. Bei jedem Statement, das einen Methodenaufruf enthält, speichert der Transformator die Datentypen der Parameter, den Datentyp des Rückgabewerts sowie eine Referenz auf das Thread-Objekt in einer Tabelle ab. Die Analyse terminiert, wenn alle Threads und die darin enthaltenen Statements untersucht worden sind. Anhand der so gewonnenen Tabelle kann der Transformator, wenn Threads und Statements nach Promela übersetzt werden, entsprechende lokale Variablen generieren bzw. zum Speichern der Methodenaufrufe bzw. Rückgabewerte verwenden.

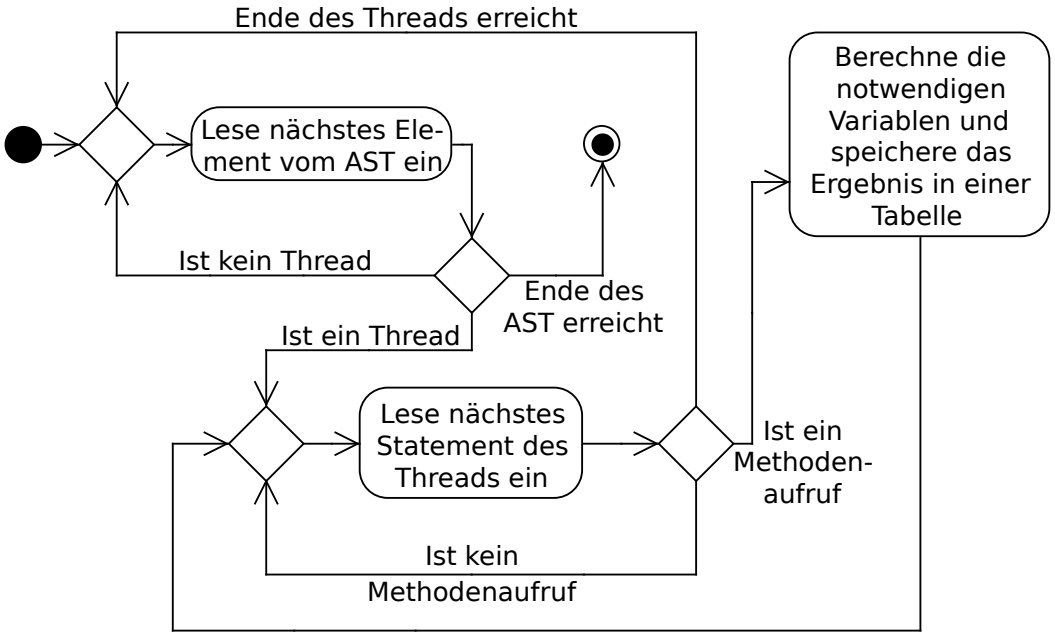

Abbildung 4.13: Analyse der Methodenaufrufe (Aktivitätsdiagramm)

Eine beispielhafte Java- bzw. Xpand-Implementierung des Analysealgorithmus gestaltet sich wie folgt: Der MWE-Workflow des Transformators enthält eine Methode, die für jedes Objekt im AST, das vom Datentyp *DVFThread* ist, aufgerufen wird. Die Methode iteriert über alle Statements und ermittelt, ob das entsprechende Objekt Methodenaufrufe enthält. Dazu müssen alle Expressions in den entsprechenden Statements analysiert werden. Sobald ein Methodenaufruf vorliegt, wird die Referenz auf das Thread-Objekt, sowie die Parameter und der Rückgabetyp in einer globalen *HashMap* [60] gespeichert.

Das folgende Beispiel verdeutlicht das Vorgehen: Gegeben sei ein CFIL-Modell, das einen Thread namens $process_0$ und zwei Methoden enthält. Die erste Methode bekommt eine Zeichenkette als Parameter übergeben und hat den Rückgabewert *void*. Die zweite Methode gibt ein Byte zurück und erhält einen Wert vom Typ Integer als Parameter. In dem Thread sind verschiedene Statements enthalten, die die beiden Methoden genau einmal aufrufen. In diesem Fall generiert die Analyse eine *HashMap* mit dem folgenden Inhalt:

Thread	Datentypen der erforderlichen Variablen
$process_0$	String
$process_0$	Byte, Integer

Für das obige Beispiel würden somit für den Thread $process_0$ drei Variablen mit den

Datentypen String, Integer und Byte generiert werden. Der Vorteil dieses Verfahrens ist, das im Rahmen von zukünftigen Arbeiten Erweiterungen mit Optimierungsalgorithmen möglich sind, die beispielsweise Duplikate entfernen und so das Generieren nicht benötigter lokaler Variablen verhindern.

4.6.3 Analyse der lokalen Variablen für Signalparameter

Modelle können Message-Queues enthalten, die mit der Produktionsregel *DVFSignalDeclare* umgesetzt sind. Jede Queue speichert Signale, die sich aus einer Menge von Variablen zusammensetzen. Mit dem Statement *recv()* kann ein Signal von einer Message-Queue abgerufen werden. Zum Speichern des Signals müssen entsprechende Variablen im vom Transformator generierten Promela-Modell vorhanden sein. Dieser Abschnitt stellt deshalb eine Analyse vor, die berechnet, wie viele Variablen zum Speichern der abgerufenen Signale in einem Modell erzeugt werden müssen. Das folgende Modell dient zur Veranschaulichung des Analyseverfahrens:

```
1 signal queue{ byte p1 };
2 byte a = 0;
3 //...
4 send(queue, 1);
5 recv(queue);
6 a = queue.p1;
7 }
```

Listing 4.57: Empfang eines Signals mit einem Parameter

Das Modell im obigen Listing ist mit einer fiktiven DSL und dem DVF umgesetzt. In dem Beispiel werden die Variable *a* und eine Message-Queue deklariert. Darauf folgen eine Reihe von Statements, die ein Signal auf der Queue speichern, es danach abrufen und das Ergebnis in *a* speichern. Nach der Ausführung des Modells nimmt *a* den Wert 1 an. Bei der Überführung nach Promela muss somit eine Variable vom Typ Byte generiert werden, die die Nachricht abspeichert, die von *recv()* in Zeile 5 abgerufen wird.

Die dafür notwendige Analyse ähnelt dem Verfahren aus dem vorangegangenen Abschnitt 4.6.2. Damit berechnet werden kann, wie viele Variablen zum Speichern der Signale notwendig sind, muss über alle Statements vom Typ *recv()* iteriert werden. Zu diesem Zweck iteriert der Transformator über alle Threads und die darin enthaltenen Statements. Wenn es sich um ein Receive-Statement handelt, werden die notwendigen Variablen zusammen mit dem Thread in eine Tabelle eingetragen. Anhand der so gewonnenen Tabelle kann der Transformator, wenn Threads und Statements nach Promela übersetzt werden, entsprechende lokale Variablen generieren bzw. zum Speichern der Signale verwenden.

4.6.4 Transformation von Aufzählungen

Nachdem die Analyse abgeschlossen ist, beginnt der Transformator mit der eigentlichen Übersetzung. Dazu wird auf den AST des Modells zugegriffen und das Ergebnis in eine Ausgabedatei, wie beispielsweise *result.pml*, geschrieben. In einem ersten Übersetzungsschritt werden die Aufzählungen des DVFs nach Promela überführt. Das folgende Beispiel dient zum Erläutern des Übersetzungsprozesses:

```
1 enum Language{ german, english };
2 enum OS{ openbsd, linux };
```

Listing 4.58: Aufzählungen

Im obigen Listing 4.58 ist eine DSL zu sehen, die mit dem DVF umgesetzt ist und zwei Aufzählungen deklariert. Jede Aufzählung beginnt mit dem Schlüsselwort *enum*. Darauf folgen der Bezeichner und die in der Aufzählung enthaltenen Elemente. In Promela können Aufzählungen mit einem Sprachkonstrukt namens *mtype* beschrieben werden. Deshalb zeigt dieser Abschnitt eine mögliche Transformation von *enum* nach *mtype*. Beide Sprachkonstrukte unterscheiden sich in den folgenden Punkten:

- *Enum* unterstützt Scopes. Dies bedeutet für Listing 4.58, dass beispielsweise das Element *linux* nur im Scope *OS* gültig ist. Somit muss mittels *OS.linux* auf das entsprechende Element zugegriffen werden.
- *Mtype* unterstützt keine Scopes. Wenn somit eine Aufzählung deklariert wird, die das Element *linux* enthält, so ist dies im gesamten Promela-Modell gültig.

Dieser Unterschied muss vom Transformator berücksichtigt werden, um Namenskonflikte zu vermeiden.

Das Konzept des Übersetzungsalgorithmus ist in Abbildung 4.14 als Aktivitätsdiagramm zu sehen. Der Transformator iteriert über den gesamten Abstract Syntax Tree und überprüft bei jedem Objekt, ob es sich um das Element einer Aufzählung handelt. Wenn es ein Enum-Element ist, wird es mit dem folgenden Bezeichner nach Promela transformiert: <Name der Aufzählung>_<Name des Elements>
Durch diese Konkatenation ist sichergestellt, dass es zu keinen Konflikten kommt, wenn in zwei Aufzählungen Elemente existieren, die den selben Bezeichner haben. Ein Problem ist, dass durch dieses Vorgehen lange Variablennamen entstehen können. Für zukünftige Arbeiten ist es deshalb wünschenswert, einen Algorithmus zu entwickeln, der die Bezeichner bei Bedarf kürzt, ohne dass es zu Konflikten kommt. Basierend auf dem vorgestellten Ansatz wird Listing 4.58 vom Transformator in das folgende Promela-Modell übersetzt:

```
1 mtype = {Language_german, Language_english
2          OS_openbsd, OS_linux}
```

Listing 4.59: Transformation von Aufzählungen nach Promela

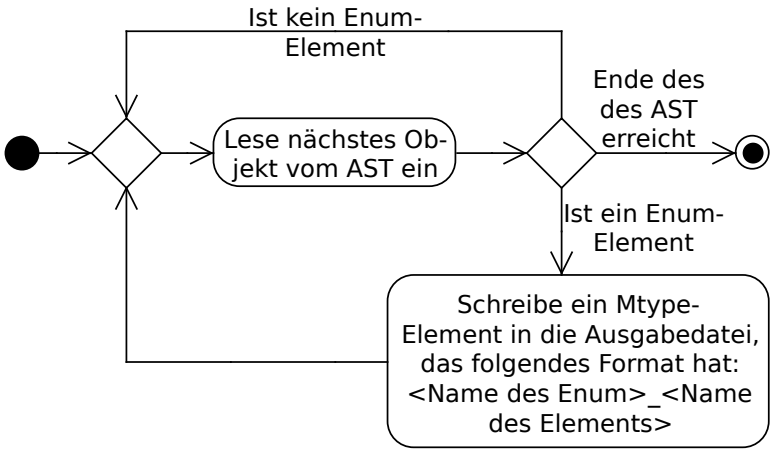

Abbildung 4.14: Transformation von Aufzählungen (Aktivitätsdiagramm)

Das obige Listing 4.59 zeigt, wie der Transformator die Aufzählungen *Language* und *OS* nach Promela überführt. Dafür wird eine Mtype-Umgebung erzeugt, die die Aufzählungselemente mit den entsprechenden Bezeichnern enthält.

4.6.5 Transformation von Zeichenketten

Domänenspezifische Sprachen, die das DVF nutzen, können neben Aufzählungen auch Zeichenketten enthalten. Deshalb wird in diesem Abschnitt eine Transformation des DVF-Datentyps *String* nach Promela vorgestellt. Promela unterstützt nativ keine Zeichenketten oder vergleichbare Elemente. Daher gibt es zwei Möglichkeiten, wie die Strings des DSL Verification Frameworks nach Promela transformiert werden können:

- Für jede Zeichenkette wird ein Byte-Array angelegt. Jedes Element des Arrays entspricht dem Ascii-Wert [96] eines Buchstabens der Zeichenkette. Das Übersetzen von Zeichenketten in ein Bytearray hat den Vorteil, dass keine Informationen verloren gehen und alle Operationen möglich sind, die auch aus Hochsprachen wie beispielsweise Java bekannt sind: Lexikalischer Vergleich, Konkatenieren, usw. Der Nachteil ist jedoch, dass jede Zeichenkette in Form eines Bytearrays die Größe des Zustandsraums beeinflusst und so zur *State Space Explosion* führen kann.

- Es wird eine Aufzählung vom Typ *mtype* angelegt und jede Zeichenkette des Modells in ein Mtype-Element überführt. Der Nachteil dieses Vorgehens ist, dass Zeichenketten nur auf lexikalische Äquivalenz hin untersucht werden können. Andere Operationen, wie beispielsweise ein lexikalischer Vergleich, sind nicht möglich. Der Vorteil ist, dass sich der Zustandsraum pro Zeichenkette nur um ein Byte vergrößert

und so das Risiko der State Space Explosion verringert wird. Zu beachten ist, dass der Model Checker Spin lediglich 255 Mtype-Elemente unterstützt. Für zukünftige Arbeiten sollte der Transformator deshalb so erweitert werden, dass er bei mehr als 255 Zeichenketten mit einer alternativen Modellierung, wie beispielsweise der Define-Direktive und dem Präprozessor, arbeitet.

Da in dieser Arbeit ein besonderer Schwerpunkt auf der Minimierung des Zustandsraums liegt, wird der zweite Ansatz verwendet und somit Zeichenketten in Mtype-Elemente überführt. Daher gliedert sich der vorliegende Abschnitt wie folgt: Zunächst wird ein Konzept vorgestellt, das die Übersetzung von Zeichenketten in Mtype-Elemente beschreibt. Darauf folgt die Betrachtung eines Sonderfalls, nämlich Zeichenketten in Methoden zur eigenschaftserhaltenden Abstraktion (vgl. Listing 4.39). Abschließend wird exemplarisch die Erweiterung der Transformation gezeigt, damit zusätzliche Operationen, wie beispielsweise das Abfragen der Länge einer Zeichenkette, möglich sind.

Die Übersetzung von Zeichenketten gestaltet sich analog zu der Transformation von Aufzählungen aus dem vorangegangenen Abschnitt. Der Transformator iteriert über das gesamte Modell und überprüft bei jedem Objekt, ob es sich um eine Zeichenkette handelt. Wenn das Element einem String entspricht, wird es in eine Mtype-Umgebung eingetragen. Das bedeutet für ein Modell, das die Zeichenketten „steve“ und „bill“ enthält, wird die folgende Promela-Ausgabedatei generiert:

```
1 mtype = {STRING_steve, STRING_bill, STRING_UNKNOWN}
```

Listing 4.60: Transformation von Zeichenketten nach Promela

Das obige Listing 4.60 zeigt, dass den Bezeichnern das Prefix *STRING* vorangestellt worden ist. Dies hat den Vorteil, das Kollisionen mit gleichnamigen Aufzählungselementen vermieden werden.

Im obigen Listing fällt auf, dass neben den beiden Zeichenketten auch das Element *STRING_UNKNOWN* generiert worden ist. Dies ist zum Umsetzen von Methoden zur eigenschaftserhaltenden Abstraktion (vgl. Listing 4.39) notwendig und wird mit dem folgenden Beispiel genauer verdeutlicht: Gegeben sei eine mit dem DVF implementierte domänenspezifische Sprache zur Beschreibung von Benutzerschnittstellen. Mit der Sprache wird eine Login-Maske modelliert. Diese ermöglicht das Eingeben eines Benutzernamens und eines Passworts. Der Benutzername und das Passwort werden von Methoden zurückgegeben, die eigenschaftserhaltende Abstraktion nutzen. Die Eingabe des Benutzernamens ist in Abbildung 4.15 schematisch als Aktivitätsdiagramm dargestellt. Gültige Benutzernamen sind die Zeichenketten aus Listing 4.60. Ein Anwender könnte jedoch auch einen ungültigen Benutzernamen, wie beispielsweise „linus“ übergeben. Eine Methode zur eigenschaftserhaltenden Abstraktion, die den Benutzernamen als Zeichenkette zurückgibt, muss also auch ungültige Eingaben berücksichtigen. Genau diese Klasse von Zeichenketten repräsentiert das Element *STRING_UNKNOWN*. Somit können

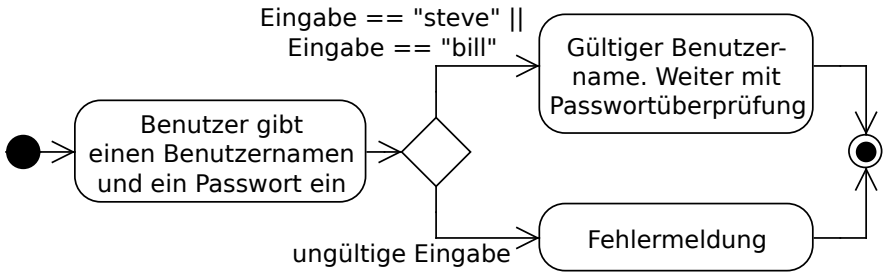

Abbildung 4.15: Eingabe eines Benutzernamens (Aktivitätsdiagramm)

die entsprechenden Methoden, neben denen im Modell enthaltenen Zeichenketten, auch *STRING_UNKNOWN* zurückgeben. Dadurch wird es möglich, dass der Model Checker das Verhalten eines Modells für alle möglichen Benutzereingaben verifiziert. Eine genaue Beschreibung der Transformation von Methoden zur eigenschaftserhaltenden Abstraktion unter Berücksichtigung von Zeichenketten erfolgt in Abschnitt 4.6.9.

Die Transformation in Mtype-Elemente hat den Vorteil, dass der Zustandsraum in nur sehr geringem Maße vergrößert wird. Dies begründetzt sich durch die Tatsache, das eine Mtype-Variable lediglich eine Größe von einem Byte aufweist. Im Vergleich dazu hätten Byte-Arrays, um die Namen „steve“ und „bill“ zu speichern, jeweils eine Größe von 5 und 4 Byte. Ein Nachteil, der durch die Nutzung von Mtype-Elementen entsteht, ist die reduzierte Anzahl von Operationen, die mit den Zeichenketten durchgeführt werden können. So können die Strings aus Listing 4.60 lediglich auf Gleichheit hin untersucht werden. Bei Bedarf kann die Implementierung des DVF-Transformators jedoch für zukünftige Arbeiten mit nur geringem Aufwand erweitert werden, damit auch weitere Operationen auf Zeichenketten möglich sind. Dazu gehört das Abfragen der Länge von Strings, das Konkatenieren oder lexikalische Vergleiche. Deshalb zeigt das folgende Promela-Beispiel exemplarisch, wie Zeichenketten mit Mtype-Elementen beschrieben und gleichzeitig ihre Länge abgefragt werden kann:

```
1 mtype = {STRING_steve, STRING_bill, STRING_UNKNOWN}
2 byte string_length[2];
3
4 init{
5   string_length[STRING_steve] = 5;
6   string_length[STRING_bill] = 4;
7 }
```

Listing 4.61: Transformation von Zeichenketten nach Promela

Im obigen Listing wird ein Byte-Array angelegt, das die Größen der einzelnen Zeichenketten enthält. Es vergrößert den Zustandsraum um zwei Bytes. Der Vorteil ist, dass

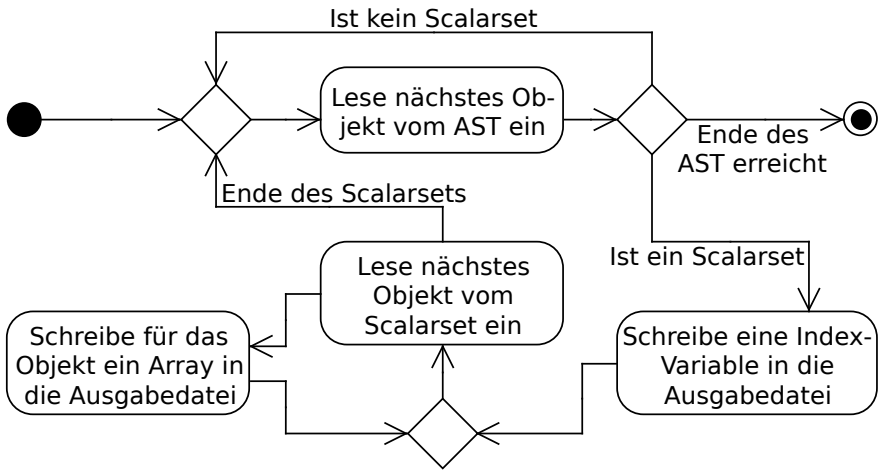

Abbildung 4.16: Transformation von Scalarsets (Aktivitätsdiagramm)

durch diese Art der Modellierung auch die Länge der Zeichenketten Teil des Modells ist. Wenn beispielsweise die Größe der Zeichenkette „steve“ abgefragt werden soll, muss lediglich folgendes Statement genutzt werden: *string_length[STRING_steve].*

4.6.6 Transformation von Scalarsets

Dieser Abschnitt stellt einen Übersetzungsalgorithmus vor, um die symmetrischen Arrays des DVF nach Promela zu überführen. Er basiert auf dem von Donaldson et al. [42] [43] entwickelten Verfahren (vgl. Abschnitt 3.1). Daraus geht hervor, dass Scalarsets in zwei Promela-Elemente überführt werden:

- Jedes Scalarset wird in entsprechende Promela-Arrays übersetzt.
- Es wird eine Variable zum Indizieren der Arrays angelegt.

Im weiteren Verlauf dieses Abschnitts wird zunächst der sich daraus ableitende Transformationsalgorithmus genauer vorgestellt. Danach verdeutlicht ein konkretes Beispiel das Verfahren. Der Übersetzungsalgorithmus ist in Abbildung 4.16 schematisch als Aktivitätsdiagramm zu sehen. Daraus geht hervor, dass der Transformator über den gesamten AST iteriert. Bei jedem Objekt wird überprüft, ob es sich um ein Scalarset handelt. Sobald ein Scalarset gefunden wird, schreibt der Transformator eine Variable in die Ausgabedatei, mit der die symmetrischen Arrays indiziert werden müssen. Danach wird über die im Scalarset enthaltenen Elemente iteriert und für jedes Objekt ein Promela-Array in die Ausgabedatei geschrieben. Der Transformator terminiert, wenn er das Ende des Abstract Syntax Trees erreicht hat.

Das generierte Array hat einen Datentyp und eine bestimmte Größe. Die vom Transformator generierten Datentypen ergeben sich aus der folgenden Tabelle:

DVF	Promela
bool	bool
byte	byte
int	int
string	mtype
enum	mtype

Bool, Byte und Integer können direkt in die entsprechenden Promela-Datentypen überführt werden. Für Zeichenketten und Aufzählungen erfolgt, wie in Abschnitt 4.6.4 bzw. 4.6.5 beschrieben, eine Transformation nach Mtype. Die Größe der Arrays ergibt sich gemäß Unterkapitel 3.1 aus der Summe aller Scalarsets. Sie wird von der Analyse in Abschnitt 4.6.1 ermittelt und kann somit vom Transformator direkt genutzt werden. Das folgende Beispiel verdeutlich den Ansatz:

```
1 scalarset scalar1[2]{
2   byte a;
3   byte b;
4 }
5
6 scalarset scalar2[3]{
7   byte c;
8 }
```

Listing 4.62: Scalarset-Beispiel

Im obigen Listing 4.62 ist ein Modell zu sehen, dass mit einer DSL und dem DVF umgesetzt ist. Es enthält die Deklarationen von zwei Scalarsets. Das erste Scalarset *scalar1* hat die Größe 2 und enthält zwei Elemente. Das zweite Scalarset *scalar2* enthält ein Element und hat eine Größe von 3. Das Modell aus Listing 4.62 wird vom Transformator mit dem in diesem Abschnitt vorgestellten Algorithmus wie folgt nach Promela überführt:

```
1 byte scalar1_a[6];
2 byte scalar2_b[6];
3 pid scalar1;
4
5 byte scalar2_c[6]
6 pid scalar2;
```

Listing 4.63: Transformation von Scalarsets nach Promela

Bei Betrachtung von Listing 4.63 fällt auf, dass in jedem symmetrischen Array Bereiche existieren, die nicht genutzt werden. So wird beispielsweise in keinem Array auf des Element mit dem Index 0 zugegriffen. Diese Elemente sind durch die interne Arbeitsweise von TopSpin notwendig (vgl. Abschnitt 3.1) und vergrößern den Zustandsraum. Hier ist es somit für zukünftige Arbeiten wünschenswert Spin so zu erweitern, dass nativ Array-Symmetrie unterstützt wird und die Nutzung von TopSpin entfällt.

4.6.7 Transformation von lokalen Variablen

Das DVF stellt Produktionsregeln zur Verfügung, mit denen Klassen in domänenspezifischen Sprachen beschrieben werden können. Die Klassen des DVF bestehen aus lokalen Variablen und einer Menge von Methoden. Dieser Abschnitt stellt einen Algorithmus vor, um die lokalen Variablen einer Klasse nach Promela zu überführen. Die Transformation der Methoden wird genauer in Unterkapitel 4.6.9 beschrieben.

Der Übersetzungsalgorithmus der lokalen Variablen wird in diesem Abschnitt anhand des folgenden Beispiels verdeutlicht:

```
1 class A{
2   bool a;
3   byte b[3];
4 }
```

Listing 4.64: Lokale Variablen in Klassen

Das obige Listing 4.64 zeigt ein Modell, das mit einer fiktiven domänenspezifischen Sprache und dem DSL Verification Framework umgesetzt ist. Es wird eine Klasse deklariert, die die Variablen a und b enthält. Das Modell hat die folgende Semantik: Die Klasse A kapselt zwei Attribute und es können Instanzen von ihr erzeugt werden.

Spins Eingabesprache enthält keine Unterstützung für Klassen oder objektorientierte Konzepte. Promela beinhaltet jedoch ein Sprachkonstrukt namens *typedef*, mit dem Variablen gekapselt werden können, um zusammengesetzte Datentypen zu erzeugen. Dies entspricht der Semantik von lokalen Variablen in den Klassen des DVF. Deshalb überführt der in diesem Abschnitt vorgestellte Algorithmus die lokalen Variablen einer Klasse in Typedef-Elemente.

Das Vorgehen des Transformators ist schematisch in Abbildung 4.17 als Aktivitätsdiagramm zu sehen. Zunächst wird über den gesamten AST iteriert und bei jedem Objekt überprüft, ob es sich um eine Klasse handelt. Wenn eine Klasse gefunden wird, schreibt der Transformator ein Typedef-Element in die Ausgabedatei. Danach beginnt eine Iteration über alle lokalen Variablen, die Teil der Klasse sind. Jede Variable wird in die zuvor generierte Typedef-Umgebung eingetragen. Der Transformator terminiert, wenn er das Ende des Abstract Syntax Trees erreicht. Auf Basis dieses Algorithmus wird das Modell in Listing 4.64 in das folgende Promela-Modell überführt:

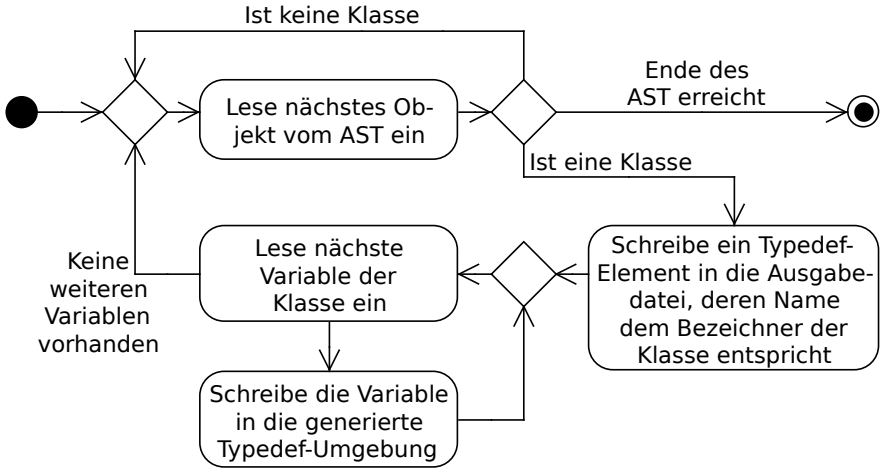

Abbildung 4.17: Transformation von Klassen nach Promela (Aktivitätsdiagramm)

```
1 typedef A{
2   bool a;
3   byte b[3];}
```

Listing 4.65: Transformation einer Klasse nach Promela

Eine Besonderheit, die beim Tranformieren von Variablen beachtet werden muss, sind Boolean-Arrays. In der Arbeit von Ruijs [127] wird darauf hingewiesen, dass derartige Arrays beim Verifizieren von Spin als Bytes betrachtet werden. Daraus ergibt sich das folgende Problem: Gegeben sei eine DSL, die mit dem DVF umgesetzt ist und das Deklarieren von Arrays ermöglicht. Gegeben sei weiterhin das folgende Modell:

```
1 bool a[8];
2 //...
3 a[3] = false;
```

Listing 4.66: DSL mit einem Boolean-Array

Das obige Listing enthält ein Array und eine Zuweisung. Bei einer direkten Übersetzung nach Promela entsteht ein Boolean-Array mit acht Elementen. Spin behandelt jedes Element des Arrays als ein Byte [68]. Somit vergrößert das Array a den Speicherverbrauch des Zustandsvektors um acht Bytes, obwohl für die darin enthaltenen acht Bits lediglich ein Byte benötigt wird. Deshalb wird das Boolean-Array aus Listing 4.66 vom Transformator gemäß [127] wie folgt nach Promela überführt:

```
1 #define SET_0(bv,i) bv=bv&(~(1<<i))
2 #define SET_1(bv,i) bv=bv|(1<<i)
```

```
3 byte a;
4 //...
5 SET_0(a,2);
```

Listing 4.67: Transformation von Boolean-Arrays nach Promela

Der Transformator fasst jeweils acht Elemente eines Boolean-Arrays zu einem Byte zusammen. Das Array a wird somit zu genau einem Byte. Der Zugriff auf die einzelnen Bits erfolgt mittels Makros gemäß des Vorgehens aus [127]. Der Vorteil dieses Ansatzes ist, dass a aus Listing 4.67 den Zustandsvektor um ein Byte vergrößert und nicht, wie im Fall eines Boolean-Arrays, um acht. Zu beachten ist, dass der Transformationsalgorithmus für Boolean-Arrays vom DVF auch für globale Variablen angewendet wird.

4.6.8 Transformation von globalen Variablen

Das DVF beinhaltet nicht nur lokale Variablen als Teil von Klassen. Auch das Deklarieren globaler Variablen ist seitens des DSL-Anwenders möglich. Deshalb stellt dieser Abschnitt einen Algorithmus vor, mit dem die globalen Variablen eines Modells nach Promela überführt werden können. Das folgende Beispiel wird genutzt, um den Ansatz zu demonstrieren:

```
1 class A{
2   byte byte_var[3];
3 }
4
5 signal queue{byte p1, byte p2};
6 A a;
7 byte b;
```

Listing 4.68: DSL mit verschiedenen globalen Variablen

Das obige Listing 4.68 enthält eine domänenspezifische Sprache, die mit dem DVF umgesetzt ist. Sie ermöglicht das Deklarieren von Klassen und globalen Variablen. In dem gezeigten Modell sind die Klasse A, der Message-Queue *queue* und zwei Variablen enthalten. Die Variable a enthält eine Referenz auf eine Instanz von A.

Abbildung 4.18 zeigt einen Übersetzungsalgorithmus als Aktivitätsdiagramm, der Modelle mit globalen Variablen, wie beispielsweise in Listing 4.68, nach Promela überführt. Zu diesem Zweck iteriert der Transformator über den gesamten AST und überprüft, ob es sich um eine globale Variable handelt. Für jede gefundene Variable wird eine Fallunterscheidung durchgeführt:

- Wenn es sich um eine Message-Queue handelt, wird er in einen Promela-Channel übersetzt. Promela-Channel finden Verwendung, da sie asynchrone Kommunikation zwischen nebenläufigen Prozessen ermöglichen, was auch dem Anwendungsfall

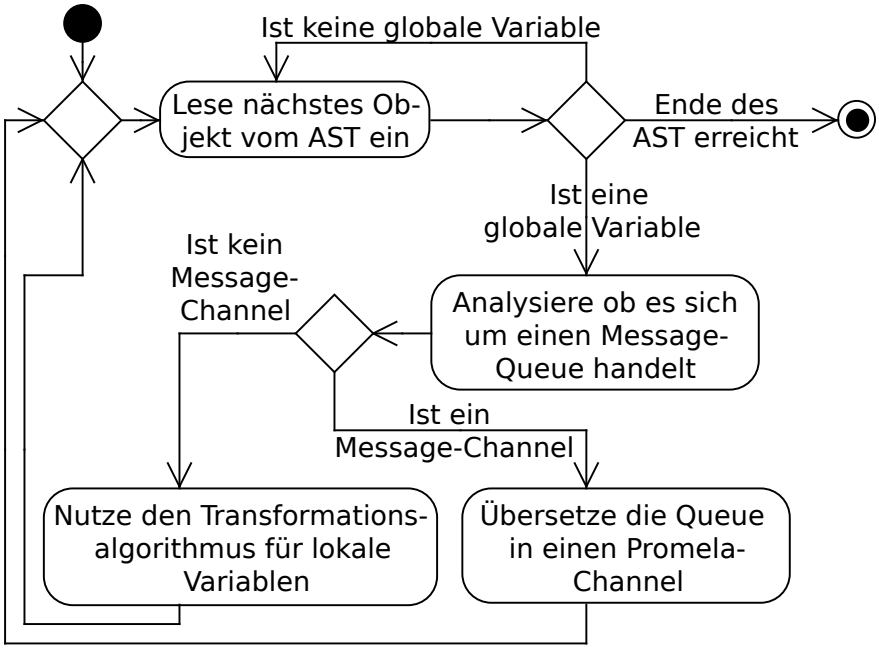

Abbildung 4.18: Transformation von globalen Variablen (Aktivitätsdiagramm)

der Message-Queues entspricht. Für jeden Message-Queue wird vom Transformator der folgende Promela-Code generiert: <Name der Variable> = { <Signal-Parameter> }.

- Wenn es sich beim Datentyp um eine Zeichenkette, Aufzählung, Bool, Byte, Integer oder Klasse handelt, wird die Transformation analog zu den lokalen Variablen aus Abschnitt 4.6.7 durchgeführt.

Beim Betrachten des Übersetzungsalgorithmus fällt auf, dass globale Variablen, im Gegensatz zu lokalen Variablen, Message-Queues enthalten. Message-Queues können im DVF nur global deklariert werden, da sie die Kommunikation zwischen verschiedenen Prozessen ermöglichen. Zu diesem Zweck müssen Prozesse lesend und schreibend auf die entsprechenden Queues zugreifen können. Dies ist jedoch nur möglich, wenn ein Queue global deklariert wird und in allen Scopes des Modells gültig ist.

Wenn Listing 4.68 unter Verwendung des in diesem Abschnitt beschriebenen Transformationsalgorithmus nach Promela überführt wird, entsteht das folgende Modell:

```
1 #define CHANSIZE 2
2
3 typedef A{
```

```
4   byte byte_var[3];
5 }
6
7 A a;
8 chan queue = [CHANSIZE] of {byte, byte}
9 byte b;
```

Listing 4.69: Transformation von globalen Variablen nach Promela

Im obigen Listing sind einige Elemente generiert worden, die einer genaueren Erklärung bedürfen. Promela-Channel sind asynchron und können eine bestimmte Menge an Nachrichten aufnehmen. Channel blockieren, wenn diese Menge erreicht ist und noch weitere Nachrichten empfangen werden. Die Größe der Channel wird vom Transformator mit einer Konstanten standardmäßig auf zwei gesetzt. Dies ist möglich, da Spin einen Präprozessor nutzt (vgl. Listing 4.69 Zeile 1 und 8). Wenn die Channel-Größe beim Verifizieren zu Deadlocks führt, meldet der Model Checker einen Fehler. In diesem Fall muss der DSL-Anwender die Konstante modifizieren und ermittelt so, mit Hilfe der formalen Verifikation, die optimale Channel-Größe für sein Modell.

4.6.9 Transformation von Methoden

Das DVF stellt dem DSL-Entwickler eine Produktionsregel zur Verfügung, mit dem Klassen in eine domänenspezifische Sprache integriert werden können. In Kapitel 4.6.7 wird die Transformation der darin enthaltenen Variablen beschrieben. Die Klassen des DVF können neben Attributen auch Verhalten in Form von Methoden enthalten. Daher stellt dieser Abschnitt einen Transformationsalgorithmus vor, um die Methoden des DVF nach Promela zu überführen. Das folgende Beispiel dient zur Demonstration des Verfahrens:

```
1 class Counter{
2   byte count=0;
3
4   byte inc(byte p1){
5     count = count + p1;
6     return count;
7   }
8 }
9
10 Counter counter;
11
12 enum Temperature{cool, hot}
13 class TempSensor{
14   abstraction Temperature read(){}}
```

Listing 4.70: Funktionen in einer Klasse

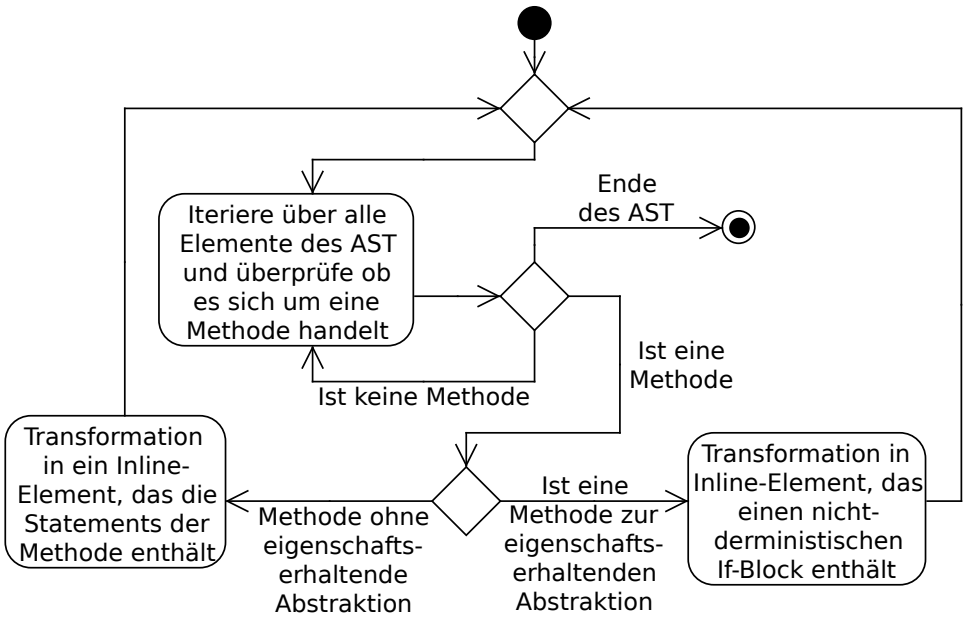

Abbildung 4.19: Transformation von Methoden (Aktivitätsdiagramm)

Im obigen Listing 4.70 werden zwei Klassen deklariert. Die erste repräsentiert einen Zähler, der von der Methode *inc()* inkrementiert wird. Im Anschluss gibt *inc()* den aktuellen Wert des Zählers zurück. Die Klasse *TempSensor* ist die Schnittstelle für einen Temperatur-Sensor, der von der Methode *read()* ausgelesen werden kann. Wenn die Temperatur sich in einem Wertebereich befindet, in dem keine Überhitzung droht, liefert *read()* den Wert *cool* zurück. Wenn die Temperatur hingegen einen bestimmten Schwellwert überschritten hat, ist der Rückgabewert *hot*.

Abbildung 4.19 zeigt einen Übersetzungsalgorithmus als Aktivitätsdiagramm, der Methoden und somit Modelle wie in Listing 4.70 nach Promela überführt. Dazu iteriert der Transformator zunächst über den gesamten Abstract Syntax Tree eines Modells und überprüft bei jedem Objekt, ob es sich um eine Methodendeklaration handelt. Für jede gefundene Methode wird eine Fallunterscheidung durchgeführt:

- Der Methodenkörper enthält Statements und die Methode setzt somit keine eigenschaftserhaltende Abstraktion um.
- Die Methode setzt eigenschaftserhaltende Abstraktion um und ihr ist das Schlüsselwort *abstraction* vorangestellt.

Promela stellt keine nativen Sprachkonstrukte zur Verfügung, um Methoden zu beschreiben. Daher müssen die Methoden des DVF mit alternativen Promela-Elementen nachge-

bildet werden. Für die Umsetzung finden gemäß Abbildung 4.19 Inline-Sprachkonstrukte [68] Verwendung. Dies begründet sich wie folgt: Die Inline-Elemente von Promela sind vergleichbar mit Makros aus der Hochsprache C [78]. Das bedeutet, eine Inline-Instruktion besteht aus einem eindeutigen Bezeichner, einer Liste von Parametern und einer Menge von Statements. Sie unterscheiden sich somit nur in zwei Punkten von den Methoden des DVF: Inline-Instruktionen haben keinen Rückgabewert und unterstützen keine Klassen bzw. die darin enthaltenen lokalen Variablen.

Daher werden Methoden ohne eigenschaftserhaltende Abstraktion wie folgt in Inline-Elemente übersetzt:
<Name der Klasse>_<Name der Methode>(<Parameter>) { <Statements> }
Die Konkatenation von Klassen- und Methoden-Name ist notwendig, da es in unterschiedlichen Klassen Methoden mit dem selben Namen geben kann und so Namenskollisionen vermieden werden. Die Parameterliste besteht aus den folgenden Elementen:
<Referenz auf Objekt>, <Referenz auf Rückgabevariable>, <Methodenparameter>
Der erste Parameter ist immer eine Referenz des Objekts, das die aufgerufene Methode enthält. Dies ermöglicht den Zugriff auf die entsprechenden lokalen Variablen. Darauf folgt eine Referenz auf die Variable, in der der Rückgabewert gespeichert wird. Den Abschluss bilden die Parameter der aufzurufenden Methode. Die Transformation der Statements im Methodenkörper wird in Abschnitt 4.6.11 vorgestellt.

Auch Methoden mit eigenschaftserhaltender Abstraktion werden vom Transformator in Inline-Elemente überführt. Der Methodenkörper enthält jedoch keine Statements, sondern einen nicht-deterministischen If-Block. Der If-Block wählt zufällig einen der möglichen Rückgabewerte aus. Das Generieren des Zufallswerts geschieht mit dem von Theodorus Ruijs [127] beschriebenen Verfahren, da es verglichen mit anderen Ansätzen den kleinsten Zustandsraum aufweist. Durch diese Art der Umsetzung verifiziert der Model Checker das Verhalten des Modells für alle möglichen Rückgabewerte der Methode, ohne dass diese implementiert werden muss und ermöglicht so die eigenschaftserhaltende Abstraktion in Promela.

Das folgende Beispiel verdeutlicht den Übersetzungsalgorithmus und zeigt, wie Listing 4.70 vom Transformator nach Promela überführt wird:

```
1  inline Counter_inc(reference, ret, p1){
2    reference.count = reference.count + p1;
3    ret = reference.count;
4  }
5
6  mtype = {Temperature_cool, Temperature_hot, Temperature_unknown}
7  inline TempSensor_read(ret){
8    if
9    :: ret = Temperature_cool;
10   :: ret = Temperature_hot;
```

```
11    :: ret = Temperature_unknown;
12    fi
13  }
```

Listing 4.71: Transformation von Methoden nach Promela

Im obigen Listing fehlen die Typedef-Elemente, um das Beispiel übersichtlicher zu gestalten. Die Inline-Umgebungen *Counter_inc* und *TempSensor_read* repräsentieren die nach Promela überführten Methoden. Die Statements in *Counter_inc()* können mittels *reference* und *ret* auf die lokalen Variablen des Objekts bzw. den Rückgabewert der Methode zugreifen. Der If-Block in *TempSensor_read()* wählt zufällig *Temperature_cool* oder *Temperature_hot* aus. Dabei fällt ein weiterer Wert namens *Temperature_unknown* auf. Dieser repräsentiert analog zu den Zeichenketten aus Abschnitt 4.6.5 alle sonstigen, nicht vorgesehenen Rückgabewerte, wie beispielsweise eine Fehlfunktion des Sensors.

4.6.10 Transformation von Threads

Das DSL Verification Framework beinhaltet neben den Produktionsregeln, die ein DSL-Entwickler in seine Grammatik integriert, auch die Sprachkonstrukte der Control Flow Intermediate Language. Die CFIL wird im Rahmen dieser Arbeit wie folgt verwendet: Der DSL-Entwickler implementiert für seine domänenspezifische Sprache einen Transformator, der alle Sprachkonstrukte, die nicht Teil des DVF sind, in die Control Flow Intermediate Language überführt. Das Ergebnis ist ein Modell, das ausschließlich die Sprachkonstrukte des DVF und der CFIL beinhaltet. Somit müssen die in diesem Abschnitt vorgestellten Transformatoren nicht nur die Sprachkonstrukte des DVF, wie beispielsweise Variablen oder Klassen, nach Promela bzw. Java überführen, sondern auch die Elemente der Control Flow Intermediate Language. Dazu gehören gemäß Abschnitt 4.4, neben speziellen Statements, auch die Sprachkonstrukte *Thread* und *Statementblock*. Daher stellt dieses Unterkapitel einen Übersetzungsalgorithmus vor, mit dem Thread und Statementblock nach Promela überführt werden können. Zur besseren Veranschaulichung enthält das folgende Listing ein Beispiel:

```
1  thread Counter entry A(byte addme){
2    byte counter=0;
3
4    statementblock B{
5      counter = counter + addme;
6      exit;
7    }
8
9    statementblock A{
10     counter = counter + 1;
```

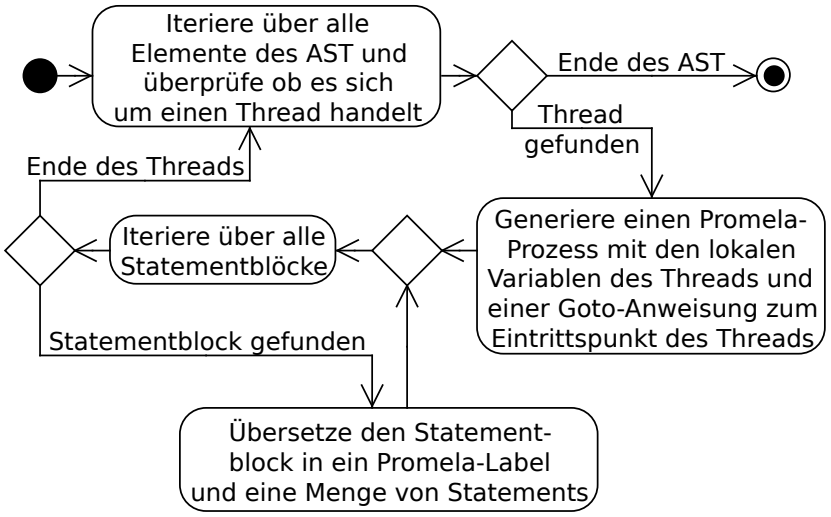

Abbildung 4.20: Transformation von Threads (Aktivitätsdiagramm)

```
11      goto B;
12    }
13  }
```

Listing 4.72: Modellierung eines Threads

Im obigen Listing 4.72 ist ein Modell zu sehen, das Sprachkonstrukte der CFIL beinhaltet. Es besteht aus dem Thread *Counter*, der zwei Statementblöcke ausführt und beim Starten ein Byte als Parameter übergeben bekommt. Das Beispiel wird im weiteren Verlauf dieses Abschnitts genutzt, um den Transformationsalgorithmus zu veranschaulichen.

Die Übersetzung von Threads und Statementblöcken ist in Abbildung 4.20 schematisch als Aktivitätsdiagramm dargestellt. Der Transformator iteriert zunächst über den gesamten AST und überprüft bei jedem Objekt, ob es sich um einen Thread handelt. Jeder Thread wird in einen Promela-Prozess überführt. Dies begründet sich durch die Tatsache, dass es sich bei den Threads des DVF um nebenläufige Elemente handelt und Promela für die Modellierung von Nebenläufigkeit das Sprachkonstrukt *Process* zur Verfügung stellt. Der erzeugte Promela-Prozess beginnt mit den lokalen Variablen. Für deren Erzeugung findet der Algorithmus aus Abschnitt 4.6.7 Verwendung. Darauf folgt eine Goto-Anweisung für einen Sprung zum Entrypoint-Statementblock.

Zum Abschluss iteriert der Transformator über alle im Thread enthaltenen Statementblöcke. Für jeden Statementblock wird zunächst ein Promela-Label erzeugt. Dies hat den Vorteil, dass der entsprechende Statementblock durch Goto-Anweisungen erreichbar ist. Danach erfolgt eine Übersetzung der einzelnen Statements. Eine genaue Vorstellung des

Transformationsalgorithmus für Statements erfolgt in Abschnitt 4.6.11.

Das nächste Beispiel zeigt wie das Modell aus Listing 4.72 vom dem in diesem Unterkapitel vorgestellten Transformationsalgorithmus nach Promela überführt wird:

```
1  process Counter(byte addme){
2    byte counter;
3    goto A;
4
5  B:
6    counter = counter + addme;
7    goto end_Counter;
8
9  A:
10   counter = counter + 1;
11   goto B;
12
13 end_Counter:
14   skip;
15 }
```

Listing 4.73: Transformation von Threads nach Promela

Das obige Beispiel enthält den Promela-Prozess *Counter* sowie die beiden Statementblöcke A und B. Damit der Eintrittspunkt A erreicht werden kann, beginnt *Counter* mit einer Goto-Anweisung. Ein Promela-Prozess terminiert erst dann, wenn seine letzte Anweisung ausgeführt ist. Deshalb wird jeder Thread so nach Promela überführt, dass an seinem Ende das Label *end_<Threadname>* steht. Dies ist in Listing 4.73 in Zeile 13 erkennbar. So kann jedes Exit-Statement in eine Sprunganweisung überführt werden, die an das Ende des Prozesses springt und ihn so terminiert.

4.6.11 Transformation von Statements und Expressions

Die vorangegangenen Abschnitte haben gezeigt, wie das DVF Klassen, Variablen, Aufzählungen, Methoden und Threads nach Promela überführt. Zwei weitere, wichtige Sprachkonstrukte des DSL Verification Frameworks sind Expressions und Statements. Daher ist es das Ziel dieses Unterkapitels, einen entsprechenden Übersetzungsalgorithmus vorzustellen.

Damit der Modelltransformator Expressions und Statements nach Promela überführen kann, iteriert er über den gesamten AST. Das Vorgehen ist ähnlich zu den vorangegangenen Abschnitten und wird beispielsweise in Abbildung 4.20 verdeutlicht: Der Transformator überprüft für jedes Objekt im AST, ob es sich um ein Statement oder eine Expression handelt, überführt es nach Promela und schreibt das Ergebnis in die entsprechende

Ausgabedatei. Dabei erleichtern die folgenden beiden Aspekte die Übersetzung:

- Die Operatorpräzedenz muss nicht ermittelt werden, sondern ergibt sich aus der Struktur des AST (vgl. Abschnitt 4.3.3).
- Ein Teil der Statements und Expressions des DVF kann ohne zusätzliche Modifikation nach Promela überführt werden. Dazu gehören alle Operatoren und damit verbundene Zugriffe auf Byte- oder Integervariablen, wie beispielsweise $a = b * (2 + c)$.

Einige Elemente können jedoch nicht direkt nach Promela überführt werden und bedürfen einer Modifkation. Sie werden im Rahmen dieses Unterkapitels genauer vorgestellt.

If-Blöcke

Das DVF ermöglicht mittels *If*, *Elseif* und *Else* das Spezifizieren von bedingter Programmausführung. Die Semantik der Sprachkonstrukte ist wie folgt:

1. Die Expressions in den If- und bzw. Elseif-Blöcken werden in absteigender Reihenfolge evaluiert.
2. Sobald eine Expression den Wert *true* annimmt, werden die in der Verzweigung enthaltenen Statements ausgeführt und abschließend an das Ende der Verzweigung gesprungen.
3. Wenn keine Expression den Wert *true* annimmt und die Verzweigung ein Else-Konstrukt enthält, werden die darin enthaltenen Statements ausgeführt.

Das folgende Beispiel verdeutlicht den Aufbau und die Semantik der bedingten Ausführung im Rahmen des DVF:

```
1 if(a){
2   var=1;
3 }
4 elseif(b){
5   var=2;
6 }
7 else{ var=3; }
8 //...
```

Listing 4.74: If-Blöcke im DVF

Im obigen Listing 4.74 ist ein Modell zu sehen, das einen If-, einen Elseif- und einen Else-Block enthält. Falls die Expression a den Wert *true* annimmt, wird *var* der Wert 1 zugewiesen und im Anschluss die Ausführung des Modells in Zeile 8 fortgesetzt. Der Elseif-Block ist ausführbar, wenn a den Wert *false* und b den Wert *true* annehmen. Auch

in diesem Fall erfolgt nach der Zuweisung $var = 2$ ein Sprung in die Zeile 8 des Modells. Wenn sowohl a, als auch b den Wert *false* zurückliefern, wird var der Wert 3 zugewiesen. Auch in Promela gibt es den If-Block als Sprachkonstrukt. Dieser unterscheidet sich jedoch in den folgenden Aspekten vom DVF:

- Es gibt keinen Elseif- oder Else-Block.
- Ein If-Block kann mehrere Verzweigungen beinhalten. Wenn mehrere Expressions den Wert *true* zurückliefern, erfolgt die Auswahl der Verzweigung nicht-deterministisch.
- Falls keine Verzweigung ausgeführt werden kann, blockiert das Modell bis eine der Expressions den Wert *true* annimmt.

