UML-basierte Action- und Constraint-Sprachen im Kontext modellbasierten Testens mit dem UML Testing Profil

Masterarbeit von Jan Sydow

Betreuer: Dipl.-Inform. M.Sc. Marc-Florian Wendland
Erstgutachterin: Prof. Dr. Ina Schieferdecker
Zweitgutachterin: Prof. Dr. Elfriede Fehr

Institut für Informatik
Freie Universität Berlin
3. Mai 2015

Abstract

Beim modellbasierten Testen werden Modelle zur Automatisierung einiger manueller Aufgaben im Bereich des Softwaretestens eingesetzt. So generieren Testfallgeneratoren Testfälle aus Testfallentwurfsmodellen, die als Zustandsmaschinen vorliegen. In dieser Arbeit wird untersucht, wie sich die Action- und Constraint-Sprachen OCL, MARTE VSL und Alf der UML für die Beschreibung von Wächtern und Effekten in Testfallentwurfsmodellen eignen. Es stellt sich heraus, dass die *Action Language for Foundational UML* (Alf) die geeignetste ist. Daraufhin wird eine Übersetzung von Alf-Code in die Eingabesprachen von drei Testfallgeneratoren, Java, QML und C#, aufgezeigt. Die Übersetzung nach C# wird am Beispiel des Testfallgenerators Spec Explorer für Fokus!MBT implementiert. Fokus!MBT ist ein Werkzeug für modellbasiertes Testen welches am Fraunhofer Institut für Offene Kommunikationssysteme (FOKUS) entwickelt wurde. Abschließend wird in einer Fallstudie der Einsatz von Alf zur Beschreibung von Wächtern und Effekten in Testfallentwurfsmodellen mit Fokus!MBT demonstriert.

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen Hilfsmittel als angegeben verwendet habe. Insbesondere versichere ich, dass ich alle wörtlichen und sinngemäßen Übernahmen aus anderen Werken als solche kenntlich gemacht habe.

Berlin, den 3. Mai 2015

Jan Sydow

Inhaltsverzeichnis

1 Einleitung und Grundlagen **1**

1.1 Modellbasiertes Testen 1

1.2 UML/UTP 2

1.3 Fokus!MBT 4

1.4 Problembeschreibung 5

1.5 Aufgabenstellung 7

1.6 Gliederung 9

2 Die Eingabesprachen der Generatoren **11**

2.1 Conformiq Designer 11

2.2 Graphwalker 14

2.3 Spec Explorer 16

2.4 Gemeinsamkeiten und Unterschiede der Sprachen 22

3 Analyse und Bewertung UML-basierter Sprachen **24**

3.1 Object Constraint Language 24

3.2 MARTE Value Specification Language 29

3.3 Action Language for Foundational UML 33

3.4 Auswertung und Vergleich 37

4 Abbildung von Alf zu Java, C# und QML **41**

4.1 Primitive Typen 41

4.2 Collectiontypen 43

4.3 Wertspezifikation und Objektnavigation 45

4.4 Operationen 49

4.5 Kontrollfluss 53

4.6 Variablen und Funktionsaufrufe 57

5 Integration in Fokus!MBT **62**

6 Fallstudie **70**

7 Auswertung **82**

7.1 Ausblick 83

1 Einleitung und Grundlagen

1.1 Modellbasiertes Testen

In der Softwareentwicklung sind Modelle mittlerweile allgegenwärtig. Modelle werden eingesetzt, um Sachverhalte zu veranschaulichen und Artefakte automatisch zu generieren. Dabei werden Modelle auf verschiedenen Ebenen eingesetzt. Der Einsatz von Modellen reicht vom schnellen Skizzieren von Klassendiagrammen zur Kommunikation zwischen zwei Entwicklern, über die Visualisierung der Softwarearchitektur bis hin zur vollständigen Modellierung des gesamten, zu bauenden Systems.

Als Modellierungssprache hat sich die Unified Modeling Language (UML)[1] der Object Management Group (OMG)[2] durchgesetzt. Modelle werden in der UML hauptsächlich grafisch mithilfe Diagrammen erstellt. Dafür UML bietet verschiedene Diagrammarten für die Struktur- und Verhaltensmodellierung. Zur strukturellen Modellierung gehört sowohl die statische Struktur, die zum Beispiel mit Klassen- oder Komponentendiagrammen definiert wird, als auch die Struktur zur Laufzeit, die mittels Objekt- oder Verteilungsdiagrammen angegeben wird. Verhalten kann in der UML zum Beispiel mit Aktivitätsdiagrammen oder Zustandsdiagrammen angegeben werden. Die UML versucht für alle von ihr definierten Konstrukte eine Semantik festzulegen. Die festgelegte Semantik sorgt dafür, dass jeder, der mit der UML vertraut ist, das gleiche Verständnis von einem Diagramm hat.