Damit die Semantik des DVF erhalten bleibt, müssen die If-Blöcke des DSL Verification Framework so nach Promela überführt werden, dass kein Nicht-Determinismus oder blockierendes Verhalten entsteht. Das folgende Beispiel verdeutlicht den Algorithmus des Transformators:

```
1  if
2  :: a ->
3       var=1;
4       goto ifblock_exit;
5  :: else -> skip;
6  fi;
7  if
8  :: b ->
9       var=2;
10      goto ifblock_exit;
11 :: else -> skip;
12 fi;
13 var=3;
14 ifblock_exit:
15 //...
```

Listing 4.75: If-Blöcke in Promela

Das obige Modell zeigt, wie der Transformator des DVF das Beispiel aus Listing 4.74 nach Promela überführt. Es wird ersichtlich, dass die If- und Elseif-Blöcke des DVF in separate Promela-If-Blöcke übersetzt werden. Damit keine Blockierungen entstehen, ergänzt der Transformator jeden If-Block durch ein „else − > skip“. Nach der Ausführung einer Verzweigung erfolgt mit einer Goto-Anweisung der Sprung an das Ende des If-Else-Konstrukts. Dies stellt sicher, dass immer nur eine Verzweigung ausführbar ist.

Das Modell ist semantisch äquivalent zu Listing 4.74, da zunächst in Zeile 2 evaluiert wird, ob a den Wert *true* annimmt. Wenn dies der Fall ist, erfolgt die Ausführung von $var = 1$ und ein Sprung an das Ende des If-Else-Konstrukts. Ein derartiges Verhalten entspricht einem If-Block des DVF. Danach wird in Zeile 8 evaluiert, ob b den Wert *true* aufweist. Wenn dies der Fall ist, erfolgt die Zuweisung $var = 2$ und die Terminierung der bedingten Ausführung durch eine Goto-Anweisung. Das Verhalten entspricht somit einem Elseif-Block des DVF. Wenn sowohl a, als auch b den Wert *false* annehmen, wird in Zeile 13 $var = 3$ ausgeführt, was äquivalent zum Else-Block aus Listing 4.74 ist.

Wait-Blöcke

Das DVF stellt ein weiteres Sprachkonstrukt zur Verfügung, mit dem die bedingte Ausführung in einem Modell umgesetzt werden kann. Dabei handelt es sich um die sogenannten Wait-Blöcke. Sie haben die folgende Semantik:

1. Die Expressions in den Wait- bzw. Elsewait-Blöcken werden in absteigender Reihenfolge evaluiert.
2. Wenn dabei eine Expression den Wert *true* annimmt, werden die Statements des Wait- bzw. Elsewait-Blocks ausgeführt und danach die bedingte Ausführung beendet.
3. Wenn alle Expressions den Wert *false* zurückliefern, blockiert das Wait-Elsewait-Konstrukt die Ausführung des Modells. Sobald mindestens eine Expression den Wert *true* annimmt, erfolgt ein Sprung zu 1.

Das folgende Beispiel demonstriert die Nutzung von Wait-Blöcken und dient im weiteren Verlauf dieses Abschnitts zur Veranschaulichung des Transformationsprozesses:

```
1 byte a=0; byte b=0;
2
3 wait(a==1){
4   a=2;
5 }
6 elsewait(b==2){
7   b=3;
8 }
9 //...
```

Listing 4.76: Wait-Blöcke im DVF

Im obigen Listing ist ein Modell zu sehen, das mit einer fiktiven DSL umgesetzt ist und die Sprachkonstrukte des DVFs nutzt. Es enthält einen Wait-Block, der bewirkt, dass die Ausführung erst dann fortgesetzt wird, wenn a den Wert 1 oder b den Wert

2 annehmen. Dies kann beispielsweise nebenläufig durch einen Thread geschehen. Es ist möglich, dass sowohl $a == 1$, als auch $b == 2$ den Wert *true* zurückliefern. Da die Expressions in absteigender Reihenfolge ausgewertet werden, würde in diesem Fall die Ausführung der Verzweigung in Zeile 3 erfolgen. Das folgende Beispiel zeigt, wie Listing 4.76 vom Transformator des DVF nach Promela überführt wird und dient zur Beschreibung des Übersetzungsalgorithmus:

```
1 byte a=0;
2 byte b=0;
3
4 do
5 :: true ->
6   if
7   :: a==1 ->
8     a=2;
9     break;
10  :: else -> skip;
11  fi;
12  if
13  :: b==2 ->
14    b=3;
15    break;
16  :: else -> skip;
17  fi
18 od;
```

Listing 4.77: Wait-Blöcke in Promela

Das obige Listing zeigt, dass für jede Wait-Elsewait-Umgebung eine Do-Schleife generiert und die einzelnen Verzweigungen darin eingebettet werden. Jeder Wait- bzw. Elsewait-Block wird in einen If-Block überführt, der aus zwei Verzweigungen besteht:

- Die erste Verzweigung enthält die Expression und die Statements des Wait- bzw. Elsewait-Blocks. Als letztes Statement ist eine Break-Anweisung eingefügt, die bewirkt, dass die Do-Schleife verlassen und somit ans Ende des Wait-Elsewait-Konstrukts gesprungen wird.
- Wenn die erste Verzweigung nicht ausführbar ist, bewirkt „else − > skip“, dass das nächste Wait- bzw. Elsewait-Konstrukt in Form eines If-Blocks evaluiert wird.

Eine alternative Transformation in einen If-Block ohne Do-Schleife ist nicht zulässig. Dies begründet sich durch die Tatsache, dass die If-Blöcke von Promela nicht-deterministisches Verhalten aufweisen, wenn mehr als eine Verzweigung ausführbar ist. Die Wait-Blöcke

des DVF sind jedoch deterministisch. Das bedeutet, die einzelnen Verzweigungen werden der Reihe nach ausgewertet. Sobald eine Expression den Wert *true* annimmt, erfolgt die Ausführung der Verzweigung und das Verlassen der Wait-Elsewait-Umgebung. Daher ist eine Aufteilung in mehrere If-Blöcke und das Einbetten in eine Do-Schleife erforderlich.

Das Promela-Modell in Listing 4.77 ist semantisch äquivalent zum DVF-Modell, das zu Beginn dieses Abschnitts vorgestellt wird: Aufgrund der Do-Schleife blockiert das Modell, bis $a == 1$ oder $b == 2$ den Wert *true* annehmen. Mit jedem Schleifendurchlauf (sogenanntes *busy waiting*) wird zunächst evaluiert, ob a den Wert 1 enthält. Wenn dies der Fall ist, erfolgt die Zuweisung $a = 2$ und ein Sprung ans Ende der Wait-Elsewait-Umgebung. Dies entspricht im DVF-Modell in Listing 4.76 dem Wait-Block in Zeile 3. Danach wird evaluiert, ob $b == 2$ den Wert *true* zurück gibt. In diesem Fall erfolgt die Zuweisung $b = 3$ und ein Sprung an das Ende der Wait-Elsewait-Umgebung, was semantisch äquivalent zur Elsewait-Verzweigung des DVF-Modells in Listing 4.76 ist.

Synchronize-Umgebungen

Neben den Wait-Blöcken ermöglicht das DSL Verification Framework auch die Modellierung sogenannter Synchronize-Umgebungen. Sie enthalten eine Menge von Statements, die nur ein Prozess gleichzeitig ausgeführen kann. Das folgende Beispiel demonstriert die Verwendung von Synchronize-Umgebungen, um daraus im weiteren Verlauf der Arbeit einen Transformationsalgorithmus abzuleiten:

```
1  byte a=0;
2  run SynchronizeMe();
3  run SynchronizeMe();
4
5  thread SynchronizeMe entry Start(){
6    statementblock Start{
7      synchronize{
8        if(a==0){
9          a=a+1;
10       }
11     }
12   }
13 }
```

Listing 4.78: Synchronize-Umgebungen im DVF

Das obige Listing 4.78 zeigt ein Modell, das aus der globalen Variable a und zwei nebenläufigen Prozessen besteht. Jede Instanz des Prozesses inkrementiert a um 1, wenn die Bedingung $a == 0$ erfüllt ist. Damit a nicht mehrfach inkrementiert werden kann, ist der If-Block Teil einer Synchronize-Umgebung. Das zu generierende Promela-Modell

muss somit die folgende Semantik aufweisen:

- Vor der Ausführung des ersten Statements muss die Synchronize-Umgebung für alle weiteren Prozesse gesperrt werden.
- Nach der Ausführung des letzten Statements muss die Synchronize-Umgebung wieder freigegeben werden.

Zur Abbildung derartig beschränkter Resourcen bietet sich die Nutzung einer Semaphore an. Jede Semaphore kann in Promela gemäß Holzmann et al. [68] mit einem Message-Channel modelliert werden. Somit überführt der Transformator Listing 4.78 in die folgende Promela-Beschreibung:

```
1 byte a = 0;
2 chan semaphor = [1] of { byte };
3
4 //...
5 semaphor!1;
6 if
7 :: a==0 -> a=a+1;
8 :: else -> skip;
9 fi;
10 semaphor?_;
```

Listing 4.79: Synchronize-Umgebung in Promela

Das obige Modell zeigt, dass der Transformator in Zeile 2 einen Message-Channel namens *semaphor* eingefügt hat. *Semaphor* speichert genau eine Nachricht vom Typ *byte*. Durch das blockierende Verhalten des Sende-Operators kann immer nur ein Prozess den If-Block in Zeile 6 bis 9 erreichen. Die Semantik des Promela-Modells ist somit äquivalent zu Listing 4.78.

Daraus ergibt sich der folgende Übersetzungsalgorithmus: Für jede Synchronize-Umgebung generiert der Transformator einen Message-Channel. Des Weiteren fügt er ein Statement ein, das zu Beginn der Synchronize-Umgebung ein *Byte* auf der Message-Queue ablegt. Diese Nachricht wird beim Verlassen der Umgebung wieder entfernt.

While-Schleifen

Das DSL Verification Framework beinhaltet das Sprachkonstrukt *while* zur Umsetzung von Schleifen. Die Syntax und Semantik der While-Schleife ist äquivalent zu den Hochsprachen C und Java: Vor jedem Schleifendurchlauf wird die Expression im Schleifenkopf ausgewertet. Wenn sie den Wert *true* annimmt, erfolgt das Ausführen der im Schleifenkörper enthaltenen Statements. Die Schleife wird so lange durchlaufen, bis die Expression im Schleifenkopf den Wert *false* zurückgibt. Das nächste Listing demonstriert die

konkrete Nutzung der While-Schleife, um im weiteren Verlauf dieses Abschnitts den Transformationsalgorithmus nach Promela zu veranschaulichen:

```
1 byte counter=0;
2 while(counter<10){
3   counter=counter+1;
4 }
```

Listing 4.80: While-Schleife im DVF

Im obigen Beispiel wird die Zählervariable *counter* deklariert und in einer While-Schleife inkrementiert. Wenn *counter* den Wert 10 annimmt, terminiert die Ausführung des Modells. Das nächste Beispiel zeigt, wie der Transformator des DVF die While-Schleife aus Listing 4.80 nach Promela überführt:

```
1 byte counter = 0;
2 do
3 :: counter<10 ->
4       counter=counter+1;
5 :: else -> break;
6 od;
```

Listing 4.81: While-Schleife in Promela

Schleifen können in Promela mit einem Do-Block modelliert werden. Die Transformation gestaltet sich wie folgt: Für jede While-Schleife wird eine Do-Umgebung generiert, die aus zwei Abschnitten besteht. Der erste Abschnitt enthält den Schleifenkopf und Schleifenkörper der While-Schleife. Der zweite Abschnitt besteht aus „else − > break;“ und bewirkt das Terminieren der Schleife, wenn die Bedingung im Schleifenkopf den Wert *false* annimmt. Diese Ergänzung ist notwendig, da die Do-Umgebung ansonsten blockiert, was nicht dem Verhalten einer While-Schleife entspricht.

Die Semantik des Promela-Modells entspricht dem DVF-Beispiel aus Listing 4.80: Die Expression in Zeile 3 bewirkt, dass die Schleife nur dann durchlaufen wird, wenn gilt $counter < 10$. Dies ist äquivalent zum Kopf der While-Schleife in Listing 4.80. Mit jedem Schleifendurchlauf wird *counter* um 1 inkrementiert. Sobald *counter* den Wert 10 erreicht, erfolgt das Verlassen der Do-Umgebung durch die Else-Anweisung in Zeile 5. Auch dies entspricht genau dem Verhalten des Modells aus Listing 4.80.

Versand von Signalen

Neben bedingter Ausführung und Schleifen unterstützt das DVF auch Statements zum Versenden bzw. Empfangen von asynchronen Signalen. Für den Versand findet das Schlüsselwort *send* Verwendung. Das Abrufen von Signalen erfolgt mittels *recv*. Beide Operationen haben die folgende Semantik:

- *Send* legt ein Signal, das sich aus einer Menge von Literalen zusammensetzt, auf einer Message-Queue ab. Queues können nur eine endliche Menge von Signalen aufnehmen. Wenn eine Sende-Operation ausgeführt wird und auf der Queue kein Platz für weitere Nachrichten vorhanden ist, blockiert *send*, bis wieder Speicherplatz zur Verfügung steht.

- *Recv* ruft ein Signal von einer Queue ab. Blockierendes Verhalten tritt auf, wenn die Queue leer ist. In diesem Fall wird die Ausführung des Threads erst dann fortgesetzt, wenn eine Sende-Operation ein Signal auf der Queue gespeichert hat.

Das folgende Beispiel demonstriert die Verwendung von *send* bzw. *recv* und dient im weiteren Verlauf dieses Abschnitts zur Veranschaulichung des Transformationsprozesses:

```
1 signal queue{byte signalvalue};
2
3 thread Sender entry A{
4   send(queue, 1);
5   recv(queue);
6   if(queue.signalvalue==1){
7     //...
8   }
9 }
```

Listing 4.82: Versand von Signalen mit dem DVF

Im obigen Listing 4.82 werden der Thread *Sender* und die Message-Queue *queue* deklariert. *Queue* ermöglicht das Senden bzw. Empfangen von Signalen, die aus einem Byte bestehen. Der Thread *Sender* speichert in Zeile 4 ein Signal auf *queue* und ruft es im Anschluss wieder ab. Die Syntax von *send()* ähnelt einem Methodenaufrauf. Der erste Parameter ist die Ziel-Queue, der zweite das zu versendende Signal. *Recv* bekommt lediglich einen Parameter übergeben, nämlich die Referenz der Queue, von der ein Signal abgerufen werden soll. Auf das zuletzt empfangene Signal kann mit dem Punkt-Operator lesend zugegriffen werden. Dies ist in Listing 4.82 in Zeile 6 verdeutlicht. Das nächste Beispiel zeigt, wie das Modell aus Listing 4.82 vom Transformator des DVF nach Promela überführt wird:

```
1 chan queue = [2] of {byte}
2
3 proctype Sender(){
4   byte queue_param1;
5   queue!1;
6   queue?queue_param1;
7
```

```
8   if
9   :: queue_param1 == 1 ->
10  //...
11  fi;
12 }
```

Listing 4.83: Send-Statement in Promela

Im obigen Listing 4.83 ist gemäß des Transformationsalgorithmus aus Abschnitt 4.6.8 die Message-Queue in einen Promela-Channel überführt worden. Auf das Sprachkonstrukt *chan* kann in Promela mit den Operatoren „?“ und „!“ zugegriffen werden. Deshalb bietet es sich an, *send* bzw. *recv* nach „?“ bzw. „!“ zu transformieren. Dabei muss ein semantischer Unterschied beachtet werden:

- *Recv* speichert implizit die Elemente des zuletzt abgerufenen Signals. Der Zugriff erfolgt mit dem Punkt-Operator (vgl. Listing 4.82 Zeile 6).
- Der Fragezeichen-Operator von Promela benötigt explizit die Namen der Variablen, in denen die Elemente des Signals gespeichert werden sollen.

Aus diesem Grund generiert der Transformator basierend auf der Analyse aus Abschnitt 4.6.3 zusätzliche lokale Variablen, in denen die Elemente eines Signals beim Abrufen gespeichert werden können. Dies ist in Listing 4.83 in Zeile 4 zu sehen, die die Variable *queue_param1* enthält.

Die Send- und Recv-Statements werden vom Transformator in die Promela-Operatoren „!“ bzw. „?“ überführt. Das Ergebnis des Übersetzungsprozesses ist in Listing 4.83 in den Zeilen 5 bis 6 enthalten. Beim Abrufen wird das Signal in der lokalen Variable *queue_param1* zwischengespeichert. Danach kann es im Rahmen der Expression im If-Block in Zeile 9 referenziert werden.

Methodenaufrufe

Dieser Abschnitt zeigt, wie der Transformator des DSL Verification Frameworks Methodenaufrufe nach Promela überführt. Die Semantik von DVF-Methodenaufrufen ist äquivalent zu C [78] bzw. Java [60]: Mit dem Aufruf einer Methode wird die Rücksprungadresse gespeichert. Danach erfolgt die Ausführung der im Methodenkörper enthaltenen Statements. Mit dem Erreichen einer Return-Anweisung terminiert die Methode und die Ausführung des Modells wird an der Rücksprungadresse fortgesetzt. Zusammen mit dem Return-Statement kann ein Rückgabewert spezifiziert werden. Des Weiteren ist beim Aufruf auch das Übergeben von Parametern möglich. Die Bezeichner der Parameter sind nur in dem entsprechenden Methodenkörper gültig. Das folgende Modell beinhaltet Methodenaufrufe und wird im weiteren Verlauf dieses Abschnitts genutzt, um den Transformationsprozess zu veranschaulichen:

```
1 class Add{
2   byte add(byte p1, byte p2){
3     return p1+p2;
4   }
5 }
6
7 thread FSM entry A{
8   byte var1;
9   Add i;
10  //...
11  var1 = i.add(i.add(1,2), i.add(3,4));
12 }
```

Listing 4.84: Aufruf von Methoden im DVF

Im obigen Listing 4.84 werden ein Thread und eine Klasse deklariert. Die Klasse *Add* beinhaltet eine Methode, um die Summe von zwei Werten zu bilden. Der Thread *FSM* ist ein nicht näher spezifizierter Automat, der die Addiererklasse instanziiert und deren Methode *add()* in Zeile 11 aufruft. Das folgende Beispiel zeigt, wie der Transformator des DVF das Modell aus Listing 4.84 nach Promela überführt. Danach wird im weiteren Verlauf dieses Abschnitts der zu Grunde liegende Übersetzungsalgorithmus vorgestellt:

```
1 inline Add_add(reference, p1, p2, ret){
2   ret = p1 + p2;
3 }
4
5 proctype FSM(){
6   byte add_param1, add_param2;
7   byte add_ret1, add_ret2;
8   byte var1;
9   Adder i;
10  //...
11  add_param1 = 1;
12  add_param2 = 2;
13  Add_add(i, add_param1, add_param2, add_ret1);
14  add_param1 = 3;
15  add_param2 = 4;
16  Add_add(i, add_param1, add_param2, add_ret2);
17  Add_add(i, add_ret1, add_ret2, var1);
18 }
19 Add adder;
```

Listing 4.85: Aufruf von Methoden in Promela

Das Modell im obigen Listing 4.85 ist vereinfacht dargestellt, um die Lesbarkeit zu erhöhen. Bei Betrachtung der Zeilen 6 bis 7 fällt auf, dass der Transformator vier zusätzliche lokale Variablen generiert hat. Sie basieren auf dem Analyseverfahren aus Abschnitt 4.6.2, um Methodenparameter bzw. Rückgabewerte zwischenspeichern zu können. Dies ist aus dem folgenden Grund notwendig: Es gibt in Promela keine nativen Elemente zum Modellieren von Methoden bzw. Methodenaufrufen. Methoden werden deshalb vom Transformator (vgl. Abschnitt 4.6.9) mit dem Promela-Sprachkonstrukt *inline* umgesetzt. Inline-Deklarationen haben keinen Rückgabewert. Des Weiteren werden Parameter nicht als Kopie, sondern als Referenz übergeben. Aus diesem Grund müssen vom Transformator zusätzliche Variablen generiert werden, um Parameter und Rückgabewerte zwischenspeichern zu können.

In Listing 4.84 wird in Zeile 11 im Rahmen einer Zuweisung dreimal die Methode *add* aufgerufen. Das Ergebnis der Transformation ist in Listing 4.85 in den Zeilen 11 bis 17 zu sehen. Inline-Aufrufe haben keinen Rückgabewert und können daher nicht als Parameter Verwendung finden. Aus diesem Grund unterteilt der Transformator *„add(add(1,2), add(3,4))“* in drei Inline-Aufrufe:

- Zeile 11 bis 13: Die Parameter 1 und 2 werden in den Variablen *add_param1* und *add_param2* zwischengespeichert. Nach dem Methodenaufruf enthält *add_ret1* den Rückgabewert.
- Zeile 14 bis 16: Die Parameter 3 und 4 werden in den Variablen *add_param1* und *add_param2* zwischengespeichert. Nach dem Methodenaufruf enthält *add_ret2* den Rückgabewert.
- Zeile 17: Die Ergebnisse von *„add(1,2)“* und *„add(3,4)“* werden *add()* als Parameter übergeben und in *var1* gespeichert.

Somit ist nach dem Ausführen von Zeile 17 in *var1* das Ergebnis von *„add(add(1,2), add(3,4))“* enthalten. Dies entspricht genau der Semantik von Listing 4.84.

Zugriff und Abfrage von Scalarsets

In Abschnitt 4.6.6 wird erläutert, wie Scalarset-Deklarationen vom Transformator des DVF nach Promela überführt werden. Neben der Deklaration beinhaltet das DSL Verification Framework auch Statements, um auf Scalarsets lesend bzw. schreibend zuzugreifen. Dazu gehören:

- *All()* entspricht dem Allquantor [82] und liefert *true* zurück, wenn alle Elemente eines symmetrischen Arrays eine bestimmte Eigenschaft erfüllen.
- *Exist()* entspricht dem Existenzquantor [82] und liefert *true* zurück, wenn mindestens ein Element eines symmetrischen Arrays eine bestimmte Eigenschaft erfüllt.

- *WriteAll()* weist allen Elementen eines symmetrischen Arrays einen bestimmten Wert zu.
- *WriteOne()* weist einem zufälligen Element eines symmetrischen Arrays einen bestimmten Wert zu. Ein mehrmaliger Aufruf von *writeOne()* greift immer auf das selbe Element des Arrays zu. Um ein neues zufälliges Element auszuwählen, muss *nextIndex()* aufgerufen werden.

Dieser Abschnitt stellt für *all*, *exist*, *writeAll()*, *writeOne* und *nextIndex()* einen Transformationsalgorithmus vor. Um die Übersetzung besser zu veranschaulichen, wird das folgende Beispiel verwendet:

```
1 scalarset scalar1[3]{
2   byte a;
3   byte b;
4 }
```

Listing 4.86: Scalarsets im DVF

Im obigen Listing 4.86 ist das Scalarset *scalar1* zu sehen. Es enthält die Elemente *a* und *b*. In Abschnitt 4.6.6 wird gezeigt, dass das Scalarset vom Transformator des DVF in die symmetrischen Arrays *scalar1_a*, *scalar2_b* und die Index-Variable *scalar1* überführt wird. Beim Zugriff auf die Arrays *scalar1_a* bzw. *scalar2_b* kann zwischen den folgenden beiden Fällen unterschieden werden:

- Es handelt sich um einen Lesezugriff mittels *exist()* oder *all()*.
- Es handelt sich um einen Schreibzugriff mittels *writeOne()* oder *writeAll()*.

Die Transformation der beiden Fälle wird im weiteren Verlauf dieses Abschnitts genauer vorgestellt.

Bei einem Lesezugriff mit Vergleichsoperatoren liefern *exist()* bzw. *all()* den Wert *true* oder *false* zurück. Eine Überführung nach Promela muss so gestaltet sein, dass sie beispielsweise als Expressions in If-Blöcke eingebettet werden können. Das folgende Listing demonstriert den Transformationsalgorithmus:

```
1 //exist(scalar1.a) == 5
2 scalar1_a[1]==5 || scalar1_a[2]==5 || scalar1_a[3]==5
3 //all(scalar1.b) <= 3
4 scalar1_b[1]<=3 && scalar1_b[2]<=3 && scalar1_b[3]<=3
```

Listing 4.87: Transformation von exist() und all() nach Promela

Im obigen Listing 4.87 sind in den Zeilen 1 und 3 Scalarset-Abfragen mittels *exist* bzw. *all* zu sehen. Die Zeilen 2 und 4 enthalten die entsprechenden Transformationen nach

Promela. Die Abfrage *exist()* liefert *true* zurück, wenn für mindestens ein Element des Scalarsets die spezifizierte Expression „==5“ gilt. Daher wird ein Promela-Ausdruck generiert, der die Expression der Exist-Abfrage für alle Elemente des Scalarsets mit einem Oder-Operator konkateniert. Ein ähnliches Verfahren findet bei *all()* Verwendung. Da in diesem Fall die Expression für alle Elemente des Scalarsets gelten muss, wird hier der Und-Operator genutzt.

WriteOne() und *writeAll()* können den Inhalt von Scalarsets verändern, wenn sie im Rahmen einer Zuweisung aufgerufen werden. *WriteAll()* weist in diesem Fall allen Elementen eines Scalarsets einen bestimmten Wert zu, während *writeOne()* lediglich ein Element manipuliert und zum Indizieren die Index-Variable verwendet. Das nächste Listing demonstriert die Transformation nach Promela:

```
1 //writeOne(scalar1.a) = 5
2 scalar1_a[scalar1] = 5;
3 //writeAll(scalar1.a) = 3;
4 scalar1 = 1;
5 scalar1_a[scalar1] = 3;
6 scalar1 = 2;
7 scalar1_a[scalar1] = 3;
8 scalar1 = 3;
9 scalar1_a[scalar1] = 3;
```

Listing 4.88: Transformation von writeAll() und writeOne() nach Promela

Im obigen Listing 4.88 sind in den Zeilen 1 bzw. 3 die Statements des DVF zu sehen und darunter die entsprechenden Transformationen nach Promela. Bei *writeOne()* wird die Indexvariable genutzt, um einem Element des Scalarsets den Wert 5 zuzuweisen. Da die Index-Variable immer einen Zufallswert enthält, entspricht die Operation genau der Semantikdefinition von *writeOne()*. Analog dazu werden bei *writeAll()* alle Elemente des Arrays beschrieben. Zu diesem Zweck wird *writeAll()* in eine Menge von Anweisungen transformiert, die jedem Element von a den Wert 3 zuweisen. Dies entspricht genau der Semantikdefinition von *writeAll()*.

Das letzte Sprachkonstrukt, das im Rahmen der Scalarsets zur Verfügung steht, ist *nextIndex()*. Es weist der Indexvariable des entsprechenden Scalarsets einen zufälligen Wert zu. Das folgende Beispiel zeigt, wie *nextIndex()* für das Scalarset aus Listing 4.86 vom Transformator des DVF nach Promela überführt wird:

```
1 //nextIndex(scalar1)
2 atomic{
3   if
4   :: true -> scalar1 = 1;
5   :: true -> scalar1 = 2;
6   :: true -> scalar1 = 3;
```

```
7   fi;
8 }
```

Listing 4.89: Transformation von nextIndex() nach Promela

Um einen Zufallswert zu generieren wird ein nicht-deterministischer If-Block verwendet. Für das Erzeugen der Zufallswerte findet das Verfahren von Ruijs Verwendung [127]. Des Weiteren wird, um den Zustandsraum des entsprechenden Promela-Modells zu reduzieren, das Erzeugen des Zufallswerts in einen Atomic-Block eingebettet. Durch diese Art der Modellierung wird der Zustandsraum des Modells reduziert und so der Speicherverbrauch von Spin minimiert.

4.6.12 Erzeugen des Init-Blocks

Als letztes Element muss im Rahmen der Übersetzung nach Promela ein Init-Block erzeugt werden. Er hat die folgende Funktionalität:

- Das erzeugte Modell besteht aus einer Menge von Prozess-Deklarationen. Diese entsprechen entweder den Threads des DVF, oder Dummy-Prozessen zum Umsetzen der Scalarsets (vgl. Abschnitt 2.1.4). Der Transformator fügt deshalb Run-Statements in den Init-Block ein, um die entsprechenden Prozesse zu starten.

- Die Promela-Modelle können Arrays enthalten. Der Transformator fügt deshalb eine Menge von Statements in den Init-Block ein, die den Arrays die entsprechenden Werte zuweisen. Dies ist notwendig, da die Arrays des DVF zusammen mit einem Initialwert deklariert werden können, was jedoch die Sprache Promela nicht zulässt.

Der Transformator geht zum Umsetzen der beiden Aspekte nach einem Algorithmus vor, den Abbildung 4.21 als Aktivitätsdiagramm visualisiert. Der Ansatz wird im weiteren Verlauf dieses Abschnitts genauer beschrieben.

Der Transformator fügt zunächst die Deklaration eines Dummy-Prozesses in das Modell ein. Die Deklaration ist in Listing 2.18 in den Zeilen 10 bis 14 gezeigt. Dummy-Prozesse sind notwendig, damit das Werkzeug Topspin Symmetrie in den Promela-Modellen erkennt und den Zustandsraum verkleinert (vgl. Abschnitt 2.1.4).

Im Anschluss wird der Init-Block generiert. Gemäß Abbildung 4.21 fügt der Transformator zunächst eine Menge von Statements ein, die den Arrays des Modells ihre Initialwerte zuweisen. Das folgende Beispiel verdeutlicht das Vorgehen:

```
1 /* byte a[2] = 255; */
2 byte a[2];
3
4 init{
```

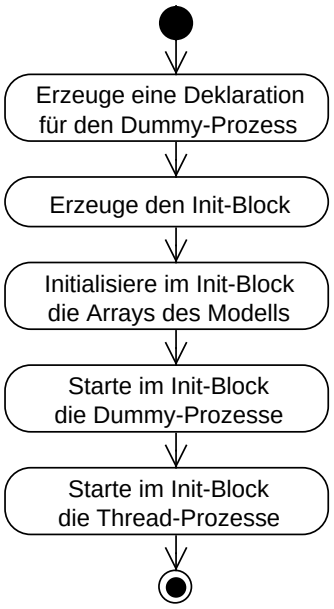

Abbildung 4.21: Erzeugen des Init-Blocks (Aktivitätsdiagramm)

```
5   d_step{
6     a[0] = 255;
7     a[1] = 255;
8   }
9 }
```

Listing 4.90: Initialisierung von Variablen

Im obigen Listing 4.90 ist in Zeile 1 die Deklaration eines Arrays a zu sehen, wie sie mit dem DSL Verification Framework möglich ist. Jedes Element des Arrays wird mit dem Wert 255 initialisiert. Die Zeilen 2 bis 8 zeigen die Transformation nach Promela. Der Init-Block enthält zwei Statements, die den Elementen des Arrays a den Wert 255 zuweisen. Dabei fällt auf, dass die Initialisierung in eine Dstep-Umgebung eingebettet ist. Dies hat den Vorteil, dass Spin beim Verifizieren alle darin enthaltenen Statements zu einem Zustand zusammenfasst und so weniger Speicher benötigt.

Als alternative Implementierungsart können die Statements auch in eine Atomic-Umgebung eingebettet werden. Spin betrachtet alle Statements in einer Atomic-Umgebung als atomar und fasst sie somit zu einem Zustand zusammen. Sie bietet also einen ähnlichen Vorteil wie *d_step*. Gemäß Holzman et al. [68] gilt für *d_step* die Einschränkung, dass alle enthaltenen Statements deterministisch sein müssen. Im Gegenzug kann Spin während des Verifikationsprozesses den Zustandsraum von Modellen mit Dstep-Umge-

bungen besser optimieren. Da es sich bei den Zuweisungen für die Initialisierungswerte der Arrays ausschließlich um deterministische Operationen handelt, sind somit Dstep-Umgebungen vorzuziehen und werden vom Transformator des DVF verwendet.

Nach dem Initialisieren der Arrays generiert der Transformator Run-Statements, um die Dummy-Prozesse zu starten. Die Anzahl der zu startenden Dummy-Prozesse ergibt sich anhand der Analyse aus Abschnitt 4.6.1. Gegeben sei beispielsweise ein DVF-Modell, das ein Scalarset enthält:

```
1 scalarset scalar1[3]{
2   byte a;
3   byte b;
4 }
```

Listing 4.91: DVF-Modell mit einem Scalarset

Im Fall des obigen Listings 4.91 kommt die Scalarset-Analyse aus Abschnitt 4.6.1 zu dem Ergebnis, dass drei Dummy-Prozesse gestartet werden müssen, damit Topspin die symmetrischen Eigenschaften des Modells zur Reduktion des Zustandsraums nutzt. Somit fügt der Transformator die folgenden Statements in den Init-Block ein:

```
1 init{
2   atomic{
3     run dummy();
4     run dummy();
5     run dummy();
6   }
7 }
```

Listing 4.92: Starten der Dummy-Prozesse

Im obigen Listing 4.92 fällt auf, dass keine Dstep-, sondern eine Atomic-Umgebung genutzt wird. Dies ist notwendig, da die Run-Anweisung von Promela nur innerhalb von Atomic-Blöcken zulässig ist [68].

Neben den Dummy-Prozessen müssen auch die Threads im Init-Block gestartet werden Deshalb iteriert der Transformator über alle Statements des DVF-Modells und überprüft, ob es sich um die Run-Instruktion handelt. Für jedes Run-Statement wird ein entsprechender Prozess im Init-Block gestartet. Das folgende Beispiel dient zur Veranschaulichung des Transformationsprozesses:

```
1 thread FSM{
2   //...
3 }
4 run FSM();
5 run FSM();
```

Listing 4.93: Deklaration und Instanziierung eines Threads

Das obige Listing 4.93 ist mit dem DVF umgesetzt. Es enthält den Thread *FSM*, von dem zwei Instanzen mittels *run* gestartet werden. Listing 4.93 wird vom Transformator des DVF in das folgende Promela-Modell überführt:

```
1 proctype FSM(){
2   //...
3 }
4
5 init{
6   atomic{
7     run FSM();
8     run FSM();
9   }
10 }
```

Listing 4.94: Starten von Threads in Promela

Im obigen Listing 4.94 ist der Promela-Prozess FSM enthalten. Des Weiteren werden vom Transformator zwei Run-Statements in den Init-Block eingefügt, um den Thread zur Laufzeit zu instanziieren.

4.6.13 Zusammenfassung

Dieses Kapitel zeigt, wie der im DVF enthaltene Transformator die Sprachkonstrukte des DSL Verification Frameworks automatisiert nach Promela überführt. Dazu gehören sowohl die Produktionsregeln, die ein DSL-Entwickler in seine domänenspezifische Sprache integriert, als auch die Control Flow Intermediate Language. Das Ziel dieses Abschnitts ist eine Zusammenfassung des Übersetzungsprozesses. Seine Struktur orientiert sich an den Unterkapiteln 4.3 und 4.4.

Die Include-Anweisung bewirkt, dass der Parser alle inkludierten Dateien in genau einen Abstract Syntax Tree überführt. Dieser wird im Anschluss vom Transformator eingelesen und nach Promela übersetzt. Eine Transformation der Include-Direktive ist somit nicht notwendig, da sie der Parser automatisiert berücksichtigt. Da Spin beim Einlesen eines Promela-Modells den *C Präprozessor* [78] aufruft, könnten jedoch für zukünftige Arbeiten Promela-Modelle erzeugt werden, die unter Verwendung der Include-Anweisung auf mehrere Quellcode-Dateien aufgeteilt sind.

Das DVF ermöglicht in einer domänenspezifischen Sprache das Deklarieren von Variablen. Als Datentypen sind Klassen, Aufzählungen, Boolean, Byte, Integer und String zulässig. Boolean, Byte und Integer können unverändert nach Promela überführt werden. Zeichenketten und Aufzählungen bildet der Transformator des DVF in Promela mit Mtype-Umgebungen ab. Klassen werden in Typedef-Elemente überführt. Eine Sonderform der Variablen sind im DVF die sogenannten Message-Queues zum Senden

bzw. Empfangen von Signalen. Message-Queues übersetzt der Transformator in Promela-Channel.

Die Deklaration einer Variable ist auch als Array möglich. Da Promela Arrays unterstützt, können die DVF-Arrays unverändert nach Promela überführt werden. Eine Sonderform sind symmetrische Arrays, für die der Transformator spezielle Variablen zum Indizieren generiert. Die Initialisierung der Arrays erfolgt im Init-Block des Promela-Modells.

Alle Variablen des DVF können in einem Modell an drei unterschiedlichen Positionen deklariert werden:

- Globale Variablen werden auch von Spin unterstützt und müssen vom Transformator nicht gesondert berücksichtigt werden.
- Die lokalen Variablen von Klassen bildet der Transformator mit Variablen innerhalb von Typedef-Elementen ab.
- Die Variablen von Threads bildet der Transformator mit lokalen Variablen innerhalb von Promela-Prozessen ab.

Neben Variablen beinhaltet das DSL Verification Framework auch Produktionsregeln für Expressions und Statements. Die darin verwendeten Operatoren werden auch von Promela unterstützt und müssen vom Transformator nicht modifiziert werden. Lediglich Methodenaufrufe innerhalb einer Expression erfordern eine gesonderte Betrachtung (vgl. Abschnitt 4.6.11).

Mit der Produktionsregel für Statements lassen sich eine Reihe unterschiedlicher Sprachkonstrukte beschreiben. Zum Abbilden der If- und Wait-Blöcke verwendet der Transformator If-Umgebungen, die ein Teil von Promela sind. Für das Sprachkonstrukt *while* nutzt er die in Promela enthaltenen Do-Schleifen. Der Versand und Empfang von Signalen erfolgt mit den Promela-Operatoren „?“ bzw. „!“. Den Lesezugriff eines Scalarsets überführt der Transformator in eine Expression, die die Elemente des symmetrischen Arrays mit Und- bzw. Oder-Operatoren verknüpft. Der Schreibzugriff auf Scalarsets wird mit einer Menge von Zuweisungen umgesetzt.

Das DSL Verification Framework beinhaltet auch eine Produktionsregel zum Deklarieren von Methoden, die der Transformator in Inline-Umgebungen übersetzt. Somit wird eine Klasse, die aus Methoden und Attributen besteht, in Inline-Umgebungen und Typedef-Elemente transformiert.

Die Control Flow Intermediate Language stellt Threads zur Verfügung, die nebenläufig ausgeführt werden. Daher werden sie vom Transformator des DVF in Promela-Prozesse überführt. Zum Starten der Threads enthält die CFIL eine spezielle Run-Direktive. Diese wird in Promela mit Run-Statements abgebildet, deren Aufruf im Init-Block erfolgt. Ein weiteres Element in der CFIL sind die sogenannten Statement-Blöcke. Sie kapseln eine Menge von Anweisungen und können mit einem Goto-Statement betreten oder verlassen

werden. Der Transformator des DVF überführt alle Anweisungen in einem Statement-Block nach Promela and stellt ihnen ein Label voran. Dadurch können sie mit den Sprung-Anweisungen von Promela betreten oder verlassen werden.

4.7 Übersetzung nach Java

Die vorangegangenen Abschnitte stellen die Sprachkonstrukte des DSL Verification Frameworks, die Validierung und die Transformation nach Promela vor. Dabei wird auch gezeigt, wie bei der Überführung nach Promela durch verschiedene Optimierungen auf Modellebene der Zustandsraum des Modells eingeschränkt werden kann. Für den in dieser Arbeit vorgestellten Ansatz, also die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation, ist ein weiterer Transformator notwendig, der die Elemente des DVF in eine Hochsprache überführt. Daher zeigt dieser Abschnitt, wie die Sprachkonstrukte des DSL Verification Frameworks nach Java übersetzt werden können.

Das DSL Verification Framework beinhaltet verschiedene Sprachkonstrukte. Einige dieser Sprachkonstrukte werden von Java nativ unterstützt. Dazu gehören beispielsweise Klassen. Für einen Teil der DVF-Sprachkonstrukte gibt es jedoch in Java keine nativen Elemente, die die selbe Semantik aufweisen. Hierzu gehören beispielsweise Signale oder Message-Queues. Aus diesem Grund besteht der Java-Transformator des DSL Verification Frameworks aus zwei Komponenten:

- Die *Java DVF Library* (JDL): Hierbei handelt es sich um eine Java-Klassenbibliothek, deren Komponenten genau den DVF-Sprachkonstrukten entsprechen, für die keine nativen Java-Elemente zur Verfügung stehen.
- Der eigentliche Transformator: Er liest den Abstract Syntax Tree ein und überführt ihn unter Verwendung der JDL nach Java.

Die Verwendung der JDL wird in Abbildung 4.22 verdeutlicht. Der Parser hat ein Modell eingelesen, das aus den Sprachkonstrukten des DSL Verification Frameworks besteht und es in einen Abstract Syntax Tree überführt. Der AST wird dem Java-Transformator übergeben, der es in Java-Quellcode übersetzt. Der Abstract Syntax Tree beinhaltet unter anderem DVF-Elemente, die Java nativ unterstützt. Hierzu gehören beispielsweise Strings. Einige DVF-Sprachkonstrukte sind jedoch in der Java-Klassenbibliothek nicht vorhanden. Diese werden stattdessen von der JDL zur Verfügung gestellt. Somit überführt der Transformator alle Elemente des Modells, die nicht nativ von Java unterstützt werden, in Klassen, die Teil der JDL sind bzw. von ihnen erben. Hierzu gehören beispielsweise Message-Queues und Signale.

Die Beschreibung des Java-Transformators gliedert sich wie folgt: Abschnitt 4.7.1 stellt die JDL und die von ihr zur Verfügung gestellten Klassen vor. Darauf folgt Unterkapitel 4.7.2, das den eigentlichen Übersetzungsalgorithmus beschreibt. Abschnitt 4.7.3 gibt

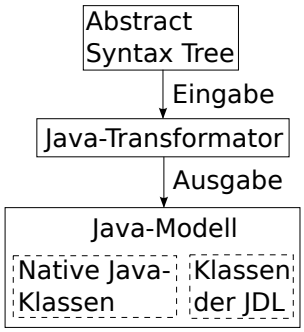

Abbildung 4.22: Transformation eines DVF-Modells nach Java

eine abschließende Zusammenfassung und weist nach, dass alle Elemente des DVF vom Transformator nach Java überführt werden.

4.7.1 Java DVF Library

In diesem Abschnitt wird die Java DVF Library (JDL) vorgestellt, die einen Teil der DVF-Sprachkonstrukte mit Java-Klassen abbildet. Die JDL bietet einen wichtigen Vorteil: Der Transformator muss DVF-Elemente, für die es keine native Java-Unterstützung gibt, nicht mit einer Menge von Java-Sprachkonstrukten nachbilden. Stattdessen können die entsprechenden DVF-Elemente direkt in Klassen überführt werden, die von der JDL erben. Dadurch wird die Übersetzung nach Java vereinfacht und somit auch die Komplexität des Transformators reduziert. In Abbildung 4.23 ist das Klassendiagramm der Java DVF Library zu sehen. Dabei ist zu beachten, dass einige Attribute und Methoden fehlen, um die Lesbarkeit zu verbessern. Die Abbildung zeigt, dass die JDL aus den folgenden Komponenten besteht:

- Das Wurzelelement der JDL ist die Klasse *Main*, die die nebenläufigen Threads eines Modells startet und verwaltet.
- Für das Umsetzen der DVF-Threads steht die Klasse *ConcurrentProc* zur Verfügung.
- Die asynchronen Signale des DVF können durch Objekte vom Typ *MessageQueue* empfangen bzw. versendet werden.
- Die Modellierung der eigentlichen Signale erfolgt mit der Klasse *Signal*.

Die einzelnen Elemente der JDL werden im weiteren Verlauf dieses Abschnitts genauer vorgestellt.

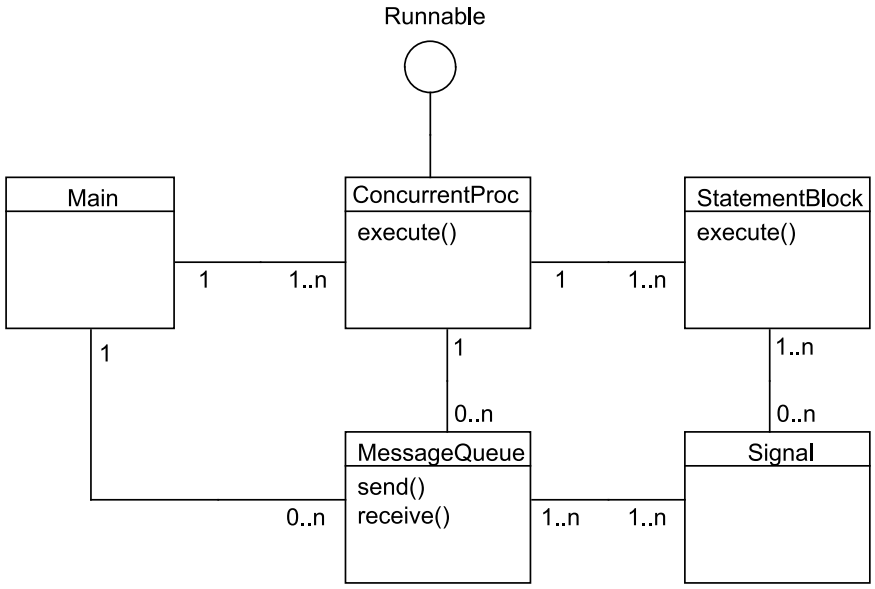

Abbildung 4.23: Struktur der Java DVF Library (Klassendiagramm)

Instanziieren von Threads

Dieser Unterabschnitt zeigt, wie mit Java und der JDL ein DVF-Modell umgesetzt werden kann, das aus einem Thread besteht. Dafür wird das folgende DVF-Modell als Beispiel verwendet:

```
1 thread FSM{
2   //...
3 }
4
5 run FSM();
```

Listing 4.95: DVF-Modell mit einem Thread

Im obigen Listing 4.95 ist ein Modell zu sehen, das mit den Sprachkonstrukten des DSL Verification Framework umgesetzt ist. Es enthält den Thread *FSM*, von dem genau eine Instanz gestartet wird. Das nächste Beispiel zeigt, wie das Modell aus Listing 4.95 mit Java unter Verwendung der JDL beschrieben werden kann:

```
1 class Main{
2   FSM instance1 = new FSM();
3
4   public static void main(){
```

```
5       new Thread(instance1).start();
6     }
7 }
8
9 class FSM extends ConcurrentProc{ /* ... */ }
```

Listing 4.96: Starten von Threads im Java-Framework

Im obigen Listing 4.96 werden zunächst die Klassen *Main* und *FSM* deklariert. Damit *FSM* als DVF-Thread ausgeführt werden kann, erbt sie von *ConcurrentProc*. In *Main* wird eine Instanz von *FSM* erzeugt. Danach erfolgt das Starten von *FSM* als nebenläufiger Thread.

Kapselung von Statements

Der letzte Abschnitt hat gezeigt, wie die Threads des DVF mit den Klassen der JDL und Java beschrieben werden können. Bei den Modellen aus den Abbildungen 4.95 bzw. 4.96 ist jedoch für den Thread *FSM* noch kein konkretes Verhalten spezifiziert. Um Verhalten mit dem DSL Verification Framework umzusetzen, enthält jeder Thread eine Menge von Statements, die wiederum in sogenannten Statement-Blöcken (vgl. beipielsweise Listing 4.3) gekapselt sind. Daher zeigt dieser Abschnitt, wie Statement-Blöcke mit der JDL modelliert und in Threads eingefügt werden können. Dafür wird zur Veranschaulichung das folgende DVF-Modell als Beispiel verwendet:

```
1 statementblock A{
2   var1 = var1 + 2;
3   goto B;
4 }
5
6 statementblock B{
7   var1 = var1 * 2;
8   exit;
9 }
```

Listing 4.97: Statement-Blöcke mit dem DVF

Im obigen Listing ist ein Modell zu sehen, das mit einer domänenspezifischen Sprache und dem DVF umgesetzt ist. Es enthält zwei Statement-Blöcke, die auf eine Variable namens *var1* zugreifen. Wenn *A* terminiert, wird mit einem Goto-Statement Statement-Block *B* betreten. Das folgende Beispiel zeigt, wie das Modell aus Listing 4.97 mit der Java DVF Library umgesetzt werden kann:

```
1 class A extends StatementBlock{
2   StatementBlock execute(){
```

```
3       var1 = var1 + 2;
4       return new B();
5     }
6   }
7
8   class B extends StatementBlock{
9     StatementBlock execute(){
10      var1 = var1 * 2;
11      return null;
12    }
13  }
```

Listing 4.98: Statement-Blöcke mit der JDL

Abbildung 4.23 macht deutlich, dass die JDL zur Umsetzung von Statement-Blöcken die Klasse *StatementBlock* enthält. Somit werden in Listing 4.98 die Klassen A und B deklariert, die beide von *StatementBlock* erben. Jede Klasse, die von *StatementBlock* erbt, muss die Methode *execute()* implementieren. Die Methode *execute()* enthält die entsprechenden Statements und liefert eine Referenz auf den Statement-Block zurück, der als nächstes ausgeführt werden soll. Wenn kein weiterer Statement-Block auszuführen ist, entspricht der Rückgabewert einem Objekt vom Typ *null*.

Gemäß der CFIL-Spezifikation (vgl. Abschnitt 4.4) sind Statement-Blöcke ein Teil der Threads des DVF. Daher wird im weiteren Verlauf dieses Unterkapitels gezeigt, wie die Statement-Blöcke der JDL in Objekte vom Typ *ConcurrentProc* zu integrieren sind. Jedes ConcurrentProc-Objekt beinhaltet eine Methode namens *add()*. Ihr werden Objekte vom Typ *StatementBlock* übergeben und so zu dem entsprechenden Thread hinzugefügt. Wenn beispielsweise ein Java-Modell beschrieben werden soll, das einen Thread *FSM* mit den beiden Statement-Blöcken *A* und *B* aus Listing 4.97 enthält, kann dies mit der JDL wie folgt umgesetzt werden:

```
1   class Main{
2     FSM instance1;
3
4     public static void main(){
5       instance1 = new FSM();
6       A a = new A(instance1);
7       B b = new B(instance1);
8       instance1.addEntry(a);
9       instance1.add(b);
10    }
11  }
```

Listing 4.99: Threads und Statement-Blöcke in der JDL

Im obigen Listing 4.99 erfolgt zunächst in Zeile 5 der Aufruf des Konstruktors der Klasse *FSM*, die von *ConcurrentProc* erbt. Danach werden in der Main-Methode zwei Instanzen von *A* und *B* gebildet. Da der Statement-Block *A* als Eintrittspunkt des Threads *FSM* fungiert, wird das Objekt *a* mit der Methode *addEntry()* übergeben. Des Weiteren fügt der Aufruf von *add()* in Zeile 9 ein Objekt vom Typ *B* hinzu. Auffällig ist, dass die Konstruktoren von *a* und *b* eine Referenz auf das Objekt *instance1* erhalten. Dies ist notwendig, damit der jeweilige Statement-Block die lokalen Variablen des entsprechenden Threads referenzieren kann.

Versenden und Empfangen von Signalen

Modelle, die mit DVF umgesetzt sind, können sensitiv auf bestimmte Signale sein. Zu diesem Zweck beinhaltet das DSL Verification Frameworks sogenannte Message-Queues. Über Sende- bzw. Empfangsoperationen können Signale auf einer Queue gespeichert bzw. abgerufen werden. Die JDL stellt zur Modellierung von Signalen und Message-Queues die Klassen *Signal* bzw. *MessageQueue* zur Verfügung. Beide werden in diesem Abschnitt genauer vorgestellt.

Alle Signale, die von einer *MessageQueue* empfangen werden sollen, müssen von der JDL-Klasse *Signal* erben. Das folgende Beispiel verdeutlicht die Modellierung eines Signals, das zwei Parameter enthält:

```
1 class TwoBytes extends Signal{
2   byte param1;
3   byte param2;
4 }
```

Listing 4.100: Erzeugen eines Signals

Im obigen Listing 4.100 wird ein DVF-Signal mit der JDL in Java umgesetzt. Das Signal hat den Bezeichner *TwoBytes* und besteht aus zwei Bytes. Den Versand und das Empfangen derartiger Signale ermöglicht die Klasse *MessageQueue*, die dem Benutzer zwei Methoden zur Verfügung stellt:

- *Receive()* ruft ein Signal von der Queue ab.
- *Send()* speichert ein Signal auf der Queue.

Die Semantik der beiden Methoden entspricht genau der DVF-Spezifikation. Das bedeutet *send()* blockiert, wenn kein weiteres Signal auf der Queue gespeichert werden kann. Analog dazu blockiert *receive()*, wenn die Queue leer ist und somit kein Signal abgerufen werden kann.

Implementierung

In den vorangegangenen Unterkapiteln werden Konzept, Aufbau und Verwendung der Java DVF Library demonstriert. Dieser Abschnitt zeigt die Implementierung der JDL.

Die Statement-Blöcke des DSL Verification Frameworks sind in Threads eingebettet. Somit müssen die Statement-Blöcke der JDL in Objekte vom Typ *ConcurrentProc* integriert werden. Die Implementierung von *ConcurrentProc* gestaltet sich wie folgt:

```
1  class ConcurrentProc implements Runnable{
2    StatementBlock current;
3
4    void addEntry(StatementBlock entry){
5      current=entry;
6    }
7
8    void run(){
9      execute();
10   }
11
12   public void execute(){
13     while(current!=null){
14       StatementBlock next = current.execute();
15       current = next;
16     }
17   }
18 }
```

Listing 4.101: Struktur von Execute()

Die Umsetzung der Klassen *ConcurrentProc* bzw. *StatementBlock* orientiert sich an dem State-Pattern [55]. Damit *ConcurrentProc* als nebenläufiger Thread ausgefügt werden kann, implementiert sie das Interface *Runnable* [60]. Der Eintrittspunkt, also der als erstes auszuführende Statement-Block, wird *ConcurrentProc* mit der Methode *addEntry()* übergeben. *Execute()* beinhaltet eine While-Schleife. Die Schleife ruft die Execute-Methode des aktuell auszuführenden Statement-Blocks auf und speichert den Rückgabewert in *current*. Sobald *current* den Wert *null* annimmt terminiert die Schleife und somit auch der Thread.

Die JDL ermöglicht das Senden und Empfangen von asynchronen Signalen. Jedes Signal wird in einer Message-Queue gespeichert. Zur Umsetzung der Queues stellt die JDL die Klasse *MessageQueue* zur Verfügung. Ihre Implementierung gestaltet sich wie folgt:

```
1 class MessageQueue{
```

```
2   final static int QUEUE_SIZE = 2;
3   public ArrayBlockingQueue<Signal> queue =
4     new ArrayBlockingQueue<Signal>(QUEUE_SIZE);
5
6   Signal receive(){
7     return queue.take();
8   }
9
10  void send(Signal s){
11    put(s);
12  }
13 }
```

Listing 4.102: Message-Queue im Java-Framework

Anhand des obigen Listings wird deutlich, dass *MessageQueue* die Klasse *ArrayBlockingQueue* verwendet. *ArrayBlockingQueue* bietet den Vorteil einer thread-sicheren Implementierung. Dies ist notwendig, damit die Queue von mehreren Threads gleichzeitig genutzt werden kann. Auf Objekte vom Typ *ArrayBlockingQueue* kann wie folgt zugegriffen werden:

- *take()*: Entfernt das erste Element der Queue und gibt es zurück. Die Methode blockiert, wenn die Queue leer ist.

- *put()*: Speichert ein Element am Ende der Queues. Die Methode blockiert, wenn bereits die maximale Anzahl von Nachrichten auf der Queue gespeichert ist.

Take() und *put()* werden nicht direkt aufgerufen. Stattdessen stellt *MessageQueue* nach außen die Methoden *send()* und *receive()* zur Verfügung. Diese Art der Implementierung hat den Vorteil, dass die JDL leicht erweitert werden kann, um andere Medien zum Versenden und Empfangen von Signalen zu nutzen. Wenn Signale beispielsweise über eine Netzwerkschnittstelle übertragen werden sollen, muss lediglich eine Klasse erzeugt werden, die von *MessageQueue* erbt und *send()* bzw. *receive()* mit neuer Funktionalität überschreibt.

4.7.2 Transformation nach Java unter Verwendung der JDL

Der vorangegangene Abschnitt stellt die Java DVF Library vor. Sie enthält Klassen, um einen Teil der DVF-Sprachkonstrukte mittels Java umzusetzen. In diesem Abschnitt wird ein Übersetzungsalgorithmus beschrieben, der die Elemente des DSL Verification Frameworks, unter Verwendung der JDL, nach Java transformiert.

Klassen

Dieser Abschnitt beschreibt die Übersetzung der Klassen des DSL Verification Frameworks. Dabei ist zu beachten, dass nur die Transformation von lokalen Variablen und der Methodenköpfe vorgestellt wird. Die Übersetzung der Methodenkörper erfolgt im nächsten Abschnitt im Rahmen von Statements und Expressions.

Der Transformator für DVF-Klassen arbeitet nach dem folgenden Algorithmus: Er iteriert über den gesamten Abstract Syntax Tree eines Modells. Bei jedem Element findet eine Überprüfung statt, ob es sich um eine Klasse handelt. Wenn der Transformator ein Objekt vom Typ *DVFClass* findet, wird eine Java-Klasse generiert, die den selben Bezeichner wie die DVF-Klasse aufweist. Danach iteriert der Transformator über alle Elemente, die Teil der gefundenen Klasse sind, überführt sie nach Java und fügt das Ergebnis in die zuvor generierte Java-Klasse ein. Dazu gehören:

- Lokale Variablen vom Typ Boolean, Byte, Integer, String, Aufzählungen oder Klassen-Instanzen. Variablen können auch als Array deklariert werden.
- Methoden zur Beschreibung von Verhalten.

Java unterstützt alle Datentypen des DSL Verification Frameworks. Deshalb können die lokalen Variablen einer DVF-Klasse unverändert nach Java überführt werden. Die einzige Ausnahme stellt das Initialisieren von Arrays dar: In diesem Fall generiert der Transformator für die entsprechende Klasse einen Konstruktor, der den Arrays ihre jeweiligen Startwerte zuweist.

Die Methoden einer DVF-Klasse können entweder eine Menge von Statements enthalten, oder eigenschaftserhaltende Abstraktion umsetzen. Syntax und Semantik von DVF-Methoden, deren Verhalten mit Statements beschrieben wird, sind äquivalent zu den entsprechenden Java-Methoden [60]. Daher können auch sie unverändert vom Transformator in das Java-Modell eingefügt werden.

Eine Besonderheit sind Methoden zur eigenschaftserhaltenden Abstraktion (vgl. beispielsweise Listing 4.41), die keinen Methodenkörper haben. Das fehlende Verhalten muss nach der Übersetzung in eine Hochsprache vom Entwickler ergänzt werden. Daher stellt dieser Abschnitt einen Java-Übersetzungsalgorithmus für Methoden mit eigenschaftserhaltender Abstraktion vor. Das folgende Beispiel enthält ein DVF-Modell und dient zur Verdeutlichung des Vorgehens:

```
1 class Sensors{
2   abstraction range(1,3) readSensor1(){}
3
4   //...
5 }
```

Listing 4.103: Beispiel für Methoden zur eigenschaftserhaltenden Abstraktion

Im obigen Listing 4.103 ist ein DVF-Modell zu sehen, das die Klasse *Sensors* deklariert. *Sensors* dient zur Steuerung nicht näher definierter Sensoren. Die Methode *readSensor1()* fragt $Sensor_1$ ab, der einen Wert zwischen 1 und 3 zurückgeben kann. Eine einfache Lösung wäre es, die Klasse *Sensors* wie folgt nach Java zu übersetzen:

```
1 class Sensors{
2   byte readSensor1(){
3     //TODO: Implement me
4   }
5 }
```

Listing 4.104: Transformation einer Abstraction-Methode nach Java

Im obigen Listing 4.104 wird die Klasse *Sensors* so nach Java überführt, dass sie eine leere Methode namens *readSensor1()* enthält, deren Körper noch implementiert werden muss. Das Problem dabei ist, dass ein Entwickler beispielsweise in Zeile 3 das Statement „return 5“ einfügen könnte, bei dem es sich um einen gültigen Rückgabewert handelt. Allerdings verifiziert der Model Checker das Modell nur für Rückgabewerte zwischen 1 und 3. Um diesem Problem entgegen zu wirken, wird Listing 4.104 stattdessen wie folgt nach Java überführt:

```
1 class Sensors{
2   byte readSensor1(){
3     byte ret = readSensor1_user();
4     if(ret<1 || ret>3)
5       assert(false);
6     return ret;
7   }
8
9   byte readSensor1_user(){
10    //TODO: Implement me
11    return 255;
12  }
13 }
```

Listing 4.105: Transformation einer Interface-Methode nach Java

Der vom Anwender zu implementierende Quellcode ist im obigen Listing in die Methode *readSensor1_user()* ausgelagert, die von *readSensor1()* aufgerufen wird. Die Methode *readSensor1()* überprüft, ob sich der Rückgabewert in dem von der DSL spezifizierten Wertebereich befindet und wirft im Fehlerfall eine entsprechende Exception. Der Wertebereich ergibt sich aus dem Datentyp bzw. dem Schlüsselwort *range* (vgl. Listing 4.39). So wird sichergestellt, dass sich die Java-Anwendung genauso verhält, wie das Promela-Modell. Damit der Quellcode kompilierbar ist, gibt *readSensor1_user()* in Zeile

11 einen der Grenzwerte zurück. Dies muss später vom Anwender beim Implementieren der Methode entsprechend angepasst werden. Des Weiteren ist bei der Umsetzung des Java-Transformators darauf zu achten, Methoden wie *readSensor1_user()* nicht zu überschreiben, wenn sie bereits manuell implementierten Quellcode enthalten. Eine Alternative für zukünftige Arbeiten ist das Generieren eines Interfaces oder einer abstrakten Klasse, die die Methoden mit eigenschaftserhaltender Abstraktion enthält.

Globale Variablen

Dieser Abschnitt zeigt, wie der Transformator globale Variablen in DVF-Modellen nach Java überführt. Zu diesem Zweck iteriert er über den gesamten AST und überprüft bei jedem Element, ob es sich um eine globale Variable handelt. Der vorangegangene Abschnitt erläuterte bereits die Transformation lokaler Variablen innerhalb von Klassen. Dabei wird deutlich, dass Syntax und Semantik äquivalent zu Java sind. Die einzige Ausnahme stellte das Initialisieren von Arrays dar. Ähnlich verhält es sich mit den globalen Variablen eines Modells. Auch hier muss lediglich die Initialisierung von Arrays gesondert betrachtet werden. Zur Veranschaulichung des Transformationsprozesses dient das folgende DVF-Beispiel:

```
1 byte counter;
2 bool[3] bits = true;
3
4 //...
```

Listing 4.106: Deklaration globaler Variablen mit dem DVF

Im obigen Listing 4.106 ist ein Modell zu sehen, das mit dem DSL Verification Framework umgesetzt ist. Es enthält zwei globale Variablen. Bei *bits* in Zeile 2 handelt es sich um ein Array, das mit dem Wert *true* initialisiert wird. Das nächste Beispiel zeigt, wie der Transformator des DVF das Modell in Listing 4.106 nach Java überführt:

```
1 class Main{
2   public static volatile byte counter;
3   public static volatile boolean[] bits = new boolean[3];
4
5   public static void main(){
6       bits[0] = true;
7       bits[1] = true;
8       bits[2] = true;
9       //...
10  }
11 }
```

Listing 4.107: Deklaration globaler Variablen mit Java

Java unterstützt keine globalen Variablen, wie sie im DSL Verification Framework zulässig sind. Das obige Listing zeigt, dass der Transformator die globalen Variablen deshalb wie folgt nach Java übersetzt: Jedes DVF-Modell, das nach Java überführt wird, enthält die Klasse *Main*. Deshalb deklariert der Transformator die globalen Variablen eines DVF-Modells als lokale Variablen innerhalb von *Main*. Sie nutzen die folgenden Schlüsselwörter:

- Durch *static* und *public* können die Variablen in jedem anderen Objekt mittels *Main.counter* bzw. *Main.bits* referenziert werden. Voraussetzung ist allerdings, dass sie sich in dem selben *Package* befinden.
- Durch *volatile* ist sichergestellt, dass ein Thread auf die globale Variable schreibt und alle weiteren Threads den aktualisierten Wert auch lesen können.

Eine alternative Implementierung ist die Nutzung einer Singleton-Klasse [55]. Dies führt jedoch zu dem Nachteil, das eine zusätzliche Klasse generiert werden muss.

Die Initialisierung des Arrays *bits* ist in Listing 4.107 in den Zeilen 6 bis 8 zu sehen. Es wird deutlich, dass das Zuweisen der Startwerte zu Beginn der Main-Methode erfolgt. Dies hat den Vorteil, dass danach die entsprechenden Threads gestartet werden können, alle Variablen bereits ihre Initialisierungswerte enthalten und das Modell in einem konsistenten Zustand ist.

Threads

Mit dem DSL Verification Framework können Modelle beschrieben werden, die nebenläufig ausgeführte Regionen enthalten. Dafür wird DSL-Entwicklern das Sprachkonstrukt *Thread* zur Verfügung gestellt. Dieser Abschnitt beschreibt, wie Threads vom Transformator des DVF nach Java überführt werden können. Dafür findet die Klasse *ConcurrentProc* aus der JDL Verwendung (vgl. Abschnitt 4.7.1). Zur Veranschaulichung des Übersetzungsprozesses wird im Rahmen dieses Abschnitts das folgende DVF-Beispiel genutzt:

```
1 thread Timer entry Run{
2   byte counter = 0;
3
4   statementblock Run{
5     counter = counter + 1;
6     counter = counter + 1;
7     counter = counter + 1;
8     exit;
9   }
10 }
```

Listing 4.108: DVF-Modell mit einem Thread und einem Statement-Block

Im obigen Listing 4.117 ist ein Modell zu sehen, dass den Thread *Timer* enthält. Die Ausführung von *Timer* beginnt aufgrund der Entry-Direktive mit dem Statement-Block *Run*. In *Run* wird die lokale Variable *counter* dreimal um 1 inkrementiert. Danach terminiert der Thread. Das nächste Beispiel zeigt, wie der Transformator Listing 4.117 nach Java überführt:

```
1  class Main{
2    public static void main(String[] argz){
3      Timer timer = new Timer();
4      Run run = new Run(timer);
5      timer.addEntry(run);
6    }
7  }
8
9  class Run extends StatementBlock{
10   StatementBlock execute(){
11     instance.var1 = instance.var1 + 1;
12     instance.var1 = instance.var1 + 1;
13     instance.var1 = instance.var1 + 1;
14     return null;
15   }
16 }
17
18 class Timer extends ConcurrentProc{
19   byte var1 = 0;
20 }
```

Listing 4.109: Transformation eines Threads nach Java

Das obige Listing 4.109 zeigt, wie der Transformator das Modell aus 4.108 nach Java überführt. Dabei geht er nach dem folgenden Algorithmus vor: Zunächst iteriert der Transformator über den AST des Modells und sucht nach Objekten vom Typ *DVF-Thread*. Jedes gefundene Objekt wird in eine Java-Klasse übersetzt, die von *Concurrent-Proc* erbt und die lokalen Variablen des entsprechenden Threads enthält. Dies verdeutlicht Listing 4.109 in den Zeilen 18 bis 20.

Im Anschluss iteriert der Transformator über alle im Thread enthaltenen Statement-Blöcke. Jeder gefundene Statement-Block wird in eine Klasse überführt, die von *StatementBlock* erbt. Listing 4.109 verdeutlicht das Vorgehen in den Zeilen 9 bis 16.

Nach der erfolgreichen Transformation aller Threads und Statement-Blöcke müssen Instanzen der Java-Klassen gebildet werden. Zu diesem Zweck fügt der Transformator in die Main-Methode des Modells entsprechende Statements ein. Listing 4.109 verdeutlicht dieses Vorgehen in den Zeilen 1 bis 7.

Statements und Expressions

Dieser Abschnitt beschreibt, wie die Statements und Expressions des DVF nach Java überführt werden. Auch hier ist es von Vorteil, dass einige der Elemente bereits nativ in Java vorkommen. Dazu gehören:

- Alle Operatoren.
- If-Blöcke und While-Schleifen.
- Synchronize-Umgebungen.
- Assert- und Return-Statements.
- Funktionsaufrufe.
- Zuweisungen.

Die entsprechenden Sprachkonstrukte kann der Transformator unverändert in das zu generierende Java-Modell einfügen. Daher werden sie in diesem Abschnitt nicht näher betrachtet. Im DSL Verification Framework sind jedoch auch Komponenten enthalten, für die keine nativen Java-Elemente existieren. Dazu gehören:

- Das Versenden bzw. Empfangen von Signalen.
- Lese- und Schreibzugriff auf symmetrische Arrays.

Damit Signale versendet werden können, müssen Message-Queues deklariert werden. Analog dazu müssen in einem Modell Scalarsets vorhanden sein, um schreibend oder lesend auf symmetrische Arrays zuzugreifen. Die entsprechenden Statements werden daher nicht in diesem Abschnitt vorgestellt, sondern in den beiden folgenden Unterkapiteln, die die Transformation von Message-Queues und Scalarsets beschreiben.

Message-Queues

Dieser Abschnitt beschreibt den Transformationsalgorithmus für Message-Queues und die dazugehörigen Statements, um Signale zu Versenden bzw. zu Empfangen. Dabei handelt es sich um eine mögliche Implementierung, die von nebenläufigen Threads genutzt werden kann, um zu kommunizieren. Für zukünftige Arbeiten sind alternative Transformationen möglich, die eine Netzwerkschnittstelle nutzen und so das Versenden von Signalen in verteilten Systemen ermöglichen. Zur Veranschaulichung des Ansatzes zeigt das nächste Listing ein DVF-Modell, das Message-Queues enthält und auf sie zugreift. Das Beispiel wird im weiteren Verlauf dieses Abschnitts genutzt, um die Transformation nach Java zu verdeutlichen:

```
signal s1{byte p1, bool p2}

thread Sender{
  //...
  send(s1, 1, false);
}

thread Receiver{
  receive(s1);
  if(s1.p2==true){
    //...
  }
}

run Sender(); run Receiver();
```

Listing 4.110: Versand von Signalen mit dem DVF

Im obigen Listing 4.110 ist ein DVF-Modell zu sehen, das eine Queue namens s_1 deklariert. Auf s_1 werden Signale gespeichert, die aus einem Byte und einem Bool bestehen. Des Weiteren enthält das Modell zwei Threads, die nebenläufig ausgeführt werden. *Sender* überträgt ein Signal auf die Queue s_1. *Receiver* ruft ein Signal von s_1 ab und wertet es im Rahmen eines If-Blocks aus. Das nächste Beispiel zeigt, wie Listing 4.110 und die darin enthaltenen Send- bzw. Receive-Operationen nach Java überführt werden können:

```
class S1 extends Signal{
  byte p1;
  bool p2;

  Queue(byte p1, boolp2){ /* ... */ }
}

class Main{
  static MessageQueue s1 = new MessageQueue();
  //...
}

class Sender extends ConcurrentProc{
  //...
  Main.s1.send(new S1(1, false));
}

```

```
18 class Receiver extends ConcurrentProc{
19   S1 s1_read;
20   s1_read = Main.s1.receive();
21   if(s1_read.p2 == false){
22     //...
23   }
24 }
```

Listing 4.111: Versand von Signalen mit Java

Das obige Beispiel zeigt, dass der Transformator das DVF-Modell aus Listing 4.110 mit dem folgenden Algorithmus nach Java überführt: In Java gibt es keine nativen Sprachkonstrukte zur Umsetzung von Message-Queues und Signalen. Deshalb verwendet der Transformator aus der JDL die Klassen *MessageQueue* und *Signal*.

Im Zuge der Übersetzung wird über den gesamten AST des Modells iteriert. Für jede gefundene Queue legt der Transformator eine Klasse an, die von *Signal* erbt und die Nachricht repräsentiert, die über die Queue versendet bzw. empfangen werden soll. Listing 4.111 enthält somit die Klasse *S1*, die das Signal der Queue *s1* darstellt. Das Signal besteht aus einem Byte und einem Bool. Somit werden in der Signal-Klasse die Variablen *p1* und *p2* generiert.

Nach dem Erzeugen des entsprechenden Signals überführt der Transformator die eigentliche Queue nach Java. Hierfür findet eine Fallunterscheidung statt:

- Wenn es sich um eine globale Queue handelt, wird in der Klasse *Main* ein Objekt vom Typ *MessageQueue* generiert und ihm das Schlüsselwort *static* vorangestellt. Dadurch kann die Queue von den verschiedenen Threads des Modells referenziert werden.

- Wenn die Deklaration der Queue innerhalb einer Klasse oder eines Threads erfolgt, legt der Transformator eine entsprechende lokale Variable an.

Da die Message-Queue *s1* in Listing 4.111 ein globales Element ist, wird sie in Zeile 9 in die Klasse *Main* integriert.

Die Transformation der Sende- und Empfangsoperationen wird in den Zeilen 13 bis 24 deutlich: *Sender* und *Receiver* erben beide von *ConcurrentProc*, da sie nebenläufige Threads sind. *Sender* erzeugt ein neues Objekt vom Typ *S1* und speichert es auf der Queue. Receiver ruft die Nachricht ab. Anschließend wird der empfangene Parameter *p2* in einem If-Block ausgewertet.

Scalarsets

Modelle können neben Message-Queues auch Scalarsets enthalten. Bei Scalarsets handelt es sich um symmetrische Arrays, deren Syntax und Semantik im Rahmen von Listing

4.10 vorgestellt wird. Im weiteren Verlauf dieses Abschnitts wird gezeigt, wie der Transformator des DVF die Scalarsets und die entsprechenden Zugriffsmethoden nach Java überführt. Das folgende DVF-Modell dient zur Veranschaulichung des Übersetzungsprozesses:

```
1 scalarset scalar1[3]{
2   byte a;
3   int b;
4 }
```

Listing 4.112: Scalarset mit dem DVF

Im obigen Listing 4.112 ist ein DVF-Modell zu sehen, in dem das Scalarset *scalar1* deklariert wird. Es enthält die beiden Elemente *a* und *b*. Das Modell besteht somit aus zwei symmetrischen Arrays, die eine Größe von drei haben. Listing 4.112 wird vom Transformator des DVF in den folgenden Java-Quellcode überführt:

```
1 public byte[] scalar1_a = new byte[3];
2 public int[] scalar1_b = new int[3];
3 public int scalar1 = 0;
```

Listing 4.113: Scalarset mit Java

Das obige Listing 4.113 zeigt, dass der Transformator wie folgt arbeitet: Jedes Element eines Scalarsets wird in ein Java-Array mit der Größe des Scalarsets überführt. Zusätzlich muss eine Index-Variable angelegt werden, die beispielsweise *writeOne()* zum Zugriff auf die entsprechenden Elemente nutzt. Daher werden in Listing 4.113 die beiden Arrays *scalar1_a* und *scalar1_b*, sowie die Indexvariable *scalar1* erzeugt.

Eine weitere Besonderheit stellen im DVF die Statements dar, die zum Lesen und Schreiben von Scalarsets eingesetzt werden. Syntax und Semantik von *writeOne()*, *writeAll()*, *exist()*, *all()* und *nextIndex()* werden in Abschnitt 4.3.3 vorgestellt, während die Transformation nach Promela Teil von Unterkapitel 4.6.11 ist. Das nächste DVF-Beispiel nutzt die Sprachkonstrukte *exist()* und *all()*, damit im Anschluss die Transformation nach Java beschrieben werden kann:

```
1 scalarset scalar1[2]{
2   int a;
3 }
4 if(exist(scalar1.a)==3){
5   //...
6 }
7 elseif(all(scalar1.a)==5){
8   //...
9 }
```

Listing 4.114: Abfrage von Scalarsets mit dem DVF

Im obigen Listing wird das Scalarset *scalar1* deklariert. Der If-Block fragt ab, ob sich in dem Array entweder ein Element mit dem Wert 3 befindet, oder alle Elemente den Wert 5 haben. Das nächste Beispiel zeigt, wie Listing 4.114 vom Transformator nach Java übersetzt wird:

```
1 byte[] scalar1_a = new byte[2];
2 int scalar1 = 0;
3
4 if(scalar1_a[0]==3 || scalar1_a[1]==3){
5   //...
6 }
7 else if(scalar1_a[0]==5 && scalar1_a[1]==5){
8   //...
9 }
```

Listing 4.115: Abfrage von Scalarsets mit Java

Das obige Listing zeigt, dass der Transormator des DVF nach dem folgenden Algorithmus vorgeht: Der Ausdruck *exist()* wird in eine Verkettung von Oder-Operatoren überführt. Dadurch ist sichergestellt, dass die Abfrage *true* zurückliefert, wenn mindestens ein Element des Scalarsets den Wert 3 enthält. Analog dazu wird das Sprachkonstrukt *all()* in eine Verkettung von Und-Operatoren übersetzt, wodurch die Expression nur dann den Wert *true* annimmt, wenn in allen Elemente des Arrays der Wert 5 gespeichert ist.

Neben *exist()* und *all()* gibt es im DVF für den Schreibzugriff auf Scalarsets die Sprachkonstrukte *writeOne()* und *writeAll()*. Ihre Transformation nach Java wird im weiteren Verlauf dieses Abschnitts vorgestellt. Zur Veranschaulichung dient das folgende DVF-Modell:

```
1 scalarset scalar1[2]{
2   int a;
3 }
4
5 writeAll(scalar1.a) = 3;
6 nextIndex(scalar1);
7 writeOne(scalar1.a) = 2;
```

Listing 4.116: Schreibzugriff auf Scalarsets mit dem DVF

Im obigen Listing 4.116 wird zunächst das Scalarset *scalar1* deklariert. Es enthält das symmetrische Array a. In Zeile 5 weist *writeAll()* allen Elementen von a den Wert 3 zu. Danach erhält die Indexvariable in Zeile 6 einen neuen, zufälligen Wert zwischen 0 und 1. Abschließend wird einem zufälligen Element in a der Wert 2 zugewiesen. Das nächste Beispiel zeigt, wie Listing 4.116 vom Transformator nach Java überführt werden kann:

```
1 Random random = new Random();
2 int[] scalar1_a;
3 int scalar1 = 0;
4
5 //writeAll()
6 scalar1_a[0] = 3;
7 scalar1_a[1] = 3;
8 //nextIndex()
9 scalar1 = random.next(0,2);
10 //writeOne()
11 scalar1_a[scalar1] = 2;
```

Listing 4.117: Schreibzugriff auf Scalarsets mit dem DVF

Das obige Listing 4.117 zeigt, wie der Transformator des DVF *writeAll()*, *nextIndex()* und *writeAll()* nach Java überführt. Die Übersetzung von *writeAll()* ist in den Zeilen 5 bis 7 zu sehen. Da *writeAll()* jedes Element des Arrays mit einem bestimmten Wert beschreibt, wird es in eine Menge von Zuweisungen überführt. In den Zeilen 8 bis 9 ist die Transformation von *nextIndex()* gezeigt. Es wird so nach Java transformiert, dass die entsprechende Indexvariable unter Verwendung der Klasse *Random* einen neuen Zufallswert erhält. Abschließend verdeutlichen die Zeilen 10 bis 11 die Übersetzung von *writeOne()*. Es nutzt die Indexvariable *scalar1*, um einem zufälligen Element von *a* den Wert 2 zuzuweisen.

4.7.3 Zusammenfassung

Der vorangegangene Abschnitt zeigt, wie der im DVF enthaltene Transformator die Sprachkonstrukte des DSL Verification Frameworks automatisiert nach Java überführt. Dieses Unterkapitel fasst die Übersetzung der verschiedenen Elemente zusammen. Die Reihenfolge der beschriebenen Elemente orientiert sich am Aufbau der Abschnitte 4.3 und 4.4.

Das DVF enthält eine Include-Anweisung, um ein Modell auf mehrere Quellcode-Dateien aufzuteilen. Diese muss gemäß Abschnitt 4.6.13 nicht nach Java überführt werden. Im Rahmen von zukünftigen Arbeiten ist es jedoch wünschenswert, dass für die generierten Java-Klassen verschiedene Dateien angelegt werden, um die Lesbarkeit eines Modells zu erhöhen.

Modelle, die mit dem DSL Verification Framework umgesetzt werden, können Variablendeklarationen enthalten. Als Datentypen sind Klassen, Aufzählungen, Boolean, Byte, Integer und String zulässig. Alle Datentypen können vom Transformator unverändert nach Java überführt werden, da Java native Elemente zum Umsetzen von Klassen, Aufzählungen, Boolean, Byte, Integer und Strings bietet. Ein besonderer Datentyp sind im DVF die sogenannten Message-Queues. Mit ihnen können Signale gesen-

det bzw. empfangen werden. Message-Queues übersetzt der Transformator in Objekte vom Typ *MessageQueue*, die von der *Java DVF Library* zur Verfügung gestellt werden.

Neben einzelnen Variablen ermöglicht das DVF auch das Beschreiben von Arrays. Diese werden vom Transformator unverändert nach Java überführt. Eine Sonderform sind symmetrische Arrays, für die der Transformator zusätzlich spezielle Variablen zum Indizieren generiert.

Alle Variablen des DVF können in einem Modell an drei unterschiedlichen Positionen deklariert werden:

- Globale Variablen werden von Java nicht direkt unterstützt. Daher überführt sie der Transformator in Klassenvariablen. Damit sie im Modell von den verschiedenen Threads referenziert werden können, stellt er ihnen die Schlüsselwörter *static* und *volatile* voran.
- Die lokalen Variablen von Klassen bedürfen keiner gesonderten Betrachtung, da sie von Java nativ unterstützt werden.
- Die lokalen Variablen von Threads bildet der Transformator mit lokalen Variablen innerhalb von Concurrent-Objekten ab.

Neben Variablen beinhaltet das DSL Verification Framework auch Produktionsregeln für Expressions und Statements. Die darin verwendeten Operatoren werden von Java nativ unterstützt. Eine Modifikation während des Übersetzungsprozesses ist daher nicht notwendig.

Mit der Produktionsregel für Statements lassen sich eine Reihe unterschiedlicher Sprachkonstrukte beschreiben. If-Blöcke und While-Schleifen kommen auch nativ in Java vor. Daher müssen sie vom Transformator nicht modifiziert werden. Der Versand und Empfang von Signalen erfolgt mit den Methoden *send()* bzw. *receive()*, die Teil der Klasse *MessageQueue* sind. Den Lesezugriff eines Scalarsets überführt der Transformator in eine Expression, die die Elemente des symmetrischen Arrays mit Und- bzw. Oder-Operatoren verknüpft. Der Schreibzugriff auf Scalarsets wird mit einer Menge von Zuweisungen umgesetzt.

Sowohl Java, als auch das DVF unterstützen Methoden bzw. Methodenaufrufe. Eine Besonderheit stellen jedoch Methoden zur eigenschaftserhaltenden Abstraktion dar, für die es in Java kein entsprechendes Sprachkonstrukt gibt. Der Transformator des DVF überführt Methoden zur eigenschaftserhaltenden Abstraktion in Java-Methoden, deren Körper keine Statements enthält. Der Entwickler muss nach dem Übersetzungsprozess das Verhalten selbst implementieren.

Die Control Flow Intermediate Language stellt Threads zur Verfügung, die nebenläufig ausgeführt werden. Sie werden vom Transformator in Klassen überführt, die von *ConcurrentProc* erben. Ein weiteres Element in der CFIL sind die sogenannten Statement-Blöcke. Sie kapseln eine Menge von Anweisungen und können mit einem Goto-Statement

betreten oder verlassen werden. Der Transformator des DVF überführt alle Statement-Blöcke in Klassen, die von *StatementBlock* erben. Sowohl *ConcurrentProc*, als auch *StatementBlock* sind Elemente der JDL.

4.8 Erweiterbarkeit

Die letzten Abschnitte haben die Komponenten des DSL Verification Frameworks vorgestellt. Dazu gehören Produktionsregeln, die in DSL-Projekte integriert werden, um die domänenspezifische Sprache mit Spracheigenschaften wie Expressions, Variablen, usw. zu ergänzen. Ein weiteres Element des DVF ist die Control Flow Intermediate Language. Sie dient als Zwischensprache, in die ein DSL-Entwickler alle Sprachkonstrukte seiner domänenspezifischen Sprache transformiert, die nicht auf Produktionsregeln des DVF basieren. Des Weiteren gehören zwei Transformatoren zum DVF, die ein Modell automatisiert nach Java bzw. Promela überführen.

Bei der Umsetzung des DSL Verification Frameworks wird darauf geachtet, dass die beschriebenen Komponenten einen breiten Sprachumfang abdecken und so in möglichst vielen DSL-Projekten einsetzbar sind. Trotzdem kann es notwendig sein, das DVF zu modifizieren:

- Das DVF soll um neue Produktionsregeln ergänzt werden können. Dabei kann es sich beispielsweise um einen bestimmten Operator handeln, der im Rahmen einer Expression Verwendung findet.
- Das DVF stellt eine automatische Validierung bereit, die unter anderem Typsicherheit verifiziert. Die Validierung soll um neue Eigenschaften ergänzt werden können.
- Das DVF soll um neue Transformatoren ergänzt werden können, die ein Modell in eine neue Zielsprache, wie beispielsweise C, übersetzen.

Das Ziel dieses Abschnitts ist es, die drei genannten Erweiterungsmöglichkeiten genauer vorzustellen. Er gliedert sich wie folgt: Unterkapitel 4.8.1 zeigt, wie das DVF um eine Produktionsregel ergänzt werden kann. Dazu gehört auch eine Modifikation der Transformatoren, die das neue Sprachkonstrukt in eine Hoch- bzw. Model Checker-Eingabesprache überführen. Danach verdeutlicht Abschnitt 4.8.2 das Erweitern der Validierung. Zum Abschluss wird in Unterkapitel 4.8.3 erläutert, wie ein neuer Transformator in das DVF integriert werden kann.

4.8.1 Hinzufügen neuer Produktionsregeln

Dieser Abschnitt zeigt, wie neue Produktionsregeln zum DSL Verification Framework hinzugefügt werden können. Diese Regeln kann der DSL-Entwickler im Anschluss in

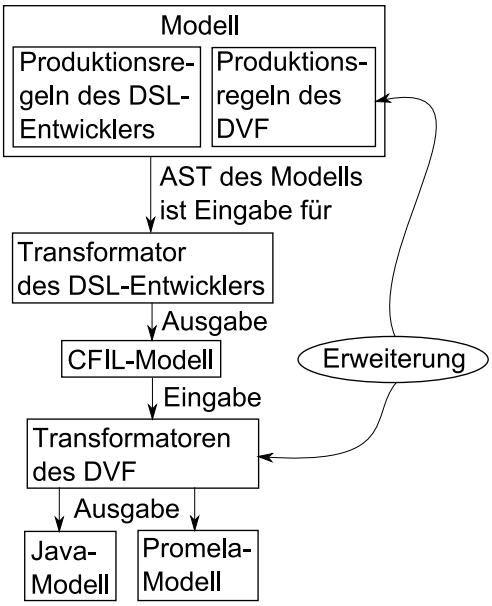

Abbildung 4.24: Erweiterung von Produktionsregeln

die Grammatik seiner domänenspezifischen Sprache integrieren, um sie so mit neuen Sprachkonstrukten zu ergänzen. Der Abschnitt gliedert sich wie folgt: Zunächst wird das allgemeine Ergänzungskonzept vorgestellt. Danach verdeutlicht ein Beispiel das konkrete Vorgehen.

Abbildung 4.24 zeigt den Erweiterungsansatz. Auf der linken Seite des Diagramms ist der bereits vorgestellte Workflow des DVF in Form von Rechtecken zu sehen: Ein DSL-Anwender hat ein Modell mit einer domänenspezifischen Sprache beschrieben. Er nutzt dafür eine DSL, die ein DSL-Entwickler mit dem DVF umgesetzt hat. Dementsprechend enthält das Modell Sprachkonstrukte, die Teil des DVF sind und Sprachkonstrukte, die der DSL-Entwickler manuell implementiert hat. Ein Parser überführt das Modell in einen Abstract Syntax Tree, den wiederum ein Transformator einliest und in ein CFIL-Modell übersetzt. Das Ergebnis können die Transformatoren des DVF nach Java bzw. Promela überführen.

Obwohl das DVF eine breite Palette von Sprachkonstrukten zur Verfügung stellt, kann es notwendig sein, neue Sprachkonstrukte einzufügen und so das DSL Verification Framework zu erweitern. Daher ist auf der rechten Seite von Abbildung 4.24 in Form einer Ellipse zu sehen, welche Komponenten des DVF angepasst werden müssen. Es wird deutlich, dass zunächst eine Modifikation der vom DVF zur Verfügung gestellten Produktionsregeln notwendig ist. Im Anschluss kann das neue Sprachkonstrukt bereits

in eine Grammatik integriert und zum Beschreiben eines Modells genutzt werden. Für eine erfolgreiche Verifikation bzw. Überführung in eine Hochsprache ist jedoch auch eine Erweiterung der Transformatoren notwendig. Das bedeutet, in einem zweiten Schritt müssen die Transformatoren des DVF so modifiziert werden, dass sie das neue Sprachkonstrukt in die jeweilige Zielsprache überführen.

Das folgende Beispiel verdeutlicht den beschriebenen Ansatz: Zu den Statements des DVF soll ein Sprachkonstrukt hinzugefügt werden, das einen Wert quadriert und das Ergebnis in einer Variable speichert. Abbildung 4.24 zeigt, dass dazu in einem ersten Schritt eine Modifikation der Produktionsregeln des DVF notwendig ist. Das folgende Listing demonstriert das Vorgehen:

```
1 DVFStatement:
2   (assign=DVFAssignmentOrCall) | (assert=DVFAssert) |
3   (send=DVFSendSignal) | (read=DVFReadSignal) |
4   (ret=DVFReturn) | (loop=DVFLoop) |
5   (ifelse=DVFIfElse) | (wait=DVFWait) |
6   (pow=EXTPow)
7 ;
```

Listing 4.118: Erweiterung von DVFStatement

Das obige Listing 4.118 enthält die Produktionsregel *DVFStatement*, die Teil des DSL Verification Frameworks ist. Verglichen mit der ursprünglichen Version aus Listing 4.22 ist *DVFStatement* in Zeile 6 um eine Ableitung erweitert worden. Die nächste Produktionsregel zeigt die Umsetzung von *EXTPow*:

```
1 EXTPow:
2   "pow" "("
3     expr=DVFExpression "," dest=[DVFVariableDeclare]
4   ")" ";"
5 ;
```

Listing 4.119: Erweiterung von DVFStatement

EXTPow setzt exemplarisch das neue Sprachkonstrukt um. Als Prefix wird *EXT* gewählt, um sich von den Produktionsregeln des DVF abzugrenzen und so die Lesbarkeit zu erhöhen. Die Methode *pow()* bekommt zwei Parameter übergeben: Die zu quadrierende Expression und den Bezeichner einer Variable, in die der Rückgabewert gespeichert werden soll. Somit ist die Syntax der Erweiterung vollständig definiert und sie kann in den Modellen der domänenspezifischen Sprache Verwendung finden.

Damit das neue Sprachkonstrukt verifiziert bzw. in eine Hochsprache überführt werden kann, ist auch eine Modifikation der Transformatoren notwendig. Für die prototypische Umsetzung des DSL Verification Frameworks sind die Transformatoren mit Xpand implementiert. Daher gibt es zwei Xpand-Methoden, die für jedes Statement in einem

Modell aufgerufen werden und es nach Promela bzw. Java überführen. Das folgende Listing zeigt eine der Methoden und verdeutlicht, wie sie erweitert werden muss:

```
1 «DEFINE transformStatementJava FOR DVFStatement-»
2 «FILE "result.dvf"-»
3   «IF assign!=null-»
4     //...
5   «ELSE IF assert!=null-»
6     //...
7   //...
8   «ELSEIF pow!=null-»
9     «pow.dest» = («pow.expr») * («pow.expr»);
10  «ELSE»
11    «error()»
12  «ENDIF»
13 «ENDFILE-»
14 «ENDDEFINE-»
```

Listing 4.120: Erweiterte Transformation von Statements

Die Methode *transformStatement()* ist Teil des DVF, erhält als Parameter ein Objekt vom Typ *DVFStatement* und überführt es nach Java. Der MWE2-Workflow (vgl. Abschnitt 2.2.1) ruft sie automatisiert für jedes Objekt vom *DVFStatement* auf, das Teil des Abstract Syntax Trees ist. Die Methode führt eine Fallunterscheidung durch und überprüft, ob es sich bei dem jeweiligen Statement um eine Zuweisung, eine Assertion, etc. handelt. Für das neue Sprachkonstrukt *pow()* wird in den Zeilen 8 bis 9 ein neuer If-Block eingefügt. Er überprüft, ob das Statement dem Sprachkonstrukt *pow()* entspricht und überführt es nach Java. Neben der Xpand-Methode *transformStatementJava()* gibt es auch die Methode *transformStatementPromela()*. Aufbau und Modifikation sind analog zu Listing 4.120, daher werden sie in diesem Abschnitt nicht weiter vorgestellt.

4.8.2 Modifizieren der Validierung

Neben dem Hinzufügen oder Modifzieren eines Sprachkonstrukts bzw. einer Transformationsregel kann es auch wünschenswert sein, die Validierung des DSL Verification Frameworks anzupassen. Eine genaue Vorstellung der Validierung erfolgt in Abschnitt 4.5. Dabei wird deutlich, dass das DSL Verification Framework eine Menge von sogenannten Validierungsmethoden enthält. Jeder dieser Methoden ist die Annotation *Check* vorangestellt. Sie bewirkt, dass die entsprechende Methode automatisch beim Validieren des Abstract Syntax Trees aufgerufen wird. Das folgende Beispiel zeigt, wie das DVF um eine Validierung des Sprachkonstrukts *pow()* (vgl. Listing 4.119) ergänzt werden kann:

```
1 @Check
2 public void checkPow(EXTPow var){
3   ValDataType dt = getExprType(var.getExpr());
4   if(dt.getType() != ValDataTypeEnum.Int)
5     error("Datentyp kein Integer", 0);
6   return;
7 }
```

Listing 4.121: Validierung eines neuen Sprachkonstrukts

Das obige Listing 4.121 zeigt eine Validierungsmethode. Sie wird durch die Check-Annotation automatisch für jedes Element im Abstract Syntax Tree aufgerufen, das vom Typ *EXTPow* ist. Das Sprachkonstrukt *pow()* quadriert einen Wert, der als Parameter in Form einer Expression übergeben wird. Expressions können unterschiedliche Datentypen, wie beispielsweise Boolean oder Integer zurückliefern. Die Validierung stellt sicher, dass der zu quadrierende Wert vom Typ Integer ist. Dafür findet die Methode *getExprType()* Verwendung, die bereits in Abschnitt 4.5.1 vorgestellt wird.

4.8.3 Hinzufügen eines neuen Transformators

Neben dem Modifizieren der Produktionsregeln oder der Validierung kann es auch wünschenswert sein, einen komplett neuen Transformator zu entwickeln, der das Modell in eine weitere Zielsprache übersetzt. Dafür sind in Abbildung 4.25 zwei verschiedene Ansätze zu sehen. Die linke Seite zeigt den bekannten Workflow des DVF: Der DSL-Anwender beschreibt mit der domänenspezifischen Sprache ein Modell, das zunächst in die CFIL und danach in die jeweiligen Zielsprachen übersetzt wird. Auf der rechten Seite ist ein neuer Transformator eingefügt. Die gestrichelten Linien zeigen an, dass dieser auf zwei verschiedene Arten umgesetzt werden kann:

- Die CFIL wird als Eingabesprache genutzt. Dies hat den Vorteil, dass der Transformator in weiteren DSL-Projekten wiederverwendet werden kann, sobald der DSL-Entwickler eine Übersetzung in die CFIL implementiert hat.
- Die DSL wird als Eingabesprache für den neuen Transformator genutzt. Dies ist vorteilhaft, wenn die DSL und die Zielsprache sehr ähnlich sind und in diesem Fall ein Transformator mit geringem Aufwand umgesetzt werden kann. Der Nachteil ist, dass der Transformator in anderen Projekten mit einer neuen DSL nicht wiederverwendet werden kann.

In beiden Fällen muss vom DSL-Entwickler auf Konsistenz geachtet werden: Die Semantik der Ausgabe des neuen Transformators darf nicht von den bestehenden Transformatoren abweichen.

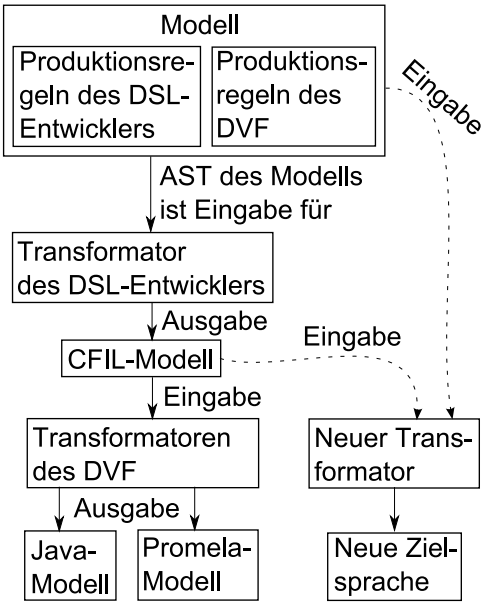

Abbildung 4.25: Zusätzlicher Transformator im DVF

5 Fallstudien

In den vorangegangenen Kapiteln ist das DSL Verification Framework vorgestellt worden. Es dient zur Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation. Zu diesem Zweck stellt das DVF Produktionsregeln für verschiedene Sprachkonstrukte zur Verfügung, die ein DSL-Entwickler in seine Grammatik integrieren kann. Des Weiteren beinhaltet das DVF Transformatoren, die eine domänenspezifische Sprache automatisiert in eine Hoch- und Model Checker-Eingabesprache überführen. Besondere Schwerpunkte des DVF sind, gemäß der Analyse des Stands der Forschung, die Reduktion des Zustandsraums durch Symmetrie, eigenschaftserhaltene Abstraktion (vgl. Abschnitt 3.1) und das Bereitstellen von Sprachkonstrukten zum Modellieren von Verhalten (vgl. Abschnitt 3.2).

Um die Anwendbarkeit des DVF nachzuweisen, sind im Rahmen dieser Arbeit und in Kooperation mit der *soft2Tec GmbH* [132] zwei Industriefallstudien durchgeführt worden. Jede Fallstudie verknüpft das Prinzip der modellgetriebenen Entwicklung mit der formalen Verifikation. Dies hat den Vorteil, dass Quellcode automatisiert aus Modellen generiert wird, die zuvor ein Model Checker überprüft hat (vgl. Kapitel 1).

Für jede Fallstudie ergibt sich der folgende Aufbau: Es wird eine Software entwickelt, die zum Teil aus einer DSL und zum Teil aus einer Hochsprache besteht. Die Umsetzung der domänenspezifischen Sprache erfolgt unter Zuhilfenahme des DVF. Nach der Entwicklung einer Grammatik und eines Transformators in die Control Flow Intermediate Language wird die DSL genutzt, um ein Modell und Anforderungen zu beschreiben. Im Anschluss überführt der Transformator das Modell in die Model Checker-Eingabesprache Promela, damit es von Spin verifiziert werden kann. Wenn Spin meldet, dass eine der Anforderungen nicht erfüllt ist, kann der DSL-Anwender das Modell modifizieren und den Fehler beheben. Sobald der Model Checker keine weiteren Fehler findet, erfolgt die Transformation in eine Hochsprache und die Integration in das entsprechende Softwareprojekt.

Die erste Fallstudie, die Teil von Abschnitt 5.1 ist, beinhaltet eine domänenspezifische Sprache für endliche Automaten. Die DSL wird in einem Projekt namens *AssyControl* genutzt, dessen Ziel die Implementierung eines Systems zur Überwachung von Handmontagen mittels Ultraschall ist. Die zweite Fallstudie in Abschnitt 5.2 entwickelt eine domänenspezifische Sprache zur Beschreibung von Web-Applikationen. Die Nutzung der DSL erfolgt in dem Projekt *inTune*, das das Testen in verteilten Systemen ermöglicht und das über eine Web-Oberfläche gesteuert wird.

5.1 Fallstudie 1: AssyControl

Dieser Abschnitt stellt die erste Fallstudie vor, um das Konzept des DSL Verification Frameworks zu evaluieren. Dabei wird untersucht, inwiefern die vordefinierten Produktionsregeln ausreichend sind, um in einem Industrieprojekt Verwendung zu finden. Von besonderem Interesse ist auch der benötigte Aufwand für die Implementierung des CFIL-Transformators und somit die Frage, ob der Einsatz des DVF in einem Softwareprojekt lohnenswert ist.

Das Ziel der ersten Fallstudie ist die Erweiterung eines Systems namens *AssyControl*, das die *soft2Tec GmbH* entwickelt. Es überwacht die Handmontage an Arbeitsplätzen mittels Ultraschall und warnt einen Arbeiter, wenn ein falsches Bauteil gegriffen oder die Zusammenbaureihenfolge nicht eingehalten wird. Ein konkretes Beispiel ist das Bestücken einer Platine mit verschiedenen Widerständen, die sich in unterschiedlichen Aufbewahrungsbehältern befinden. In diesem Fall kann AssyControl den Arbeiter warnen, wenn er einen falschen Widerstand greift. Das Ziel der ersten Fallstudie ist es, AssyControl mit einem speziellen Konfigurationsmodus zu erweitern, der die Nutzung des Systems erleichtert, wenn sich mehrere Arbeiter einen Montageplatz teilen.

Bei AssyControl handelt es sich um eine reaktive Anwendung, die die Signale eines Ultraschallsystems empfängt. Jedes Signal beinhaltet Informationen über die Bewegungen des Arbeiters an seinem Montageplatz. AssyControl analysiert sie und informiert den Anwender, wenn ein Fehler beim Zusammenbau gemacht wurde. Endliche Automaten sind gut geeignet, um derartige reaktive Systeme zu modellieren [111]. Deshalb wird die AssyControl-Erweiterung mit einer domänenspezifischen Sprache für Statecharts implementiert.

Die DSL, die Transformatoren und ihre Umsetzung mit dem DVF sind Gegenstand dieses Abschnitts. Die Anwendung des DSL Verification Framework stellt Abbildung 1.8 in allgemeiner Form vor. Der sich daraus ergebende Workflow für die erste Fallstudie ist in Abbildung 5.1 visualisiert. Abgerundete Ecken stehen für Komponenten, die der DSL-Entwickler implementiert. Gestrichelte Ränder symbolisieren Elemente, die vom DVF bereitgestellt werden. Abbildung 5.1 zeigt, dass der DSL-Entwickler zunächst eine Grammatik beschreibt. Sie setzt sich aus Produktionsregeln des DVF und benutzerspezifischen Produktionsregel zusammen. Zu den DVF-Sprachkonstrukten gehören beispielsweise Statements, um Verhalten zu beschreiben. Zustände und Transitionen sind hingegen benutzerspezifische Komponenten.

Modelle, die mit der domänenspezifischen Sprache für Statecharts beschrieben sind, sollen in eine Hoch- und eine Model Checker-Eingabesprache überführt werden. Zu diesem Zweck stellt das DVF einen Transformator bereit, der Sprachkonstrukte des DVF automatisiert nach Java und Promela überführt. Die domänenspezifische Sprache für endliche Automaten enthält jedoch auch Elemente, die nicht Teil des DSL Verification Frameworks sind. Deshalb implementiert der DSL-Entwickler einen Transformator, der

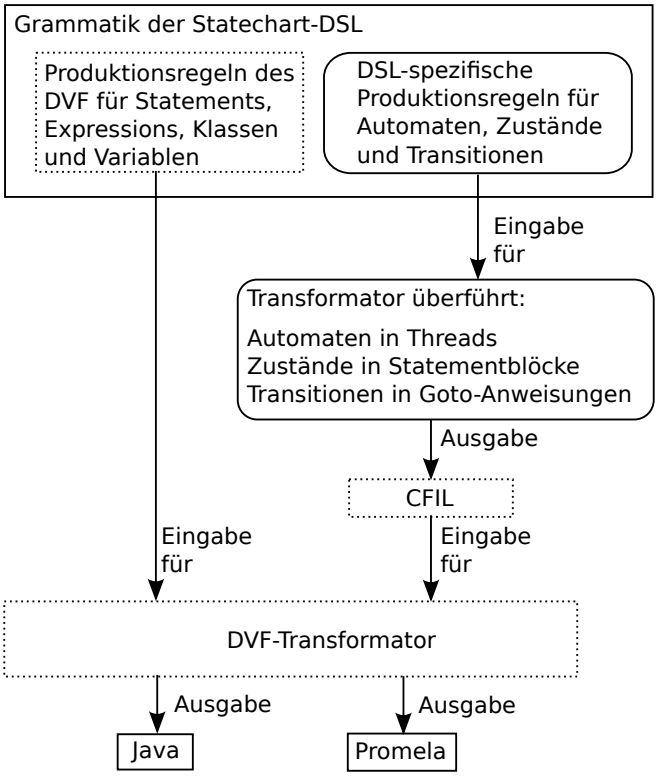

Abbildung 5.1: Umsetzung der ersten Fallstudie mit dem DVF

alle benutzerspezifischen Sprachkonstrukte in die Control Flow Intermediate Language (CFIL) übersetzt. Er überführt beispielsweise Transitionen in Goto-Anweisungen.

Aus der ersten Fallstudie und dem DVF leitet sich für dieses Unterkapitel die folgende Struktur ab: Zunächst gibt Abschnitt 5.1.1 eine detaillierte Einführung in AssyControl bzw. die zu implementierende AssyControl-Erweiterung. Daraus wird in Unterkapitel 5.1.2 eine domänenspezifische Sprache namens Statechart-DSL abgeleitet und eine entsprechende Grammatik umgesetzt. Sie besteht sowohl aus benutzerspezifischen Produktionsregeln, als auch aus Produktionsregeln des DSL Verification Frameworks. Danach stellt Abschnitt 5.1.3 einen Transformator vor, der Modelle, die mit der Statechart-DSL implementiert wurden, in die CFIL übersetzt. Anschließend zeigt Unterkapitel 5.1.4, wie mit der Statechart-DSL die AssyControl-Erweiterung modelliert, nach Java bzw. Promela transformiert und durch einen Model Checker verifiziert werden kann. Zum Abschluss erfolgt in Abschnitt 5.1.5 eine Auswertung, ob der Einsatz des DSL Verification Framework von Vorteil ist.

Abbildung 5.2: AssyControl-Arbeitsplatz

5.1.1 Beschreibung der AssyControl-Erweiterung

Das Ziel der ersten Fallstudie ist die Erweiterung von AssyControl. Deshalb stellt dieser Abschnitt zunächst die Funktionsweise des AssyControl-Systems bzw. ihrer Erweiterung vor. Daraus wird im weiteren Verlauf der vorliegenden Arbeit eine DSL abgeleitet, um die erste Fallstudie zu modellieren bzw. eine Übersetzung nach Java und Promela durchzuführen.

AssyControl basiert auf einem PC, drei Ultraschallempfängern und mindestens einem Ultraschallsender. Abbildung 5.2 zeigt wie das System in der Praxis eingesetzt werden kann: Ein Arbeiter steht an einer Montagewerkbank und baut beispielsweise PKW-Steuergeräte zusammen. Deshalb greift er der Reihe nach verschiedene Komponenten aus den entsprechenden Aufbewahrungsbehältern und baut diese auf einer Arbeitsfläche zusammen.

Durch die sich immer wiederholende, monotone Arbeit kann es zu Fehlern kommen. Daher trägt der Arbeiter auf seiner linken und rechten Hand zwei sogenannte *Marker*. Bei einem Marker handelt es sich um einen Ultraschallsender, der in regelmäßigen Abständen ein Ultraschallsignal erzeugt. Die drei Ultraschallempfänger von AssyControl sind in Abbildung 5.2 nicht zu sehen. Sie sind über der Werkbank angebracht und empfangen die entsprechenden Ultraschallsignale. Aufgrund der Signallaufzeiten kann AssyControl die Koordinaten der Marker im Raum berechnen und erkennen, wenn ein Arbeiter die vorgeschriebene Zusammenbaureihenfolge nicht einhält oder die falschen

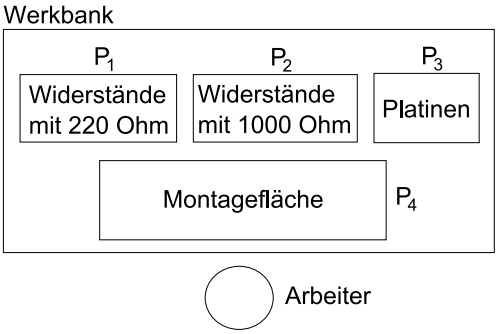

Abbildung 5.3: Beispielhafter Arbeitsplatz mit AssyControl

Komponten aus Aufbewahrungsbehältern greift. In diesem Fall kann der Arbeiter visuell oder akkustisch gewarnt werden und sofern dies möglich ist über einen Bildschirm einen Korrekturvorschlag erhalten.

Bevor ein Montageprozess das erste Mal ausgeführt wird, muss AssyControl von einem Vorarbeiter mit den folgenden Daten programmiert werden:

- Die Werkbank bzw. Montagefläche wird in mehrere, dreidimensionale Bereiche unterteilt. Diese Bereiche werden im weiteren Verlauf der Arbeit als *Positions* bezeichnet.
- Die verschiedenen Komponenten, die sich auf dem Montageplatz befinden.
- Eine Zuordnung, welche Komponente sich auf welcher Position befindet. Dabei kann eine Komponente auch mehreren Positions zugeordnen werden. Dies bildet den Umstand ab, dass sich an einer Werkbank auch mehrere Behälter befinden können, die beispielsweise den selben Schraubentyp beinhalten.
- Die Zusammenbaureihenfolge der einzelnen Komponenten.

Die genaue Funktionsweisweise von AssyControl wird anhand des folgenden Beispiels demonstriert: In Abbildung 5.3 ist eine Werkbank von oben zu sehen, die von einem Vorarbeiter in die vier Positions P_1, P_2, P_3 und P_4 eingeteilt worden ist. Die Positions P_1, P_2 und P_3 enthalten Widerstände [136] und Platinen. P_4 ist leer und dient als Montagefläche. Der Arbeiter trägt lediglich einen Marker an einer seiner beiden Hände, die zu Beginn des Montageprozesses bei P_4 ruht. Die Zusammenbaureihenfolge sieht vor, dass der Arbeiter zunächst eine Platine greift und auf P_4 ablegt. Danach nimmt er einen Widerstand mit 1000 Ohm und befestigt ihn auf der Platine. Zum Abschluss wird ein Widerstand mit 220 Ohm gegriffen und auf der Platine angebracht. Der Montageprozess besteht für AssyControl somit aus einer Reihe von Bewegungen zu den folgenden

Positions: P_4, P_3, P_4, P_2, P_4, P_1, P_4. Wenn ein Fehler beim Montageprozess gemacht wird, erkennt AssyControl, dass der Marker zu einer Position bewegt wird, die nicht der vorgesehenen Zusammenbaureihenfolge entspricht und meldet einen Fehler.

Wenn sich Zusammenbaureihenfolge, Komponenten oder Positions ändern, muss AssyControl neu konfiguriert werden. Um den Konfigurationsaufwand zu minimieren, erfolgt im Rahmen dieser Fallstudie die Entwicklung einer AssyControl-Erweiterung. Sie ergänzt AssyControl um einen Konfigurationsmodus, dessen Konzept das folgende Beispiel verdeutlicht: Gegeben sei eine Werkbank, die bereits vom Vorarbeiter in Positions eingeteilt worden ist. Auch die Komponenten und ihre Zusammenbaureihenfolge sind bekannt. Der Arbeiter, der an der Werkbank die Komponenten montiert, ist Rechtshänder. Nach einem Schichtwechsel nimmt jedoch ein Arbeiter an der Werkbank Platz, der Linkshänder ist. Damit er komfortabler an der Werkbank arbeiten kann, verändert er den Arbeitsplatz. Bezogen auf Abbildung 5.3 kann dies beispielsweise bedeuten, dass die Platinen nach P_1 und die Widerstände mit 220 Ohm nach P_3 verschoben werden. Die Größe bzw. Koordinaten der Positions und die Zusammenbaureihenfolge der Komponenten bleiben jedoch unverändert.

Damit der Arbeiter an der Werkbank mit dem Montageprozess beginnen kann, muss AssyControl neu programmiert werden. Der Konfigurationsmodus, der im Rahmen dieser Fallstudie implementiert wird, hat das Ziel, die Neuprogrammierung zu erleichtern bzw. zu beschleunigen. Er hat die folgende Funktionalität: Nachdem der neue Arbeiter an der Werkbank Platz genommen hat, wird AssyControl in den Konfigurationsmodus versetzt. Der Arbeiter führt den Montagesprozess einmalig aus. Dabei wird davon ausgegangen, dass dies fehlerfrei geschieht. Die AssyControl-Erweiterung analysiert die Bewegungen und detektiert, da die Zusammenbaureihenfolge bekannt ist, bei welcher Position sich welche Komponente befindet. Danach kann der Konfigurationsmodus beendet und AssyControl wie gewohnt verwendet werden, um Fehler beim Zusammenbau zu erkennen.

Die Erweiterung wird exemplarisch für einen Arbeitsplatz entwickelt, der aus sechs Positions besteht und an dem Modellautos zusammengebaut werden. Neben den Positions P_1 bis P_6 gibt es drei Komponenten:

- Eine Box mit Reifen.
- Eine Box mit Achsen.
- Eine Box mit Cockpits.

Die drei Komponenten sind auf drei Positions verteilt, die jedoch im Konfigurationsmodus zu Beginn des Montageprozesses nicht bekannt sind. Zur Vereinfachung wird davon ausgegangen, dass der Arbeiter nur einen Marker trägt. Die Zusammenbaureihenfolge ist wie folgt festgeschrieben: Zuerst greift der Arbeiter eine Achse und legt sie auf eine der Arbeitsflächen. Danach greift er zweimal hintereinander einen Reifen und montiert

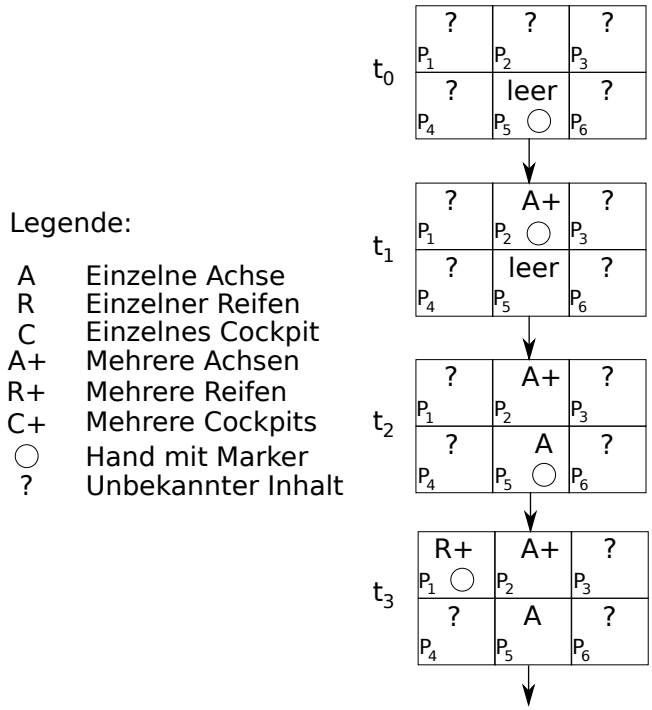

Abbildung 5.4: Zusammenbau von Achsen und Reifen

ihn an der Achse. Im Anschluss werden erneut nacheinander eine Achse und zwei Reifen gegriffen und auf einer der Arbeitsflächen montiert. Abschließend greift sich der Arbeiter ein Cockpit, bringt die beiden Achsen daran an und der Montageprozess gilt als beendet. Daraus ergibt für das Greifen der Komponenten die folgende Reihenfolge: Achse, Reifen, Reifen, Achse, Reifen, Reifen, Cockpit. Zu beachten ist, dass es drei Positions gibt, die zur Montage genutzt werden können. Somit kann es beispielsweise passieren, dass ein Arbeiter die erste Achse mit den entsprechenden Reifen auf der Postion P_5 zusammenbaut. Der Rest des Autos wird jedoch auf der Position P_6 montiert. In diesem Fall muss am Ende des Montageprozesses die Achse von P_5 nach P_6 bewegt und dort alle Komponenten zusammengefügt werden.

Abbildung 5.4 verdeutlicht den Ansatz der AssyControl-Erweiterung und zeigt die verschiedenen Phasen des Montageprozesses im Konfigurationsmodus zu den Zeitpunkten t_0, t_1, usw. Die Positions tragen von links nach rechts bzw. oben nach unten die Bezeichner P_1 bis P_6. Der Montageprozess beginnt zum Zeitpunkt t_0. Die Hand mit dem Marker ruht bei der leeren Position P_5. Der Inhalt der anderen Positions ist unbekannt und soll anhand der Bewegungen des Arbeiters erkannt werden. In einem ersten Montageschritt

bewegt der Arbeiter seine Hand von P_5 nach P_2. Das System geht im Konfigurationsmodus davon aus, dass die korrekte Zusammenbaureihenfolge eingehalten wird und der Arbeiter somit eine Achse greift. Deshalb detektiert der Konfigurationsmodus zum Zeitpunkt t_1: Bei Position P_1 befindet sich die Box mit den Modellbauachsen und es wird eine der Achsen gegriffen. Im Anschluss bewegt der Arbeiter seine Hand zurück nach P_4. Das System macht die folgende Annahme: Wenn die Hand etwas gegriffen hat und sich zu einer leeren Position bewegt, wird die entsprechende Komponente dort abgelegt. Danach greift der Arbeiter ein Objekt, das sich bei P_1 befindet. Da auch hier davon ausgegangen wird, dass der initiale Montageprozess fehlerfrei verläuft und der Arbeiter die Zusammenbaureihenfolge einhält, erkennt das System bei P_1 eine Box mit Reifen. Dieses Vorgehen wird so lange fortgesetzt, bis alle Komponenten zusammengebaut und der Inhalt aller Positionen erkannt ist. Danach kann der Konfigurationsmodus beendet werden.

Eine DSL zur Beschreibung des Konfigurationsmodus muss somit die folgenden Modellierungsmöglichkeiten bieten:

- Es muss eine Komponente geben, die auf die Bewegungen des Arbeiters reagiert und daraus den Aufbau des Montageplatzes ableitet. Sie soll als reaktives System mit einem endlichen Automaten umgesetzt werden können.
- Die Komponenten, die Zusammenbaureihenfolge und die *Positions* müssen abstrahiert werden können.
- Es muss eine Schnittstelle zur Ultraschall-Hardware geben, von der die Koordinaten der Marker abgefragt werden können.

Im weiteren Verlauf dieses Kapitels wird aus den genannten Anforderungen eine domänenspezifische Sprache abgeleitet. Um die Entwicklung bzw. Transformation der DSL zu erleichtern, findet das DVF Verwendung. Anschließend kann der DSL-Anwender die AssyControl-Erweiterung mit der DSL beschreiben, verifizieren und in ausführbare Software übersetzen.

5.1.2 Entwicklung der Statechart-DSL

Dieser Abschnitt beschreibt eine DSL zum Modellieren von endlichen Automaten, mit der im weiteren Verlauf der Arbeit die AssyControl-Erweiterung umgesetzt wird. Sie ist Teil einer Fallstudie, um die Anwendbarkeit des DVF zu evaluieren. Der Workflow des DSL Verification Frameworks ist in Abbildung 5.1 zu sehen. Es wird deutlich, dass ein DSL-Entwickler wie folgt vorgeht, um eine domänenspezifische Sprache mit dem DVF zu entwickeln: Zunächst muss eine Grammatik spezifiziert werden, die sich aus einer Menge von Produktionsregeln zusammensetzt. Das DVF stellt vorgefertigte Produktionsregeln zur Verfügung, um die Entwicklung der DSL zu beschleunigen. Wenn

bestimmte Sprachkonstrukte jedoch nicht Teil des DSL Verification Frameworks sind, muss sie der DSL-Entwickler manuell beschreiben.

Daraus ergibt sich für diesen Abschnitt das folgende Vorgehen: Zunächst wird der Aufbau einer DSL für endliche Automaten diskutiert. Anschließend beschreibt eine Grammatik die sich daraus ergebende domänenspezifische Sprache. Für Sprachkonstrukte, die bereits im DVF enthalten sind, finden die entsprechenden vorgefertigten Produktionsregeln Verwendung. Dabei wird auch ersichtlich: Das DVF kann zwar die Entwicklung einer Grammatik vereinfachen, es muss jedoch trotzdem seitens des DSL-Entwicklers ein gewisses Maß an Expertenwissen im Bereich Compilerbau [1] vorhanden sein, um eigene Produktionsregeln umzusetzen. Zum Abschluss verdeutlicht ein Beispiel die Anwendung der domänenspezifischen Sprache durch die Modellierung zweier endlicher Automaten.

Die domänenspezifische Sprache wird im weiteren Verlauf dieses Abschnitts als Statechart-DSL [5][4] bezeichnet. Sie setzt, basierend auf den Anforderungen der AssyControl-Erweiterung, die folgenden Eigenschaften der UML-Statecharts (vgl. Abschnitt 2.3) um:

- Das Verhalten von Klassen kann mit endlichen Automaten beschrieben werden. Jeder Automat besteht aus einer Menge von Zuständen und einer Menge von Transitionen. Die Automaten finden Verwendung, um den Teil der AssyControl-Erweiterung zu implementieren, der auf die Signale der Ultraschall-Hardware reagiert.
- Neben einfachen Zuständen (den sogenannten Simple-States) beinhaltet ein Automat genau einen Startzustand und mindestens einen Endzustand.
- Hierarchie wird mit zusammengesetzten Zuständen, den sogenannten Composite-States, umgesetzt.
- Jeder Simple-State kann eine Entry- und eine Exit-Action enthalten. Entry- und Exit-Actions kapseln eine Menge von Statements, die beim Betreten oder Verlassen eines Zustands ausgeführt werden.
- Nebenläufige Automaten können über den Versand asynchroner Nachrichten miteinander kommunizieren.
- Die Ausführbarkeit einer Transition kann von einer Expression (Guard) oder dem Erhalt einer asynchronen Nachricht (Trigger) abhängig sein.
- Die Statechart-DSL ermöglicht das Deklarieren von Klassen, Aufzählungen und globalen Variablen, um Komponenten bzw. Positions modellieren zu können.
- Eigenschaftserhaltende Abstraktion für die Schnittstelle zur Ultraschall-Hardware.

Im weiteren Verlauf dieses Abschnitts wird gezeigt, wie die genannten Eigenschaften als domänenspezifische Sprache, unter Verwendung des DVF, umgesetzt werden können. Ein

DSL-Entwickler muss dafür gemäß Unterkapitel 2.2 eine Grammatik entwerfen. Dabei ist darauf zu achten, dass für Sprachkonstrukte, die bereits im DSL Verification Framework vorhanden sind, die vorgefertigten Produktionsregeln des DVF Verwendung finden. Es gilt die folgende Namenskonvention: Sprachkonstrukte, die der DSL-Entwickler manuell beschreibt, haben das Prefix *STM*. Die Produktionsregeln des DSL Verification Frameworks beginnen hingegen mit der Signatur *DVF*.

Die Referenzimplementierung des DSL Verification Frameworks basiert auf Xtext und Xpand. Deshalb wird im weiteren Verlauf dieses Abschnitts die Grammatik der Statechart-DSL mit Xtext beschrieben:

```
1 Model:
2   (enums+=DVFEnumDeclare)*
3   (globalVariables+=DVFVariableDeclare)*
4   (scalarsets+=DVFScalarDeclare)*
5   (fsminstances+=STMStmInstance)+
6   (classes+=DVFClass)*
7   (fsms+=STMStmClass)*
8 ;
```

Listing 5.1: Startregel der Statechart-DSL

Das obige Listing 5.1 enthält die Produktionsregel *Model*. Dabei handelt es sich um die Startregel der Grammatik und somit die spätere Wurzel des Abstract Syntax Trees. Jedes *Model* der Statechart-DSL besteht aus Aufzählungen, Klassendeklarationen, globalen Variablen und endlichen Automaten. Das DVF beinhaltet bereits Sprachkonstrukte für Variablen, Klassen und Aufzählungen. Damit diese Elemente in einem Modell deklariert werden können, muss der DSL-Entwickler die entsprechenden Produktionsregeln, also *DVFEnumDeclare*, *DVFVariableDeclare*, *DVFScalarDeclare* und *DVFClass*, in seine Grammatik einfügen und referenzieren. Die Referenzen sind in Listing 5.1 in den Zeilen 2, 3, 4 und 6 zu sehen. Eine Alternative ist die manuelle Umsetzung der Produktionsregeln, was jedoch den Arbeitsaufwand beim Schreiben der Grammatik erhöht.

Im Gegensatz zu Variablen und Klassen gibt es im DVF für die Modellierung endlicher Automaten keine vorgefertigten Sprachkonstrukte. Somit muss der DSL-Entwickler die entsprechenden Produktionsregeln manuell umsetzen. Des Weiteren sollen Automaten nicht nur deklariert, sondern auch instanziiert werden können. Dies ermöglicht es dem DSL-Anwender in seinem Modell genau zu spezifizieren, wie viele nebenläufige Automaten auszuführen sind. Aus diesem Grund gibt es in Listing 5.1 in den Zeilen 5 und 7 zwei Referenzen auf die Produktionsregeln *STMStmClass* und *STMStmInstance*:

- *STMStmClass* repräsentiert einen endlichen Automaten, der aus Transitionen, Zuständen und einem eindeutigen Bezeichner besteht.
- *STMStmInstance* repräsentiert die Instanz eines Automaten und hat einen eindeutigen Bezeichner.

Eine alternative Implementierung ist das Weglassen von *STMStmInstance*. In diesem Fall müsste die Semantik der DSL (und somit der in Abschnitt 5.1.3 vorgestellte Transformator in die CFIL) so angepasst werden, dass jede Automatendeklaration implizit auch eine Instanz darstellt. Dies führt jedoch zu dem Nachteil, dass mehrere Instanzen eines Automaten nicht mehr referenziert werden können, um beispielsweise ein asynchrones Signal zu übertragen. Die nächsten Listings zeigen die Umsetzung *STMStmInstance* und *STMStmClass* im Detail:

```
1 STMStmInstance:
2   "fsm" type=[STMStmClass]
3   name=ID
4   (array="[" size=INT "]")? ";"
5 ;
```

Listing 5.2: Produktionsregel für Automaten-Instanzen der Statechart-DSL

Das obige Listing 5.2 zeigt die Produktionsregel *STMStmInstance*, um Instanzen von endlichen Automaten bilden zu können. Jede Instanziierung beginnt mit dem Schlüsselwort *fsm*. Darauf folgt der Name des Automaten und ein eindeutiger Bezeichner. Es können auch mehrere Instanzen eines Automaten durch das Spezifizieren von eckigen Klammern gestartet werden. Der Vorteil dieser Art der Implementierung ist, dass jede Automateninstanz in einem Modell durch Angabe des entsprechenden Bezeichners referenziert werden kann, um Signale zu übertragen oder auf lokale Variablen zuzugreifen.

Damit die Instanz eines Automaten ausgeführt werden kann, muss er zunächst deklariert werden. Dafür steht das folgende Sprachkonstrukt zur Verfügung:

```
1 STMStmClass:
2   "stm_class" name=ID "{"
3     (variables+=DVFVariableDeclare ";")*
4     (sensitives+=STMSensitive)*
5     (states+=STMState)*
6   "}"
7 ;
```

Listing 5.3: Produktionsregel für Automaten-Deklarationen der Statechart-DSL

Das obige Listing 5.3 enthält die Produktionsregel *STMStmClass*, die das Deklarieren endlicher Automaten ermöglicht. Endliche Automaten werden in der Statechart-DSL mit sogenannten Automaten-Klassen beschrieben, die das Schlüsselwort *stm_class* einleitet. Da es sich bei den Automaten-Klassen um Klassen handelt, deren Verhalten mit einem endlichen Automaten beschrieben ist, soll das Deklarieren lokaler Variablen möglich sein. Das DVF enthält bereits eine Produktionsregel für Variablendeklarationen. Deshalb wird *DVFVariableDeclare* in Zeile 3 referenziert, um lokale Variablen innerhalb eines Automaten zu ermöglichen.

Gemäß der UML-Spezifikation [111] müssen Automaten eine Menge von Zuständen enthalten. Des Weiteren kann ein Automat sensitiv auf bestimmte Signale sein. Signale und Zustände (vgl. Listing 5.3 Zeilen 4 und 5) sind jedoch nicht Teil des DVF. Deshalb muss der DSL-Entwickler sie manuell beschreiben. Dabei ist zu beachten: Das DVF beinhaltet ein Sprachkonstrukt, um Message-Queues zu deklarieren (vgl. Listing 4.11). Dabei handelt es sich jedoch nicht um eine Menge von Signalen, auf die ein Automat sensitiv ist. Deshalb kann es an dieser Stelle nicht genutzt werden. Der DSL-Entwickler muss stattdessen eine eigene Produktionsregel entwerfen, die ein Signal repräsentiert, das ein Automat empfängt oder versendet:

```
1 STMSensitive:
2   "sensitive" name=ID "{"
3     (firstVariable=DVFVariableDeclare
4     "," variables+=DVFVariableDeclare)*)?
5   "}"
6 ;
```

Listing 5.4: Produktionsregel für eine Sensitivitätsliste

Das obige Listing 5.4 enthält die Produktionsregel *STMSensitive*, zum Beschreiben eines Signals. Gemäß der UML-Spezifikation [111] kann ein Signal zusammen mit einer Menge von Parametern übertragen werden. Dementsprechend besteht die Produktionsregel *STMSensitive* aus dem Schlüsselwort *sensitive*, einem Bezeichner und optional einer Liste von Parametern. Die Parameter eines Signals entsprechen den Parametern einer Methode. Sie bestehen somit aus einem Datentyp und einem Bezeichner. Aus diesem Grund muss eine entsprechende Produktionsregel nicht von Hand implementiert werden. Der DSL-Entwickler kann stattdessen die im DSL Verification Framework enthaltene Produktionsregel *DVFVariableDeclare* referenzieren.

Neben einer Liste von Signalen besteht ein Automat aus einer Menge von Zuständen. Die Produktionsregel für das Ableiten von Zuständen wird mit dem folgenden Listing vorgestellt:

```
1 STMState:
2   type=STMStateType name=ID
3   "{"
4     ("entry" "{" entryStatements+=DVFStatement+ "}")?
5     ("exit" "{" exitStatements+=DVFStatement+ "}")?
6     states+=STMState*
7     out+=STMTransition+
8   "}"
9 ;
10
```

```
11 STMStateType: "initial" | "final" | "state" | "composite"
```

Listing 5.5: Xtext-Grammatik für Zustände

Gemäß der Statechart-Spezifikation aus Abschnitt 2.3 gibt es verschiedene Zustandstypen. Dieser Umstand muss auch in der Statechart-DSL berücksichtigt werden. Deshalb beginnt jede Zustandsdeklaration in Listing 5.5 mit einem Schlüsselwort, das angibt, ob es sich um einen einfachen, zusammengesetzten, Start- oder Endzustand handelt. Darauf folgt ein eindeutiger Bezeichner als Zustandsname.

In der UML können Zustände Entry- und Exit-Actions beinhalten [111]. Sie kapseln wiederum eine Menge von Statements, die beim Betreten oder Verlassen des Zustands ausgeführt werden. Das DSL Verification Framework enthält bereits ein Sprachkonstrukt zum Umsetzen von Statements. Eine entsprechende Produktionsregel muss also seitens des DSL-Entwicklers nicht mehr von Hand implementiert werden. Daher wird in Listing 5.5 in den Zeilen 4 bis 5 die Produktionsregel *DVFStatement* referenziert.

Ein zusammengesetzter Zustand kann weitere Zustände enthalten [111]. Daher erfolgt in der Grammatik der Statechart-DSL eine rekursive Ableitung von *STMState*. Dies ist in Zeile 6 zu sehen. Des Weiteren müssen Transitionen in der Grammatik enthalten sein, um Zustandsübergänge modellieren zu können. In Listing 5.5 ist in Zeile 7 zu sehen, dass jeder Zustand eine Liste von ausgehenden Transitionen beinhaltet. Eine alternative Implementierung wäre es, die Transitionen innerhalb der Produktionsregel *STMStmClass* global (vgl. Listing 5.3) umzusetzen. Dies führt jedoch zu dem Nachteil, dass neben dem Zielzustand auch der Startzustand spezifiziert werden muss, was die Grammatik und die damit beschriebenen Modelle komplexer macht. Das nächste Listing zeigt den Aufbau von *STMTransition* im Detail:

```
1 STMTransition:
2   "->" destination=[STMState] "{"
3     ("trigger" "{" trigger=[STMSensitive] "}")?
4     ("guard" "{" guard=DVFExpression "}")?
5     ("effect" "{" effect+=DVFStatement* "}")?
6   "}"
7 ;
```

Listing 5.6: Xtext-Grammatik für Transitionen

Jede ausgehende Transition muss einen Zielzustand haben. Daher beginnt eine Transition in der Statechart-DSL mit dem Schlüsselwort -> und dem Namen des Zielzustands. Darauf folgen in geschweiften Klammern die optionalen Elemente. Dabei handelt es sich gemäß der Statechart-Spezifikation [111] um:

- einen sogenannten Trigger, der auf einen Signalnamen verweist. Die Transition kann nur ausgeführt werden, wenn auf der Message-Queue das entsprechende Signal anliegt.

- einen sogenannten Guard, der eine Expression enthält. Die Transition kann nur ausgeführt werden, wenn die Expression den Wert *true* zurückgibt.

- einen sogenannten Effect, der eine Menge von Statements enthält. Wenn die Transition ausführbar ist, werden vor dem eigentlichen Zustandswechsel alle Statements ausgeführt, die im Effect enthalten sind.

Listing 5.6 zeigt in Zeile 3, wie ein Trigger im Rahmen einer Produktionsregel beschrieben werden kann: Das Schlüsselwort *trigger* leitet die Deklaration eines Triggers ein. Darauf folgt eine Referenz auf die Produktionsregel *STMSensitive*, (vgl. Listing 5.4) die sicherstellt, dass nur der Name eines zuvor deklarierten Signals angegeben werden kann. Alternativ zu *STMSensitive* könnte auch die Produktionsregel *ID* Verwendung finden. Dies führt jedoch zu dem Nachteil, dass der Parser des DVF nicht automatisiert überprüft, ob der angegebene Signalname wirklich einem Signal entspricht, das zuvor deklariert wurde.

Die Umsetzung der Guard-Expression ist in Listing 5.6 in Zeile 4 gezeigt. Da das DVF bereits ein Sprachkonstrukt für Expressions enthält, muss der DSL-Entwickler keine eigene Produktionsregel umsetzen. Stattdessen kann *DVFExpression* referenziert werden.

Zeile 5 verdeutlicht die Umsetzung von Effect. Auch an dieser Stelle stellt das DVF eine passende Produktionsregel zur Verfügung: Der DSL-Entwickler muss die Statements, die in Effect gekapselt sind, nicht manuell beschreiben. Stattdessen integriert er aus dem DSL Verification Framework die Produktionsregel *DVFStatement*.

Bevor die Transformation der DSL beschrieben werden kann, ist eine Erweiterung des DVF gemäß Abschnitt 4.8.1 notwendig: Das DSL Verification Framework beinhaltet Statements, um Signale an eine Message-Queue zu senden oder sie zu empfangen. In der Statechart-DSL sollen Signale jedoch nicht direkt an eine Message-Queue gesendet werden, sondern an die Instanz einer Automaten-Klassen. Damit dies möglich ist, muss ein Sprachkonstrukt für eine neue Sende-Operation in das DVF eingefügt werden. Das folgende Listing zeigt die entsprechende Produktionsregel:

```
1 STMSend:
2   "sendToSTM" "("
3       destination=[STMStmInstace]
4       (array="[" index=INT "]")?
5       "," signal=[STMSensitive]
6       ("," params+=DVFExpression)*
7   ")"
8 ;
```

Listing 5.7: Xtext-Grammatik für Transitionen

Das obige Listing zeigt ein Sprachkonstrukt, mit dem Nachrichten versendet werden können. Es besteht aus dem Schlüsselwort *sendToSTM*, dem Zielautomaten, dem Signalnamen und optional einer Menge von Parametern. Als Ziel wird immer der Bezeichner einer Instanz angegeben, die mit der Produktionsregel aus Listing 5.2 umgesetzt ist. Zum Abschluss muss *STMSend* gemäß Abschnitt 4.8.1 in *DVFStatement* integriert werden:

```
1 DVFStatement:
2   (assign=DVFAssignmentOrCall) | (assert=DVFAssert) |
3   (send=DVFSendSignal) | (read=DVFReadSignal) |
4   (ret=DVFReturn) | (loop=DVFLoop) | (ifelse=DVFIfElse) |
5   (wait=DVFWait) | (scalaraccess=DVFScalarAccess) |
6   (sendstm=STMSend)
7 ;
```

Listing 5.8: Xtext-Grammatik für Transitionen

Das obige Beispiel enthält die Produktionsregel *DVFStatement* aus Listing 4.22. Damit das neue Statement *STMSend* in einer DSL genutzt werden kann, um ein Signal an einen endlichen Automaten zu übertragen, ist *DVFStatement* in Zeile 6 entsprechend erweitert worden.

5.1.3 Transformation in die CFIL

Im letzten Abschnitt ist eine domänenspezifische Sprache zur Beschreibung von endlichen Automaten vorgestellt worden. Sie besteht aus Produktionsregeln des DVF und Produktionsregeln, die der DSL-Entwickler manuell umgesetzt hat. Für eine erfolgreiche Verknüpfung von formaler Verifikation und modellgetriebener Entwicklung müssen neben den Produktionsregeln auch Transformatoren implementiert werden, die ein Modell automatisiert in eine Hoch- bzw. Model Checker-Eingabesprache überführen. Um den DSL-Entwickler bei der Implementierung entsprechender Transformatoren zu unterstützen, stellt das DVF die Control Flow Intermediate Language (CFIL) zur Verfügung. Ihre genaue Verwendung wird in Abbildung 5.1 im Rahmen des DVF-Workflows erläutert: Der DSL-Entwickler beschreibt einen Transformator, der alle Sprachkonstrukte seiner DSL, die nicht Teil des DVF sind, in die CFIL überführt. Dazu gehören im Fall der Statechart-DSL Automaten, Zustände und Transitionen. Das Ergebnis ist ein Modell, dessen AST nur noch aus vorgefertigten Sprachkonstrukten des DVF und der CFIL besteht. Ein derartiger Abstract Syntax Tree kann den Transformatoren des DVF übergeben werden, die ihn automatisiert in eine Hoch- bzw. Model Checker-Eingabesprache überführen. Deshalb beschreibt dieser Abschnitt einen Transformationsalgorithmus, der Modelle der Statechart-DSL in die CFIL übersetzt.

Die domänenspezifische Sprache aus Abschnitt 5.1.2 beinhaltet die folgenden Sprachkonstrukte, die nicht Teil des DVF sind:

- Deklarationen von Klassen, deren Verhalten mit einem endlichen Automaten beschrieben ist. Sie bestehen eine Liste von Signalen, Zuständen und ausgehenden Transitionen.
- Instanzen der Automaten-Klassen, die nebenläufig ausgeführt werden.
- Ein Statement, um ein Signal an die Instanz einer Automaten-Klasse zu senden.

Der DSL-Entwickler muss somit einen Transformator implementieren, der über den Abstract Syntax Tree iteriert und die drei genannten Elemente mit den Sprachkonstrukten der CFIL substituiert. Der entsprechende Übersetzungsalgorithmus wird im Verlauf dieses Abschnitts genauer vorgestellt.

Transformation von Automaten-Klassen

Dieses Unterkapitel beschreibt einen Transformationsalgorithmus für Automaten-Klassen. Sie setzen sich aus den folgenden Komponenten zusammen:

- Der eigentlichen Klasse bzw. ihre Deklaration.
- Den Signalen, auf die der Automat sensitiv ist.
- Den Zuständen und ihren ausgehenden Transitionen.

Die Transformation der drei genannten Komponenten ist Gegenstand dieses Unterkapitels.

Gemäß Listing 5.3 werden Automaten-Klassen in der Statechart-DSL mit der Produktionsregel *STMStmClass* beschrieben. Für eine Transformation in die CFIL muss der DSL-Entwickler die Eigenschaften von *STMStmClass* analysieren und evaluieren, mit welchem Element der CFIL es sich abbilden lässt: Automaten setzen das Konzept der nebenläufigen Ausführung um. Des Weiteren kapseln sie lokale Variablen und Zustände. In der CFIL hat das Thread-Sprachkonstrukt (vgl. Listing 4.44 und 4.45) eine ähnliche Semantik: Es wird nebenläufig ausgeführt und kann lokale Variablen bzw. Verhalten beinhalten. Aus diesem Grund bietet es sich an, dass der Transformator Automaten-Klassen in CFIL-Threads überführt. Der Übersetzungsalgorithmus ist in Abbildung 5.5 als Aktivitätsdiagramm visualisiert. Er zeigt, dass der Transformator über den AST iteriert und das nächste Objekt vom Typ *STMStmClass* sucht. Für jedes gefundene Objekt erzeugt er einen CFIL-Thread und fügt die lokalen Variablen der Automaten-Klasse hinzu. Im Anschluss wird die Automaten-Klasse im AST mit dem zuvor erzeugten Thread ersetzt. Das folgende Beispiel verdeutlicht, wie der DSL-Entwickler den Algorithmus mit Xpand umsetzen kann:

```
1 «DEFINE transformFSM FOR STMStmClass-»
2 thread «name» entry «getInitialState(this)»{
```

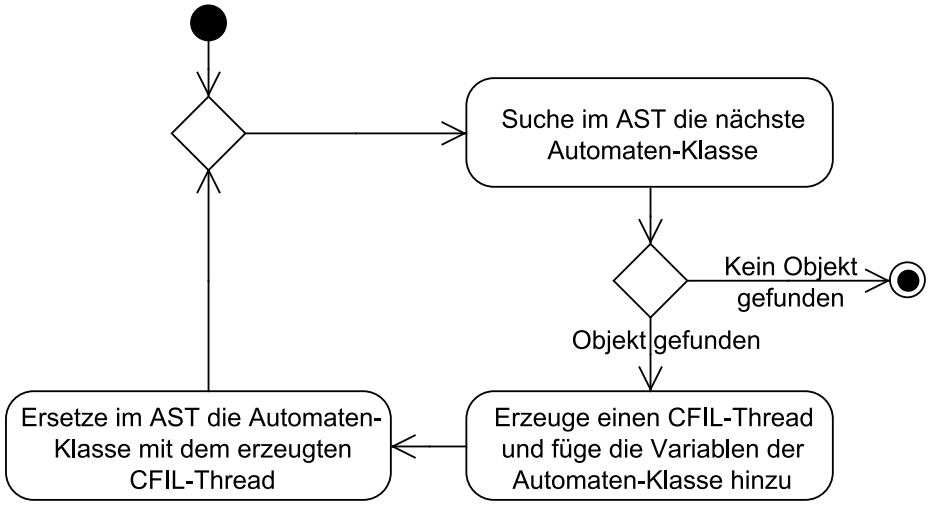

Abbildung 5.5: Transformation von Automaten-Klassen (Aktivitätsdiagramm)

```
3   «FOREACH variables AS variable-»
4     «addLocalVariable(variable)»
5   «ENDFOREACH-»
6 }
7 «ENDDEFINE»
```

Listing 5.9: Transformation von Automaten-Klassen

Der Xpand-Transformator im obigen Listing 5.9 arbeitet wie folgt: Jede Automaten-Klasse wird in einen Thread überführt. Der Bezeichner des Threads entspricht dem Namen der Automaten-Klasse. Jeder Thread hat einen Einstiegspunkt. Deshalb wird die Java-Methode *getInitialState()* aufgerufen. *GetInitialState()* evaluiert alle Zustände des endlichen Automaten und gibt den Namen des Startzustands zurück. Nach dem Ermitteln des Einstiegspunkts iteriert der Transformator in den Zeilen 3 bis 5 über alle lokalen Variablen der Automaten-Klasse und fügt sie in den Thread ein.

Neben der Automaten-Deklaration müssen auch die darin enthaltenen Signale bzw. Zustände in die CFIL überführt werden. Auch hier ist es erforderlich, dass der DSL-Entwickler zunächst die Semantik des entsprechenden Sprachkonstrukts analysiert und im Anschluss entscheidet, mit welchen Elementen der CFIL es sich abbilden lässt. Deshalb werden beide Sprachkonstrukte im weiteren Verlauf dieses Abschnitts untersucht und daraus ein Transformationsalgorithmus abgeleitet.

Gemäß Listing 5.4 kann eine Automaten-Klasse eine Liste von Signalen enthalten, auf die sie sensitiv ist. Ein Signal besteht aus einem Bezeichner und optional einer Liste von

Parametern. Damit Signale empfangen werden können, besitzt jeder Automat gemäß der UML-Beschreibung aus Abschnitt 2.3 eine eigene Message-Queue. Das DVF stellt die folgenden Sprachkonstrukte zur Verfügung, die eine ähnliche Semantik aufweisen:

- Das DSL Verification Framework ermöglicht in einer domänenspezifischen Sprache die Nutzung von Message-Queues (vgl. Listing 4.11). Daher bietet es sich an, die Queues der endlichen Automaten mit den Queues der CFIL umzusetzen.
- Jedes Signal der Statechart-DSL hat einen eindeutigen Bezeichner. Das DVF stellt Byte-Werte oder Aufzählungen zur Verfügung, um die Signalnamen zu abstrahieren. Der Vorteil von Aufzählungen ist eine bessere Lesbarkeit des generierten Modells. Deshalb wird vom Transformator für jedes übersetzte Modell eine Aufzählung generiert, deren Elemente die Signalnamen repräsentieren.

Der Transformationsalgorithmus für Signale wird anhand des folgenden Beispiels verdeutlicht:

```
1 stm_class Sender{
2    sensitive sig1{byte p1}
3    sensitive sig2{byte p2, byte p3}
4 }
```

Listing 5.10: Signale in einem endlichen Automaten

Das obige Listing zeigt ein Modell der Statechart-DSL. Es enthält einen endlichen Automaten, der auf die Signale sig_1 und sig_2 sensitiv ist. Eine Übersetzung in die CFIL gestaltet sich wie folgt:

```
1 enum Signal{sig1, sig2}
2
3 thread Sender{
4    signal queue[2]{Signal signal, byte p1, byte p2};
5 }
```

Listing 5.11: Transformation von Signalen

Das obige Beispiel 5.11 zeigt, wie der Transformator die Automaten-Klasse aus Listing 5.10 in die CFIL überführt. Das Statechart-Modell enthält die Signale sig_1 und sig_2. Aus diesem Grund wird in Listing 5.11 eine Aufzählung namens *Signal* generiert, die die Elemente *sig1* und *sig2* kapselt und die in Zeile 1 zu sehen ist.

Das Übertragen der Signale erfolgt mittels Message-Queues, die in der Statechart-DSL implizit Teil jeder Automaten-Klasse sind. Daher wird in Listing 5.11 in Zeile 4 eine entsprechende Queue vom Transformator angelegt. Die auf ihr zu speichernden Nachrichten bestehen aus drei Elementen: *Signal*, *p1* und *p2*. Die entsprechenden Sendeoperationen können wie folgt umgesetzt werden:

- Das Signal *sig1* wird zusammen mit einem Byte als Parameter übertragen. Zum Senden finden daher die Felder *signal* und *p1* Verwendung.

- Das Signal *sig2* wird zusammen mit zwei Bytes als Parameter übertragen. Zum Senden finden daher die Felder *signal*, *p1* und *p2* Verwendung.

Die Automaten-Klassen beinhalten neben der Sensitivitätsliste auch eine Menge von Zuständen, die in die CFIL zu überführen sind. Deshalb muss der DSL-Entwickler die Semantik der Zustände genauer untersuchen und im Anschluss entscheiden, wie sie in die CFIL transformiert werden können. Jeder Zustand der Statechart-DSL hat die folgenden Eigenschaften:

- Entry-Actions kapseln eine Menge von Statements, die beim Betreten des Zustands ausgeführt werden.

- Exit-Actions kapseln eine Menge von Statements, die beim Verlassen des Zustands ausgeführt werden.

- Zustände können durch Transitionen erreicht bzw. verlassen werden.

In der CFIL gibt es ein ähnliches Element, das Verhalten kapselt und durch Sprunganweisungen betreten bzw. verlassen werden kann. Dabei handelt es sich um den Statement-Block (vgl. Abschnitt 4.4.2). Daher wird jeder Zustand in einen Statement-Block überführt und die Statements der Exit- bzw. Entry-Actions entsprechend integriert. Das Verlassen eines Zustands kann mit einer Goto-Anweisung erfolgen (vgl. Listing 4.48). Da es in der Statechart-DSL verschiedene Zustandstypen gibt, ist eine Fallunterscheidung notwendig:

- Initialzustände bestehen gemäß der UML-Spezifikation [111] lediglich aus einer ausgehende Transition, die keinen Guard oder Trigger aufweist. Deshalb wird der Initialzuständ lediglich in einen Statement-Block mit einer Goto-Anweisung überführt, die zum Folgezustand führt.

- Endzustände beinhalten keine Entry-Action, Exit-Action oder ausgehende Transitionen. Beim Erreichen eines Endzustands terminiert der endliche Automat. Deshalb wird ein Endzustand in einen Statement-Block überführt, der eine Exit-Anweisung (vgl. Listing 4.49) enthält.

- Einfache Zustände sind komplexer als Start- bzw. Endzustände, da sie ausgehende Transitionen enthalten, deren Ausführung von Signalen oder Guard-Expressions abhängig ist. Deshalb wird ihre Transformation im weiteren Verlauf dieses Abschnitts genauer untersucht.

Einfache Zustände mit ausgehenden Transitionen setzen den Run-To-Completion-Step (siehe Abschnitt 2.3) um. Das bedeuten, sie müssen zunächst eine potentielle Entry-Action ausführen und im Anschluss ein Signal der Message-Queue abrufen. Wenn keine der ausgehenden Transitionen ausführbar ist, wird ein weiteres Signal der Queue verarbeitet. Dieser Vorgang wiederholt sich sich, bis der Zustand durch eine der ausgehenden Transitionen verlassen werden kann. Es bietet sich somit an, den Run-To-Completion Step in eine Schleife und die Transitionen in If-Blöcke zu transformieren. Die Übersetzung gestaltet sich wie folgt:

1. Erzeuge für den einfachen Zustand einen Statement-Block, der die Statements der Entry-Action enthält.
2. Auf die Entry-Action folgt eine Endlosschleife, die mit jeder Iteration ein Signal der Message-Queue abruft.
3. In der Schleife wird für jede ausgehende Transition ein If-Block generiert, der in Abhängigkeit des Signal-Triggers und der Guard-Expression ausführbar ist.
4. Jeder dieser If-Blöcke enthält die Effect-Statements der ausgehenden Transition, die Exit-Action des Zustands und abschließend eine Goto-Anweisung zum Zielzustand.

Das folgende Beispiel verdeutlicht den Transformationsalgorithmus für Zustände und ihre ausgehenden Transitionen:

```
1  stm_class SimpleFSM{
2    sensitive sig1{}
3    byte a=0;
4
5    initial start{ -> A{} }
6
7    state A{
8      -> B{
9        guard{a>=5}
10       trigger{sig1}
11     }
12     -> C{
13       guard{a>=10}
14       trigger{sig1}
15     }
16   }
17
18   final B{}
```

```
19   final C{}
20 }
```

Listing 5.12: Beispiel für Zustände und Transitionen

Das obige Modell zeigt einen endlichen Automaten, der mit der Statechart-DSL umgesetzt ist. Der Automat enthält die lokale Variable a und ist sensitiv für das Signal sig_1. *SimpleFSM* beginnt mit der Ausführung im Initialzustand *start*. Danach wird in den Zustand A gewechselt und auf den Erhalt von sig_1 gewartet. Nach dem Empfang des Signals wechselt *SimpleFSM* in die Zustände B bzw. C und terminiert. Das nächste Listing zeigt, wie *SimpleFSM* von dem in diesem Abschnitt beschriebenen Algorithmus in die CFIL überführt werden kann:

```
1 enum Signal{sig1}
2 signal queue{Signal signal};
3
4 thread SimpleFSM entry start (byte id){
5   byte a = 0;
6
7   statementblock start{goto A;}
8
9   statementblock A{
10    while(true){
11      receive(queue);
12      if(queue.signal==sig1 && a>=5){
13        goto B;
14      }
15      elseif(queue.signal==sig1 && a>=10){
16        goto C;
17      }
18    }
19  }
20
21  statementblock B{exit;}
22  statementblock C{exit;}
23 }
```

Listing 5.13: Transformation von Zuständen und Transitionen

Das obige Modell zeigt, wie der Transformator des DSL-Entwicklers das Beispiel aus Listing 5.12 in die CFIL überführt. In den Zeilen 1 und 2 sind eine Aufzählung und eine Queue zu sehen, um die Sensitivitätsliste der Automaten-Klasse abzubilden. Die Statement-Blöcke, die die Zustände des Automaten repräsentieren, werden in den Zeilen

7, 9, 21 und 22 deklariert. Dabei fällt auf, dass die Initial- bzw. Endzustände lediglich aus einem Goto- bzw. Exit-Statement bestehen. Der Run-To-Completion-Step ist in den Zeilen 10 bis 18 in Form einer While-Schleife umgesetzt. Mit jeder Iteration wird ein Signal abgerufen und überprüft, ob die ausgehenden Transitionen ausführbar sind. Bei Erreichen von B oder C terminiert der Thread und somit der endliche Automat.

In den bisherigen Beschreibungen und Beispielen sind Composite-States nicht betrachtet worden. Sie stellen eine Besonderheit dar und werden vom Transformator durch einen Flatten-Algorithmus, wie beispielsweise [150] oder [152] entfernt. Die Entfernung von Composite-States hat den Vorteil, dass die dafür notwendigen Verfahren bereits gut erforscht sind und die Transformation vereinfacht wird. Ein Nachteil ist, dass durch das Entfernen von Composite-States der Zustandsraum des Modells vergrößert wird und so das Risiko der State Space Explosion steigt. Für zukünftige Arbeiten ist es daher wünschenswert diese Fallstudie so zu erweitern, dass Composite-States direkt unterstützt werden und das Anwenden eines Flatten-Algorithmus entfällt.

Transformation von Automaten-Instanzen

Neben der Deklaration einer Automaten-Klasse ermöglicht die Statechart-DSL auch das Starten entsprechender Instanzen. Das dafür notwendige Sprachkonstrukt ist in Listing 5.2 enthalten und nicht Teil des DVF. Daher zeigt dieses Unterkapitel, wie es in die CFIL übersetzt werden kann.

Jede Instanziierungsanweisung der Statechart-DSL besteht gemäß Listing 5.2 aus dem Schlüsselwort *fsm*, einem eindeutigen Bezeichner und optional eckigen Klammern. Die Klammern ähneln einem Array und bewirken, dass entsprechend des darin enthaltenen ganzzahligen Werts mehrere Instanzen einer Klasse gebildet werden. In der CFIL gibt es ein ähnliches Statement, das zum Starten nebenläufiger Prozesse Verwendung findet: Die Run-Anweisung (vgl. Listing 4.46). Aus diesem Grund bietet es sich für den DSL-Entwickler an, jede Instanziierungsanweisung in ein Run-Statement zu überführen. Das folgende Beispiel verdeutlicht den Übersetzungsalgorithmus:

```
1 fsm RunMe single;
2 fsm RunMe more[2];
3
4 stm_class RunMe{
5   //...
6 }
```

Listing 5.14: Beispiel für das Instanziieren von Automaten

Im obigen Listing 5.14 erfolgt in Zeile 4 das Deklarieren der Automaten-Klasse *RunMe*. Des Weiteren werden von *RunMe* drei Instanzen gebildet. Das nächste Beispiel zeigt, wie Listing 5.14 vom Transformator des DSL-Entwicklers in die CFIL überführt werden kann:

```
1 run runMe();
2 run runMe();
3 run runMe();
4
5 thread RunMe{
6   //...
7 }
```

Listing 5.15: Transformation von Automaten-Instanzen in die CFIL

Im obigen Listing 5.15 ist zu sehen, dass der Automat *RunMe* vom Transformator in einen Thread überführt wird. Des Weiteren sind drei Run-Statements in dem Modell enthalten, die drei Instanzen von *RunMe* erzeugen. Das obige CFIL-Modell hat somit anschaulich die selbe Semantik wie Listing 5.14.

Transformation des Send-Statements

Die Statechart-DSL ermöglicht das Deklarieren und Instanziieren endlicher Automaten. Ein wesentlicher Bestandteil endlicher Automaten ist ihre Kommunikation untereinander. Die Statechart-DSL ortientiert sich an der *Unified Modeling Language* [111] und stellt dem DSL-Anwender zu diesem Zweck asynchrone Nachrichten zur Verfügung.

Die vorangegangenen Unterkapiteln machen deutlich, dass Automaten-Klassen eine Queue besitzen, über die sie Signale empfangen können. Der Transformator überführt daher jede Automaten-Klasse in einen Thread, der eine Message-Queue aus der CFIL enthält. Nachrichten sollen jedoch nicht nur empfangen, sondern auch abgerufen werden können. Aus diesem Grund wird das DSL Verification Framework im Rahmen der ersten Fallstudie erweitert und ein zusätzliches Statement namens *sendToSTM()* (vgl. Listing 5.7) eingeführt. In einem Modell kann *sendToSTM()* Verwendung finden, um ein Signal mitsamt seinen Parametern an die Instanz einer Automaten-Klasse zu übertragen. Da *sendToSTM()* nicht Teil des DSL Verification Frameworks ist, muss es vom Transformator des DSL-Entwicklers in die CFIL überführt werden. Ein entsprechender Algorithmus wird in diesem Abschnitt vorgestellt.

Für eine Transformation in die CFIL muss der DSL-Entwickler evaluieren, mit welchen Sprachkonstrukten das Statement *sendToSTM()* abgebildet werden kann. Bei Betrachtung des DSL Verification Frameworks fällt auf, dass das Statement *send()* zur Verfügung gestellt wird (vgl. Listing 4.26). Mit ihm kann ein asynchrones Signal an eine Message-Queue übertragen werden. *sendToSTM()* und *send()* unterscheiden sich lediglich in einem Punkt: Das Ziel von *sendToSTM()* ist die Instanz einer Klasse, während *send()* das Signal einer Message-Queue zuweist. Aus diesem Grund bietet es sich an, dass der Transformator *sendToSTM()* mit *send()* substitutiert. Dafür kann der folgende Algorithmus Verwendung finden:

1. Iteriere über den AST und suche das nächste Objekt vom Typ *sendToSTM*. Terminiere, wenn kein weiteres Objekt vorhanden ist.
2. Erzeuge ein Send-Statement. Der erste Parameter entspricht dem Signalnamen. Darauf folgenen die Parameter von *sendToSTM()*. Das Ziel von *send()* ist die Queue, die zu der Instanz der Automaten-Klasse gehört.
3. Ersetze *sendToSTM()* mit dem generierten Send-Statement.
4. Springe zu 1.

Der Algorithmus wird mit dem folgenden Beispiel verdeutlicht:

```
1 fsm Sender sender;
2 fsm Receiver receiver;
3
4 stm_class Sender{
5   state A{
6     entry{
7       sendToSTM(receiver, 1);
8     }
9   }
10 }
11
12 stm_class Receiver{
13   sensitive sig1{ byte p1 }
14 }
```

Listing 5.16: Versand eines Signals

Im obigen Listing 5.16 ist ein Modell der Statechart-DSL zu sehen. Es enthält die Automaten *Sender* und *Receiver*. Von beiden wird jeweils eine Instanz gestartet. *Receiver* ist sensitiv auf das Signal sig_1. *Sender* überträgt im Zustand A das Signal sig_1 an die Receiver-Instanz. Das nächste Beispiel zeigt, wie Listing 5.16 vom Transformator in die CFIL überführt wird:

```
1 run Sender();
2 run Receiver();
3
4 enum Signal{ sig1 }
5 signal receiver_queue{Signal sig, byte p1};
6
7 thread Sender{
8   statementblock A{
```

```
9       send(receiver_queue, Sender.sig1, 1);
10    }
11  }
12
13  thread Receiver{ }
```

Listing 5.17: Transformation von sendToSTM()

Das obige Listing 5.17 startet in den Zeilen 1 und 2 die Instanzen der beiden Automaten-Klassen. Die Zeilen 4 bis 5 zeigen, dass die Sensitivitätsliste von *Receiver* in eine Aufzählung und eine Message-Queue übersetzt wird. Die Transformation von *sendToSTM()* ist Zeile 9 zu sehen: *sendToSTM()* wird durch ein Send-Statement ersetzt. Das Ziel ist nicht mehr die Instanz der Automaten-Klasse, sondern die zuvor generierte Message-Queue. Das übertragene Signal setzt sich aus einem Byte und dem Element der Aufzählung zusammen. Das generierte CFIL-Modell hat somit die selbe Semantik wie Listing 5.16.

5.1.4 Umsetzung des Modells und Verifikation

Das vorangegangene Unterkapitel stellt die erste Fallstudie vor, bei der es sich um eine Erweiterung für AssyControl handelt. AssyControl ist ein System zur Überwachung von Handmontage an einer Werkbank. Abschnitt 5.1.1 beschreibt die Erweiterung im Detail. Daraus ergeben sich Anforderungen an eine domänenspezifische Sprache namens Statechart-DSL. Die Grammatik der Statechart-DSL ist Teil von Unterkapitel 5.1.2 und nutzt sowohl spezielle Sprachkonstrukte des DSL-Entwicklers, als auch Elemente des DVF. Der aus der Grammatik generierte Parser kann Modelle der Statechart-DSL einlesen und daraus einen Abstract Syntax Tree erzeugen. Dieser AST dient einem Transformator als Eingabe, der ihn in die Control Flow Intermediate Language übersetzt. Abschnitt 5.1.3 beschreibt die Umsetzung des Transformators bzw. den von ihm verwendeten Algorithmus. Das vom Transformator generierte CFIL-Modell kann in einem letzten Schritt vom DSL Verification Framework nach Promela bzw. Java überführt werden (vgl. Abbildung 5.1). Genau dieser letzte Schritt, also der Entwurf eines Modells und seine Transformation in die Zielsprachen, ist Gegenstand des vorliegenden Unterkapitels. Dabei ist auch von Interesse, ob das gesamte Modell verifizierbar ist, oder es einen zu großen Zustandsraum aufweist.

Abschnitt 5.1.1 beschreibt eine AssyControl-Erweiterung, die eine einfache Rekonfiguration des Systems ermöglicht, wenn ein Arbeiter den Montageplatz verändert. Vor der konkreten Implementierung empfiehlt es sich, die Softwarearchitektur graphisch zu planen. Daraus ergibt sich für diesen Abschnitt die folgende Gliederung: Zunächst visualisieren zwei Diagramme die Architektur der ersten Fallstudie. Danach wird die Statechart-DSL vom DSL-Anwender genutzt, um die die AssyControl-Erweiterung zu modellieren. Dabei ist darauf zu achten, dass das entsprechende Modell alle Anforderungen der AssyControl-Erweiterung erfüllt. Es muss jedoch kein bestimmter Modellierungsstil Ver-

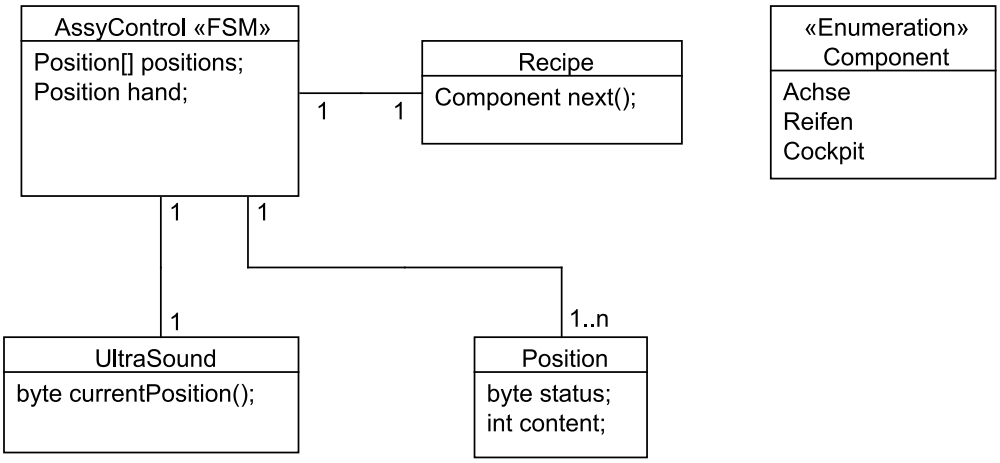

Abbildung 5.6: Erste AssyControl-Erweiterung (Klassendiagramm)

wendung finden, um beispielsweise Optimierungen auf Modellebene zu erreichen: Sie werden automatisiert von den Transformatoren des DVF berücksichtigt. Modelle mit der selben Semantik, aber unterschiedlicher Syntax, sind somit zulässig. Zum Abschluss erfolgen zwei Transformationen in eine Hoch- bzw. Model Checker-Eingabesprache.

Listing 5.6 zeigt die erste Fallstudie als Klassendiagramm. Die Klasse *AssyControl* ist der Kern des Modells. Sie reagiert auf die Bewegungen des Arbeiters. Da es sich um ein reaktives System handelt, soll ihr Verhalten mit einem endlichen Automaten beschrieben werden. Das Ziel von *AssyControl* ist die automatische Zuordnung von Komponenten zu Positions. Deshalb enthält sie zwei Variablen:

- Das Array *positions* gibt darüber Aufschluss, welche Komponenten sich bei welcher Position befinden. Es enthält sechs Elemente, da der Arbeitsplatz aus sechs Positions besteht.
- Ein Arbeiter kann Komponenten mit seiner Hand aufnehmen. Die aktuell gegriffene Komponente wird von *AssyControl* in der Variable *hand* gespeichert.

Für eine automatische Zuordnung von Komponenten zu Positions, muss die Zusammenbaureihenfolge bekannt sein. Sie wird von der Klasse *Recipe* bzw. der Methode *next()* abstrahiert. *Next()* liefert die als nächstes zu greifende bzw. montierende Komponente zurück. Das bedeutet, wenn *next()* das erste mal aufgerufen wird, ist der Rückgabewert *Component.Achse* (vgl. Abschnitt 5.1.1). Darauf folgen *Component.Reifen*, *Component.Reifen*, *Component.Achse*, *Component.Reifen*, *Component.Reifen* und zum Abschluss *Component.Cockpit*.

Neben der Zusammenbaureihenfolge müssen auch die Positions abstrahiert werden.

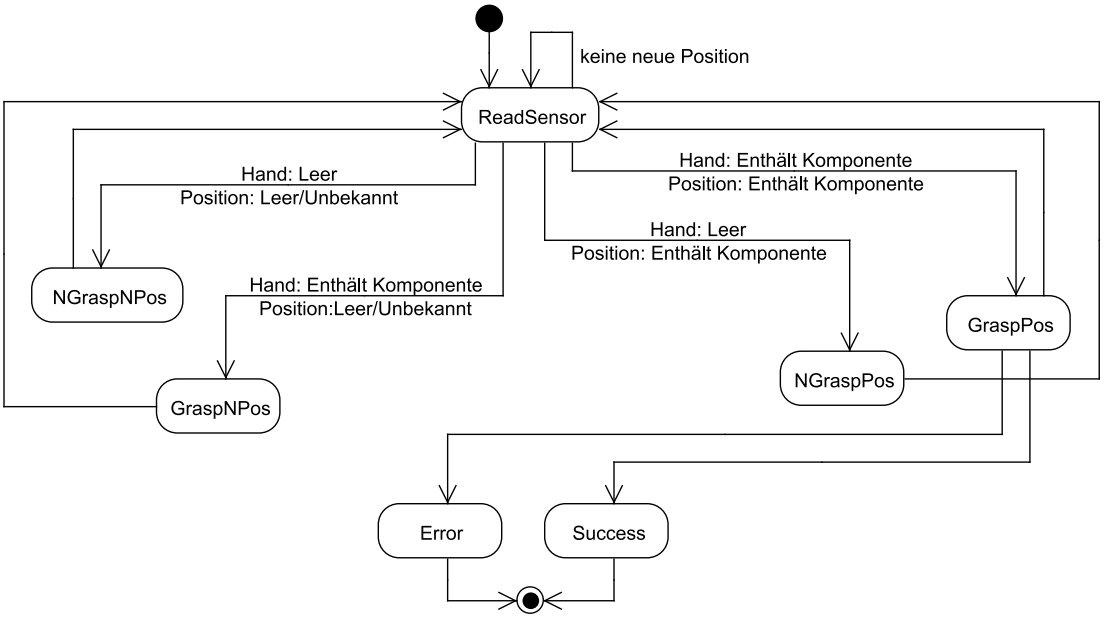

Abbildung 5.7: Erste AssyControl-Erweiterung (Zustandsdiagramm)

Zu diesem Zweck enthält das Modell die Klasse *Position*. Sie enthält zwei Variablen. *Content* gibt darüber Aufschluss, welche Komponente bzw. Komponenten sich bei der der entsprechenden Position befinden. Die zweite Variable hat den Bezeichner *status* und nimmt einen der drei folgenden Werte an:

- 0: Die Position ist leer.
- 1: Auf der Position befindet sich entweder ein einzelnes Element, oder mehrere, die bereits zusammengebaut wurden.
- 2: Die Position enthält eine Box mit Komponenten.

Die AssyControl-Erweiterung reagiert auf die Signale des Ultraschallsystems. Deshalb muss das Modell eine entsprechende Schnittstelle enthalten. Sie wird mit der Klasse *UltraSound* bzw. der Methode *currentPosition()* abgebildet. *CurrentPosition()* liefert die Position zurück, bei der sich der Marker gerade befindet.

Mit den vorgestellten Klassen können Positions, Komponenten und der Marker in einem Modell abgebildet werden. Es fehlt jedoch die konkrete Umsetzung der Klasse *AssyControl*. Ihr Verhalten soll als reaktives System mit einem endlichen Automaten beschrieben werden. Abbildung 5.7 zeigt eine mögliche Implementierung. Der Automat

ruft im Zustand *ReadSensor* die aktuelle Position des Markers ab. Danach wird zwischen vier verschiedenen Fällen unterschieden:

- *NGraspNPos*: Die Hand des Arbeiters hat kein Objekt gegriffen und die aktuelle Position ist leer bzw. ihr Inhalt unbekannt. In diesem Fall wird davon ausgegangen, dass sich bei der Position eine Box befindet. Die Box enthält, in Abhängigkeit des Rückgabewerts von *receipe.next()*, entweder Achsen, Reifen oder Cockpits. Der Arbeiter greift ein Element aus der entsprechenden Box.
- *GraspNpos*: Die Hand des Arbeiters hat ein Objekt gegriffen und die aktuelle Position ist leer bzw. ihr Inhalt unbekannt. Es wird davon ausgegangen, dass der Arbeiter die gegriffene Komponente bei der leeren Position ablegt.
- *NGraspPos*: Die Hand des Arbeiters ist leer und die aktuelle Position enthält ein oder mehrere Komponenten. Es wird davon ausgegangen, dass der Arbeiter die entsprechende Komponente greift.
- *GraspPos*: Die Hand des Arbeiters hat ein Objekt gegriffen und die aktuelle Position enthält ein oder mehrere Komponenten. Es wird davon ausgegangen, dass der Arbeiter die Komponente an der Position ablegt und mit den bereits vorhandenen kombiniert. Wenn das Ergebnis nicht der Zusammenbaureihenfolge entspricht, wird ein Fehler gemeldet. Falls hingegen alle Komponenten erfolgreich zusammengebaut werden konnten, meldet der Konfigurationsmodus das Ende des Montageprozesses und terminiert.

Nach der Fallunterscheidung wird erneut in den Zustand *ReadSensor* gesprungen und die Position des Markers abgefragt. Das System läuft in einer Endlosschleife, bis der Inhalt aller Positions erkannt wurde oder ein Fehler auftritt.

Nach der Planung des Gesamtsystems durch die Abbildungen 5.6 und 5.7 kann mit der Implementierung durch die DSL begonnen werden. Das folgende Listing zeigt die Umsetzung der Klasse *Recipe*, die die Zusammenbaureihenfolge darstellt:

```
1  class Recipe{
2    byte index;
3    Component recipe[7];
4    Component ret;
5
6    void init(){
7      index=0;
8      recipe[0] = Component.Achse;
9      recipe[1] = Component.Reifen;
10     recipe[2] = Component.Reifen;
11     recipe[3] = Component.Achse;
```

```
12      recipe[4] = Component.Reifen;
13      recipe[5] = Component.Reifen;
14      recipe[6] = Component.Cockpit;
15    }
16
17    Component next(){
18      ret = recipe[index];
19      index = index+1;
20      return ret;
21    }
22  }
```

Listing 5.18: Aufbau von Recipe

Das obige Beispiel zeigt, wie die Klasse *Recipe* mit der Statechart-DSL umgesetzt ist. Sie enthält ein Array mit Elementen, die der Zusammenbaureihenfolge entsprechen. Bei jedem Aufruf von *next()* wird das Element zurückgegeben, dass als nächstes gegriffen bzw. zusammengebaut werden muss.

Neben der Zusammenbaureihenfolge müssen auch die Schnittstelle zum Ultraschallsystem und der endliche Automat modelliert werden. Das folgende Beispiel zeigt ihre Umsetzung:

```
1  class UltraSound{
2    abstraction range(0,5) nextPosition(){}
3  }
4
5  fsm AssyControl assycontrol;
6
7  stm_class AssyControl{
8    Position positions[6];
9    Position hand;
10   byte cp;
11   byte cp_next;
12   bool newpos;
13   UltraSound us;
14   Receipe receipe;
15
16   initial start{
17     -> ReadSensor{
18       effect{ receipe.init(); }
19     }
20   }
```

```
21
22    //...
23 }
```

Listing 5.19: Struktur der AssyControl-Erweiterung

Das obige Listing 5.19 zeigt die Klassen *UltraSound* und *AssyControl*. *UltraSound* ist in Zeile 1 zu sehen, enthält die Methode *nextPosition()* und ist die Schnittstelle zum Ultraschallsystem. Sie wird mit eigenschaftserhaltender Abstraktion implementiert. *NextPosition()* gibt einen Wert zwischen 0 und 5 zurück, was der Position entspricht, bei der sich der Marker gerade befindet. Die Verwendung von eigenschaftserhaltender Abstraktion ist von Vorteil, da die konkrete Umsetzung des Ultraschallsystems nicht Teil des Modells bzw. der AssyControl-Erweiterung ist.

In Zeile 7 ist die Klasse *AssyControl* zu sehen, deren Verhalten mit einem endlichen Automaten beschrieben werden soll. Sie enthält ein Array namens *positions*, das aus 6 Elementen besteht. Es repräsentiert die Positions P_1 bis P_6. Die Variable *cp* entspricht der Position, bei der sich der Marker aktuell befindet und wird zum Indizieren von *positions* genutzt. Diese Art der Implementierung hat den Vorteil, dass der aktuellen Position oder der Hand des Arbeiters durch einfache Zuweisungen Komponenten entnommen bzw. hinzugefügt werden können.

Gemäß Zeile 5 gibt es von *AssyControl* genau eine Instanz. Nach dem Starten wechselt der Automat in den Zustand *ReadSensor*, den das nächste Listing genauer vorstellt:

```
1  state ReadSensor{
2    entry{
3      cp_next = us.readPosition();
4      if(cp==cp_next){
5        newpos=false;
6      }
7      else{
8        newpos=true;
9        cp=cp_next;
10     }
11   }
12
13   -> ReadSensor{
14     guard { !newpos }
15   }
16   -> NGraspNPos{
17     guard{
18       newpos && positions[cp].content==0 && hand.content==0
19     }
```

```
20    }
21    -> GraspNPos{
22      guard{
23        newpos && positions[cp].content==0 && hand.content!=0
24      }
25    }
26    -> NGraspPos{
27      guard{
28        newpos && positions[cp].content!=0 && hand.content==0
29      }
30    }
31    -> GraspPos{
32      guard{
33        newpos && positions[cp].content!=0 && hand.content!=0
34      }
35    }
36  }
```

Listing 5.20: Aufbau von ReadSensor

ReadSensor beinhaltet eine Entry-Action, die von dem Ultraschallsystem die aktuelle Position des Markers abfragt. Das Ergebnis wird in *cp_next* zwischengespeichert. Im Anschluss wird ermittelt, ob sich die Position des Markers verändert hat und in Abhängigkeit des Ergebnisses die Variable *newpos* gesetzt. Danach erfolgt das Ausführen einer der fünf Transitionen: Wenn der Marker keine neue Position erreicht hat, wird wieder der Zustand *ReadSensor* betreten, andernfalls erfolgt eine Transition nach *NGraspNPos*, *GraspNPos*, *NGraspPos* oder *GraspPos*. Das nächste Listing zeigt die Umsetzung des Zustands *NGraspNPos*:

```
1  state NGraspNPos{
2    entry{
3      hand.content = receipe.next();
4      hand.status = 1;
5      position[cp].content = hand.content;
6      position[cp].status = 2;
7    }
8    -> ReadSensor{}
9  }
```

Listing 5.21: Aufbau von NGraspNPos

NGraspNPos führt eine Entry-Action aus, die der aktuellen Position das nächste Element der Zusammenbaureihenfolge zuweist. Die Variable *status* wird auf zwei gesetzt, was einer

Box mit Komponenten entspricht. Da davon ausgegangen wird, dass der Arbeiter eine Komponente aus der Box greift, werden auch *hand.status* und *hand.content* entsprechend initialisiert. Die Implementierung der verbleibenden Zustände *NGraspPos*, *GraspNPos* und *GraspPos* ist vergleichbar mit dem Modell aus Listing 5.21 und wird daher im Rahmen dieser Arbeit nicht im Detail beschrieben.

Nach der Beschreibung des Modells muss es in die beiden Zielsprachen übersetzt werden. Zu diesem Zweck beschreibt Unterkapitel 5.1.3 einen Transformator, der das Statechart-Modell in die CFIL überführt. CFIL-Modelle dienen einem Transformator als Eingabe, der Teil des DSL Verification Frameworks ist. Durch ihn kann eine automatisierte Übersetzung nach Promela und Java erfolgen. Daher sind zwei Schritte notwendig, um die erste Fallstudie abzuschließen:

- Der DSL-Anwender nutzt das DVF, um sein Statechart-Modell nach Promela zu überführen. Der Model Checker Spin verifiziert das Ergebnis. Sofern Fehler vorhanden sind, müssen sie im Statechart-Modell behoben werden. Darauf folgt eine erneute Übersetzung nach Promela bzw. Verifikation mit Spin. Dieser Schritt wird wiederholt, bis Spin keine weiteren Fehler findet.
- Der DSL-Anwender nutzt das DVF, um sein Statechart-Modell nach Java zu überführen.

Die entsprechenden Tranformationen werden in den beiden folgenden Abschnitten genauer vorgestellt.

Transformation nach Promela und Verifikation mit Spin

Abschnitt 5.1.4 hat ein Statechart-Modell vorgestellt, um AssyControl mit einem Konfigurationsmodus zu erweitern. Dieses Unterkapitel zeigt, wie es nach Promela überführt und von Spin verifiziert werden kann. Das Ziel ist es, nicht erfüllte Anforderungen zu finden. Des Weiteren soll auch untersucht werden, ob der gesamte Zustandsraum des Modells verifizierbar ist, oder die State Space Explosion [29] dies verhindert.

Das DSL Verification Framework stellt eine Assert-Anweisung zur Verfügung, mit der ein DSL-Anwender Anforderungen in seinem Modell spezifizieren kann. In dem Modell aus Abschnitt 5.1.4 sind jedoch keine expliziten Assert-Anweisungen enthalten. Deshalb werden von Spin implizit die folgenden Aspekte untersucht:

- Es darf keine Deadlocks geben.
- Es darf keine Zugriffe über Array-Grenzen hinaus geben.

Sobald der DSL-Anwender das Statechart-Modell in die Model Checker-Eingabesprache Promela transformiert hat, kann die formale Verifikation erfolgen. Das nächste Listing zeigt die Ausgabe des Model Checkers:

```
1 pan:1: assertion violated - invalid array index (at depth 45)
2 pan: wrote assy.promela.trail
3
4 (Spin Version 6.2.3 -- 24 October 2012)
5 Warning: Search not completed
6         + Partial Order Reduction
7
8 Full statespace search for:
9         never claim             - (none specified)
10        assertion violations    +
11        cycle checks            - (disabled by -DSAFETY)
12        invalid end states      +
13
14 State-vector 40 byte, depth reached 45, errors: 1
15       86 states, stored
16       12 states, matched
17       98 transitions (= stored+matched)
18        0 atomic steps
19 hash conflicts:         0 (resolved)
20
21 Stats on memory usage (in Megabytes):
22    0.004        equivalent memory usage for states
23                 (stored*(State-vector + overhead))
24    0.287        actual memory usage for states
25   64.000        memory used for hash table (-w24)
26    0.343        memory used for DFS stack (-m10000)
27   64.539        total actual memory usage
```

Listing 5.22: Verifikation der AssyControl-Erweiterung

Gemäß Zeile 1 erfolgte ein Zugriff über eine Array-Grenze hinweg. Das Modell ist somit fehlerhaft. Zeile 2 weist auf einen automatisch generierten Fehlerpfad hin. Dieser gestaltet sich wie folgt:

1. Das System befindet sich in Ausgangslage. Der Inhalt der sechs Positions ist unbekannt.

2. Der Arbeiter greift bei P_1 eine Achse, bewegt seine Hand zu P_4 und legt sie dort ab.

3. Der Arbeiter greift bei P_2 einen Reifen, bewegt seine Hand zu P_4 und befestigt den Reifen an der Achse.

4. Der Arbeiter greift bei P_2 einen Reifen, bewegt seine Hand zu P_4 und befestigt den Reifen an der Achse.

5. Der Arbeiter wiederholt die Schritte eins bis drei. Dadurch liegen zwei Achsen, an denen die entsprechenden Reifen angebracht sind, bei der Position P_4. Des Weiteren hat die AssyControl-Erweiterung erfolgreich detektiert, dass sich bei P_1 und P_2 die Schachteln mit den Achsen bzw. Reifen befinden.

6. Der Arbeiter bewegt seine Hand zur Position P_3 und greift ein Cockpit. Im Anschluss führt er seine Hand jedoch nicht nach P_4, um dort alle Komponenten zusammenzubauen. Stattdessen bewegt er die Hand nach P_5 und legt dort das Cockpit ab.

7. Der Arbeiter bewegt seine Hand zur Position P_6 und der ungültige Array-Zugriff wird ausgelöst.

Der Fehler hat den folgenden Hintergrund: Nach Ausführung der ersten sechs Schritte hat die Erweiterung detektiert, dass sich die Behälter mit Achsen, Reifen und Cockpits bei den Positions P_1, P_2 und P_3 befinden. Die zusammengebauten Achsen liegen bei P_4 und ein einzelnes Cockpit bei P_5. Der Inhalt von P_6 ist unbekannt.

Wenn der Arbeiter abschließend seine Hand nach P_6 bewegt, betritt die AssyControl-Erweiterung den Zustand *NGraspNPos* aus Listing 5.21. Im Rahmen der Entry-Action wird die Methode *recipe.next()* aufgerufen. Ihre Implementierung ist in Listing 5.18 zu sehen. Da alle Komponenten, die Teil der Zusammenbaureihenfolge sind, bereits vom Arbeiter gegriffen wurden, beinhaltet die Variable *index* in der Klasse *Recipe* den Wert 7. Dadurch wird ein Zugriff auf *recipe[7]* durchgeführt, die Array-Grenze überschritten und der Model Checker meldet einen entsprechenden Fehler. Der Fehler kann durch eine If-Abfrage behoben werden, die sicherstellt, dass *index* vor einem Array-Zugriff nicht größer als 6 ist. Nach dem Beheben des Fehlers wird das Modell erneut in die Model Checker-Eingabesprache Promela überführt. Die Verifikation führt zu dem folgenden Ergebnis:

```
1 pan: out of memory
2
3 (Spin Version 6.2.3 -- 24 October 2012)
4 Warning: Search not completed
5         + Partial Order Reduction
6
7 State-vector 40 byte, depth reached 635, errors: 0
8  34000000 states, stored
9  14390167 states, matched
10 48390167 transitions (= stored+matched)
```

```
11         0 atomic steps
12 hash conflicts:  20732988 (resolved)
13
14 Stats on memory usage (in Megabytes):
15  1686.096       equivalent memory usage for states
16                 (stored*(State-vector + overhead))
17  1365.160       actual memory usage for states
18                 (compression: 80.97%)
19                 state-vector as stored =
20                 30 byte + 12 byte overhead
21   256.000       memory used for hash table (-w26)
22     0.343       memory used for DFS stack (-m10000)
23  1621.179       total actual memory usage
```

Listing 5.23: Erneute Verifikation der AssyControl-Erweiterung

Das obige Listing zeigt, dass keine weiteren Fehler im Modell enthalten sind. Ein Problem ist in Zeile 1 zu erkennen: Der Verifier konnte nicht den gesamten Zustandsraum des Modells untersuchen, da nicht genügend Hauptspeicher zur Verfügung stand. Deshalb kann nicht ausgeschlossen werden, dass sich weitere Fehler im Modell befinden, die der Model Checker lediglich nicht detektiert hat. Der DSL-Anwender hat an dieser Stelle zwei Möglichkeiten:

- Er übergibt Spin spezielle Parameter, um andere Suchalgorithmen, wie beispielsweise Tiefensuche, zu nutzen [68].
- Er passt sein Statechart-Modell an, um den Zustandsraum zu verkleinern.

Da in dieser Arbeit untersucht werden soll, wie ein Anwender das DVF auch ohne Expertenwissen im Bereich der formalen Verifikation nutzt, ist das Umkonfigurieren von Spin keine Option. Der DSL-Anwender probiert daher, den Zustandsraum durch Anpassungen am Modell zu reduzieren.

Gemäß Abschnitt 3.1 gibt es verschiedene Optimierungen, um den Zustandsraum eines Modells zu verkleinern. Einige, wie beispielsweise Bitstate-Hashing [67], werden automatisch seitens des Model Checkers durchgeführt. Das DSL Verification Framework stellt dem DSL-Anwender jedoch auch eine Optimierung zur Verfügung, die manuell auf Modellebene (vgl. Abschnitt 1.2.2) angewendet werden kann. Dabei handelt es sich um sogenannte symmetrische Arrays, deren Konzept in Unterkapitel 2.1.4 detailliert beschrieben wird.

Der DSL-Anwender kann also, wenn der Zustandsraum zu groß ist, sein Modell auf symmetrische Eigenschaften hin untersuchen. Bei Betrachtung von Listing 5.19 fällt auf, dass die Permutation der Elemente im Array *positions* keine Auswirkung auf die Semantik des Modells hat. Dies verdeutlicht das folgende Beispiel:

- Zu Beginn des Montageprozesses bewegt der Arbeiter seine Hand mit dem Marker wie folgt: P_4, P_1, P_4, P_2. In diesem Fall detektiert die AssyControl-Erweiterung, dass sich bei P_1 eine Box mit Achsen und bei P_2 eine Box mit Reifen befindet.
- Zu Beginn des Montageprozesses bewegt der Arbeiter seine Hand mit dem Marker wie folgt: P_4, P_2, P_4, P_1. In diesem Fall detektiert die AssyControl-Erweiterung, dass sich bei P_2 eine Box mit Achsen und bei P_1 eine Box mit Reifen befindet.

Beim Vergleich der Montageprozesse wird ersichtlich, dass sich die Achsen und Reifen bei unterschiedlichen Positions befinden. Trotzdem ist das System in beiden Fällen in dem selbem Zustand. Somit kann das Array *positions* aus Listing 5.19 als Scalarset implementiert werden. Daraus ergeben sich zwei weitere Anpassungen:

- Generieren eines Zufallswerts, um ein Scalarset zu indizieren: Vor der Modifikation wird *positions* mit einem Byte indiziert, das die Methode *nextPosition()* zurückliefert. Dies ist für Scalarsets nicht zulässig. Stattdessen muss ein neuer Index-Wert durch Aufruf von *nextIndex()* generiert werden.
- Indizierung des Scalarsets: Vor der Modifikation wird *positions* mit der Variable *cb* indiziert. Dies ist für Scalarsets nicht zulässig. Stattdessen muss *current()* Verwendung finden, um auf *positions* zuzugreifen.

Diese Art der Implementierung hat folgenden Vorteil: Der Model Checker kann durch Ausnutzung der symmetrischen Eigenschaften den Zustandsraum verkleinern. Des Weiteren erhält der Aufruf von *nextIndex()* und *current()* die ursprüngliche Semantik des Modells: Beim Betreten des Zustands *ReadSensor* (vgl. Listing 5.20) wird ein Zufallswert generiert, der als Index für *positions* dient.

Nach dem Anpassen des Modells erfolgt eine erneute Übersetzung nach Promela und Verifikation mit Spin. Der entsprechende Transformationsalgorithmus für Scalarsets wird in Abschnitt 2.1.4 beschrieben. Die Ausgabe von Spin ist im folgenden Listing zu sehen:

```
1 (Spin Version 6.1.0 -- 4 May 2011)
2         + Partial Order Reduction
3
4 Full statespace search for:
5         never claim             - (none specified)
6         assertion violations    +
7         cycle checks            - (disabled by -DSAFETY)
8         invalid end states      +
9
10 State-vector 124 byte, depth reached 110, errors: 0
11     17087 states, stored
12      7696 states, matched
```

```
13     24783 transitions (= stored+matched)
14         6 atomic steps
15 hash conflicts:         53 (resolved)
16
17     4.687        memory usage (Mbyte)
```

Listing 5.24: Ausgabe von Spin nach EInfügen symmetrischer Arrays

Das obige Listing 5.24 zeigt, dass der vollständige Zustandsraum des Modells untersucht werden konnte. Durch das Ausnutzen der symmetrischen Eigenschaften belegt der Verifier lediglich 4.6 Megabyte des Hauptspeichers. Da Spin das Modell vollständig untersucht hat und keine weiteren Fehler findet, kann in einem letzten Schritt eine Transformation nach Java erfolgen.

Transformation nach Java

Im vorangegangenen Abschnitt wird das Statechart-Modell der ersten Fallstudie in die Model Checker-Eingabesprache Promela übersetzt. Darauf folgt die Verifikation mit Spin. Das Modell muss zweimal angepasst werden, um einen Fehler zu beheben und symmetrische Eigenschaften auszunutzen. Neben der Verifikation ist jedoch auch eine Transformation in eine Hochsprache erforderlich. Daher schließt dieses Unterkapitel die erste Fallstudie ab und beschreibt die Übersetzung nach Java.

Um den generierten Quellcode testen zu können, wird im Rahmen des Forschungsprojekts *KoverJa* [79] eine Java-Applikation namens *AssyControl-GUI* entwickelt [9]. Die AssyControl-GUI ist in Abbildung 5.8 zu sehen und visualisiert die Bewegungen eines Markers mittels OpenGL [154] bzw. Swing [125]:

- Die gelbe Sphäre stellt den Marker dar.
- Die grünen Quader repräsentieren die Positions.
- Textnachrichten können ausgegeben werden.

Die AssyControl-GUI muss nicht direkt mit einem Ultraschallsystem verbunden sein. Stattdessen können die Bewegungsdaten des Markers auch aus einer Datei eingelesen werden.

Zusammen mit der AssyControl-GUI gestaltet sich die Umsetzung der ersten Fallstudie wie folgt: Der DSL-Anwender nutzt das DVF und überführt sein Statechart-Modell in die Hochsprache Java. Um den generierten Java-Quellcode testen zu können, wird er in die AssyControl-GUI integriert. Das genaue Vorgehen ist in Abbildung 5.9 zu sehen. Zunächst führt ein Arbeiter verschiedene Montageprozesse an der Werkbank durch. Er trägt dabei einen Marker. Ein Ultraschallsystem zeichnet die Bewegungsdaten auf. Die sich daraus ergebenden Datensätze dienen der AssyControl-GUI als Eingabe. Sie besteht durch die Integration der AssyControl-Erweiterung aus zwei nebenläufigen Prozessen:

Abbildung 5.8: Screenshot der AssyControl-GUI

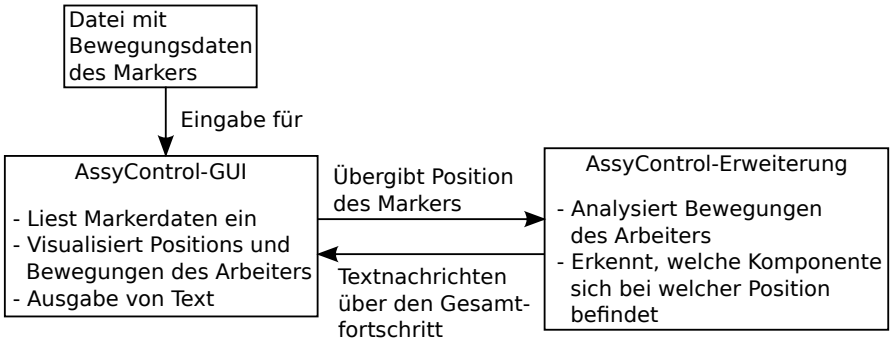

Abbildung 5.9: Integration der ersten Fallstudie in die AssyControl-GUI

- Die AssyControl-GUI liest die Bewegungsdaten ein und visualisiert sie.
- Der Automat der AssyControl-Erweiterung wartet auf die Bewegungen des Markers, um den Aufbau des Montageplatzes zu detektieren.

Zur Umsetzung der ersten Fallstudie wird die GUI so modifiziert, dass sie dem UltraSound-Objekt (vgl. Listing 5.19) der AssyControl-Erweiterung die aktuelle Position des Markers mitteilt. Durch dieses Vorgehen kann der endliche Automat der AssyControl-Erweiterung die Bewegungen des Arbeiters analysieren und den Aufbau des Arbeitsplatzes erkennen. Den aktuellen Fortschritt sendet er an die AssyControl-GUI zurück. Sie gibt ihn in Form von Textnachrichten aus.

5.1.5 Auswertung

Die vorangegangenen Abschnitte stellen die erste Fallstudie vor. Dieses Unterkapitel fasst die dabei gewonnenen Ergebnisse zusammen. Es soll auch untersucht werden, ob die Verwendung des DSL Verification Frameworks den Entwicklungsprozess beschleunigt und so die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation erleichtert. Dabei sind die folgenden Aspekte von zentraler Bedeutung:

- Kann das DVF den Implementierungssaufwand reduzieren?
- Ist durch das DVF weniger Expertenwissen zur Umsetzung der DSL notwendig?
- Konnte der gesamte Zustandsraum des Modells verifiziert werden?

Die drei Fragestellungen werden im weiteren Verlauf des Abschnitts genauer untersucht.

Implementierungsaufwand

Das DVF verfolgt den Ansatz, den Implementierungsaufwand einer DSL durch das Bereitstellen vorgefertigter Elemente zu reduzieren. Die folgende Tabelle zeigt deshalb, welche Sprachkonstrukte mit dem DVF abgebildet werden können und wie sich eine Umsetzung der ersten Fallstudie ohne DVF gestaltet:

Manuelles Beschreiben bzw. Übersetzen von:	Mit DVF	Ohne DVF
Klassen		×
Variablen		×
Statements		×
Expressions		×
Endliche Automaten	×	×
Zustände	×	×
Transitionen	×	×
Sende-Operationen	×	×

Die obige Tabelle besteht aus drei Spalten: Die erste zeigt alle Sprachkonstrukte, aus denen sich die Statechart-DSL zusammensetzt. Die zweite und dritte Spalte vergleichen, welche Aspekte der Statechart-DSL manuell implementiert werden müssen, wenn ein DSL-Entwickler das DVF verwendet bzw. wenn er es nicht verwendet. Es wird deutlich, dass fast die Häfte aller Sprachkontrukte vom DVF bereitgestellt werden. Dazu gehören Klassen mit Methoden, die ein endlicher Automat aufruft. Des Weiteren können die Ergebnisse von arithmetischen Operationen in Variablen gespeichert werden. Die Zuweisung erfolgt mittels Statements. Expressions ermöglichen das Modellieren von Bedingungen, die erfüllt sein müssen, damit eine Transition ausführbar ist. Der DSL-Entwickler kann somit die folgenden Produktionsregeln des DVF in seine Grammatik integrieren: *DVFClass*, *DVFVariableDeclare*, *DVFEnumDeclare*, *DVFStatement* und *DVFExpression*. Ohne das DSL Verification Framework ist eine manuelle Umsetzung der entsprechenden Produktionsregeln notwendig, die den Implementierungsaufwand der domänenspezifischen Sprache erhöht.

Einige Aspekte der Statechart-DSL sind jedoch nicht Teil des DVF. Der DSL-Entwickler muss daher die folgenden Produktionsregeln manuell umsetzen:

- *STMStmInstance* zum Instanzieren der endlichen Automaten.
- *STMStmClass* zum Deklarieren der endlichen Automaten.
- *STMSensitive* ist eine Liste von Signalen, auf die der Automat sensitiv ist.
- *STMState* und *STMTransition* repräsentieren Zustände bzw. ihre ausgehenden Transitionen.
- *STMSend* ist ein Statement, um ein Signal an einen Automaten zu übertragen.

Neben der Menge bedarf auch der Komplexitätsgrad einer genaueren Betrachtung. Die Listings 5.2 bis 5.7 enthalten die manuell umgesetzten Produktionsregeln der Statechart-DSL. Sie haben eine Größe von 5 bis 11 Zeilen. Der Umfang der Produktionsregeln, die Teil des DVF sind, ist deutlich höher. Hierzu gehört *DVFExpression* (vgl. Abschnitt 4.3.3), das sich aus mehreren Produktionsregeln zusammensetzt, die sich gegenseitig referenzieren um unterschiedliche Operatoren abzubilden. Das DVF reduziert also den Implementierungsauf der Statechart-DSL, da der DSL-Entwickler lediglich die Hälfte aller Produktionsregeln manuell implementieren muss und diese einen geringen Komplexitätsgrad aufweisen.

Expertenwissen

Neben dem Implementierungsaufwand soll das DVF auch die Notwendigkeit von Expertenwissen reduzieren. Die aus der ersten Fallstudie gewonnenen Ergebnisse sind in der folgenden Tabelle zusammengefasst und werden im weiteren Verlauf des Abschnitts genauer vorgestellt:

Eigenschaft	Mit DVF	Ohne DVF
Expertenwissen Model Checking (Promela)		×
Expertenwissen Symmetrie (TopSpin)		×
Expertenwissen Linksrekursion		×
Grundkenntnisse im Bereich Compilerbau (Xtext)	×	×

Die obige Tabelle besteht aus drei Spalten: Die erste enthält alle Wissensgebiete, die zur Umsetzung der ersten Fallstudie relevant sind. Die zweite und dritte Spalte vergleichen, in welchen Bereichen der Statechart-DSL Expertenwissen vorhanden sein muss, wenn ein DSL-Entwickler das DVF verwendet bzw. wenn er es nicht verwendet. Die erste Spalte zeigt, dass die AssyControl-Erweiterung ohne Expertenwissen im Bereich der formalen Verifikation umgesetzt werden kann. Dies begründet sich durch die Tatsache, dass der DSL-Entwickler einen Transformator in die CFIL implementiert und nicht in eine Model Checker-Eingabesprache. Auch der DSL-Anwender muss lediglich mit der Statechart-DSL vertraut sein, um die AssyControl-Erweiterung zu modellieren. Das DVF abstrahiert somit erfolgreich die Konzepte von Spin und Promela.

Des Weiteren ist es für die erste Fallstudie nicht notwendig, dass der DSL-Entwickler mit dem Konzept der Symmetrie im Kontext der formalen Verifikation vertraut ist. Er ergänzt lediglich die Grammatik der Statechart-DSL mit der Produktionsregel *DVFScalarset* (vgl. Listing 4.9). Dies ermöglicht es dem DSL-Anwender, Modelle zu beschreiben, die symmetrische Arrays bzw. Scalarsets enthalten (vgl. Listing 5.1). Die Transformatoren des DVF erkennen symmetrische Arrays und überführen sie so nach Promela, dass Spin bzw. TopSpin den Zustandsraum entsprechend verkleinern. Im Fall der AssyControl-Erweiterung führt dieses Vorgehen dazu, dass Spin lediglich 4.6 Megabyte

benötigt, um das gesamte Modell im Hauptspeicher abzubilden.

Neben der formalen Verifikation soll das DVF auch die Notwendigkeit von Expertenwissen im Bereich Compilerbau reduzieren. Bei Betrachtung der Statechart-Grammatik fällt auf, dass der DSL-Entwickler lediglich Sprachkonstrukte manuell beschreiben muss, die strukturelle Aspekte der DSL abbilden. Hierzu gehören Transitionen und Zustände. Sie enthalten keine Rekursion oder Vererbung (vgl. Abschnitt 2.2). Komplexere Sprachkonstrukte, wie Statements oder Expressions, die hingegen Kenntnisse im Bereich Linksrekursion oder Mehrdeutigkeit [1] erfordern, sind ausschließlich dem DSL Verification Framework entnommen worden. Durch Nutzung des DVF benötigt der DSL-Entwickler somit weniger Expertenwissen. Zu beachten ist jedoch, dass der DSL-Entwickler auch unter Verwendung des DVF einige Produktionsregeln manuell beschreiben muss, was zumindest Grundkenntnisse im Bereich Compilerbau erforderlich macht.

Verifikation

Nach der Implementierung der Grammatik und des Transformators kann die AssyControl-Erweiterung mit der Statechart-DSL beschrieben werden. Das entsprechende Modell und die Transformationen nach Java bzw. Promela sind Teil von Abschnitt 5.1.4. Spin fand beim Verifizieren des Promela-Modells einen ungültigen Zugriff über eine Array-Grenze hinweg. Der Fehler wurde behoben und das Modell erneut verifiziert. Spin konnte jedoch nicht den gesamten Zustandsraum des Modells untersuchen, da der zur Verfügung stehende Hauptspeicher in Höhe von 1.7 Gigabyte nicht ausreichte. Da das Modell einen symmetrischen Aspekt enthält, wurde vom DSL-Anwender ein Array in ein Scalarset umgewandelt.

Nach einer erneuten Transformation benötigt Spin lediglich 4.6 Megabytes, um den gesamten Zustandsraum des Modells zu untersuchen. Dabei werden die folgenden Vor- bzw. Nachteile des DVF ersichtlich:

- Wenn ein Modell Scalarsets enthält, erzeugt das DVF automatisch Promela-Quellcode, der so strukturiert ist, dass Spin bzw. TopSpin [43] die symmetrischen Eigenschaften ausnutzen und den Zustandsraum reduzieren.
- Der DSL-Entwickler muss mit dem Konzept der Scalarsets vertraut sein, um zu erkennen, dass bei einigen Modellen die Permutation der Array-Elemente keine Auswirkung auf dessen Semantik hat (vgl. Abschnitt 2.1.4).

Nach der Verifikation kann das Modell in die Hochsprache Java übersetzt und die erste Fallstudie abgeschlossen werden. Zusammenfassend wird deutlich, dass das DVF die Anforderung erfüllt, automatisiert Promela-Modelle zu erzeugen, die nicht von dem Problem der *State Space Explosion* betroffen sind. Modellgetriebene Entwicklung und formale Verifikation sind somit erfolgreich im Rahmen der ersten Fallstudie miteinander verknüpft worden.

5.2 Fallstudie 2: InTune

Die AssyControl-Fallstudie zeigt, dass mit dem DVF und dessen Optimierungen auf Modellebene erfolgreich ein Modell beschrieben, verifiziert und in eine Hochsprache übersetzt werden kann. Sie weist jedoch eine Besonderheit auf: Es gibt keine Nebenläufigkeit. Deshalb stellt dieses Kapitel eine zweite Industriefallstudie vor, deren Ziel die Implementierung einer nebenläufigen Web-Anwendung names *InTune* ist. Zu diesem Zweck wird mit dem DVF eine domänenspezifische Sprache namens GWT-DSL entwickelt. Von besonderem Interesse ist der Implementierungsaufwand des CFIL-Transformators und somit die Beantwortung der Frage, ob der Einsatz des DVF lohnenswert ist. Des Weiteren wird auch untersucht, inwiefern die vordefinierten Produktionsregeln des DVF ausreichend sind, um in einem Industrieprojekt Verwendung zu finden.

Die DSL, die Transformatoren und ihre Umsetzung mit dem DVF sind Gegenstand dieses Abschnitts. Das allgemeine Vorgehen des DVF, um eine Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation zu erreichen, wird im Rahmen von Abbildung 1.8 beschrieben. Daraus leitet sich für die zweite Fallstudie ein konkreter Workflow ab. Er ist in Abbildung 5.10 zu sehen. Abgerundete Ecken stehen für Komponenten, die der DSL-Entwickler implementiert. Gestrichelte Ränder symbolisieren Elemente, die vom DVF bereitgestellt werden. Abbildung 5.1 zeigt, dass der DSL-Entwickler zunächst eine Grammatik beschreibt. Sie besteht sowohl aus Produktionsregeln des DVF, als auch benutzerspezifischen Elementen. Zu den DVF-Produktionsregeln gehören beispielsweise Statements, um Verhalten abzubilden. Die Client- und Server-Seite der Web-Anwendung werden mit benutzerspezifischen Sprachkonstrukten umgesetzt.

Modelle, die mit der domänenspezifischen Sprache für Web-Anwendungen beschrieben sind, sollen in eine Hoch- und eine Model Checker-Eingabesprache überführt werden. Zu diesem Zweck stellt das DVF einen Transformator bereit, der Sprachkonstrukte des DVF automatisiert nach Java und Promela überführt. Die domänenspezifische Sprache für Web-Anwendungen enthält jedoch auch Elemente, die nicht Teil des DSL Verification Frameworks sind. Deshalb implementiert der DSL-Entwickler einen Transformator, der alle benutzerspezifischen Sprachkonstrukte in die Control Flow Intermediate Language (CFIL) übersetzt. Dazu gehören beispielsweise GUI-Elemente, die der Transformator in Klassen übersetzt.

Aus dem Vorgehen in Abbildung 5.10 leitet sich für die zweite Fallstudie die folgende Gliederung ab: Abschnitt 5.2.1 stellt die zu implementierende Web-Anwendung vor. Dabei handelt es sich um ein Projekt namens *InTune*, das das Testen in verteilten Systemen ermöglicht. Aus den Anforderungen an InTune wird in Unterkapitel 5.2.2 eine domänenspezifische Sprache namens *GWT-DSL* abgeleitet. Ihre Grammatik besteht sowohl aus Produktionsregeln des DSL-Entwicklers, als auch aus Produktionsregeln des DVF. Nach der Beschreibung der DSL muss ein Transformator in CFIL implementiert werden. Die Vorstellung eines entsprechenden Transformationsalgorithmus erfolgt in Ab-

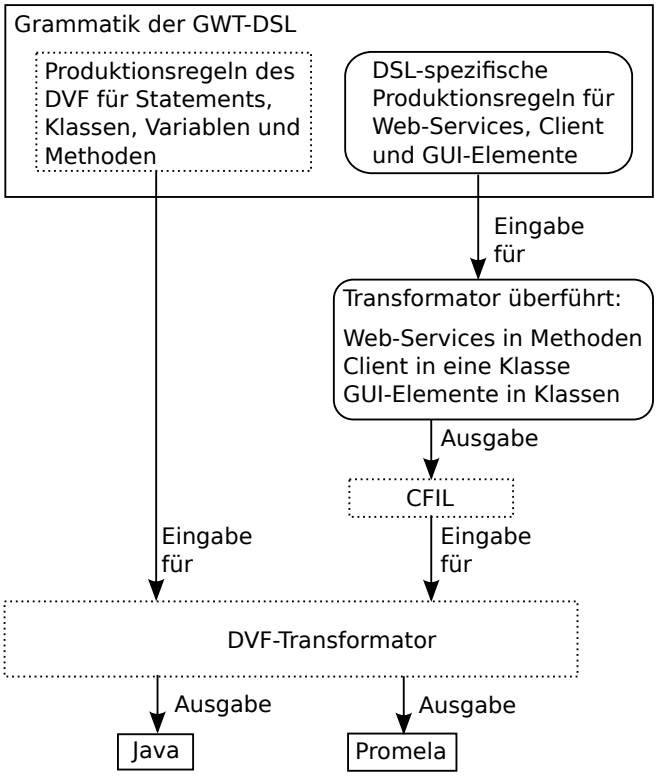

Abbildung 5.10: Umsetzung der zweiten Fallstudie mit dem DVF

schnitt 5.2.3. Anschließend kann die zweite Fallstudie mit der GWT-DSL modelliert werden. Unterkapitel 5.2.4 stellt das entsprechende Modell und die Übersetzung nach Promela bzw. Java vor. Zum Abschluss werden die Ergebnisse der zweiten Fallstudie in Abschnitt 5.2.5 ausgewertet.

5.2.1 Beschreibung von InTune

Auch die zweite Fallstudie ist eine Applikation der *soft2Tec GmbH* und heisst *InTune*. Mit ihr können automatisiert Testfälle in verteilten Systemen erzeugt, ausgeführt und ausgewertet werden. InTune besteht aus zwei Komponenten, die in Abbildung 5.11 zu sehen sind:

- Conductor
- Orchestra

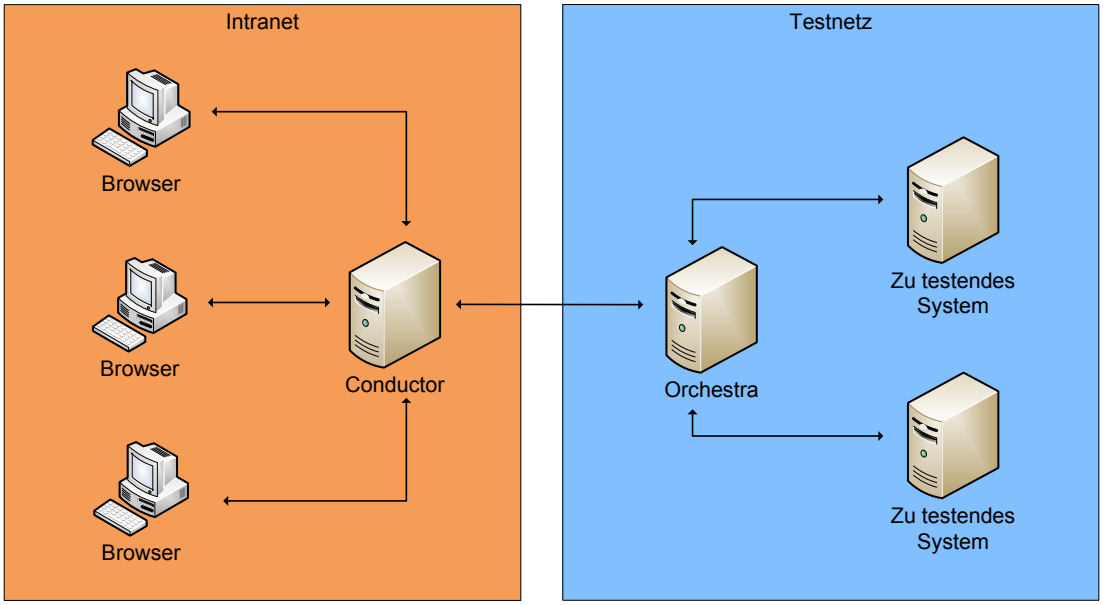

Abbildung 5.11: Aufbau von InTune

Der Conductor ist das zentrale Element von InTune, auf dem sich Benutzer einloggen, um Testfälle zu verwalten oder zu starten. Der Conductor ist als Web-Anwendung implementiert. Anwender nutzen somit einen Web-Browser. Wenn ein Anwender auf den Conductor zugreift, kann er die dort hinterlegten Testfälle auswählen und starten. Anschließend verbindet sich der Conductor mit der zweiten InTune-Komponente namens Orchestra, die sich im Testnetzwerk befindet und die entsprechenden Tests durchführt.

Das folgende Beispiel verdeutlicht den Ansatz: Es soll ein System getestet werden, das aus einem Web-Server und einer Datenbank besteht. Sowohl Web-Server, Datenbank als auch Orchestra befinden sich in einem separaten Testnetzwerk. Sobald ein Anwender über den Conductor den Test gestartet hat, verbindet sich dieser mit Orchestra. Orchestra überprüft, ob beide Systeme im Netzwerk vorhanden sind und startet die Testfälle. Dabei ist eine Priorisierung wichtig: Der Web-Server soll erst getestet werden, wenn sichergestellt ist, dass die Datenbank korrekt arbeitet. Nach Ausführung der Tests sendet Orchestra die entsprechenden Log-Files zurück an den Conductor. Dieser bereitet sie graphisch auf und präsentiert sie dem Anwender.

Bei deren Implementierung von Orchestra und Conductor kommen verschiedene Technologien zum Einsatz. Der Conductor ist eine Web-Anwendung, die sich auf einem Web-Server befindet. Sie besteht typischerweise aus HTML-Dokumenten [155], Javascript [49] und Java-Servlets [157]. Wenn ein Anwender einen Testfall startet, müssen die ent-

sprechenden Daten an Orchestra übertragen werden. Dafür kann beispielsweise ein Web-Service Verwendung finden. Bei der Implementierung von InTune sind daher die folgenen Aspekte zu berücksichtigen:

- Es ist wünschenswert, dass neben Software-Entwicklern auch Experten für graphische Oberflächen partizipieren.
- Die Entwickler müssen mehrere Technologien beherrschen, wie HTML, Java und Javascript.
- Das System ist hochgradig nebenläufig: Mehrere Anwender können gleichzeitig auf den Conductor zugreifen, der die parallele Ausführung verschiedener Testfälle auf Orchestra initiiert.

Die Nutzung einer domänenspezifischen Sprache in Kombination mit dem DVF bietet die folgenden Vorteile: Eine DSL zur Implementierung von Conductor und Orchestra kann automatisiert in die verschiedenen Zielsprachen, wie HTML oder Java, überführt werden. Die automatische Übersetzung nach Promela ermöglicht zusätzlich eine Verifikation mit Spin. Dies ist von Vorteil, um beispielsweise Probleme zu entdecken, die durch die nebenläufigen Aspekte hervorgerufen werden. Des Weiteren können Domänenexperten für graphische Benutzeroberflächen die DSL verwenden, ohne tiefergehende Kenntnisse im Bereich der Software-Entwicklung zu haben.

Das Ziel der zweiten Fallstudie ist somit die Entwicklung bzw. Nutzung einer domänenspezifischen Sprache für Web-Anwendungen. Dabei muss das folgende Problem beachtet werden: Das DVF überführt eine DSL automatisiert nach Java und Promela. Im Rahmen der zweiten Fallstudie soll jedoch eine Übersetzung in Zielsprachen wie HTML oder Javascript erfolgen. Diesem Problem wird mit dem Google Web Toolkit (GWT) [59] begegnet, das Abschnitt 2.4 detailliert vorgestellt hat. Beim GWT handelt es sich um ein Framework, das Java-Quellcode automatisiert in eine Web-Anwendung überführt. Daraus ergibt sich für die zweite Fallstudie das folgende Vorgehen:

- Es werden eine domänenspezifische Sprache für Web-Anwendungen und ein entsprechender Transformator in die CFIL entwickelt. Dabei kommt das DVF zum Einsatz.
- Das DVF überführt ein Modell nach Promela. Das Ergebnis wird von Spin verifiziert.
- Wenn der Model Checker keine weiteren Fehler findet, überführt das DVF das Modell nach Java. Das Ergebnis wird vom GWT eingelesen und in eine Web-Anwendung übersetzt.

Aus der Beschreibung von InTune ergeben sich die folgenden Anforderungen an eine domänenspezifische Sprache: InTune besitzt eine graphische Benutzeroberfläche. Wenn

ein Anwender darauf zugreift, kann er nach Eingabe von Benutzername und Passwort seine Testfälle verwalten. Die domänenspezifischen Sprache sollte daher Elemente wie Textboxen oder Eingabefelder enthalten, um den Entwicklern die Modellierung einer entsprechenden Benutzerschnittstelle zu ermöglichen.

Für einige InTune-Eigenschaften ist die Anbindung an eine Datenbank von Vorteil. Dazu gehört die Abfrage des Benutzerpassworts oder das Auslesen der zur Verfügung stehenden Testfälle. Die domänenspezifische Sprache zur Modellierung von InTune sollte deshalb die Möglichkeit bieten, Datenbankzugriffe abzubilden.

InTune besteht aus Conductor und Orchestra. Der Conductor verbindet sich mit Orchestra und übergibt eine Liste von auszuführenden Testfällen. Die Kommunikation der beiden Komponenten soll mit einem Web-Service modelliert werden. Aus diesem Grund muss die domänenspezifische Sprache das Modellieren von Web-Services ermöglichen, die sich auf einem Web-Server befinden und die ein Client aufruft.

5.2.2 Entwicklung der GWT-DSL

Im vorangegangenen Unterkapitel wird InTune vorgestellt und Anforderungen an eine domänenspezifische Sprache formuliert. Dieser Abschnitt leitet daraus eine DSL zur Beschreibung von Web-Anwendungen ab. Sie trägt den Namen GWL-DSL und wird im weiteren Verlauf der Arbeit genutzt, um InTune zu implementieren.

Gemäß Abbildung 5.10 gestaltet sich die Entwicklung einer domänenspezifischen Sprache in Kombination mit dem DVF folgendermaßen: Der DSL-Entwickler beschreibt die GWT-DSL mit einer Grammatik. Sofern dies möglich ist, nutzt er dafür die vorgefertigten Produktionsregeln aus dem DVF. Da die Referenzimplementierung des DSL Verification Frameworks mittels Xtext erfolgt (vgl. Kapitel 4), wird auch die Grammatik der GWT-DSL unter Zuhilfenahme von Xtext beschrieben. Der vorliegende Abschnitt stellt die entsprechenden Produktionsregeln genauer vor.

InTune wird im Web-Browser des Anwenders ausgeführt, kann aber auch mit einem Server kommunizieren. Dementsprechend soll die GWT-DSL die folgenden beiden Sprachkonstrukte zur Verfügung stellen:

- Einen Client, der in einem Web-Browser ausgeführt wird.
- Einen Web-Server, der verschiedene Web-Services zur Verfügung stellt. Clients können die Web-Services aufrufen.

Daher implementiert der DSL-Entwickler für die GWT-DSL das folgende Wurzelelement:

```
1 Model:
2   server=GWTServer?
3   client=GWTClient
4   session=GWTSession?
```

```
5   classes+=DVFClass*
6 ;
```

Listing 5.25: Xtext-Grammatik für die GWT-DSL

Jedes Modell besteht aus einem Web-Client und einem Web-Server. Da das DVF hierfür keine vorgefertigten Produktionsregeln zur Verfügung stellt, muss sie der DSL-Entwickler manuell umsetzen. *GWTServer* wird in einen Web-Server abgeleitet, der eine Liste von Methoden enthält. Der Client, den die Produktionsregel *GWTClient* repräsentiert, kann diese Methoden aufrufen. Eine Besonderheit ist *GWTSession*: Web-Anwendungen nutzen zur Kommunikation mit dem Web-Server das zustandslose *Hypertext Transfer Protocol* (HTTP) [156]. In einigen Fällen, wie beispielsweise einer Passwortauthentifizierung, ist jedoch das Speichern des Zustands wünschenswert. Zu diesem Zweck wird *GWTSession* in die GWT-DSL integriert.

Objektorientierte Komponenten können hilfreich sein, um ein Modell besser zu strukturieren. Aus diesem Grund ist neben Client und Server auch die Produktionsregel *DVFClass* in die Gramatik integriert (vgl. Listing 5.25 Zeile 5). Die Klassen des DVF ermöglichen die Umsetzung eigenschaftserhaltender Abstraktion. Dies bietet in Web-Applikationen den Vorteil, dass beispielsweise die Schnittstelle zu einer Datenbank beschrieben werden kann, ohne dass deren konkrete Implementierung Teil des Modells ist.

Daraus ergibt sich für diesen Abschnitt das folgende Vorgehen: Der DSL-Entwickler muss *GWTSession*, *GWTServer* und *GWTClient* implementieren, um die Grammatik der GWL-DSL zu vervollständigen. Die entsprechenden Produktionsregeln werden in den folgenden Unterkapiteln genauer vorgestellt.

Sessions

Mit *GWTSession* kann der Zustand einer HTTP-Kommunikation gespeichert werden. Die Produktionsregel hat den folgenden Aufbau:

```
1 GWTSession:
2   "session" "{"
3     (variables+=DVFVariableDeclare ";")+
4   "}"
5 ;
```

Listing 5.26: Xtext-Grammatik für Sessions

Jede Session beginnt mit dem Schlüsselwort *session* und beinhaltet eine Menge von Variablen. Da das DVF bereits eine eine Produktionsregel für Variablendeklarationen enthält, wird sie in Zeile 3 in die Grammatik integriert. Das Session-Element hat in einem Modell die folgende Semantik: Wenn sich ein Client das erste mal mit dem Server verbindet, wird ihm eine Session-ID zugewiesen. Bei jedem weiteren Verbindungsaufbau

überträgt der Client die ID zurück an den Server, damit er ihn eindeutig identifizieren kann. Die Variablen in *GWTSession* sind der jeweiligen Session-ID zugeordnet. Das bedeutet, wenn sich beispielsweise fünf Clients mit dem Server verbinden, erzeugt dieser fünf Session-Objekte und verknüpft sie mit den entsprechenden Session-IDs.

Eine Passwortabfrage kann damit wie folgt umgesetzt werden: Der DSL-Anwender legt im Session-Objekt die Boolean-Variable *login_success* an. Sobald sich ein Benutzer erfolgreich mit dem System authentifiziert hat, setzt der Server *login_success* auf *true*. Wenn sich der Nutzer ein weiteres Mal mit dem Server verbindet, kann dieser *login_success* abfragen und erkennen, ob es sich um einen authentifizierten Anwender handelt.

Diese Art der Implementierung hat den Vorteil, dass der DSL-Anwender das Umsetzen von Cookies oder einer Session-ID nicht manuell vornehmen muss. Stattdessen kann implizit davon ausgegangen werden, dass die Variablen in *GWTSession* zu einem bestimmten Client gehören. Dies ermöglicht das Modellieren von Zuständen, wie eine Benutzerauthentifizierung, ohne dass Kenntnisse im Bereich des HTTP notwendig sind.

Server

Die GWT-DSL enthält die Produktionsregel *GWTSession* zur Beschreibung von Web-Services. Ihre Xtext-Umsetzung gestaltet sich wie folgt:

```
1 GWTServer:
2   "server" "{"
3     (variables+=DVFVariableDeclare ";")*
4     (functions+=DVFFunctionDeclare)+
5   "}"
6 ;
```

Listing 5.27: Xtext-Grammatik für den Server

Die Server-Seite wird durch das Schlüsselwort *server* eingeleitet. Sie besteht aus einer Menge von Variablen und einer Menge von Methoden. Jede Methode repräsentiert einen Web-Service und kann von einem Web-Client aufgerufen werden. Das DVF beinhaltet vorgefertigte Produktionsregeln zur Beschreibung von Methoden und Variablen. Daher werden sie in Zeile 3 und 4 in die Grammatik der domänenspezifischen Sprache integriert.

Client

Neben dem Server ermöglicht die GWT-DSL auch das Beschreiben der Client-Seite. Der Client wird in einem Browser ausgeführt. Er besteht typischerweise aus einer Menge von Web-Seiten. Der DSL-Entwickler muss deshalb ein Sprachkonstrukt in seine domänenspezifische Sprache integrieren, das den Client abbildet und eine Menge von Webseiten enthält. Die entsprechende Produktionsregel hat gemäß Listing 5.25 den Bezeichner *GWTClient*. Die Xtext-Implementierung gestaltet sich wie folgt:

```
1 GWTClient:
2   "client" "entry" "=" entry=[GWTPage] "{"
3     (pages+=GWTPage)+
4   "}"
5 ;
```

Listing 5.28: Xtext-Grammatik für den Client

Listing 5.28 ist so umgesetzt, dass *GWTClient* in eine Menge von Page-Elementen abgeleitet wird. Eine Besonderheit ist in Zeile 2 zu sehen: Wenn sich der Client das erste mal mit einem Web-Server verbindet, muss dieser eine Startseite anzeigen. Zu diesem Zweck wird die Entry-Direktive in die Grammatik aufgenommen. Sie referenziert die Seite, die der Client beim erstmaligen Verbinden mit dem Server aufruft.

Jedes GWTPage-Element entspricht einer HTML-Seite. Sie besteht aus einer Reihe von GUI-Elementen, wie Schaltern oder Textfeldern, mit denen der Benutzer interagiert. Wenn der Nutzer einen Schalter betätigt, wird Verhalten ausgeführt. Dazu gehören Berechnungen oder der Aufruf eines Web-Services. Die Ergebnisse von Berechnungen können in Variablen gespeichert werden. Daher bietet es sich an, die Deklaration von Variablen innerhalb von HTML-Seiten zu ermöglichen. Bei den genannten Web-Services handelt es sich um Methodenaufrufe. Methodenaufrufe in Hochsprachen, wie beispielsweise Java, sind synchron [60]. Bei einer Web-Anwendung bedeutet ein Methodenaufruf jedoch das Aufbauen einer Netzwerkverbindung, was bei hohen Latenzen zu Verzögerungen führen kann. Daher ist es wünschenswert, die Web-Services als asynchrone Methoden umzusetzen. Zu diesem Zweck werden Callback-Objekte in das Modell integriert. Sie kapseln Verhalten, das genau dann ausgeführt wird, wenn der aufgerufene Web-Service terminiert. Daraus ergeben sich für GWTPage-Elemente die folgenden Anforderungen:

- Jede HTML-Seite kann GUI-Elemente enthalten.
- Innerhalb einer HTML-Seite sollen Variablen deklariert werden können.
- Für den Aufruf von Web-Services soll die Deklaration entsprechender Callback-Objekte möglich sein.

Die genaue Umsetzung der Produktionsregel *GWTPage* gestaltet daher sich wie folgt:

```
1 GWTPage:
2   "page" name=ID "{"
3     (callbacks+=GWTCallBack)*
4     (variables+=DVFVariableDeclare ";")*
5     (elements+=GWTElement)+
6   "}";
```

Listing 5.29: Xtext-Grammatik für HTML-Seiten

Gemäß des obigen Listings kann *GWTPage* in eine Menge von GWTElement-Objekten abgeleitet werden, die die entsprechenden GUI-Elemente repräsentieren (vgl. Zeile 5). Da das DVF bereits eine vorgefertigte Produktionsregel für Variablendeklarationen zur Verfügung stellt, wird diese in Zeile 4 eingefügt. Das Ableiten in Callback-Objekte ist in Zeile 3 umgesetzt. Die Anforderungen und der Aufbau von *GWTCallback* bzw. *GWT-Element* werden im weiteren Verlauf dieses Abschnitts genauer vorgestellt.

Der Aufruf eines Web-Services kann fehlschlagen, wenn Netzwerkprobleme auftreten. Jedes Callback-Objekt soll deshalb zwei Methoden enthalten:

- *OnSuccess* wird ausgeführt, wenn der Web-Service erfolgreich aufgerufen werden konnte.
- *OnFailure* wird ausgeführt, wenn beim Aufruf des Web-Service ein Fehler auftrat.

Web-Services entsprechen einem Methodenaufruf. Sie können daher einen Rückgabewert enthalten. Da Web-Services asynchron sind, steht der Rückgabewert erst mit dem Ausführen von *onSuccess()* zur Verfügung. Er wird deshalb als Parameter von *onSuccess()* modelliert. Aus diesen Anforderungen ergibt sich für die Produktionsregel *GWTCallback* der folgende Aufbau:

```
1  GWTCallback:
2    "callback" name=ID "{"
3      "onFailure" "(" ")" "{"
4        (statements+=DVFStatement)*
5      "}"
6      "onSuccess" "(" ret=DVFVariableDeclare ")" "{"
7        (statements+=DVFStatement)*
8      "}"
9    "}"
10 ;
```

Listing 5.30: Xtext-Grammatik für Callback-Objekte

Die Ableitung der beiden Methoden erfolgt im obigen Listing in den Zeilen 3 und 6. Da das DSL Verification Framework bereits die Produktionsregel *DVFStatement* zur Beschreibung von Verhalten zur Verfügung stellt, wird diese in Zeile 4 bzw. 7 in die Xtext-Grammatik integriert. Auch der übergebene Parameter von *onSuccess()* wird bereits vom DVF unterstützt und ist in Zeile 6 zu sehen.

Jedes Page-Element kann neben Callback-Objekten auch verschiedene GUI-Elemente, wie Schalter oder Textfelder enthalten. Die Anforderungen aus dem vorangegangenen Abschnitt 5.2.1 zeigen, dass die graphische Schnittstelle sowohl Informationen darstellen, als auch als auch Interaktion mit dem Benutzer ermöglichen muss. Daher soll die GWT-DSL die folgenden GUI-Elemente unterstützen:

- Jede HTML-Seite soll Text anzeigen können.
- Es sollen Schalter zur Verfügung stehen. Wenn der Nutzer einen Schalter betätigt, wird Verhalten in Form von Statements ausgeführt.
- Es gibt Textfelder, in die der Anwender Daten eingeben kann.
- GUI-Elemente sollen horizontal oder vertikal ausgerichtet werden können, um die Lesbarkeit zu erhöhen.

Daraus leitet sich die folgende Grammatik für *GWTElement* (vgl. Listing 5.29) ab:

```
1 GWTElement:
2   button=GWTButton | label=GWTLabel | text=GWTTextBox |
3   horizontal=GWTHorizintal | vertical=GWTVertical
4 ;
```

Listing 5.31: Xtext-Grammatik für GUI-Elemente

Das obige Listing zeigt, dass *GWTElement* in einen Schalter, ein Label zur Textanzeige, ein Texteingabefeld oder in eine Struktur zur vertikalen bzw. horizontalen Ausrichtung abgeleitet werden kann. Die folgenden Listings stellen die verschiedenen Sprachkonstrukte genauer vor.

Jeder Schalter setzt sich aus einer Beschriftung und Verhalten in Form von Statements zusammen. Die Statements werden ausgeführt, wenn der Benutzer den Schalter betätigt. Daraus leitet sich die folgende Xtext-Grammatik ab:

```
1 GWTButton:
2   "button" "{"
3     "label" "{" text=STRING "}"
4     "behaviour" "{"
5       statements+=DVFStatement+
6     "}"
7   "}"
8 ;
```

Listing 5.32: Xtext-Grammatik für Buttons

Gemäß des obigen Listings beginnt jede Schalterdeklaration mit dem Schlüssselwort *button*. Darauf folgt gemäß Zeile 3 ein Label, das der Beschriftung dient. Auf das Label folgt ein Behaviour-Block, in dem der DSL-Anwender Statements spezifizieren kann. Da das DVF eine entsprechende Produktionsregel zur Verfügung stellt, wird diese in Zeile 5 in die Xtext-Grammatik integriert.

Neben Schaltern sollen HTML-Seiten auch Text darstellen können. Dadurch kann der DSL-Anwender Überschriften modellieren oder Elemente mit Bezeichnern versehen. Das nächste Beispiel zeigt deshalb die Umsetzung von *GWTLabel*:

```
1 GWTLabel:
2   "label" name=ID "{"
3     text = STRING
4   "}"
5 ;
```

Listing 5.33: Xtext-Grammatik für Label

Das obige Listing zeigt, dass jeder Text mit dem Schlüsselwort *label* beginnt. In Zeile 3 ist eine Zeichenkette referenziert, die den anzuzeigenden Text repräsentiert.

Neben dem Darstellen von Zeichenketten soll die DSL auch Texteingabefelder beinhalten. Diese können von den Statements im Behaviour-Block eines Schalters ausgelesen und weiter verarbeitet werden. Das nächste Listung zeigt deshalb die Umsetzung von *GWTTextBox*:

```
1 GWTTextBox:
2   "textbox" name=ID "{" "}"
3 ;
```

Listing 5.34: Xtext-Grammatik für Textfelder

Jedes Textfeld beginnt mit dem Schlüsselwort *textbox* und hat einen eindeutigen Bezeichner. Der Bezeichner ist notwendig, damit das Element von Statements referenziert werden kann, um seinen Inhalt auszulesen.

Neben der Eingabe von Text soll es auch möglich sein, GUI-Elemente horizontal oder vertikal auszurichten. Ein entsprechendes Sprachkonstrukt muss daher Schalter, Textflächen und Beschriftungen kapseln können. Das nächste Listing zeigt die Umsetzung der Produktionsregeln *GWTHorizontal* bzw. *GWTVertical*:

```
1 GWTHorizontal:
2   "horizontal" "{" elements+=GWTElement+ "}"
3 ;
4
5 GWTVertical:
6   "vertical" "{" elements+=GWTElement+ "}"
7 ;
```

Listing 5.35: Xtext-Grammatik für horizontale bzw. vertikale Panels

Sowohl *GWTHorizontal*, als *GWTVertical* dienen der Strukturierung von GUI-Elementen. Alle Elemente in *GWTHorizontal* werden horizontal zueinander angeordnet. Dementsprechend bewirkt *GWTVertical* eine vertikale Verteilung. Da beide Produktionsregeln von *GWTElement* abgeleitet sind, kann eine horizontale Umgebung auch weitere *GWTHorizontal* bzw. *GWTVertical-Komponenten* enthalten.

Die Menge der vorgestellten GUI-Elemente wird so gewählt, dass sie dem Nutzer Informationen anzeigen, er Datensätze eingibt und Verhalten durch das Betätigen von Schaltern ausführt. Bei Bedarf kann die domänenspezifische Sprache jedoch auch um neue Elemente ergänzt werden. In diesem Fall muss der DSL-Entwickler die Produktionsregel *GWTElement* aus Listing 5.31 erweitern. Ein möglicher Anwendungsfall wäre beispielsweise die Integration von Objekten zur Darstellung von Grafiken.

Im bisherigen Verlauf dieses Abschnitts sind die Sprachkonstrukte der GWL-DSL vorgestellt worden. Auf einige Elemente kann lesend bzw. schreibend zugegriffen werden. Hierzu gehören:

- Auf die in Sessions gekapselten Variablen (vgl. Listing 5.26) kann lesend oder schreibend zugegriffen werden.
- Die Benutzereingabe in Textfeldern kann ausgelesen werden (vgl. Listing 5.34).
- Jedem Label kann eine Zeichenkette zugewiesen werden, die die entsprechende Web-Seite anzeigt (vgl. Listing 5.33).

Der Zugriff auf Sprachkonstrukte der GWT-DSL wird mittels Statements umgesetzt. Für Statements stellt das DVF die Produktionsregel *DVFStatement* zur Verfügung. Der DSL-Entwickler muss daher *DVFStatement* so erweitern, dass es auch den Zugriff auf *GWTSession*, *GWTTextBox* und *GWTLabel* unterstützt. Der generelle Erweiterungsansatz wird in Abschnitt 4.8.1 beschrieben. Für *DVFStatement* führt dies zu den folgenden Anpassungen:

```
1 DVFStatement:
2   (assign=DVFAssignmentOrCall) | (assert=DVFAssert) |
3   (send=DVFSendSignal) | (read=DVFReadSignal) |
4   (ret=DVFReturn) | (loop=DVFLoop) | (ifelse=DVFIfElse) |
5   (wait=DVFWait) | (scalaraccess=DVFScalarAccess) |
6   (readtext=GWTReadText) | (writelabel=GWTWriteLabel) |
7   (readsession=GWTReadSession) | (writesession=GWTWriteSession)
8 ;
```

Listing 5.36: Erweiterung von DVFStatement

Die obige Produktionsregel ist eine Erweiterung von Listing 4.22 und enthält in den Zeilen 6 bis 7 vier neue Ableitungen. Sie ergänzen das DSL Verification Framework mit vier Statements, um die entsprechenden Elemente der GWT-DSL referenzieren zu können.

Das folgende Listung zeigt *readTextBox()*. Es ermöglicht das Auslesen von Textfeldern und ist als Methode umgesetzt, die den Bezeichner eines Textfeldes als Parameter übergeben bekommt. Der Rückgabewert ist der Inhalt des Textfelds als Zeichenkette.

```
1 GWTReadText:
2   "readTextBox" "("
3     textbox=[GWTTextBox]
4   ")" ";"
5 ;
```

Listing 5.37: Xtext-Grammatik zum Auslesen von Eingabefeldern

Die Produktionsregel im obigen Listing beginnt mit dem Schlüsselwort *readTextBox*. Darauf folgt in Zeile 3 eine Referenz auf ein Objekt vom *GWTTextBox*

Das nächste Listing zeigt die Umsetzung von *GWTWriteLabel*. Das Statement kann Textfeldern zur Laufzeit eine Zeichenkette zuweisen, die sie auf der HTML-Seite anzeigen. Bei *writeLabel()* handelt es sich um eine Methode, die zwei Parameter übergeben bekommt. Der Erste ist der Bezeichner des *Labels*, auf das zugegriffen werden soll. Der zweite Parameter repräsentiert die anzuzeigende Zeichenkette:

```
1 GWTWriteLabel:
2   "writeLabel" "("
3     label=[GWTLabel], content=DVFExpression
4   ")" ";"
5 ;
```

Listing 5.38: Xtext-Grammatik zum Beschreiben von Textfeldern

Gemäß des obigen Listings beginnt das Statement mit dem Schlüsselwort *writeLabel*. Darauf folgt in Zeile drei eine Referenzen auf ein Label und einen Ausdruck. Sie repräsentieren die beiden Parameter. Die Verwendung von *DVFExpression* hat zwei Vorteile: Die Produktionsregel ist bereits Teil des DVF und muss somit nicht manuell implementiert werden. Des Weiteren ermöglicht *DVFExpression* auch die Verwendung komplexerer, zusammengesetzter Ausdrücke.

Das nächste Listing zeigt die Umsetzung von *GWTReadSession*, um auf die Variablen von Session-Elementen zugreifen zu können:

```
1 GWTReadSession:
2   "readSession" "("
3     variable=[DVFVariableDeclare]
4   ")" ";"
5 ;
```

Listing 5.39: Xtext-Grammatik zum Auslesen von Session-Variablen

Auch das Auslesen von Session-Variablen ist als Methode umgesetzt. *ReadSession()* bekommt den Bezeichner einer Session-Variable übergeben und liefert ihren Wert zurück. Das nächste Listing zeigt den Aufbau von *GWTWriteSession*:

```
1 GWTWriteSession:
2   "writeSession" "("
3     variable=[DVFVariableDeclare] "," value=DVFExpression
4   ")" ";"
5 ;
```

Listing 5.40: Xtext-Grammatik zum Schreiben von Session-Variablen

Wenn der Webserver schreibend auf eine Session-Variable zugreift, muss die Methode *writeSession()* aufgerufen werden. Sie erhält als Paramater den Namen der entsprechenden Session-Variable und den ihr zuzuweisenden Wert.

5.2.3 Transformation in die CFIL

Der vorherige Abschnitt stellt eine DSL zur Beschreibung von Web-Applikationen vor. Damit die domänenspezifische Sprache genutzt werden kann, um die InTune-Fallstudie zu beschreiben, muss gemäß Abbildung 5.10 ein Transformator implementiert werden. Dieser Transformator übersetzt die Modelle der GWT-DSL in die CFIL. Aus dem Ergebnis generiert das DSL Verification Framework Java- bzw. Promela-Quellcode. Deshalb zeigt dieser Abschnitt, wie der DSL-Entwickler einen entsprechenden CFIL-Transformator implementieren kann. Das Vorgehen ist wie folgt strukturiert: Zunächst wird aus der Semantik der GWT-DSL ein Übersetzungsalgorithmus abgeleitet. Danach erfolgt die Vorstellung des eigentlichen Transformators, der ein Modell in die CFIL überführt.

Beschreibung des Transformationsalgorithmus

Dieser Abschnitt analysiert die Struktur bzw. die Semantik der GWT-DSL, um daraus einen Transformationsalgorithmus abzuleiten. Abbildung 5.12 zeigt den Aufbau einer Web-Anwendung, die mit der GWT-DSL modelliert ist. Sie besteht aus drei zentralen Komponenten:

- Ein Nutzer verwendet einen Web-Browser, der die HTML-Seiten anzeigt. Er nutzt GUI-Elemente, um mit dem Gesamtsystem zu interagieren.
- Der Client repräsentiert den Browser des Nutzers und die darin angezeigten HTML-Seiten. Durch die Nutzerinteraktion werden die Methoden des Servers aufgerufen. Da die GWT-DSL Klassen mit eigenschaftserhaltender Abstraktion unterstützt, können auch entsprechende Objekte Teil des Modells sein.
- Der Server beinhaltet Methoden, die die nebenläufigen Clients aufrufen. Da die GWT-DSL Klassen mit eigenschaftserhaltender Abstraktion unterstützt, können auch entsprechende Objekte Teil des Modells sein.

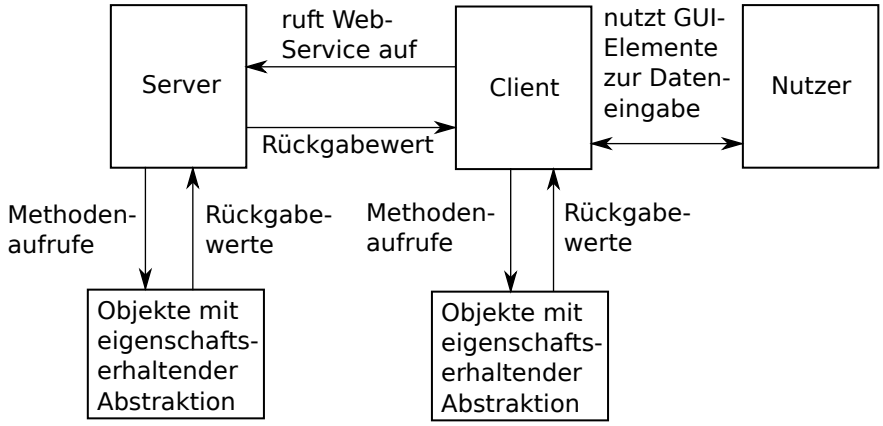

Abbildung 5.12: Struktur einer Web-Anwendung

Bei der genauen Betrachtung fällt auf, dass ein Modell aus zwei Entitäten besteht: Dem Server und einer Menge von Clients. Daher bietet es sich an, auch die Transformation in zwei Komponenten aufzuteilen. Zunächst wird der Server in die CFIL überführt. Darauf folgt die Transformation des Clients. Diese Art der Übersetzung bietet zwei Vorteile: Die Modularität führt zu einer besseren Wartbarkeit des Transformators. Des Weiteren können sowohl Client, als auch Server getrennt voneinander verifiziert werden. Das folgende Beispiel verdeutlicht den sich daraus ergebenden Vorteil: Gegeben sei ein System, das aus einem Server und zwei Clients besteht. Die Menge S repräsentiert den Zustandsraum des Servers. Die Zustände des Clients sind in C enthalten. Ein Model Checker müsste somit den Zustandsraum $G \subseteq S \times C \times C$ verifizieren. Bei dem vorgestellten modularen Ansatz werden hingegen S bzw. C getrennt voneinander betrachtet. Dies kann zu einer Reduktion des Zustandsraums führen und den Speicherverbrauch des Model Checkers positiv beeinflussen.

Das folgende Beispiel verdeutlicht den Transformationsalgorithmus: Gegeben sei ein Modell, das aus einem Server und zwei Clients besteht. Der Server beinhaltet die Web-Services *a()* und *b()*. Da die Transformation in zwei Schritte unterteilt ist, wird zunächst der Server in die CFIL überführt. Das Ergebnis ist in Abbildung 5.13 zu sehen. Für den Server wird eine Klasse generiert, die aus den Methoden *a()* und *b()* besteht. Der Model Checker muss jedoch auch das Verhalten der beiden Clients berücksichtigen. Daher wird der Thread *Client-Driver* erzeugt und zwei entsprechende Instanzen zum Modell hinzugefügt. Sie haben die folgende Funktionalität:

1. Wähle nicht-deterministisch die Methode *a()* oder *b()* aus.

2. Aufruf der entsprechenden Methode.

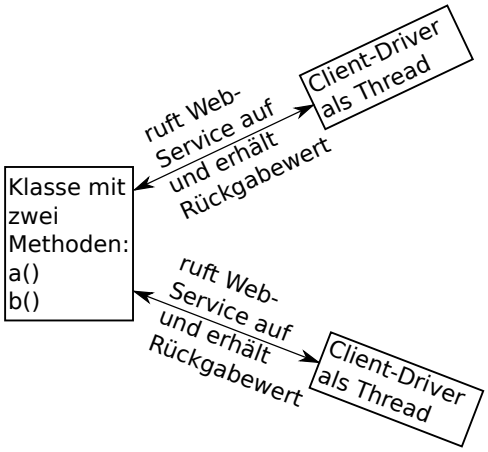

Abbildung 5.13: Transformation des Servers

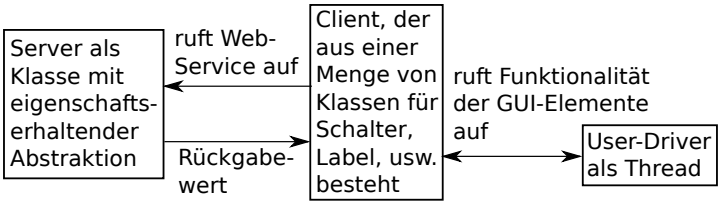

Abbildung 5.14: Transformation des Clients

3. Weiter bei 1.

Der Thread wird somit in einer Endlosschleife ausgeführt, ruft mit jedem Durchlauf einen zufälligen Web-Service auf und simuliert das Verhalten der Clients. Diese Art der Transformation hat zwei Vorteile:

1. Wenn das CFIL-Modell aus Abbildung 5.13 nach Java überführt wurde, kann das GWT die Server-Klasse einlesen, um daraus einen Teil der Web-Anwendung zu generieren.

2. Wenn das CFIL-Modell aus Abbildung 5.13 nach Promela überführt wurde, kann Spin den Server verifizieren, ohne dass die vollständige Implementierung des Clients darin enthalten ist. Dies reduziert den Zustandsraum und wirkt dem Risiko der State Space Explosion entgegen.

Neben dem Server muss auch der Client in die CFIL überführt werden. Abbildung 5.14 verdeutlicht das genaue Vorgehen: Zunächst überführt der Transformator den Client und

die in ihm enthaltenen Komponenten in eine Menge von Klassen. Der dafür notwendige Algorithmus ist Teil des nächsten Unterkapitels. Zum besseren Verständnis wird jedoch bereits an dieser Stelle ein Beispiel für das GUI-Element *GWTButton* gegeben. Es besteht gemäß Listing 5.32 aus einem Bezeichner und einer Liste von Statements. Die Statements stellen Verhalten dar, das ausgeführt wird, wenn der Nutzer den Schalter betätigt. Daher bietet es sich an, für jeden Schalter eine Klasse zu generieren, die eine Zeichenkette und eine Behaviour-Methode enthält.

Nach Abschluss der Transformation wird eine Menge von Klassen generiert, die den Client repräsentieren. Für die Untersuchung mit einem Model Checker muss jedoch auch das Verhalten des Servers und die Interaktion des Nutzers berücksichtigt werden. Die Verifikation des Servers erfolgt bereits im ersten Transformationsschritt. Daher generiert der Transformator eine Klasse, die die Methoden des Servers mit eigenschaftserhaltender Abstraktion umsetzt. Dieser Ansatz hat den Vorteil, dass der Client die entsprechenden Web-Services aufrufen kann, ohne dass ihre vollständige Implementierung ein Teil des Modells ist.

Zur Abbildung des Nutzerverhaltens wird der Thread *Client-Driver* zum Modell hinzugefügt. Er hat die folgende Funktionalität:

1. Rufe nicht-deterministisch eine HTML-Seite des Clients auf.
2. Wähle nicht-deterministisch ein GUI-Element der entsprechenden HTML-Seite aus.
3. Wenn es sich bei dem Element um ein Textfeld handelt, trage eine zufällige Zeichenkette ein.
4. Wenn es sich bei dem Element um einen Schalter handelt, betätige ihn durch Aufrufen der entsprechenden Behaviour-Methode.
5. Weiter bei 1.

Der Client-Driver wird somit als Endlosschleife ausgeführt, die zufällige GUI-Elemente nutzt und dadurch beliebiges Nutzerverhalten simuliert. Diese Art der Transfomation hat zwei Vorteile:

1. Wenn das CFIL-Modell aus Abbildung 5.14 nach Java überführt wurde, kann das GWT die Client-Klassen einlesen und daraus den noch fehlenden Teil der Web-Anwendung generieren.
2. Wenn das CFIL-Modell aus Abbildung 5.14 nach Promela überführt wurde, kann es Spin für alle möglichen Interaktionen des Nutzers verifizieren. Auch das Verhalten des Servers wird berücksichtigt, ohne dass seine vollständige Implementierung Teil des Modells ist, was den Zustandsraum verkleinert und somit dem Risiko der State Space Explosion begegnet.

Nach Abschluss des Übersetzungsprozesses werden Client und Server in die CFIL überführt. Bei einer genauen Betrachtung der GWT-DSL ergeben sich jedoch zwei Detailfragen, die untersucht werden müssen, bevor eine Implementierung des Transformators im Rahmen des nächsten Unterkapitels möglich ist.

Gegeben sei eine Web-Anwendung, die einen Schalter enthält. Der Client führt die Statements s_0 bis s_n aus, wenn der Nutzer den Schalter betätigt. Hier ist folgendes Szenario möglich: Der Anwender betätigt den Schalter und ruft in seinem Web-Browser eine HTML-Seite auf, bevor das Statement s_n ausgeführt werden konnte. Es stellt sich die Frage, ob der Client die Ausführung der Statements abbricht oder das Laden der neuen Seite erst nach Erreichen von s_n erfolgt. Dieser Aspekt wird empirisch in [7] [4] untersucht. Es ergaben sich für die verschiedenen Browser unterschiedliche Ergebnisse:

- Der *Internet Explorer 9* [99] lädt die neue Seite und führt parallel dazu im Hintergrund die noch verbleibenden Statements aus.
- *Chrome 14* [58] lädt erst dann eine neue Seite, wenn das Statement s_n erreicht wird.

Im Rahmen dieser Dissertation wird die Web-Anwendung so in die CFIL überführt, dass das Verhalten des Promela-Modells dem Chrome-Browser entspricht. Transformationen bezüglich *Internet Explorer* und *Firefox* sind Gegenstand zukünftiger Arbeiten.

Neben den Statements innerhalb eines Schalters bedürfen auch die Callback-Objekte einer genaueren Untersuchung. Sie enthalten zwei Methoden namens *onSuccess()* und *onError()*, deren Aufruf erfolgt, wenn der entsprechende Web-Service terminiert. Das folgende Modell verdeutlicht eine Fragestellung, die sich bei mehreren, parallel ausgeführten Web-Services ergibt:

```
1  page ExecutionTimeExample{
2    byte a;
3
4    callback callback1{
5      void onSuccess(){ a = 1; }
6      void onError(){ }
7    }
8    callback callback2{
9      void onSuccess(){ a = 2; }
10     void onError(){ }
11   }
12   onLoad{
13     rpc1(callback1);
14     rpc2(callback2);
15     a = 3; }
```

Listing 5.41: Modell mit zwei Callback-Objekten

Das obige Listing 5.41 zeigt ein Modell der GWT-DSL, das aus einer HTML-Seite namens *ExecutionTimeExample* besteht. Die darin eingebettete onLoad-Methode ruft die nicht näher definierten Web-Services *rpc1()* und *rpc2()* auf. Zum Abschluss wird a der Wert 3 zugewiesen und *onLoad* terminiert.

Das Modell enthält auch zwei Callback-Objekte namens *callback1* und *callback2*. Die Ausführung von *callback1.onSuccess()* erfolgt, wenn *rpc1()* terminiert. Analog dazu wird *callback2.onSuccess()* nach dem Beenden von *rpc2()* aufgerufen. Aufgrund des asynchronen Verhaltens und der unbekannten Laufzeit der beiden Web-Services stellt sich die Frage, in welcher Reihenfolge die Abarbeitung der Statements $a = 1$, $a = 2$ und $a = 3$ erfolgt. Auch dieser Aspekt ist Teil einer empirischen Untersuchung in [4], die für GWT-Anwendungen zu dem folgenden Ergebnis kam:

1. Nach dem Laden einer HTML-Seite beginnt, sofern vorhanden, die Ausführung von *onLoad*. Die darin aufgerufenen Web-Services sind nebenläufig.

2. Nach dem Abarbeiten des letzten Statements von *onLoad* wartet der Client, bis die aufgerufenen Web-Services terminieren.

3. Danach erfolgt die Ausführung der onSuccess- bzw onError-Methoden. Die Reihenfolge entspricht dem Aufruf der entsprechenden Web-Services.

Somit hat Listing 5.41 die folgende Semantik: Nach der Zuweisung $a = 3$ wartet der Client, bis *rpc1()* und *rpc2()* terminieren. Danach erfolgt der Aufruf von *callback1.onSuccess()* mit $a = 1$. Zum Abschluss wird das Statement $a = 2$ ausgeführt, das Teil von *callback2.onSuccess()* ist. Die Variable a enthält also am Ende der Ausführung immer den Wert 2 und das System ist deterministisch.

Transformation des Servers

Das vorangegangene Unterkapitel analysiert den Aufbau von GWT-DSL-Modellen und leitet daraus ein Übersetzungsalgorithmus ab. Dazu gehörte auch die semantische Analyse von Schaltern und Callback-Objekten. Das Ergebnis findet in diesem Abschnitt Verwendung, um einen CFIL-Transformator für die Server-Seite zu beschreiben. Zum Abschluss wird der Übersetzungsprozess anhand eines Beispiels demonstriert.

Die Aufgabe des Transformators ist es, alle Elemente im AST, die nicht Teil des DVF sind, mit Sprachkonstrukten aus der CFIL zu substituieren. Daher muss der DSL-Entwickler den Transformator so implementieren, dass er im Abstract Syntax Tree nach einem Objekt vom Typ *GWTServer* sucht (vgl. Listing 5.27). Gemäß des Vorgehens aus dem vorangegangenen Abschnitt wird *GWTServer* in eine Klasse überführt, die die Web-Services mit gleichnamigen Methoden umsetzt. Dafür kann der folgende Algorithmus Verwendung finden:

1. Generiere eine Klasse vom Typ *DVFClass* mit dem Bezeichner *Server*.

2. Iteriere über alle Variablen, die in *GWTServer* enthalten sind. Generiere in der Server-Klasse für jedes gefundene Element eine Variable. Diese Variable hat den selben Typ und den selben Bezeichner.

3. Iteriere über alle Web-Services, die in *GWTServer* enthalten sind. Generiere in der Server-Klasse für jedes gefundene Element eine Methode. Diese Methode besteht aus dem selben Rückgabewert, Parametern und Statements.

4. Ersetze *GWTSession* im AST mit der erzeugten Server-Klasse.

Auch das Session-Objekt (vgl. Listing 5.26) muss im Rahmen der Server-Transformation berücksichtigt werden. Damit eine Erkennung seitens des Google Web Toolkits möglich ist, bietet es sich an, das Session-Objekt in eine weitere Klasse zu überführen. Der DSL-Entwickler kann diesbezüglich den folgenden Algorithmus implementieren:

1. Suche im Abstract Syntax Tree nach einem Objekt vom Typ *GWTSession*.

2. Generiere eine Klasse mit dem Bezeichner *Session*. Erzeuge darin für jede Variable, die Teil von *GWTSession* ist, ein gleichnamiges Element.

3. Lösche *GWTSession* aus dem AST und erstelle ein Session-Objekt in der Server-Klasse.

Gemäß Abbildung 5.13 muss auch ein Thread erzeugt werden, der den Client während der formalen Verifikation mit Spin emuliert. Er trägt den Bezeichner Client-Driver und ruft die Methoden des Servers in einer Endlosschleife auf. Um ihn zu generieren, kann der Transformator wie folgt implementiert werden:

1. Erzeuge einen CFIL-Thread und integriere ihn in den AST.

2. Füge zum Thread eine While-Schleife hinzu.

3. Füge zur While-Schleife einen nicht-deterministischen If-Block hinzu. In jeder Verzweigung des If-Blocks wird ein Web-Service aufgerufen.

4. Web-Services können die Übergabe von Parametern erfordern. In diesem Fall muss mittels eigenschaftserhaltender Abstraktion eine Methode erzeugt werden, die entsprechende Zufallswerte generiert. Durch diese Art der Implementierung berücksichtigt der Model Checker alle möglichen Parameter, die ein Web-Client an den Server übertragen kann.

Mit dem bisher vorgestellten Algorithmus kann der Transformator den Server in die CFIL überführen. Zur besseren Veranschaulichung verdeutlicht ein Beispiel den Übersetzungsprozess:

```
server{
  void increaseCounter(){
    writeSession(counter, readSession(counter) + 1);
  }

  void decreaseCounter(){
    writeSession(counter, readSession(counter) - 1);
  }
}

session{
  byte counter;
}
```

Listing 5.42: Server in der GWT-DSL

Im obigen Listing 5.42 ist ein Server zu sehen, der zwei Web-Services enthält. Jeder Client-Session ist eine Variable namens *counter* zugeordnet. Sie kann durch Aufruf von *increaseCounter()* bzw. *decreaseCounter* inkrementiert oder dekrementiert werden. Die nächsten beiden Beispiele verdeutlichen die Transformation in die CFIL:

```
class Server{
  Session session;

  void increaseCounter(){
    session.counter = session.counter + 1;
  }

  void decreaseCounter(){
    session.counter = session.counter - 1;
  }
}

class Session{
  byte counter;
}
```

Listing 5.43: Server in der CFIL

Das obige Listing 5.43 zeigt das Ergebnis der CFIL-Transformation für *Server* und *Session*. Jedes Element wird in eine Klasse überführt. Der *Server* enthält eine Referenz auf das Session-Objekt, damit *increaseCounter()* und *decreaseCounter()* darauf zugreifen können. Das nächste Beispiel zeigt den dazugehörigen Client-Driver:

```
1  thread ClientDriver entry verify{
2    Server server;
3    Random random;
4    byte webservice;
5
6    statementblock verify{
7      while(true){
8        webservice = random.getWebService();
9        if(webservice==0){
10         server.increaseCounter();
11       }
12       else if(webservice==1){
13         server.decreaseCounter();
14       }
15     }
16   }
17 }
18
19 class Random{
20   interface range(0,1) getWebService(){}
21 }
```

Listing 5.44: Client-Thread in der CFIL

Das obige Listing 5.44 enthält einen Client-Driver, den der Transformator zur Verifikation des Modells generiert. Die Klasse *Random* ist notwendig, um einen zufälligen Web-Service auszuwählen. Der Thread besteht aus einer Endlosschleife. Sie ruft mit jedem Durchlauf einen zufälligen Web-Service auf. Das Verhalten des Modells entspricht somit genau dem Algorithmus, der im vorangegangenen Abschnitt im Rahmen der Server-Transformation vorgestellt wird.

Transformation des Clients

Die CFIL-Transformation besteht aus zwei Schritten. Der letzte Abschnitt hat die Übersetzung des Servers vorgestellt. Daher zeigt dieses Unterkapitel die Transformation des Clients. Gemäß Abbildung 5.14 müssen der Client und die darin enthaltenen GUI-Elemente in eine Menge von Klassen überführt werden. Daraus ergibt sich für dieses Unterkapitel die folgende Gliederung: Zunächst wird die Übersetzung der GUI-Elemente diskutiert. Darauf folgt die Transformation der einzelnen HTML-Seiten. Zum Abschluss wird ein sogenannter Client-Driver in Form eines Threads generiert, der das Verhalten des Anwenders im Rahmen der formalen Verifikation mit Spin simuliert.

Die GWT-DSL unterstützt gemäß Listing 5.31 die folgenden GUI-Elemente: *Horizontal*, *Vertical*, *Label*, *Textbox* und *Button*. *Horizontal* und *Vertical* beeinflussen nicht die Semantik des Modells, sondern werden vom Google Web Toolkit benötigt, um die GUI-Elemente auf einer HTML-Seite anzuordnen. Es bietet sich daher an, die folgende Klasse zu generieren:

```
1 class Horizontal{
2   interface void addButton(Button b){}
3   interface void addLabel(Label l){}
4   interface void addTextBox(TextBox t){ }
5   interface void addHorizontal(){Horizontal h}
6   interface void addVertical(){Vertical h}
7 }
```

Listing 5.45: Transformation der Horizontal-Umgebungen

Das obige Listing 5.45 zeigt, dass der CFIL-Transformator die Horizontal-Umgebungen in Objekte vom Typ *Horizontal* überführt. Sie enthalten fünf Methoden, um GUI-Elemente wie *Button* oder *Label* hinzufügen zu können. Das Verhalten entspricht somit genau der Semantik der GWT-DSL, die das horizontale Anordnen von beliebigen GUI-Elementen ermöglicht.

Diese Art der Transformation bietet den folgenden Vorteil: Die Methoden von *Horizontal* sind mit eigenschaftserhaltender Abstraktion umgesetzt und haben somit keinen Einfluss auf die Größe des Zustandsraums im Rahmen der formalen Verifikation mit Spin. Trotzdem sind die Horizontal-Umgebungen ein Teil des Modells und können somit vom Google Web Toolkit beim Generieren der Web-Anwendung berücksichtigt werden. Neben *Horizontal* erzeugt der Transformator auch eine Vertical-Klasse. Sie enthält die selben Methoden, die in Listing 5.45 zu sehen sind und ermöglicht das vertikale Anordnen von GUI-Elementen.

Neben *Horizontal* und *Vertical* sind auch die sogenannten *Label* ein Teil der GWT-DSL. Das nächste Beispiel verdeutlicht ihre Übersetzung in die CFIL:

```
1 class Label{
2   interface void setLabel(String s){}
3 }
```

Listing 5.46: Transformation von Label

Das obige Listing 5.46 zeigt, dass der CFIL-Transformator die Klasse *Label* generiert. Instanzen von *Label* ermöglichen das Modellieren von Beschriftungen innerhalb einer Web-Applikation. Die Methode in Zeile 2 ist mit eigenschaftserhaltender Abstraktion umgesetzt. Diese Art der Implementierung wird aus dem selben Grund gewählt, wie bei den bereits vorgestellten Horizontal- bzw. Vertikal-Umgebungen: Beschriftungen haben keinen Einfluss auf die Semantik des Modells. Trotzdem müssen *Label* in

die CFIL überführt werden, damit sie vom GWT im Rahmen der Java-Transformation berücksichtigt werden können.

Die GWT-DSL ermöglicht innerhalb einer Web-Seite nicht nur das Modellieren von Beschriftungen. Auch Texteingaben seitens des Anwenders sind möglich. Zu diesem Zweck kann das GUI-Element *Textbox* Verwendung finden. Das folgende Beispiel verdeutlicht, wie die entsprechenden Textfelder in die CFIL überführt werden können:

```
1 class Textbox{
2   interface String getText(){}
3 }
```

Listing 5.47: Transformation von Texteingabefeldern

Das obige Listing 5.47 zeigt, dass der Transformator für Textfelder die Klasse *Textbox* generiert. Sie beinhaltet eine Methode namens *getText()*, die mit eigenschaftserhaltender Abstraktion umgesetzt ist. *GetText()* liefert den Inhalt des Textfelds als Zeichenkette zurück. Die Nutzung von eigenschaftserhaltender Abstraktion hat den Vorteil, dass der Model Checker nach der Promela-Transformation alle möglichen Rückgabewerte der Methode in Betracht zieht. Somit wird das Modell für alle möglichen Benutzereingaben verifiziert.

Neben den Textfeldern gibt es in der GWT-DSL auch die Möglichkeit, Schalter zu modellieren. Zu diesem Zweck steht das GUI-Element *Button* zur Verfügung. Es kapselt eine Menge von Statements, die ausgeführt werden, wenn der Anwender den Schalter betätigt. Das nächste Beispiel verdeutlicht, wie der Transformator einen *Button* in die CFIL überführt:

```
1 class MyButton{
2   interface void setLabel(String s){}
3   void behaviour(){
4     //..
5   }
6 }
```

Listing 5.48: Transformation von Schaltern

Das obige Listing 5.48 zeigt, dass der Transformator für jeden *Button* eine eigene Klasse generiert. In Zeile 2 ist eine Methode zu sehen, die eine Zeichenkette übergeben bekommt. Sie modifiziert zur Laufzeit die Beschriftung des Schalters. Zeile 3 zeigt die Methode *behaviour()*. In ihr sind alle Statements enthalten (vgl. Listing 5.32), die ausgeführt werden, wenn der Anwender den Schalter betätigt.

Diese Art der Übersetzung begründet sich wie folgt: Die Beschriftung des Schalters hat keine Auswirkung auf das Verhalten des Modells. Daher kann sie mittels eigenschaftserhaltender Abstraktion umgesetzt werden. Im Gegensatz dazu beeinflusst das Betätigen

des Schalters das Verhalten einer Web-Anwendung. Für *Behaviour()* wird daher keine eigenschaftserhaltende Abstraktion verwendet.

Im Rahmen dieses Abschnitts wird gezeigt, wie GUI-Elemente in die CFIL überführt werden können. Jedes GUI-Element ist Teil einer HTML-Seite. Daher muss im weiteren Verlauf der Arbeit demonstriert werden, wie der Transformator die HTML-Seiten in die CFIL übersetzt und die entsprechenden GUI-Elemente integriert. Dafür kann der folgende Algorithmus Verwendung finden: Zunächst werden die Klassen *Horizontal*, *Vertical*, *Label* und *Textbox* zum Abstract Syntax Tree hinzugefügt. Danach iteriert der Transformator über den AST und sucht nach Objekten vom Typ *GWTPage* (vgl. Listing 5.29). Jedes gefundene Objekt wird aus dem AST entfernt und stattdessen eine gleichnamige Klasse erzeugt. Sie hat den folgenden Aufbau:

1. Generiere für jeden Schalter, der in *GWTPage* enthalten ist, eine gleichnamige Klasse. Nutze dafür den Algorithmus aus Listing 5.48.

2. Instanziiere für jedes GUI-Element, das ein Teil von *GWTPage* ist, ein entsprechendes Objekt. Somit wird beispielsweise ein Objekt vom Typ *Label* erzeugt, wenn die HTML-Seite eine Beschriftung enthält.

3. GWTPage-Objekte können OnLoad-Umgebungen enthalten. Sie kapseln Statements, die beim Laden der HTML-Seite ausgeführt werden. Der Transformator erzeugt deshalb eine Methode namens *onLoad()*. Sie besteht aus zwei Abschnitten: Zunächst werden die Objekte zur Abbildung der GUI-Elemente initialisiert. Darauf folgen die Statements aus der OnLoad-Umgebung.

Unter Verwendung des beschriebenen Verfahrens ersetzt der Transformator alle GUI- und GWTPage-Elemente durch Sprachkonstrukte der CFIL. Der Vorteil des Ansatzes ist, dass das erzeugte CFIL-Modell durch Methodenaufrufe gesteuert werden kann. Das bedeutet, wenn der Anwender eine bestimmte HTML-Seite besucht und einen Schalter betätigt, wird dies durch den Aufruf von *onLoad()* bzw. *behaviour()* abgebildet. Das folgende Beispiel verdeutlicht den Transformationsansatz:

```
1  client{
2    page Counter{
3      int i;
4      button{
5        label="increase"
6        behaviour{
7          i =i+1;
8        }
9      }
10     onLoad{
```

```
11          i=0;
12        }
13    }
14 }
```

Listing 5.49: HTML-Seite mit einem Counter

Das obige Listing 5.49 ist ein Modell der GWT-DSL, das aus einer HTML-Seite besteht. Sie enthält eine Variable, die beim Laden mit dem Wert 0 initialisiert wird. Des Weiteren ist auch ein Schalter Teil des Modells. Wenn ein Nutzer den Schalter betätigt, wird i um 1 inkrementiert. Das nächste Beispiel zeigt, wie der Transformator das Modell aus Listing 5.49 unter Verwendung des beschriebenen Algorithmus in die CFIL überführt:

```
1  class Counter{
2    Button button;
3    int i;
4
5    void onLoad(){
6      button.setLabel("increase");
7      i=0;
8    }
9
10   //encapsulated class
11   class Button{
12     interface void setLabel(String s){}
13     behaviour(){
14       i=i+1;
15     }
16   }
17 }
```

Listing 5.50: Transformation des Counters in die CFIL

Das obige Listing 5.50 demonstriert den Übersetzungsalgorithmus für GWTPage-Elemente. Es wird deutlich, dass die HTML-Seite und der darin gekapselte Schalter in die Klassen *Counter* bzw. *Button* überführt werden. In *Counter* ist sowohl eine Instanz von *Button*, als auch eine OnLoad-Methode enthalten, die das Modell initialisiert. Um das Betätigen des Schalters abzubilden, stellt *Button* die Methode *behaviour()* zur Verfügung. Durch diese Art der Transformation sind die folgenden Aspekte gewährleistet:

- Alle Elemente des Abstract Syntax Trees, die nicht Teil des DVF sind, werden durch Klassen und Methoden ersetzt. Der Transformator des DSL Verification Frameworks kann das Modell somit einlesen und nach Java bzw. Promela überführen.

- Das Verhalten des Modells wird durch das Aufrufen von *onLoad()* und *behaviour()* gesteuert. *OnLoad()* initialisiert die Variable i mit dem Wert 0 und entspricht dem Neuladen der Seite in einem Web-Browser. *Behaviour()* inkrementiert i um 1 und repräsentiert das Betätigen des Schalters. Das generierte Modell hat somit die selbe Semantik wie Listing 5.49.

Gemäß Abbildung 5.14 muss der Transformator, neben den HTML-Seiten und GUI-Elementen, auch einen sogenannten User-Driver generieren. Dieser simuliert das Verhalten eines Anwenders, der auf die Web-Anwendung zugreift. Dadurch kann Spin das Modell für alle möglichen Nutzerinteraktionen verifizieren. Für das Erzeugen des User-Drivers findet der folgende Algorithmus Verwendung:

1. Erzeuge einen CFIL-Thread und integriere ihn in den AST.
2. Füge zum Thread eine While-Schleife hinzu.
3. Füge zur While-Schleife einen nicht-deterministischen If-Block hinzu. In jeder Verzweigung des If-Blocks wird ein GUI-Element genutzt oder eine HTML-Seite aufgerufen.

Das folgende Beispiel verdeutlicht das Generieren des User-Drivers für das Modell aus Listing 5.50:

```
1  thread UserDriver entry verify{
2    Counter counter;
3    Random random;
4    byte useraction;
5
6    statementblock verify{
7      while(true){
8        useraction = random.getUserAction();
9        if(useraction==0){
10         counter.onLoad()
11       }
12       else if(useraction==1){
13         counter.button.behaviour();
14       }
15     }
16   }
17 }
18 class Random{
19   interface range(0,1) getUserAction(){}
20 }
```

Listing 5.51: Generieren des User-Drivers

Das obige Listing 5.51 zeigt einen Thread, der den User-Driver repräsentiert und aus einer Endlosschleife besteht. Mit jeder Iteration der Schleife wird eine zufällige Aktion des Nutzers ausgewählt. Das Modell aus Listing 5.49 besteht aus einer HTML-Seite und einem Schalter. Daher kann der Nutzer lediglich zwei Aktionen ausführen: Den Schalter betätigen oder die Seite neu laden. Diese Interaktionsmöglichkeiten werden in den Zeilen 10 und 13 umgesetzt. Nach dem Generieren des User-Drivers ist die CFIL-Transformation abgeschlossen.

5.2.4 Umsetzung des Modells und Verifikation

Der vorangegangene Abschnitt stellt die InTune-Fallstudie vor. Sie dient zum Testen von verteilten Systemen. Der Zugriff auf InTune erfolgt mittels einer Web-Anwendung. Um sie zu implementieren, wird in Unterkapitel 5.2.2 eine domänenspezifische Sprache namens GWT-DSL vorgestellt. Die Umsetzung der domänenspezifischen Sprache erfolgte unter Zuhilfenahme des DSL Verification Frameworks (DVF). Es ermöglicht die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation. Zu diesem Zweck müssen Modelle in die CFIL überführt werden, damit das DVF im Anschluss Java- bzw. Promela-Code generieren kann. Abschnitt 5.2.3 stellte einen entsprechenden Transformationsalgorithmus vor.

Um die zweite Fallstudie abzuschließen, muss InTune mit der GWT-DSL modelliert, verifiziert und nach Java überführt werden. Alle drei Aspekte sind Gegenstand dieses Unterkapitels. Es gliedert sich wie folgt: Zunächst wird die Web-Anwendung unter Zuhilfenahme eines Aktivitätsdiagramms beschrieben. Danach erfolgt die Umsetzung mit der GWT-DSL. Das so entstandene Modell wird nach Promela überführt. Im Anschluss verifiziert Spin, dass alle Anforderungen erfüllt sind. Sofern dies nicht der Fall ist, wird das GWT-DSL-Modell entsprechend angepasst und daraus erneut Promela-Code generiert. Nachdem Spin keine weiteren Fehler bzw. nicht erfüllte Anforderungen findet, erfolgt die Transformation in die Hochsprache Java.

Modellierung von InTune

InTune ermöglicht es einem Nutzer, Testfälle zu verwalten bzw. auszuführen. Daher soll die Web-Anwendung die folgenden Anwendungsfälle berücksichtigen:

- Jeder Nutzer betritt zunächst eine Login-Seite. Dort spezifiziert er einen Benutzernamen und ein Passwort, um sich zu authentifizieren.
- Sofern Benutzername und Passwort korrekt sind, ruft InTune aus einer Datenbank alle Testfälle ab, die dem Anwender zur Verfügung stehen.
- Dem Nutzer werden auf einer zweiten HTML-Seite alle Testfälle angezeigt. Er kann sie starten oder die Anwendung mit einem Logout-Button beenden.

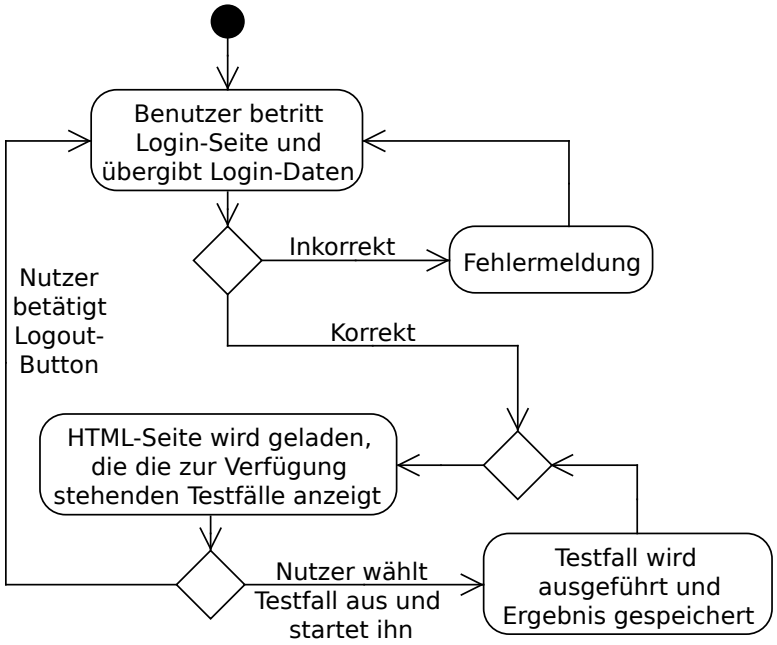

Abbildung 5.15: Nutzung von InTune (Aktivitätsdiagramm)

Bevor eine derartige Anwendung mit der GWT-DSL implementiert werden kann, empfiehlt es sich, das System zu visualisieren. Dafür findet ein Aktivitätsdiagramm Verwendung, das in Abbildung 5.15 zu sehen ist. Es zeigt die Funktionalität der Web-Anwendung. Ein Anwender betritt zunächst die Login-Seite. Diese fordert ihn auf, Nutzername und Passwort zu spezifizieren. Nach Eingabe der Daten werden sie an den Web-Server übertragen. Er überprüft, ob das Tupel korrekt ist. Dazu wird eine entsprechende Anfrage an eine Datenbank gestellt. Sofern Benutzername und Passwort ungültig sind, erfolgt die Ausgabe einer Fehlermeldung. Im Falle einer korrekten Authentifizierung wird eine zweite Seite geladen, auf der alle Testfälle des Anwenders zu sehen sind. Ihm stehen zwei verschiedene Aktionen zur Verfügung:

- Es wird ein Testfall ausgewählt und gestartet.
- Der Anwender beendet die Sitzung und gelangt wieder zur Login-Seite.

Bei der Implementierung der Web-Anwendung ist zu beachten, dass einige Aspekte nicht mit der GWT-DSL modelliert werden können. Hierzu gehört beispielsweise die Datenbankanbindung. In diesem Fall muss eigenschaftserhaltende Abstraktion Verwendung finden, da die GWT-DSL nur das Beschreiben von Web-Anwendungen ermöglicht.

Die Web-Anwendung besteht gemäß Abbildung 5.15 aus zwei HTML-Seiten. Sie werden als Login-Page und ListTestCases-Page bezeichnet. Die Login-Page ermöglicht die Nutzerauthentifizierung. Die Anzeige bzw. Auswahl der verfügbaren Testfälle ist Aufgabe der ListTestCases-Page. Daneben gibt es auch einen Server, der an eine Datenbank angebunden ist und die folgenden Web-Services zur Verfügung stellt:

- Login: Der Client übergibt Nutzername und Passwort, um sich zu authentifizieren. Der Web-Server speichert das Ergebnis des Login-Vorgangs in einer Session-Variable.
- Logout: Der Client teilt dem Server mit, dass er die InTune-Sitzung beendet.
- GetTestCases: Der Web-Server liefert dem Client eine Liste von verfügbaren Testfällen zurück. Diese werden dem Nutzer auf der ListTestCases-Page angezeigt.
- StartTestCase: Der Client übergibt eine Testfall-ID. Der Web-Server führt den entsprechenden Testfall aus.

Die HTML-Seiten und Web-Services ermöglichen die Umsetzung aller Anwendungsfälle, die zu Beginn dieses Abschnitts genannt sind. Das folgende Listing verdeutlicht die Modellierung der Login-Page:

```
1  page Login{
2    String user;
3    String pass;
4
5    vertical{
6      horizontal{
7        label{"login: "},
8        textbox username{}
9      }
10     horizontal{
11       label{"password: "},
12       textbox password{}
13     }
14     button{ label="submit"
15       behaviour{
16         user = username.getText();
17         pass = password.getText();
18         server.login(user, pass, login_request);
19       }
20     }
21   }
```

```
22
23   callback login_request{
24     onFailure(){ }
25     onSuccess(){ loadPage(ListTestCases); }
26   }
27 }
```

Listing 5.52: HTML-Seite für Nutzerauthentifizierung

Das obige Modell 5.52 zeigt die Umsetzung der HTML-Seite für die Authentifizierung des Anwenders. Sie besteht aus GUI-Elementen zur Eingabe von Nutzername und Passwort. Die entsprechenden Textfelder sind in den Zeilen 8 und 12 zu sehen. Nutzername und Passwort müssen an den Web-Server übertragen werden. Zu diesem Zweck steht in Zeile 13 ein Schalter zur Verfügung. Wenn der Anwender ihn betätigt, wird der Inhalt der beiden Textfelder in den Variablen *user* und *pass* gespeichert. Danach erfolgt der Aufruf des Web-Services *login()*, um sie zur Überprüfung an den Server zu senden. Das Callback-Objekt *login_request* ruft die Seite *ListTestCases* auf, sobald der Web-Service terminiert:

```
1  page ListTestCases{
2    int testcase_id;
3
4    onLoad{
5      server.getTestCases(get_tc_request);
6    }
7
8    vertical{
9      label testcase_list{""},
10     horizontal{
11       label{"testcase number:"}
12       textbox testcase_enter{},
13       button{
14         label="start",
15         behaviour{
16           testcase_id = testcase_enter.getText();
17           server.startTestCase(testcase_id, start_tc_request);
18         }
19       }
20       button{
21         label="logout",
22         behaviour{
23           server.logout(logout_request);
24         }
```

```
25          }
26        }
27    }
28 }
```

Listing 5.53: HTML-Seite für die Auswahl der Testfälle

Das obige Listing 5.53 zeigt die Umsetzung der HTML-Seite *ListTestCases*. Beim Laden wird zunächst der Web-Service *getTestCases()* aufgerufen. Er liefert alle verfügbaren Testfälle als Zeichenkette zurück. Das dazugehörige Callback-Objekt ist nicht Teil des Modells. Es gibt die Testfälle mit dem GUI-Element *testcase_list* (vgl. Zeile 9) aus, wenn *getTestCases()* terminiert. Das bedeutet, nach dem Laden einer HTML-Seite *ListTestCases* sind darauf alle verfügbaren Testfälle des Anwenders zu sehen.

Jeder Testfall besteht aus einer Identifikationsnummer und einem Bezeichner. Der Anwender kann den Testfall ausführen, indem er die Identifikationsnummer in ein Textfeld eingibt und einen Schalter betätigt. Schalter und Textfeld sind in den Zeilen 12 bis 19 zu sehen. Durch das Benutzen des Schalters wird der Web-Service *startTestCase()* aufgerufen. Er überträgt die Identifikationsnummer des Testfalls an den Server, der im Anschluss mit dessen Ausführung beginnt. Alternativ kann der Anwender die Sitzung durch den Logout-Schalter in Zeile 20 beenden.

Im bisherigen Verlauf dieses Unterkapitels sind die beiden HTML-Seiten der Web-Anwendung vorgestellt. Sie rufen vier verschiedene Web-Services auf. Die konkrete Implementierung der Web-Services demonstrieren die folgenden beiden Modelle:

```
1  server{
2    int user_amount = 0;
3    int max_user = 4;
4
5    void login(String u, String p){
6      if(user_amount < max_user){
7        writeSession(login_success,
8          DBInterface.loginDB(u,p));
9        if(readSession(login_success)){
10         user_amount = user_amount + 1;
11       }
12     }
13   }
14 }
15
16 session{
17   boolean login_success;
18 }
```

```
19
20 class DBInterface{
21   interface boolean loginDB(String u, String pass){}
22 }
```

Listing 5.54: Web-Service für den Login

Das obige Listing 5.54 zeigt den Web-Service *login()*. Ein Client kann ihn aufrufen, um sich zu authentifizieren. Der Web-Service hat die folgende Semantik: Die zwei Parameter u bzw. p in Zeile 4 repräsentieren Nutzername und Passwort. Der Web-Service vergleicht sie mit einer Datenbank, die Nutzerdaten bereitstellt. Die Umsetzung von Datenbanken ist nicht Aufgabe der GWT-DSL. Deshalb wird eine entsprechende Schnittstelle mit eigenschaftserhaltender Abstraktion umgesetzt. Sie ist in Zeile 20 bis 22 zu sehen und enthält eine Methode namens *loginDB()*. *LoginDB()* liefert *true* zurück, wenn Nutzername und Passwort mit dem Inhalt der Datenbank übereinstimmen.

Der Web-Service ruft *loginDB()* in Zeile 8 auf und speichert das Ergebnis in einer Session-Variable. Der Vorteil ist, dass andere Web-Services, wie beispielsweise *getTest-Cases()*, zu einem späteren Zeitpunkt überprüfen können, ob der Client sich korrekt mit dem System authentifiziert hat und somit berechtigt ist, eine Liste der verfügbaren Testfälle einzusehen.

In Zeile 2 und 3 sind zwei Variablen zu sehen, die der Erklärung bedürfen. Sie haben die folgende Semantik: InTune wird auf einem Server ausgeführt, der über eine begrenzte Anzahl von Ressourcen, wie beispielsweise Hauptspeicher oder CPU-Kerne, verfügt. Somit ist es wünschenswert, die maximale Anzahl von Anwendern, die das System parallel nutzen, zu begrenzen. Daher werden die beiden Variablen wie folgt genutzt:

- User_amount: Die Anzahl der Nutzer, die sich mit dem System authentifiziert haben.
- Max_user: Die maximale Anzahl von Nutzern, die sich gleichzeitig mit InTune verbinden dürfen. Sie ist im Rahmen der Fallstudie exemplarisch auf 4 gesetzt.

Somit überprüft der Web-Service in Zeile 6, ob die aktuelle Anzahl von Nutzern den Maximalwert in *max_user* übersteigt. Des Weiteren wird *user_amount* nach einer erfolgreichen Authentifizierung um 1 inkrementiert.

Neben *login()* gibt es auch einen Web-Service namens *logout()*. Durch den Aufruf von *logout()* kann ein Client seine InTune Sitzung beenden. Er muss somit die folgenden Aktionen durchführen:

- Setze *login_success* in der Session-Variable auf *false*.
- Dekrementiere *user_amount* um 1.

Im bisherigen Verlauf dieses Abschnitts wurden zwei Web-Services vorgestellt, mit denen der Client eine Sitzung starten oder beenden kann.

InTune ermöglicht, neben dem Login bzw. Logout, auch das Einsehen und Ausführen von Testfällen. Das folgende Modell stellt die entsprechenden Web-Services genauer vor:

```
1  server{
2    String getTestCases(){
3      assert(user_amount <= max_user);
4
5      if(readSession(login_success)){
6        return DBInterface.getTestCases();
7      }
8      else{
9        return "please login first";
10     }
11   }
12 }
13
14 class DBInterface{
15   interface String getTestCases();
16 }
```

Listing 5.55: Web-Service zum Anzeigen der Testfälle

Das obige Listing 5.55 zeigt die Umsetzung des Web-Services *getTestCases()*. Er liefert alle Testfälle zurück, die ein Client aufrufen kann. Seine Implementierung gestaltet sich wie folgt: Zunächst wird die Session-Variable ausgelesen, um zu ermitteln, ob sich der Anwender erfolgreich mit InTune authentifiziert hat (vgl. Zeile 5). Danach erfolgt die Abfrage der verfügbaren Testfälle. Auch sie sind Teil einer Datenbank. Daher ist die entsprechende Schnittstelle mit eigenschaftserhaltender Abstraktion umgesetzt (vgl. Zeile 14 bis 16).

Eine Besonderheit ist in Zeile 3 zu sehen: Das Ziel dieser Arbeit ist es, die formale Verifikation mit der modellgetriebenen Entwicklung zu verknüpfen. Somit müssen neben dem eigentlichen Modell auch Anforderungen spezifiziert werden. Eine Anforderung an AssyControl ist, dass nur eine beschränkte Anzahl von Nutzern gleichzeitig das System nutzen darf. Um auszuschliessen, dass diese Beschränkung umgegangen werden kann, ist eine entsprechende Assertion in Zeile 3 eingefügt. Durch sie stellt Spin während des Verifikationsprozesses sicher, dass immer gilt $user\_amount \leq max\_user$.

Neben *getTestCases()* ist auch *startTestCase()* ein Teil des Servers. Wenn ein Client *startTestCase()* aufruft, erfolgt die Ausführung des entsprechenden Testfalls. Die Implementierung mit der GWT-DSL gestaltet sich wie folgt:

- Der Client übergibt die Identifikationsnummer des Testfalls als Parameter.

- Der Server überprüft durch Abrufen der Session-Variable, dass der Nutzer sich bei InTune authentifiziert hat.
- Der Server führt den Testfall aus.

Um den Testfall auszuführen, verbindet sich der Web-Server gemäß Abbildung 5.11 mit dem Testnetzwerk. Da derartige Aspekte nicht Teil der GWT-DSL sind, muss die Umsetzung einer entsprechenden Schnittstelle durch eigenschaftserhaltende Abstraktion erfolgen.

Transformation nach Promela

Im bisherigen Verlauf dieses Unterkapitels findet die GWT-DSL Verwendung, um die zweite Fallstudie zu beschreiben. Gemäß des DVF-Workflows (vgl. Abbildung 1.8) muss im Anschluss die formale Verifikation mit Spin erfolgen. Daher wird das InTune-Modell dem Transformator aus Abschnitt 5.2.3 übergeben, der es in die CFIL überführt. Das Ergebnis dient dem DVF als Eingabe, um daraus eine Promela-Beschreibung zu generieren. Das Ziel dieses Abschnitts ist es, die anschließende Verifikation mit Spin genauer vorzustellen. Sie besteht gemäß Abbildung 5.12 aus zwei Schritten: Zunächst stellt Spin sicher, dass der Server alle Anforderungen erfüllt. Danach erfolgt die formale Verifikation des Clients.

Die Verifikation des Servers gestaltet sich wie folgt: Das Ziel ist es sicherzustellen, dass die Anforderung aus Listing 5.55, nämlich die Beschränkung der maximalen Nutzeranzahl, immer erfüllt ist. Der Server besteht gemäß Listung 5.54 bzw. 5.55 aus vier Web-Services. Der CFIL-Transformator generiert somit eine Server-Klasse, die vier Methoden enthält. Des Weiteren wird ein Client-Driver erzeugt, der das Verhalten des Clients simuliert. Nach der Promela-Transformation beginnt die Verifikation mit Spin. Der Verifier gestaltet sich wie folgt:

```
1 pan:1: assertion violated (user_amount<=max_user) (at depth 41)
2 pan: wrote intune_server_with_assert.pml.trail
3 ...
4
5 Stats on memory usage (in Megabytes):
6    20.451        equivalent memory usage for states
7    (stored*(State-vector + overhead))
8    69.806        actual memory usage for states
9   128.000        memory used for hash table (-w24)
10    0.002        memory used for DFS stack (-m41)
11  197.715        total actual memory usage
12 ...
13
```

```
14 pan: elapsed time 0.03 seconds
15 pan: rate 212433.33 states/second
```

Listing 5.56: Ausgabe der Verifiers für den Server

Das obige Listing 5.56 zeigt die Ausgabe des Verifiers. Zeile 1 bis 2 verdeutlichen, dass der Verifier eine nicht erfüllte Anforderung gefunden hat. Dabei handelt es sich um die Assert-Anweisung mit der Expression $user\_amount \leq max\_user$. Somit ist es möglich, dass die beschränkte Anzahl der Clients aufgrund eines Fehlers im Modell umgangen werden kann. Die Auswertung des vom Verifier erzeugten Fehlerpfads führte zu dem Ergebnis, dass die verletzte Anforderung durch den nebenläufigen Aufruf des Web-Services *login()* entsteht. Das folgende Modell zeigt dessen Umsetzung im Rahmen der zweiten Fallstudie:

```
1  byte user_amount = 0;
2  byte max_user = 4;
3
4  void login(String u, String p){
5    if(user_amount < max_user){
6      writeSession(login_success,
7        DBInterface.loginDB(u,p));
8      if(readSession(login_success)){
9        user_amount = user_amount + 1;
10     }
11   }
12 }
```

Listing 5.57: Web-Service für den Login

Zu Beginn des Fehlerpfads wird davon ausgegangen, dass sich noch kein Nutzer erfolgreich mit dem System authentifiziert hat und *user_amount* den Wert 0 enthält. Somit kann die folgende Situation eintreten: Fünf Clients rufen *login()* nahezu zeitgleich auf. Es wird der Ausdruck in Zeile 5 ausgewertet. Da sich noch kein Client erfolgreich authentifiziert hat, liefert die Expression den Wert *true* zurück. Anschließend erfolgt in Zeile 7 der Vergleich mit einer angebundenen Datenbank. Es wird davon ausgegangen, dass jeder Nutzer gültige Authentifizierungsdaten übergeben hat. Danach setzen alle fünf Clients die Session-Variable auf *true* und inkrementieren *user_amount*. Erst zu diesem Zeitpunkt kann sich kein weiterer Client authentifizieren, da *user_amount* den Wert von *max_user* übersteigt. Da aber bereits fünf Nutzer den Login-Prozess erfolgreich durchlaufen haben, ist die Anforderung $user\_amount \leq max\_user$ nicht erfüllt und der Verifier meldet einen Fehler.

Mehrere nebenläufige Prozesse teilen sich somit eine Ressource und greifen lesend bzw. schreibend darauf zu. Das Modell muss deshalb so angepasst werden, dass im-

mer nur ein nebenläufiger Client die Expression in Zeile 5 auswertet und anschließend *user_amount* inkrementiert. Dafür bietet sich die Synchronize-Umgebung an. Sie wird in Abschnitt 4.3.4 vorgestellt und ist Teil des DSL Verification Frameworks. Jede Synchronize-Umgebung kann nur von einem nebenläufigen Prozess gleichzeitig betreten werden. Daher wird das Modell aus Listing 5.57 wie folgt modifiziert:

```
1 void login(String u, String p){
2   sync{
3     if(user_amount < max_user){
4       writeSession(login_success,
5         DBInterface.loginDB(u,p));
6       if(readSession(login_success)){
7         user_amount = user_amount + 1;
8       }
9     }
10  }
11 }
```

Listing 5.58: Web-Service für den Login mit Synchronize-Umgebung

Das obige Listing 5.58 zeigt das angepasste Modell. In Zeile 2 und 10 ist eine Synchronize-Umgebung eingefügt. Somit kann *login()* lediglich sequentiell aufgerufen werden. Nach dem Anpassen des Modells ist eine erneute Verifikation notwendig, um sicherzustellen, dass keine weiteren Fehler enthalten sind. Die Ausgabe des Verifiers gestaltet sich wie folgt:

```
1 (Spin Version 6.2.3 -- 24 October 2012)
2         + Partial Order Reduction
3
4 ...
5
6 Stats on memory usage (in Megabytes):
7  1337.564       equivalent memory usage for states
8                 (stored*(State-vector + overhead))
9  1011.632       actual memory usage for states
10                (compression: 75.63%)
11                state-vector as stored =
12                69 byte + 28 byte overhead
13   128.000      memory used for hash table (-w24)
14   534.058      memory used for DFS stack (-m10000000)
15  1673.093      total actual memory usage
16
17 ...
```

```
18
19 pan: elapsed time 53.6 seconds
20 pan: rate 204503.99 states/second
```

Listing 5.59: Ausgabe des Verifiers für den modifizierten Server

Das obige Listing 5.59 zeigt, dass Spin 1.6 Gigabyte RAM benötigte, um das Modell im Hauptspeicher abzubilden (vgl. Zeile 15). Es werden keine weiteren nicht-erfüllten Anforderungen gefunden. Daher kann in einem zweiten Schritt die Verifikation des Clients erfolgen.

Der Client besteht aus zwei HTML-Seiten: *Login* und *ListTestCases* (vgl. Listing 5.52 und 5.53). Der CFIL-Transformator generiert daraus zwei Klassen, die GUI-Elemente und OnLoad-Methoden enthalten. Des Weiteren wird ein Thread namens User-Driver erzeugt, der die Benutzerinteraktion simuliert. Er besteht aus einer Endlosschleife, die nicht-deterministisch eine der folgenden Aktionen ausführt:

- Lade die Login-Seite durch Aufruf der OnLoad-Methode.
- Erzeuge zwei zufällige Zeichenketten, die Nutzername und Passwort repräsentieren. Betätige anschließend den Login-Schalter.
- Lade die ListTestCases-Seite durch Aufruf der OnLoad-Methode.
- Betätige den Start-Schalter.
- Betätige den Logout-Schalter.

Das CFIL-Modell wird vom DVF nach Promela überführt und anschließend mit Spin verifiziert. Die Beschreibung des Clients enthält keine Anforderungen in Form von Assert-Statements. Daher untersucht der Model Checker, ob Deadlocks enthalten sind oder Zugriffe über Array-Grenzen hinweg existieren. Das Ergebnis des Verifikationsprozesses ist im folgenden Listing zu sehen. Die Ausgaben von Spin sind leicht gekürzt, um die Lesbarkeit zu verbessern:

```
1 (Spin Version 6.2.3 -- 24 October 2012)
2         + Partial Order Reduction
3 ...
4
5 State-vector 36 byte, depth reached 33345, errors: 0
6    248706 states, stored
7    192844 states, matched
8    441550 transitions (= stored+matched)
9         0 atomic steps
10 hash conflicts:        482 (resolved)
```

```
11
12 Stats on memory usage (in Megabytes):
13    15.180        equivalent memory usage for states
14                  (stored*(State-vector + overhead))
15    13.546        actual memory usage for states
16                  (compression: 89.24%)
17                  state-vector as stored =
18                  29 byte + 28 byte overhead
19   128.000        memory used for hash table (-w24)
20    53.406        memory used for DFS stack (-m1000000)
21   194.882        total actual memory usage
22
23
24 unreached in proctype UserDriver
25         intune_client.pml:237, state 157, "-end-"
26         (1 of 157 states)
27
28 pan: elapsed time 0.23 seconds
29 pan: rate 1081330.4 states/second
```

Listing 5.60: Ausgabe des Verifiers für den Client

Das obige Listing 5.60 zeigt, dass der Model Checker keine nicht-erfüllten Anforderungen findet. Spin benötigt ca. 195 Megabyte, um den gesamten Zustandsraum des Modells im Hauptspeicher abzubilden (vgl. Zeile 21). Des Weiteren kann der Endzustand des Client-Drivers nicht erreicht werden (vgl. Zeile 24 bis 26). Dieses Verhalten ist jedoch zu erwarten, da der User-Driver eine Endlosschleife ausführt und somit nicht terminiert. Da Server und Client verifiziert sind, kann das InTune-Modell in einem finalen Schritt nach Java überführt und die zweite Fallstudie abgeschlossen werden.

Transformation nach Java

Der vorangegangene Abschnitt zeigt eine InTune-Modellierung und überführt sie nach Promela. Danach erfolgte die formale Verifikation mit Spin. Sie ergab eine nicht-erfüllte Anforderung. Durch den daraus resultierenden Fehler konnte die beschränkte Anzahl von gleichzeitig authentifizierten Nutzern umgangen werden. Um diesem Problem zu begegnen, erweitert der DSL-Anwender das Modell mit einer Synchronize-Umgebung. Im Rahmen einer erneuten Verifikation findet Spin keine weiteren, nicht-erfüllten Anforderungen. Daher ist es das Ziel dieses Unterkapitels, die abschließende Java-Transformation genauer vorzustellen.

InTune wird mit der GWT-DSL umgesetzt. Das entsprechende Modell besteht aus einem Client und einem Server. Der Client repräsentiert HTML-Seiten, die ein Web-

Browser ausführt. Jede HTML-Seite kapselt eine OnLoad-Umgebung und verschiedene GUI-Elemente. Des Weiteren kann sie Web-Services aufrufen, die der Server zur Verfügung stellt.

Das InTune-Modell enthält die HTML-Seiten *Login* und *ListTestCases*. Durch *Login* kann sich ein Nutzer authentifizieren, indem er den gleichnamigen Web-Service aufruft. *ListTestCases* bekommt vom Server eine Auflistung der verfügbaren Testfälle übergeben. Der Nutzer kann sie entweder ausführen oder die Sitzung mittels *logout()* beenden. Das entsprechende Modell ist mit der GWT-DSL beschrieben und in den Listings 5.52 bzw. 5.53 zu sehen. Der Transformator aus Abschnitt 5.2.3 überführte es in die CFIL. Im Anschluss konnte das DSL Verification Framework Java-Quellcode generieren. Das Ergebnis ist in Abbildung 5.16 als Klassendiagramm zu sehen. Die HTML-Seiten *Login* bzw. *ListTestCases* enthalten Referenzen auf GUI-Elemente. Dazu gehören auch Schalter wie beispielsweise *StartButton*, deren Verhalten entsprechende Beaviour-Methoden abbilden. Auch für den Server wird eine Klasse generiert. Sie kapselt die jeweiligen Web-Services in Form von Methoden.

Nach der Java-Transformation kann die zweite Fallstudie abgeschlossen werden. Der DSL-Anwender muss dabei jedoch die folgenden Aspekte beachten: InTune greift auf eine Datenbank zu, die die Nutzerdaten enthält. Die Schnittstellen zur Datenbank werden mit eigenschaftserhaltender Abstraktion beschrieben. Der DSL-Entwickler muss deshalb, um die zweite Fallstudie abzuschließen, eine geeignete Datenbank auswählen und sie in sein Projekt integrieren.

Neben den Schnittstellen muss der DSL-Anwender auch eine Eigenschaft des Google Web Toolkits (GWT) berücksichtigen, das in Abschnitt 2.4 vorgestellt wird: Ein Entwickler nutzt die Hochsprache Java, um mit dem GWT eine Web-Anwendung zu implementieren. Zur Beschreibung von GUI-Elementen, Callback-Objekten, usw. stellt das GWT eigene Klassen zur Verfügung. Somit muss der Entwickler beispielsweise ein Objekt vom Typ *Button* instanziieren, um sein Modell mit einem Schalter zu ergänzen. Abschließend wird der Java-Quellcode dem GWT übergeben und daraus die Web-Anwendung in Form von Servlets, HTML und Javascript generiert.

Bei Betrachtung von InTune in Abbildung 5.16 fällt auf: Das DSL Verification Framework hat Klassen erzeugt, um GUI-Elemente abzubilden. Dabei handelt es sich jedoch nicht um die Komponenten des GWT. Der DSL-Anwender muss das Modell somit modifizieren:

- Alle GUI-Elemente müssen von den entsprechenden GWT-Klassen erben. Das bedeutet beispielsweise, jeder Schalter wird von *Button* abgeleitet.
- Das Verhalten von Schaltern muss mit der Klasse *ClickHandler* beschrieben werden.
- Callback-Objekte müssen von *AsyncCallback* erben.

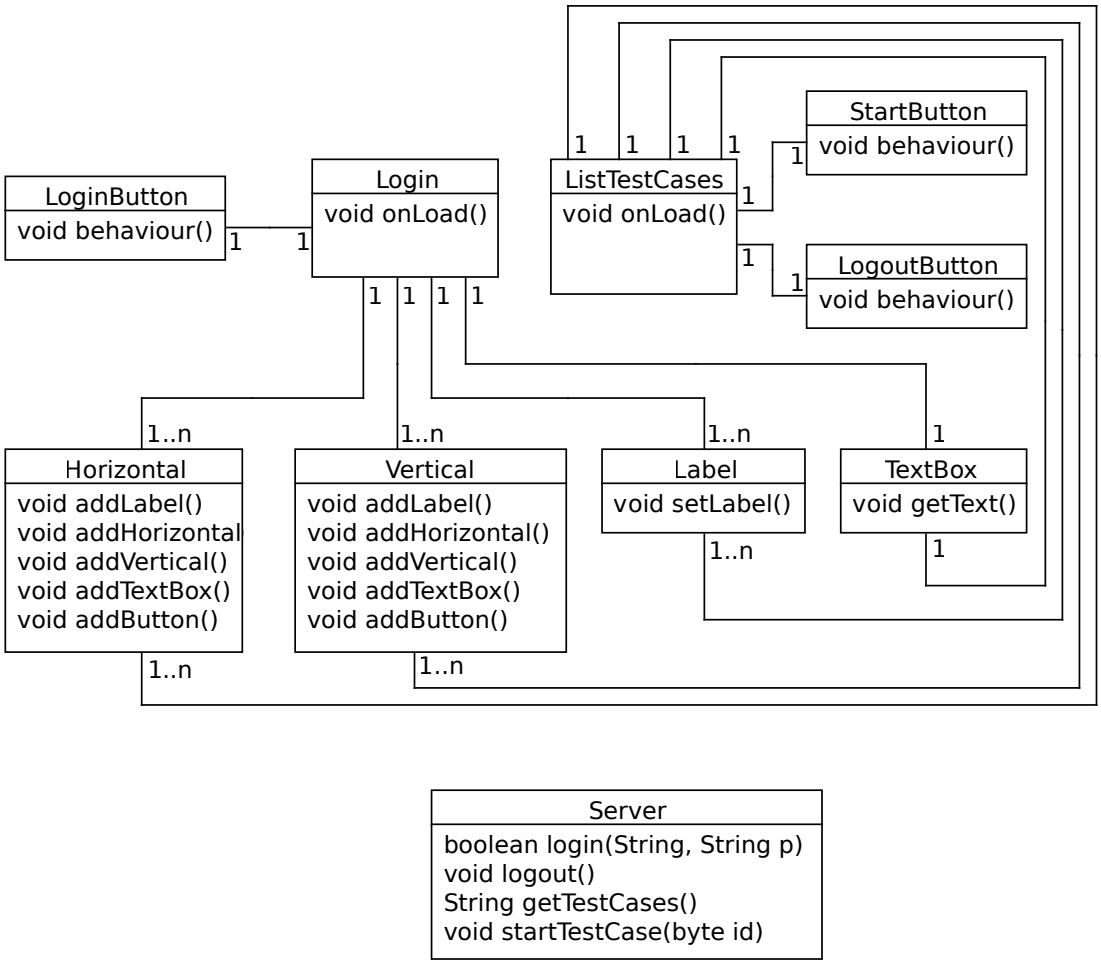

Abbildung 5.16: Ergebnis der Java-Transformation (Klassendiagramm)

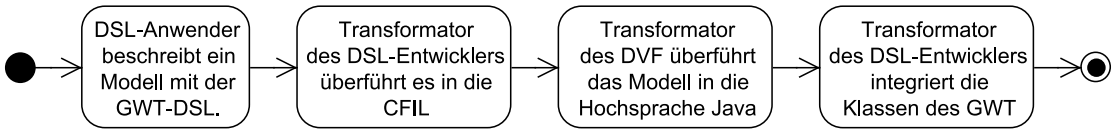

Abbildung 5.17: Erweiterung des DVF für die zweite Fallstudie (Aktivitätsdiagramm)

- Jede Klasse, die eine HTML-Seite darstellt, muss das Interface *EntryPoint* implementieren. *EntryPoint* beinhaltet eine Methode namens *onModuleLoad()*, die beim Laden der Seite ausgeführt wird, was genau der Semantik von *onLoad()* entspricht (vgl. beispielsweise Listing 5.49 Zeile 10).

Die Modifikationen sind notwendig, damit das GWT die Semantik der entsprechenden Komponenten erkennt und sie in eine Web-Anwendung überführt. Ein wesentlicher Bestandteil dieser Arbeit ist jedoch die modellgetriebene Entwicklung, die abstrakte Modelle automatisiert in ausführbare Software überführt. Daher ist es wünschenswert, dass die oben genannten Modifikationen automatisch erfolgen. Eine Erweiterung des CFIL-Transformators sollte vermieden werden, da es sich bei der *Control Flow Intermediate Language* um eine plattform-unabhängige Sprache handelt und das GWT eine konkrete Technologie darstellt. In der zweiten Fallstudie wird deshalb vom DSL-Entwickler ein weiterer Transformator implementiert. Seine Integration in den DVF-Workflow verdeutlicht Abbildung 5.17: Der DSL-Anwender beschreibt ein Modell und nutzt dafür die GWT-DSL. Es wird einem Transformator übergeben, den der DSL-Entwickler umgesetzt hat und der es in die CFIL überführt. Danach erfolgt zusammen mit dem DVF eine Übersetzung in die Hochsprache Java. Der finale Schritt in Abbildung 5.17 repräsentiert die Erweiterung und ergänzt das Java-Modell mit den Klassen des GWT. Eine Diskussion über die Vorteile dieses Ansatzes und möglicher Alternativen erfolgt in Abschnitt 5.2.5, das die Umsetzung von InTune zusammenfasst und die zweite Fallstudie auswertet.

5.2.5 Auswertung

In den vorangegangenen Abschnitten wird die zweite Fallstudie vorgestellt und mit dem DSL Verification Framework umgesetzt. Das Ziel dieses Unterkapitels ist die Zusammenfassung des Vorgehens. Des Weiteren soll auch evaluiert werden, ob die Nutzung des DVF die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation erleichterte. Dabei sind die folgenden Aspekte von zentraler Bedeutung:

- Kann das DVF den Implementierungssaufwand reduzieren?
- Ist durch das DVF weniger Expertenwissen zur Umsetzung der DSL notwendig?

- Konnte der gesamte Zustandsraum des Modells verifiziert werden?
- Ist eine Erweiterung des DVF möglich, wenn der Funktionsumfang für ein DSL-Projekt nicht ausreichend ist?

Die vier Fragestellungen werden im weiteren Verlauf des Abschnitts genauer untersucht.

Implementierungsaufwand

Das DVF soll den Implementierungsaufwand in einem DSL-Projekt reduzieren, indem es den Entwicklern vorgefertigte Komponenten zur Verfügung stellt. Das Ziel der zweiten Fallstudie ist die Implementierung eines Systems namens InTune. Es unterstützt das Testen in verteilten Systemen. Die Anforderungen an InTune stellt Abschnitt 5.2.1 vor. Daraus kann eine domänenspezifische Sprache und ein CFIL-Transformator abgeleitet werden. Die Sprache trägt den Namen GWT-DSL und ermöglicht das Modellieren von Web-Anwendungen. Der Implementierungsaufwand für die GWT-DSL wird von zwei Faktoren beeinflusst:

- Umsetzung der Grammatik.
- Umsetzung des CFIL-Transformators.

Das Ziel dieses Unterkapitels ist es, beide Aspekte genauer zu untersuchen.

Abschnitt 5.2.2 beschreibt die Syntax der GWT-DSL mit einer Grammatik. Sie besteht aus einer Menge von Produktionsregeln. Das DSL Verification Framework stellt vorgefertigte Produktionsregeln zur Verfügung, um den DSL-Entwickler bei seiner Arbeit zu unterstützen. Er integriert deshalb die vorgefertigten Sprachkonstrukte in seine Grammatik und muss nur die Aspekte manuell beschreiben, die nicht Teil des DSL Verification Frameworks sind. Die folgende Tabelle vergleicht anhand der genutzten Produktionsregeln, wie sich eine Umsetzung der Grammatik mit und ohne DVF gestaltet:

Manuelles Beschreiben von:	Mit DVF	Ohne DVF
Klassen		×
Variablen		×
Methoden		×
Expressions		×
Server	×	×
Session	×	×
Client	×	×
GUI-Elemente	×	×
Callback-Objekt	×	×

Die obige Tabelle besteht aus drei Spalten: Die erste zeigt alle Sprachkonstrukte, aus

denen sich die GWT-DSL zusammensetzt. Die zweite und dritte Spalte vergleichen, welche Aspekte der GWT-DSL manuell implementiert werden müssen, wenn ein DSL-Entwickler das DVF verwendet bzw. wenn er es nicht verwendet. Es wird deutlich, dass fast die Häfte aller Sprachkonstrukte Teil des DVF sind und es so den Implementierungsaufwand reduziert.

Zu den vorgefertigten Sprachkonstrukten gehören Klassen, um Schnittstellen von Komponenten, die nicht Teil der Web-Anwendung sind, mittels eigenschaftserhaltender Abstraktion abzubilden. Ein konkretes Anwendungsbeispiel ist die Anbindung einer Datenbank. Des Weiteren sind auch die Variablen des DVF ein Teil der GWT-DSL. Sie ermöglichen das Beschreiben der Session-Variablen bzw. des Zustands von Client- und Server-Seite. Da das DVF auch Methoden bereitstellt, finden die entsprechenden Produktionsregeln in der GWT-DSL Verwendung, um Web-Services modellieren zu können. Die Statements des DSL Verification Frameworks spezifizieren das Verhalten des Modells beim Betreten einer HTML-Seite oder Betätigen eines Schalters. Auch die Semantik der Web-Services und Callback-Objekte werden durch sie beschrieben.

Für einige Sprachkonstrukte gibt es jedoch keine vorgefertigten Produktionsregeln. Diese muss der DSL-Entwickler manuell implementieren. Dazu gehören:

- *GWTServer* stellt den Server dar und beinhaltet eine Menge von Web-Services.
- *GWTSession* repräsentiert Variablen, die einem bestimmten Client zugeordnet sind.
- *GWTClient* entspricht dem Client. Er enthält eine Menge von HTML-Seiten, die die Produktionsregel *GWTPage* abbildet.
- *GWTElement* kann in ein GUI-Element abgeleitet werden, die Teil einer HTML-Seite sind. Dazu gehören beispielsweise Schalter und Textfelder.
- *GWTCallback* repräsentiert ein Callback-Objekt, um asynchrone Aufrufe von Web-Services zu beschreiben.
- *GWTReadText* und *GWTWriteLabel* erweitern die Statements des DVF, um lesend bzw. schreibend auf die Komponenten der GWT-DSL zuzugreifen.

Neben der Quantität muss auch der Komplexitätsgrad ausgewertet werden (vgl. Abschnitt 5.1.5 für die erste Fallstudie). Bei Betrachtung der manuell implementierten Produktionsregeln fällt auf, dass sie primär strukturelle Aspekte eines Modells abbilden. Im Fall der GUI-Elemente gehören dazu beispielsweise Schalter, die aus einem Bezeichner und einer Beschriftung bestehen. Das Verhalten des jeweiligen Schalters wird jedoch durch Referenzieren von *DVFStatement* (vgl. Listing 5.32) umgesetzt und somit einer Produktionsregel aus dem DVF.

Analog dazu gestaltet sich die Modellierung des Clients und seinen HTML-Seiten: Die manuell implementierten Produktionsregeln bilden den Namen, die Startseite und die bereits erwähnten GUI-Elemente ab. Variablen und Verhalten werden durch Referenzen auf Produktionsregeln des DVF umgesetzt (vgl. Listing 5.28 und 5.29). Auch die Web-Services der Server-Seite sind lediglich Methoden des DSL Verification Frameworks. Zusammenfassend wird deutlich, dass komplexere Aspekte der GWT-DSL, die das Verhalten des Modells betreffen, durch Integration von Produktionsregeln des DVF umgesetzt werden können. Lediglich strukturelle Eigenschaften muss der DSL-Entwickler manuell umsetzen. Das DVF hat somit den Implementierungsaufwand der Produktionsregeln sowohl Quantitativ, als auch ihren Komplexitätsgrad betreffend positiv beeinflusst.

Um eine Verknüpfung von formaler Verifikation und modellgetriebener Entwicklung zu erreichen, muss neben der Grammatik auch ein CFIL-Transformator implementiert werden. Der entsprechende Transformator bzw. Transformationsalgorithmus ist Teil von Abschnitt 5.2.3. Er iteriert über den Abstract Syntax Tree eines Modells und substituiert alle Sprachkonstrukte, die nicht Teil des DVF sind, mit CFIL-Komponenten: Für den Server wird eine Klasse generiert, deren Methoden die einzelnen Web-Services repräsentieren. Auch GUI-Elemente und HTML-Seiten bildet der Transformator mit Klassen ab. Die Steuerung erfolgt durch Methodenaufrufe:

- Die Methode *behaviour()* repräsentiert das Betätigen eines Schalters.
- Der Inhalt eines Textfelds kann mit *getText()* ausgelesen werden.
- Das Setzen einer Beschriftung erfolgt durch den Aufruf von *setLabel()*.
- Das Laden einer Seite bildet die Methode *onLoad()* ab.

Verglichen mit der ersten Fallstudie aus Abschnitt 5.1 handelt es sich bei InTune um einen Sonderfall: Das generierte Modell besteht lediglich aus Klassen. Es gibt keine CFIL-Threads, die beispielsweise einen Schalter betätigen. Dies erschwert eine Verifikation mit einem Model Checker und begründet sich durch die Tatsache, dass die Semantik der GUI-Elemente erst durch das Google Web Toolkit definiert wird. Um dem Problem zu begegnen, generiert der CFIL-Transformator zwei Prozesse, die das Verhalten der Anwender simulieren. Durch diese Art der Übersetzung wird sichergestellt, dass Spin das Modell für jede mögliche Nutzerinteraktion verifiziert. Daraus ergeben sich für die Transformation die folgenden Vor- bzw. Nachteile:

- Es musste lediglich einen Transformator in die CFIL implementiert werden. Danach konnte das DVF die Modelle in zwei Zielsprachen, nämlich Promela und Java, übersetzen.
- Es musste nicht der gesamte AST in die CFIL überführt werden, sondern lediglich die Elemente, die auf manuell umgesetzten Sprachkonstrukten basieren. Der

DSL-Entwickler musste beispielsweise keinen Transformator für Statements oder Expressions entwickeln, da diese bereits Teil des DVF sind.

- Der DSL-Entwickler muss erkennen, dass eine Verifikation mit Spin aufgrund fehlender CFIL-Threads problematisch ist und *User-Driver* bzw. *Client-Driver* generieren lassen (vgl. Abbildung 5.13 und 5.14).

Das DSL Verification Framework konnte somit die Transformation der GWT-DSL vereinfachen. Allerdings sollte für zukünftige Arbeiten untersucht werden, wie das DVF mit dem GWT vergleichbare Frameworks unterstützen kann, damit die Generierung zusätzlicher Threads entfällt.

Expertenwissen

Das DSL Verification Framework soll nicht nur den Implementierungsaufwand, sondern auch die Notwendigkeit von Expertenwissen reduzieren. Dies betrifft die Grammatik, den CFIL-Transformator und die Modellierung der InTune-Anwendung. Die Ergebnisse bezüglich der Notwendigkeit von Expertenwissen sind in der folgenden Tabelle zusammengefasst und werden im weiteren Verlauf des Abschnitts genauer vorgestellt:

Notwendige Wissensdomäne	Mit DVF	Ohne DVF
Expertenwissen Model Checking (Promela)		×
Expertenwissen Symmetrie (TopSpin)		
Expertenwissen Linksrekursion		×
Grundkenntnisse im Bereich Compilerbau (Xtext)	×	×
Erweiterbarkeit des DVF	×	

Die obige Tabelle besteht aus drei Spalten: Die erste enthält alle Wissensgebiete, die zur Umsetzung der zweiten Fallstudie relevant sind. Die zweite und dritte Spalte vergleichen in welchen Bereichen der GWT-DSL Expertenwissen vorhanden sein muss, wenn ein DSL-Entwickler das DVF verwendet bzw. wenn er es nicht verwendet.

Die erste Spalte zeigt, dass das DVF dazu beiträgt, InTune ohne Expertenwissen im Bereich der formalen Verifikation umzusetzen. Dies begründet sich wie folgt: Der DSL-Anwender nutzt die GWT-DSL, um Modelle zu beschreiben. Sie enthalten einen Client und einen Server. Der Client besteht aus GUI-Elementen und Callback-Objekten. Der Server setzt sich aus einer Menge von Methoden zusammen. Das Verhalten der entsprechenden Komponenten wird mit den Statements des DVF beschrieben, die sich an Hochsprachen wie Java orientieren. Kenntnisse im Bereich der formalen Verifikation sind somit für den DSL-Anwender nicht notwendig. Dies gilt auch für den DSL-Entwickler: Er implementiert lediglich einen Transformator in die CFIL, die plattformunabhängig ist und kein Wissen im Bereich der formalen Verifikation voraussetzt. Ein Sonderfall ist jedoch beim CFIL-Transformator zu beachten: Die Modelle enthalten keinen aktiven

Prozess, was eine Verifikation mit Spin erschwert. Daher wird ein Transformationszwischenschritt durchgeführt (vgl. Abschnitt 5.2.4). Für zukünftige Arbeiten ist eine Erweiterung des DVF wünschenswert, die den DSL-Entwickler warnt, wenn wichtige Elemente fehlen und automatisiert eine Korrektur vorschlägt.

Sowohl GWT-DSL, als auch InTune enthalten keine Scalarsets. Expertenwissen zum Ausnutzen symmetrischer Eigenschaften ist deshalb nicht notwendig, unabhängig davon ob das DVF Verwendung findet oder nicht. Anders verhält es sich jedoch mit Linksrekursion: Gemäß des vorangegangenen Unterkapitels muss der DSL-Entwickler einige Sprachkonstrukte für strukturelle Aspekte manuell implementieren. Hierzu gehören GUI-Elemente oder HTML-Seiten. Sie nutzen keine Linksrekursion, referenzieren jedoch Produktionsregeln des DVF, wie *DVFStatement* und *DVFExpression*. Beide weisen eine erhöhte Komplexität auf (vgl. Abschnitt 4.3.3). Des Weiteren erfordert ihre Implementierung Expertenwissen im Bereich der rekursiven Ableitungsregeln, um die verschiedenen Operatoren modellieren zu können. Der DSL-Entwickler benötigt daher keine tiefergehenden Kenntnisse für den Umgang mit Linksrekursion und LL-Parsern [1], wenn er auf das DVF zur Umsetzung der GWT-DSL zurückgreift. Selbst mit dem DVF müssen jedoch einige Produktionsregeln, wie die GUI-Elemente, manuell beschrieben werden. Das bedeutet, Grundkenntnisse im Bereich Compilerbau sind in jedem Fall erforderlich. Trotzdem wird zusammenfassend deutlich, dass das DVF auch im Rahmen der zweiten Fallstudie die Notwendigkeit von Expertenwissen erfolgreich reduziert.

Verifikation

Das DSL Verification Framework beinhaltet einen Transformator, der CFIL-Modelle automatisiert nach nach Java und Promela überführt. Dies ermöglicht eine Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation. Da eine entsprechende Verknüpfung den zentralen Anwendungsfall des DSL Verification Frameworks darstellt, wertet dieses Unterkapitel den Verifikationsprozess aus.

Nach der Implementierung der Grammatik und des Transformators erfolgt in Abschnitt 5.2.4 die Beschreibung von InTune unter Verwendung der GWT-DSL. Das InTune-Modell enthält eine Server- und eine Client-Seite. Der Server stellt verschiedene Web-Services zur Verfügung, mit denen sich ein Nutzer beim System anmelden, abmelden, Testfälle anzeigen und ausführen lassen kann. Der Client besteht aus zwei HTML-Seiten: *Login* ermöglicht das Spezifizieren von Nutzername und Passwort, um sich mit dem System zu authentifizieren. *ListTestCases* zeigt die zur Verfügung stehenden Testfälle an. Des Weiteren kann ein Testfall gestartet oder die InTune-Sitzung beendet werden. Nach der Umsetzung des InTune-Modells erfolgt die CFIL-Transformation. Das Ergebnis wird vom DVF nach Promela überführt, um es mit Spin zu verifizieren.

Spin fand eine nicht-erfüllte Anforderung. Sie ermöglicht es Nutzern, sich mit dem System zu authentifizieren, wenn dieses bereits ausgelastet ist. Der DSL-Anwender passte

das Modell entsprechend an und fügte in den Web-Service *login()* eine Synchronize-Umgebung ein. Die erneute Verifikation mit Spin ergab keine weiteren nicht-erfüllten Anforderungen. Der Speicherverbrauch für den Server beträgt 200 Megabytes. Zum Verifizieren des Clients sind 1600 Megabytes erforderlich. Somit erzeugt das DVF auch im Rahmen der zweiten Fallstudie Modelle, die vollständig verifizierbar sind und trägt dazu bei, nicht-erfüllte Anforderungen erfolgreich zu detektieren.

Erweiterbarkeit

Im Gegensatz zur ersten Fallstudie weist die GWT-DSL eine Besonderheit auf: Es ist eine Erweiterung des DVF notwendig, um das InTune-Modell nach Java zu überführen. Der Erweiterungsansatz des DSL Verification Frameworks ist Gegenstand von Abschnitt 4.8. Dieses Unterkapitel diskutiert deshalb, ob die Erweiterungsmöglichkeiten des DVF ausreichend sind, um die InTune-Fallstudie erfolgreich abzuschließen.

Der vorangegangene Abschnitt evaluiert die Verifikation des Modells. Nach der erfolgreichen Verifikation mit Spin wird das Modell in die Hochsprache Java überführt. Da die GWT-DSL nur die Web-Applikation von InTune beschreibt, muss der DSL-Entwickler eine Datenbank in sein Software-Projekt integrieren. Anschließend wird InTune dem GWT übergeben, um daraus eine Web-Anwendung zu generieren. Dies führt zu einem Problem: Das Google Web Toolkit stellt verschiedene Klassen zur Verfügung, um GUI-Elemente, HTML-Seiten und Callback-Objekte abzubilden. Die CFIL ist jedoch eine plattformunabhängige Sprache, die keine GWT-Elemente unterstützt. Der DSL-Entwickler muss deshalb einen zweiten Transformator implementieren, der das Java-Modell modifiziert, damit es die Klassen des Google Web Toolkits verwendet. Gemäß Abschnitt 4.8 entspricht dieses Vorgehen dem Erweiterungsansatz des DVF. Trotzdem sollten zukünftige Arbeiten untersuchen, wie das GWT oder vergleichbare Frameworks besser in das DVF eingebunden werden können. Eine mögliche Lösung des Problems gestaltet sich wie folgt:

Die CFIL bzw. das DVF werden mit GUI-Elementen ergänzt, die der DSL-Entwickler in seine domänenspezifische Sprache integriert. Dazu gehören beispielsweise Schalter und Textfelder. Des Weiteren werden die Transformatoren modifiziert, damit sie die GUI-Elemente nach Promela und Java überführen. Da Java bereits über eine graphische Benutzerschnittstelle im JavaFX-Framework [153] verfügt, bietet es sich an, die GUI-Elemente mit JavaFX abzubilden. Dieser Ansatz hat den Vorteil, dass das DVF mit einem zweiten Java-Transformator ergänzt werden kann (vgl. Abschnitt 4.8.3), der die GUI-Elemente in die GWT überführt und so die Realisierung eines Projekts ermöglicht, das ein externes Framework verwendet.

6 Zusammenfasssung und Ausblick

Die vorangegangenen Abschnitte beschreiben einen Ansatz, um die Qualität in Software-Projekten durch eine Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation zu erhöhen. Dieses Kapitel fasst die gesamte Dissertation zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

6.1 Zusammenfassung

Die modellgetriebene Entwicklung sieht vor, dass Software nicht manuell implementiert werden muss. Stattdessen beschreiben Entwickler formale Modelle mit domänenspezifischen Sprachen und generieren daraus automatisiert ausführbaren Quellcode. Obwohl durch diesen Ansatz die Anzahl der Fehler in einer Software-Lösung reduziert wird, können jedoch immer noch die Modelle selbst fehlerhaft sein. Deshalb ist die Verwendung eines Model Checkers sinnvoll: Er stellt vor der Quellcode-Generierung sicher, dass die Modelle alle an sie gestellten Anforderungen erfüllen. Wenn dies nicht der Fall ist, muss das entsprechende Modell angepasst und erneut verifiziert werden. Sobald alle Fehler entfernt sind, erfolgt eine finale Transformation in eine Hochsprache. Eine derartige Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation führt zu mehreren Vorteilen:

- Durch die Verifikation werden nicht-erfüllte Anforderungen gefunden und so die Software-Qualität erhöht.
- Abstrakte Modelle ermöglichen es auch Domänenexperten ohne Programmierkenntnisse an der Entwicklung zu partizipieren.
- Automatische Quellcode-Generierung beschleunigt den Entwicklungsprozess.

Für eine erfolgreiche Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation müssen eine Reihe von Problemstellungen betrachtet werden: Damit eine domänenspezifische Sprache in einem Software-Projekt Verwendung finden kann, muss ein Parser implementiert werden. Er überführt ein Modell in einen Abstract Syntax Tree (AST). Der AST wird von einem Transformator entgegen genommen und daraus die entsprechende Hochsprache generiert. Um das Modell zusätzlich von einem Model Checker untersuchen zu lassen, muss der AST in eine Model Checker-Eingabesprache

übersetzt werden. Daraus leiten sich zunächst zwei Rollen ab: Der DSL-Entwickler implementiert Parser und die beiden Transformatoren. Der DSL-Anwender nutzt die domänenspezifische Sprache, um damit ein Modell zu beschreiben und lässt es automatisiert in die Zielsprachen überführen bzw. verifizieren. Somit ergeben sich für eine erfolgreiche Verknüpfung von MDD und formaler Verifikation die folgenden Problemstellungen:

- Der Aufwand in einem Software-Projekt steigt, da ein Parser und zwei Transformatoren implementiert werden müssen.
- Der DSL-Entwickler benötigt Expertenwissen im Bereich Compilerbau [1].
- Der DSL-Entwickler benötigt Expertenwissen im Bereich der formalen Verifikation bzw. der entsprechenden Model Checker-Eingabesprache.

Um diesen Problemen zu begegnen stellt diese Arbeit das DSL Verification Framework (DVF) als Lösungsansatz vor. Es beinhaltet eine Menge von vorgefertigten Produktionsregeln. Somit muss ein DSL-Entwickler, wenn er für seine domänenspezifische Sprache eine Grammatik beschreibt, um daraus einen Parser zu generieren, nicht alle Produktionsregeln manuell implementieren. Stattdessen verwendet er, sofern vorhanden, die Produktionsregeln des DVF.

Des Weiteren enthält das DSL Verification Framework zwei vorgefertigte Transformatoren. Sie überführen alle Sprachkonstrukte, die Teil des DVF sind, in eine Hoch- und Model Checker-Eingabesprache. Dabei wird ein Problem deutlich: Der DSL-Entwickler nutzt in seiner Grammatik nicht nur die vorgefertigten Sprachkonstrukte des DVF, sondern auch eigene Produktionsregeln. Die Transformatoren des DVF können daher ein entsprechendes Modell nicht automatisiert in die jeweiligen Zielsprachen übersetzen. Diesem Umstand wird mit der Control Flow Intermediate Language begegnet. Sie basiert auf dem folgenden Konzept (vgl. Abbildung 1.8): Ein DSL-Entwickler schreibt einen Transformator, der alle Elemente seiner DSL, die nicht Teil des DVF sind, in die CFIL überführt. Der Vorteil dieses Ansatzes ist, dass lediglich ein Transformator implementiert werden muss, der einen Teil der DSL in die CFIL übersetzt. Danach übernehmen die Transformatoren des DVF die Generierung von Hoch- und. Model Checker-Eingabesprache.

Das Ziel dieser Dissertation ist die Ausarbeitung, Implementierung und Evaluierung des DVF. Dabei ist von besonderem Interesse, ob das DVF die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation tatsächlich erleichert und die Notwendigkeit von Expertenwissen reduziert. Des Weiteren soll auch untersucht werden, ob Spin die generierten Promela-Modelle vollständig verifizieren kann oder Probleme durch die State Space Explosion entstehen.

Die Konzepte des DSL Verification Frameworks sind unabhängig von einer bestimmten Technologie. Trotzdem müssen für eine Referenzimplementierung geeignete Werkzeuge

ausgewählt werden. Um die Machbarkeit des DVF nachzuweisen, werden deshalb im Rahmen dieser Arbeit die folgenden Werkzeuge eingesetzt:

- Das Xtext-Framework beinhaltet zwei Beschreibungssprachen für Grammatiken und die dazugehörigen Transformatoren. Sie werden genutzt, um sowohl die Produktionsregeln des DVF, als auch die Transformatoren in eine Hoch- bzw. Model Checker-Eingabesprache umzusetzen.
- Diese Arbeit nutzt den Model Checker Spin. Somit überführt einer der beiden Transformatoren des DVF die CFIL-Modelle nach Promela und verifiziert sie mit Spin.

Vor einer detaillierten Ausarbeitung des DSL Verification Frameworks muss der Stand der Forschung analysiert werden. Er gliedert sich in mehrere Teilbereiche: Die formale Verifikation mit einem Model Checker kann zur sogenannten State Space Explosion führen [29]. Sie bewirkt, dass ein Modell nicht mehr vollständig im Hauptspeicher abgebildet werden kann. Daraus ergibt sich für das DVF die folgende Problemstellung: Es muss vermieden werden, dass der CFIL-Transformator Modelle erzeugt, die zur State Space Explosion führen, da in diesem Fall keine erfolgreiche Verknüpfung von MDD und formaler Verifikation möglich ist. Der Stand der Forschung begegnet diesem Problem mit dem Ausnutzen von Symmetrie und eigenschaftserhaltender Abstraktion. Diese Methoden müssen in das DVF integriert werden, damit der Model Checker die generierten Modelle vollständig verifizieren kann, ohne dass es zur State Space Explosion kommt.

Das DVF stellt dem DSL-Entwickler vorgefertigte Produktionsregeln zur Verfügung. Somit ist es wichtig, dass das DSL Verification Framework Sprachkonstrukte enthält, die in möglichst vielen domänenspezifischen Sprachen eingesetzt werden können. Im Rahmen dieser Arbeit sind deshalb auch verwandte Arbeiten von Bedeutung, die sich mit dem praktischen Einsatz von DSLs in Software-Projekten befassen. Die darin enthaltenen Sprachkonstrukte werden analysiert und in Kategorien eingeteilt. Daraus erfolgt im Anschluss die Auswahl geeigneter Sprachkonstrukte, um sie als vorgefertigte Produktionsregeln in das DSL Verification Framework zu integrieren.

Ein wesentlicher Aspekt dieser Arbeit ist das Beschreiben von DSLs und ihre automatisierte Übersetzung in eine Hoch- bzw. Model Checker-Eingabesprache. Auch in diesem Bereich gibt es verwandte Arbeiten. Dabei fällt auf, dass es sich in vielen Fällen um Speziallösungen handelt, die eine bestimmt Domäne betrachten. Dazu gehören beispielsweise endliche Automaten, eine Steuerung für Bahnhöfe oder die Spezifikation von Anforderungen. Ein Problem dieser Ansätze ist, dass sie keine nachträglichen Erweiterungen vorsehen und mit jedem DSL-Projekt eine vollständige Neuimplementierung von Parser bzw. Transformator vorgenommen werden muss. Es gehören jedoch auch generische Ansätze zum Stand der Technik. Sie lassen sich in zwei Kategorien einteilen: Entweder wird die Grammatik der DSL mit semantischen Informationen angereichert, oder eine Übersetzung in eine Zwischensprache durchgeführt. Im Gegensatz zu

den Speziallösungen ermöglichen sie nachträgliche Erweiterungen. Nachteilig sind die Notwendigkeit von Expertenwissen und die fehlende Berücksichtigung der State Space Explosion. Des Weiteren sehen die verwandten Arbeiten nicht vor, dass die DSL in ein Projekt integriert werden kann, das zum Teil in einer Hochsprache und zum Teil mit der DSL umgesetzt ist.

Nach der Analyse des Stands der Technik und den daraus abgeleiteten Problemstellungen erfolgt die detaillierte Vorstellung des DVF bzw. seinen Konzepten. Es setzt sich aus den folgenden Komponenten zusammen:

- Vorgefertigte Produktionsregeln, die der DSL-Entwickler in seine Grammatik integriert. Dazu gehören beispielsweise Ausdrücke, Zuweisungen und Variablen.
- Die Control Flow Intermediate Language: Alle Sprachkonstrukte einer DSL, die nicht Teil des DVF sind, müssen vom DSL-Entwickler in die CFIL überführt werden, damit sie das DVF in die entsprechenden Zielsprachen transformieren kann.
- Wenn der DSL-Anwender die domänenspezifische Sprache verwendet, ist es wünschenswert, Syntaxfehler zu erkennen. Aus diesem Grund verfügt das DVF über eine automatische Validierung.
- Das DVF verfügt über zwei Transformatoren, die ein Modell nach Java und Promela überführen. Die Übersetzungsalgorithmen, mögliche Alternativen und die konkrete Implementierung werden detailliert beschrieben.

Das DVF verfügt über vorgefertigte Produktionsregeln und Transformatoren. Der DSL-Entwickler kann es jedoch auch modifizieren und mit neuen Produktionsregeln oder Transformatoren ergänzen. Deshalb ist auch die Erweiterbarkeit des DVF ein Teil dieser Arbeit.

Im Gegensatz zum Stand der Technik hat das DSL Verification Framework die folgenden Vorteile: Der DSL-Entwickler benötigt weniger Expertwissen im Bereich Compilerbau, um seine domänenspezifische Sprache umzusetzen, da er die mitgelieferten Produktionsregeln und Transformatoren nutzen kann. Sie führen eine automatische Übersetzung in eine Model Checker-Eingabesprache durch, ohne dass der DSL-Entwickler mit den Konzepten der formalen Verifikation vertraut sein muss. Die Produktionsregeln des DVF ermöglichen das Modellieren von eigenschaftserhaltender Abstraktion. Sie bewirkt die Reduktion des Zustandsraums und eine Integration in Software-Projekte, die aus mehreren DSLs und Hochsprachen bestehen. Des Weiteren minimieren die vorgefertigten Elemente des DVF die Entwicklungszeit, da nicht alle Produktionsregeln bzw. Transformatoren manuell vom DSL-Entwickler implementiert werden müssen.

Nach der Beschreibung des Konzepts und dem Nachweis der Implementierbarkeit erfolgt die Vorstellung von zwei Industriefallstudien. Mit ihnen soll die praktische Nutzung des DSL Verification Frameworks evaluiert werden. Das Ziel ist der ersten Fallstudie ist

die Erweiterung von *AssyControl*. Dabei handelt es sich um ein System, das die manuelle Montage an Arbeitsplätzen überwacht. Zu diesem Zweck trägt ein Arbeiter spezielle Handschuhe, die ein Ultraschallsignal aussenden. Anhand der Signallaufzeiten kann AssyControl berechnen, wo sich die Hände des Arbeiters befinden und ihn warnen, wenn er die Zusammenbaureihenfolge nicht einhält.

AssyControl ist ein reaktives System. Daher wird mit dem DVF eine domänenspezifische Sprache für endliche Automaten namens Statechart-DSL entwickelt. Die Grammatik der Statechart-DSL enthält Sprachkonstrukte, mit denen der DSL-Anwender nebenläufige Automaten, Zustände und Transitionen modellieren kann. Auch Produktionsregeln des DVF finden Verwendung, um das Verhalten des Modells zu beschreiben. Dazu zählen beispielsweise Statements, die ein Automat ausführt, wenn er einen Zustand betritt oder verlässt. Die Expressions des DVF bilden Bedingungen ab, die erfüllt sein müssen, damit eine bestimmte Transition ausführbar ist. Auch die Message-Queues des DSL Verification Framework werden in die Grammatik integriert, damit nebenläufige Automaten miteinander über asynchrone Signale kommunizieren können. Die Produktionsregeln des DVF konnten somit einen Beitrag leisten, um die Entwicklung der Grammatik zu beschleunigen.

Auf die Grammatik folgt die Ausarbeitung eines Transformationsalgorithmus bzw. Implementierung eines Transformators. Da die Transformatoren des DVF bereits alle vorgefertigten Sprachkonstrukte unterstützen, muss lediglich ein Teil der DSL in die CFIL überführt werden. Dabei ist von Bedeutung, dass die Produktionsregeln des DVF, wie beispielsweise Expressions, einen hohen Komplexitätsgrad aufweisen. Diese Komplexität erhöht auch den Aufwand, der bei der Implementierung eines entsprechenden Transformators entsteht. Da das DVF jedoch bereits über vorgefertigte Transformatoren verfügt, wird der DSL-Entwickler entlastet. Des Weiteren berücksichtigen die Produktionsregeln des DVF bereits Probleme des Compilerbaus wie Linksrekursion [1]. Somit wird auch die Notwendigkeit von Expertenwissen im Rahmen der ersten Fallstudie reduziert.

Zum Abschluss modellierte der DSL-Anwender mit der Statechart-DSL eine Erweiterung für AssyControl. Sie ermöglicht die schnelle Neukonfiguration des Systems, wenn sich der Aufbau eines Montageplatzes ändert. Zu dem entsprechenden Modell gehört auch seine Verifikation und die Transformation in eine Hochsprache.

Die formale Verifikation des erzeugten Promela-Modells führt zu den folgenden Ergebnissen: Spin meldete einen Fehler, der einen Zugriff über eine gültige Array-Grenze hinweg ermöglicht. Nach der Modifikation des Modells wird erneut von Spin verifiziert. Der Model Checker kann den Verifikationsprozess jedoch nicht beenden, da das Problem der State Space Explosion auftrat. Der DSL-Entwickler erweitert deshalb das Modell mit einem symmetrischen Array. Dabei handelt es sich um ein Konzept, das auch Teil der verwandten Arbeiten ist, um den Zustandsraum zu verkleinern. Anschließend kann Spin unter Ausnutzung der symmetrischen Eigenschaften den Zustandsraum minimieren und das vollständige Modell verifizieren. Da keine weiteren Fehler gefunden werden, erfolgte

die Java-Transformation und somit der Abschluss der ersten Fallstudie.

Das Ziel der zweiten Fallstudie ist die Entwicklung eines Systems namens *InTune*. Es unterstützt Software-Entwickler beim Testen in verteilten System. InTune soll als Web-Applikation umgesetzt werden. Das bedeutet, ein Entwickler verwendet einen Web-Browser, um sich mit dem System zu authentifizieren, Testfälle zu starten oder auszuwerten. Aus den Anforderungen an InTune wird eine domänenspezifische Sprache Namens GWT-DSL abgeleitet. Sie dient zum Modellieren von Web-Anwendungen. Die Sprachkonstrukte der GWT-DSL ermöglichen das Beschreiben eines Clients und eines Servers. Der Client besteht aus HTML-Seiten und verschiedenen GUI-Elementen. Der Server stellt Web-Services zur Verfügung, die der Client aufruft. Auch Produktionsregeln des DVF finden Verwendung, um beispielsweise das Verhalten der Web-Services mit Statements modellieren zu können. Auch die Session-Variablen eines Modells basieren auf den Produktionsregeln des DVF. Die vorgefertigten Sprachkonstrukte können somit auch im Rahmen der zweiten Fallstudie einen Beitrag leisten, um die Verknüpfung von modellgetriebener Entwicklung und formaler Verifikation in einem Software-Projekt zu vereinfachen.

Damit der DSL-Anwender die GWT-DSL nutzen kann, um InTune zu beschreiben, muss ein Transformationsalgorithmus entwickelt bzw. ein Transformator implementiert werden. Er übersetzt alle Sprachkonstrukte, die nicht Teil des DVF sind, in die CFIL. Danach erfolgt die Umsetzung des InTune-Modells, seine Verifikation und eine abschließenende Java-Transformation. Beim Verifizieren mit Spin wird eine nicht-erfüllte Anforderung gefunden. Sie ermöglicht es Nutzern, sich mit dem System zu authentifizieren, wenn dieses bereits ausgelastet ist. Der DSL-Anwender passt sein Modell an und verifizierte es erneut. Da keine weiteren Fehler gefunden werden, erfolgt die Transformation in eine Hochsprache. Durch das Entdecken der nicht erfüllten Anforderung wird deutlich, dass eine Verknüpfung von modellgetrieber Entwicklung und formaler Verifikation die Software-Qualität erhöht.

Die Java-Transformation verdeutlicht ein Problem: Das Ziel der zweiten Fallstudie ist die Generierung einer Web-Anwendung. Sie besteht typischerweise aus HTML und Javascript. Daher soll das erzeugte Java-Modell dem Google Web Toolkit (GWT) übergeben werden. Das GWT beinhaltet einen Transformator, der Java nach HTML und Javascript überführt, damit es ein Browser als Web-Anwendung darstellen kann. Das GWT stellt verschiedene Java-Klassen zur Verfügung, die Schalter, Textfelder, usw. abstrahieren. Diese Klassen müssen verwendet werden, damit das GWT aus einem Java-Modell eine Web-Anwendung erzeugen kann. Das DVF unterstützt jedoch keine GWT-Klassen, da die CFIL eine plattformunabhängige Sprache ist. Deshalb wird vom DSL-Entwickler ein zweiter Transformator mit geringer Komplexität implementiert. Er modifiziert das Java-Modell und integriert die notwendigen GWT-Klassen.

6.2 Ausblick

Die vorangegangenen Abschnitte haben deutlich gemacht, dass das DSL Verification Framework erfolgreich eingesetzt werden kann, um modellgetriebene Entwicklung und formale Verifikation in Software-Projekten zu verknüpfen. Trotzdem gibt es eine Reihe von möglichen zukünftigen Arbeiten, um den Ansatz weiter zu verbessern. Sie werden im Rahmen dieses Abschnitts genauer vorgestellt.

Das DVF stellt symmetrische Arrays zur Verfügung, um den Zustandsraum eines Modells zu minimieren und so der State Space Explosion entgegen zu wirken. Der Model Checker Spin berücksichtigt jedoch keine Symmetrie. Deshalb muss eine Spin-Erweiterung namens TopSpin genutzt werden, die den Zustandsraum eines Modells durch Ausnutzen von Symmetrie verkleinert. Die erste Fallstudie macht deutlich, dass der Speicherverbrauch eines Model Checkers durch Berücksichtigung symmetrischer Aspekte um mehrere Zehnerpotenzen reduziert werden kann. Daher ist es für zukünftige Arbeiten wünschenswert, einen Model Checker zu finden, der Symmetrie nativ unterstützt oder die Funktionalität von TopSpin in Spin zu integrieren. Dieses Vorgehen hat den Vorteil, dass der Promela-Transformator des DVF an Komplexität verliert, da die spezielle Modellierung der symmetrischen Arrays (vgl. Abschnitt 2.1.4) entfällt.

Des Weiteren kann das DVF nur dann Symmetrie ausnutzen, wenn der DSL-Anwender Scalarsets in sein Modell integriert. Dies ist nur möglich, wenn er mit den Konzepten des DVF vertraut ist und selbstständig symmetrische Eigenschaften erkennt. Hier wäre es für zukünftige Arbeiten wünschenswert, an automatischer Erkennung von Symmetrie zu forschen.

Im Rahmen der zweiten Fallstudie zeigten sich die GUI-Elemente des GWT als problematisch. Der DSL-Entwickler musste selbstständig erkennen, dass das Modell nur dann von Spin verifiziert werden kann, wenn er spezielle Threads generiert. Für zukünftige Arbeiten sollte das DVF daher das Fehlen von Threads detektieren und den DSL-Entwickler darauf hinweisen. Alternativ könnte das DVF modifiziert werden, damit es die entsprechenden Threads automatisiert und ohne Zutun des DSL-Entwicklers generiert.

Ein weiterer Aspekt der zweiten Fallstudie ist das nachträgliche Einfügen von GWT-Klassen. Um diesem Problem zu begegnen, ist es empfehlenswert, das DVF mit neuen Produktionsregeln zu erweitern. Sie repräsentieren GUI-Elemente, wie Schalter oder Textfelder, die ein DSL-Entwickler in seine Grammatik integrieren kann. Im Anschluss wird das DVF mit einem zweiten Java-Transformator ergänzt. Dieser überführt alle GUI-Elemente in GWT-Klassen. Der Ansatz hat zwei Vorteile:

- Der DSL-Entwickler muss keinen zweiten Transformator implementieren, der die GWT-Klassen nachträglich einfügt.
- In zukünftigen Projekten kann der GWT-Transformator wiederverwendet werden, um domänenspezifische Sprachen für Web-Anwendungen umzusetzen.

Zum gegenwärtigen Zeitpunkt wird das DVF auch am *Fraunhofer-Institut für integrierte Schaltungen* eingesetzt und weiter evaluiert. Das Ziel ist die Entwicklung von möglichst kleinen und leistungsfähigen Digitalkameras [52]. Sie enthalten das Android-Betriebssystem. Es verfügt über einen offenen Quellcode und kann beliebig erweitert werden. Deshalb werden unter Verwendung des DVF Modelle beschrieben, die Hintergrundprozesse für das Android-Betriebssystem repräsentieren. Dies ermöglicht beispielsweise das Einfügen eines Prozesses, der automatisiert überprüft, ob ein Datenträger mit einem Firmware-Update eingelegt wurde und ihn gegebenenfalls installiert. Die entsprechenden Modelle sollen sowohl in eine Hochsprache übersetzt, als auch durch einen Model Checker verifiziert werden.

Literaturverzeichnis

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] M. H. Alalfi, J. R. Cordy, and T. R. Dean. Modelling Methods for Web Application Verification and Testing: State of the Art. *Softw. Test. Verif. Reliab.*, 19:265–296, December 2009.

[3] C. Ammann. Integration von Model-Driven Development und formaler Verifikation in den Softwareentwicklungsprozess - eine Fallstudie mit einem 3D-Tracking-System. *Softwaretechnik-Trends, Band 30, Heft 10*, 2010. ISSN 0720-8928.

[4] C. Ammann. Formal Verification of Web Applications. *Softwaretechnik-Trends, Band 32, Heft 1*, 2011. ISSN 0720-8928.

[5] C. Ammann. Verifikation von UML-Statecharts unter besonderer Berücksichtigung von Speicherverbrauch und Laufzeit des Model Checkers. *Softwaretechnik-Trends, Band 31, Heft 3*, 2011. ISSN 0720-8928.

[6] C. Ammann. Verification of Behavioral Domain-Specific Languages with a Model Checker. In T. Zhang, editor, *Mechanical Engineering and Technology*, volume 125 of *Advances in Intelligent and Soft Computing*, pages 779–782. Springer Berlin / Heidelberg, 2012.

[7] C. Ammann. Verification of Web Applications with a Model Checker. In T. Gonzalez and M. Hamza, editors, *The 16th IASTED International Conference on Software Engineering and Applications*. ACTA Press, 2012.

[8] C. Ammann, S. Kleuker, and E. Pulvermüller. From Business Modeling to Verified Applications. In *Informatik 2011*, GI-Edition - Lecture Nodes in Informatics (LNI). Protocol based Modelling of Business Interactions Workshop.

[9] C. Ammann, T. Müller, and L. Knemeyer. Java-Applikation zur Visualisierung der Assycontrol-Erweiterung, 2011. Technischer Bericht, Hochschule Osnabrück.

[10] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *Model Driven Engineering Languages and Systems*, pages 436–450. 2007.

[11] H. Andersen. Partial Model Checking. In *Logic in Computer Science, 1995. LICS '95. Proceedings., Tenth Annual IEEE Symposium on*, pages 398 –407, jun 1995.

[12] Association for Computing Machinery. System Software Award. http://awards.acm.org/homepage.cfm?awd=149; accessed 18-August-2011.

[13] J. Bauer, I. Schaefer, T. Toben, and B. Westphal. Specification and Verification of Dynamic Communication Systems. In *ACSD '06: Proceedings of the Sixth International Conference on Application of Concurrency to System Design*, pages 189–200, Washington, DC, USA, 2006. IEEE Computer Society.

[14] M. E. Beato, M. Barrio-Solórzano, C. E. Cuesta, and P. de la Fuente. UML Automatic Verification Tool with Formal Methods. *Electron. Notes Theor. Comput. Sci.*, 127(4):3–16, Apr. 2005.

[15] K. Bergner, M. Broy, K.-R. Moll, M. Pizka, A. Rausch, and T. Seifert. Erfolgreiches Management von Softwareprojekten. *Informatik Spektrum*, 27(5), Oct. 2004.

[16] L. Bettini. A DSL for writing type systems for Xtext languages. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 31–40, New York, NY, USA, 2011. ACM.

[17] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The Software Model Checker BLAST: Applications to Software Engineering. *INT. J. SOFTW. TOOLS TECHNOL. TRANSFER*, 2007.

[18] B. Bordbar and K. Anastasakis. MDA and Analysis of Web Applications. In *Proceedings of the 31st VLDB conference on Trends in Enterprise Application Architecture*, TEAA'05, pages 44–55, Berlin, Heidelberg, 2006. Springer-Verlag.

[19] E. Börger. *Specification and Validation Methods*. International schools for computer scientists. Oxford University Press, 1995.

[20] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible Markup Language (XML) 1.0 (fifth edition). World Wide Web Consortium, Recommendation REC-xml-20081126, November 2008.

[21] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentelli, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Staple, G. Swamy, and T. Villa. VIS : A System for Verification and Synthesis. pages 428–432. Springer-Verlag, 1996.

[22] Y. Cao, T. Xu, T. Tang, H. Wang, and L. Zhao. Automatic Generation and Verification of interlocking Tables based on Domain Specific Language for Computer

based Interlocking Systems (DSL-CBI). In *Computer Science and Automation Engineering (CSAE), 2011 IEEE International Conference on*, volume 2, pages 511 –515, June 2011.

[23] V. d. Castro, J. M. V. Mesa, E. Herrmann, and E. Marcos. A Model-Driven Approach for the Alignment of Business and Information Systems Models. In *Proceedings of the 2008 Mexican International Conference on Computer Science*, ENC '08, pages 33–43, Washington, DC, USA, 2008. IEEE Computer Society.

[24] S. Ceri, P. Fraternali, A. Bongio, M. Brambilla, S. Comai, and M. Matera. *Designing Data-Intensive Web Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[25] K. M. Chandy. *Parallel Program Design: A Foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[26] K. M. Chandy and J. Misra. The Drinking Philosophers Problem. *ACM Trans. Program. Lang. Syst.*, 6(4):632–646, Oct. 1984.

[27] K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic Anchoring with Model Transformations. In *In ECMDA-FA, volume 3748 of LNCS*, pages 115–129. Springer, 2005.

[28] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. Nusmv 2: An Opensource Tool for Symbolic Model Checking. pages 359–364. Springer, 2002.

[29] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, 2001. Springer-Verlag.

[30] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, Sept. 1994.

[31] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[32] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A Finite State Verification Technique for Software Systems. *IBM Systems Journal*, 41:2002, 2001.

[33] D. Conan, R. Rouvoy, and L. Seinturier. Scalable Processing of Context Information with COSMOS. In *Proceedings of the 7th IFIP WG 6.1 international conference on Distributed applications and interoperable systems*, DAIS'07, pages 210–224, Berlin, Heidelberg, 2007. Springer-Verlag.

[34] G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. Viatra - Visual Automated Transformations for Formal Verification and Validation of UML Models. In *Proceedings of the 17th IEEE international conference on Automated software engineering*, ASE '02, pages 267–, Washington, DC, USA, 2002. IEEE Computer Society.

[35] A. David, M. Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In R.-D. Kutsche and H. Weber, editors, *Fundamental Approaches to Software Engineering*, volume 2306 of *Lecture Notes in Computer Science*, pages 218–232. Springer Berlin Heidelberg, 2002.

[36] R. Detmer. *Introduction to 80x86 Assembly Language and Computer Architecture.* Jones and Bartlett, 2001.

[37] A. Deutsch, L. Sui, and V. Vianu. Specification and Verification of Data-Driven Web Applications. *J. Comput. Syst. Sci.*, 73(3):442–474, May 2007.

[38] E. W. Dijkstra. Go To Statement considered Harmful. *Comm. ACM*, 11(3):147–148, 1968. letter to the Editor.

[39] E. W. Dijkstra. Letters to the Editor: Goto Statement considered Harmful. *Commun. ACM*, 11(3):147–148, Mar. 1968.

[40] E. W. Dijkstra. Guarded Commands, Non-Determinacy and a Calculus for the Derivation of Programs. June 1974.

[41] D. L. Dill. The Murphi Verification System. In *CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification*, pages 390–393, London, UK, 1996. Springer-Verlag.

[42] A. Donaldson. *Automatic Techniques for detecting and exploiting Symmetry in Model Checking.* PhD thesis, University of Glasgow, 2007.

[43] A. F. Donaldson and A. Miller. A Computational Group Theoretic Symmetry Reduction Package for the SPIN model checker. In *Proceedings of the 11th International Conference on Algebraic Methodology and Software Technology (AMAST'06)*, volume 4019 of *Lecture Notes in Computer Science*, pages 374–380. Springer, 2006.

[44] W. Dong, J. Wang, X. Qi, and Z.-C. Qi. Model Checking UML Statecharts. In *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pages 363 – 370, dec. 2001.

[45] R. Drechsler and B. Becker. *Binary Decision Diagrams: Theory and Implementation.* Springer, 1998.

[46] Eclipse Foundation. Eclipse Xtext - Language Development Framework. http://www.eclipse.org/Xtext; accessed 31-January-2011;.

[47] M. El Amine Matougui and S. Leriche. Validation of COSMOS DSL Programs. In *Computer Engineering and Systems (ICCES), 2010 International Conference on*, pages 307 –313, Dezember 2010.

[48] G. Fischermanns. *Praxishandbuch Prozessmanagement.* Ibo-Schriftenreihe. Schmidt, 2009.

[49] D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 1998.

[50] M. Fowler. *Domain-Specific Languages.* The Addison-Wesley Signature Series. Addison-Wesley, 2010.

[51] M. Frappier, B. Fraikin, R. Chossart, R. Chane-Yack-Fa, and M. Ouenzar. Comparison of Model Checking Tools for Information Systems. In *Proceedings of the 12th international conference on Formal engineering methods and software engineering*, ICFEM'10, pages 581–596, Berlin, Heidelberg, 2010. Springer-Verlag.

[52] Fraunhofer Gesellschaft. INCA - Intelligente Kamera. http://www.iis.fraunhofer.de/de/bf/bsy/fue/digicam/pov/inca.html; accessed 3-April-2014.

[53] G. Frehse. Phaver: Algorithmic Verification of Hybrid Systems Past Hytech. *Int. J. Softw. Tools Technol. Transf.*, 10(3):263–279, May 2008.

[54] P. Freudenstein, J. Buck, M. Nussbaumer, and M. Gaedke. Model-Driven Construction of Workflow-Based Web Applications with Domain-specific Languages. In *Proceedings of the 3rd International Workshop on Model-Driven Web Engineering MDWE 2007, Como, Italy*, volume 261 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[55] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.

[56] J. J. Garrett. Ajax: A New Approach to Web Applications, 2005. Adaptive Path LLC, http://www.adaptivepath.com/publications/essays/archives/000385.php.

[57] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[58] Google Inc. Google Chrome. https://www.google.com/chrome; accessed 20-March-2012;.

[59] Google Inc. Google Web Toolkit. http://code.google.com/webtoolkit; accessed 18-August-2011.

[60] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java (Addison-Wesley))*. Addison-Wesley Professional, 2005.

[61] R. C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison-Wesley Professional, 1 edition, 2009.

[62] T. Haitzer and U. Zdun. DSL-Based Support for Semi-Automated Architectural Component Model Abstraction throughout the Software Lifecycle. In *Proceedings of the 8th international ACM SIGSOFT conference on Quality of Software Architectures*, QoSA '12, pages 61–70, New York, NY, USA, 2012. ACM.

[63] J. N. Hall and R. L. Schwartz. *Effective Perl Programming: Writing better Programs with Perl*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998.

[64] M. Han and C. Hofmeister. Modeling and Verification of adaptive Navigation in Web Applications. In *Proceedings of the 6th international conference on Web engineering*, ICWE '06, pages 329–336, New York, NY, USA, 2006. ACM.

[65] D. Harel. Statecharts: A visual Formalism for complex Systems. *Science of Computer Programming*, 8(3):231 – 274, 1987.

[66] W. Heijstek and M. R. V. Chaudron. Empirical Investigations of Model Size, Complexity and Effort in a Large Scale, Distributed Model Driven Development Process. In *EUROMICRO-SEAA*, pages 113–120. IEEE Computer Society, 2009.

[67] G. J. Holzmann. An Analysis of Bitstate Hashing. *Form. Methods Syst. Des.*, 13(3):289–307, 1998.

[68] G. J. Holzmann. *The Spin Model Checker: Primer and reference Manual*. Addison-Wesley Professional, 2003.

[69] P. Hudak. Modular Domain Specific Languages and Tools. In *Software Reuse, 1998. Proceedings. Fifth International Conference on*, pages 134–142, Jun 1998.

[70] P. Hudak, S. Peyton Jones, P. Wadler, B. Boutel, J. Fairbairn, J. Fasel, M. M. Guzmán, K. Hammond, J. Hughes, T. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, and J. Peterson. Report on the Programming Language Haskell: a non-strict, purely functional Language Version 1.2. *SIGPLAN Not.*, 27:1–164, May 1992.

[71] IEEE. IEEE Standard VHDL Language Reference Manual. Standard, IEEE Press, 1987.

[72] International Business Machines Corporation. IBM Rational Rhapsody. http://www.ibm.com/software/rational; accessed 14-September-2010.

[73] International Business Machines Corporation. IBM Rational Software. http://www.ibm.com/software/de/rational/; accessed 02-May-2012.

[74] ISO/IEC JTC1 SC22 WG14. ISO/IEC 9899:TC2 Programming Languages - C. Technical report, May 2005.

[75] P. Iyenghar, E. Pulvermüller, C. Westerkamp, and J. Wübbelmann. Integrated Model-Based Approach and Test Framework for embedded Systems. In *Specification and Design Languages (FDL), 2011 Forum on*, pages 1–8, 2011.

[76] D. Jackson. Alloy: a lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002.

[77] G. Karsai, A. Agrawal, F. Shi, and J. Sprinkle. On the Use of Graph Transformations for the formal Specification of Model Interpreters. *Journal of Universal Computer Science*, 9:1296–1321, 2003.

[78] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[79] S. Kleuker. KoverJa Projekt - Korrekte verteilte Java Applikationen. http://www.edvsz.hs-osnabrueck.de/kleuker/CSI/KoverJa; accessed 14-September-2010.

[80] S. Kleuker. *Formale Modelle der Softwareentwicklung*. Vieweg+Teubner Verlag, 2009.

[81] A. Knapp and G. Zhang. Model Transformations for Integrating and Validating Web Application Models. In H. C. Mayr and R. Breu, editors, *Modellierung*, volume 82 of *LNI*, pages 115–128. GI, 2006.

[82] U. Knauer. *Diskrete Strukturen, kurz gefasst*. Spektrum Akademischer Verlag, 2001.

[83] N. Koch, G. Zhang, and H.-M. Hassler. Modeling Business Processes in Web Applications with ArgoUWE. In *Proc. 7 Int. Conf. Unified Modeling Language, Volume 3273 of Lect. Notes Comp. Sci*, pages 69–83. Springer, 2004.

[84] A. . T. B. Laboratories, S. Johnson, and B. Kernighan. *The Programming Language B*. Computing science technical report. Bell Laboratories, 1973.

[85] J. P. Lang. Redmine - A flexible Project Management Web Application. http://www.eclipse.org/Xtext; accessed 13-September-2011;.

[86] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a Nutshell. *In Springer International Journal of Software Tools for Technology Transfer*, 1(1+2):134–152, May 1997.

[87] D. Latella, I. Majzik, and M. Massink. Automatic Verification of a Behavioural Subset of UML Statechart Diagrams Using the SPIN Model-Checker. *Formal Asp. Comput.*, 11(6):637–664, 1999.

[88] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, pages 465–, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[89] J. Lee. ANSI C Yacc Grammar, 1985. http://www.lysator.liu.se/c/ANSI-C-grammar-y.html; accessed 25-August-2012.

[90] K. R. P. H. Leung, L. C. K. Hui, S. M. Yiu, and R. W. M. Tang. Modeling Web Navigation by Statechart. In *24th International Computer Software and Applications Conference*, COMPSAC '00, pages 41–47, Washington, DC, USA, 2000. IEEE Computer Society.

[91] J. R. Levine, T. Mason, and D. Brown. *Lex & Yacc (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1992.

[92] F. Leymann. Web Services Flow Language (WSFL 1.0). *IBM Corporation*, page 107, 2001.

[93] D. R. Licata and S. Krishnamurthi. Verifying Interactive Web Progams. In *IEEE International Symposium on Automated Software Engineering*. IEEE Press, 2004.

[94] J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of the 2nd international conference on The unified modeling language: beyond the standard*, UML'99, pages 430–444, Berlin, Heidelberg, 1999. Springer-Verlag.

[95] J. Lind-Nielsen, H. Andersen, H. Hulgaard, G. Behrmann, K. Kristoffersen, and K. Larsen. Verification of Large State/Event Systems Using Compositionality and Dependency Analysis. *Formal Methods in System Design*, 18:5–23, 2001.

[96] E. L. Lohse, R. V. Smith, J. F. Traub, and S. Gorn. Character Structure and Character Parity Sense for Serial-By-Bit Data Communication in the American Standard Code for Information Interchange. *Commun. ACM*, 8(9):553–556, 1965.

[97] T. Lukman, M. Mernik, Z. Demirezen, B. Bryant, and J. Gray. Automatic Generation of Model Traversals from Metamodel Definitions. In *Proceedings of the 48th Annual Southeast Regional Conference*, ACM SE '10, pages 78:1–78:6, New York, NY, USA, 2010. ACM.

[98] S. Merz. Model Checking and Code Generation for UML State Machines and Collaborations. In *In G. Schellhorn and W. Reif. 5 th Workshop on Tools for System Design and Verification (FM-TOOLS*, pages 59–64, 2002.

[99] Microsoft Corporation. Microsoft Internet Explorer. http://www.windows.microsoft.com; accessed 2-January-2014;.

[100] Microsoft Corporation. Microsoft Windows. http://windows.microsoft.com; accessed 16-May-2012;.

[101] Microsoft Research. AsmL: Abstract State Machine Language. http://research.microsoft.com/en-us/projects/asml/, zuletzt besucht am 31.11.2011.

[102] E. Mikk, Y. Lakhnech, M. Siegel, and G. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on*, pages 90 –101, 1998.

[103] J. Miller and J. Mukerji. MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG), 2003.

[104] Mozilla Foundation. Mozilla Firefox. http://www.firefox.com; accessed 20-March-2012;.

[105] F. Müller. *Systemprogrammierung in Google Go: Grundlagen - Skalierbarkeit - Performanz - Sicherheit*. iX-Edition. Dpunkt.Verlag GmbH, 2011.

[106] S. Nakajima. Verification of Web Service Flows with Model-Checking Techniques. In *Proceedings of the First International Symposium on Cyber Worlds (CW'02)*, CW '02, pages 0378–, Washington, DC, USA, 2002. IEEE Computer Society.

[107] M. Nelson and J.-L. Gailly. *The Data Compression Book (2nd ed.)*. MIS:Press, New York, NY, USA, 1996.

[108] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. John Wiley & Sons, Inc., New York, NY, USA, 1992.

[109] C. Norris IP and D. L. Dill. Better Verification through Symmetry. *Formal Methods in System Design*, 9:41–75, 1996. 10.1007/BF00625968.

[110] Object Management Group. MOF 2.0/XMI Mapping, Version 2.1.1. http://www.omg.org/spec/XMI/2.1.1; accessed 14-September-2010.

[111] Object Management Group. OMG Unified Modeling Language (OMG UML) infrastructure version 2.3. Technical Report formal/2010-05-03, 2010.

[112] T. O'Hear and Y. Boudjenane. Using Activity Descriptions to Generate User Interfaces for ERP software. In *Proceedings of the 13th International Conference on Human-Computer Interaction. Part IV: Interacting in Various Application Domains*, pages 577–586, Berlin, Heidelberg, 2009. Springer-Verlag.

[113] Oracle Inc. Mysql Homepage. http://www.mysql.com/; accessed 12-February-2013.

[114] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen, 5. Auflage.* Spektrum Akademischer Verlag, 2012.

[115] T. Parr. A Java 1.5 Grammar for ANTLR v3. http://www.antlr.org/grammar/1152141644268/Java.g; accessed 18-July-2012.

[116] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.

[117] F. Paternò, C. Santoro, and L. D. Spano. Designing Usable Applications based on Web Services. In S. M. Abrahão, E. L.-C. Law, J. Stage, K. Hornbæk, and N. J. Juzgado, editors, *I-USED*, volume 407 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[118] L. Pedro, L. Lucio, and D. Buchs. System Prototype and Verification Using Metamodel-Based Transformations. *Distributed Systems Online, IEEE*, 8(4):1, april 2007.

[119] J. L. Peterson. *Petri Net Theory and the Modeling of Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.

[120] M. Pilgrim. *Dive Into Python.* APress, 2004.

[121] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[122] A. Pnueli. The Temporal Logic of Programs. In *SFCS '77: Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[123] I. Porres. Modeling and Analyzing Software Behavior in UML, 2001.

[124] E. Pulvermüller, S. Feja, and A. Speck. Developer-Friendly Verification of Process-Based Systems. *Know.-Based Syst.*, 23(7):667–676, Oct. 2010.

[125] M. Robinson and P. Vorobiev. *Swing (Second Edition)*. Dreamtech Press, 2004.

[126] G. Rossi, H. A. Schmidt, and F. Lyardet. Customizing Business Processes in Web Applications. In K. Bauknecht, A. M. Tjoa, and G. Quirchmayr, editors, *Proceedings of the 4th International Conference on E-Commerce and Web Technologies (Ec-WEB 2003)*, number 2738 in LNCS, pages 359–368. Springer Verlag, 2003.

[127] T. C. Ruijs. *Towards Effective Model Checking*. PhD thesis, Enschede, March 2001.

[128] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *Electronic Notes in Theoretical Computer Science*, 55(3):357 – 369, 2001.

[129] W. Shen, K. Compton, and J. Huggins. A Toolset for Supporting UML Static and Dynamic Model Checking. In *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, COMPSAC '02, pages 147–152, Washington, DC, USA, 2002. IEEE Computer Society.

[130] L. Simon, A. Mallya, and G. Gupta. Design and Implementation of AT: a Real-time Action Description Language. In *Proceedings of the 15th international conference on Logic Based Program Synthesis and Transformation*, LOPSTR'05, pages 44–60, Berlin, Heidelberg, 2006. Springer-Verlag.

[131] Y. Singh and M. Sood. Model Driven Architecture: A Perspective. In *Advance Computing Conference, 2009. IACC 2009. IEEE International*, pages 1644 –1652, March 2009.

[132] soft2tec GmbH. Creativity.Knowledge.People Consulting. http://soft2tec.com; accessed 14-September-2010.

[133] M. Spielmann. Verification of Relational Transducers for Electronic Commerce. *JCSS*, 66:92–103, 2000.

[134] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung - Techniken, Engineering, Management*. dpunkt.verlag, 2007. zweite Auflage.

[135] G. Stuurman and I. Kurtev. Action Semantics for defining Dynamic Semantics of Modeling Languages. In *Proceedings of the Third Workshop on Behavioural Modelling*, BM-FA '11, pages 64–71, New York, NY, USA, 2011. ACM.

[136] B. Taylor. *Guide for the Use of the International System of Units (SI): The Metric System*. DIANE Publishing Company, 1995.

[137] The Eclipse Foundation. Eclipse IDE. http://www.eclipse.org; accessed 16-October-2012;.

[138] The Eclipse Foundation. Eclipse Modeling Framework Project. http://www.eclipse.org/modeling/emf; accessed 27-June-2012;.

[139] The FreeBSD Project. FreeBSD Homepage. http://www.freebsd.org; accessed 16-May-2012;.

[140] The GNU Project. GCC, the GNU Compiler Collection. http://www.gnu.org; accessed 16-May-2012;.

[141] The PHP Group. Php: Hypertext Preprocessor Homepage. http://www.php.net/; accessed 12-February-2013.

[142] S. Thibault, R. Marlet, and C. Consel. Domain-specific Languages: From Design to Implementation Application to Video Device Drivers Generation. *Software Engineering, IEEE Transactions on*, 25(3):363 –377, may/jun 1999.

[143] V. Torres, J. Munoz, and V. Pelechano. A Model Driven Method for the Integration of Web Applications. In *Web Congress, 2005. LA-WEB 2005. Third Latin American*, page 10 pp., Oct.-2 Nov. 2005.

[144] L. Torvalds. The Linux Kernel Archive. http://kernel.org; accessed 16-May-2012;.

[145] A. Valmari. *State Space Generation: Efficiency and Practicality*. PhD thesis, 1988.

[146] A. Valmari. A Stubborn Attack On State Explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, CAV '90, pages 156–165, London, UK, UK, 1991. Springer-Verlag.

[147] M. van Amstel, M. van den Brand, and L. Engelen. An Exercise in iterative Domain-Specific Language Design. In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 48–57, New York, NY, USA, 2010. ACM.

[148] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 3–, Washington, DC, USA, 2000. IEEE Computer Society.

[149] J. von Neumann. First Draft of a Report on the EDVAC. *IEEE Ann. Hist. Comput.*, 15(4):27–75, Oct. 1993.

[150] A. Wasowski. On efficient Program Synthesis from Statecharts. *SIGPLAN Not.*, 38:163–170, June 2003.

[151] A. Wasowski. On Efficient Program Synthesis from Statecharts. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 163–170, New York, NY, USA, 2003. ACM.

[152] A. Wasowski. Flattening Statecharts without Explosions. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '04, pages 257–266, New York, NY, USA, 2004. ACM.

[153] J. Weaver, W. Gao, S. Chin, D. Iverson, and J. Vos. *Pro JavaFX 2: A Definitive Guide to Rich Clients with Java Technology.* Apresspod Series. Apress, 2012.

[154] M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 1.2.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 1999.

[155] World Wide Web Consortium Working Group. HTML 4.01 Specification - W3C Recommendation., 1999. http://www.w3.org/TR/html401; accessed 20-March-2012;.

[156] World Wide Web Consortium Working Group. Hypertext Transfer Protocol – HTTP/1.1., 1999. http://www.w3.org/Protocols/rfc2616/rfc2616.html; accessed 20-March-2012;.

[157] World Wide Web Consortium Working Group. Web Services Architecture - W3C Working Group Note., 2004. http://www.w3.org/TR/ws-arch; accessed 20-March-2012;.