Modelle können aber nicht nur zur Kommunikation benutzt werden. Aus ihnen lässt sich auch Code generieren. Ein gängiges Einsatzgebiet von Codegenerierung ist das Generieren von Klassen aus einem Klassendiagramm mit Stub-Implementierungen für die Operationen. Mit der Zeit haben sich verschiedene Bewegungen in der Softwareentwick-

[1] http://www.uml.org
[2] http://www.omg.org

lung gebildet, die Modelle als den zentralen Bestandteil der Entwicklung sehen. Beispiele dafür sind die Model-Driven Architecture der OMG [3] oder das Model-Driven Softwaredevelopment aus dem Eclipseumfeld mit dem Eclipse Modeling Framework[4].

Beim modellbasierten Testen (MBT) wird das Konzept der modellbasierten Softwareentwicklung auf den Bereich des Softwaretestens übertragen. MBT stellt Modelle in den Mittelpunkt des Testprozesses[16]. Alle für den Testprozess relevanten Informationen werden dabei in einem Testmodell abgelegt. Benötigte Artefakte können dann per Modelltransformation aus dem Modell erzeugt werden. Dies betrifft zum Beispiel Berichte oder Testfälle. Als großer Vorteil gegenüber traditionellen Testansätzen wird die Automatisierung durch Testgenerierung gesehen.

Bei der Testgenerierung werden Testfälle und Testdaten generiert. Hierfür wird ein Testentwurfsmodell des zu testenden Systems (System under test, SUT) erstellt[16]. Dieses modelliert das abstrakte Verhalten des SUTs an dessen Schnittstelle. Meist werden Zustandsdiagramme für das Testentwurfsmodell verwendet. Mithilfe von Testfallgeneratoren werden dann aus den Zustandsdiagrammen Testfälle generiert. Der Testfallgenerator exploriert das Zustandsdiagramm und schneidet einzelne Testfälle aus diesem heraus. Dabei versucht der Testfallgenerator Testfälle zu erstellen, die den eingestellten Anforderungen genügen. Abdeckungskriterien können die besuchten Zustände, die gemachten Zustandsübergänge oder eine Anforderungsabdeckung sein. Die erstellten Testfälle werden zum Beispiel als Interaktionsdiagramme wieder in das Modell integriert. Der Tester muss sich die Testfälle nicht mehr selbst ausdenken, um die gewünschten Abdeckungskriterien zu erreichen. Des Weiteren ist die Erzeugung der Testfälle transparent für andere Tester und wiederholbar. Die generierten Testfälle beschreiben den Testfall auf einer logischen Ebene. Ein Adapter ist notwendig um diese logischen Testfälle technisch zu implementieren[16].

1.2 UML/UTP

Die UML erhebt den Anspruch eine Modellierungssprache zu sein, die so allgemein ist, dass sie für möglichst viele Fachdomänen anwendbar ist. Gleichzeitig sorgt die Standardisierung von UML, dass UML-Modelle zwischen Modellierungswerkzeugen ausgetauscht

[3] http://www.omg.org/mda/
[4] http://www.eclipse.org/modeling/emf/

werden können und diese auch die gleichen Modellelemente unterstützen[Vgl. 11, Kap. 1-2]. Um allen Anforderungen gerecht zu werden, gibt es in der UML 14 verschiedene, teilweise sehr ähnliche Diagrammarten. Auch bei der Modellierung werden dem Benutzer viele Freiheiten gelassen. Zum Beispiel können in einem Klassendiagramm Klassen entweder per Assoziation oder per Attribut ein Objekt einer anderen Klasse referenzieren.

Die UML kann auch für verschiedene Fachdomänen und Plattformen mithilfe von Profilen spezialisiert und erweitert werden. Profile bestehen aus Paketimporten für die Teile der UML, die von dem Profil unterstützt werden und Stereotypen, um die generischen UML Konstrukte mit Konzepten aus der gewählten Domäne oder Plattform zu verknüpfen. [11, Kap. 18] Die UML bietet jedoch von sich aus keine besonderen Mittel für die Modellierung des Testens.

Aus diesem Grund wurde das UML Testing Profile (UTP)[5] entwickelt. Es erweitert die UML um Konzepte des Softwaretestens.

Das UTP wurde so konzipiert, dass es sich möglichst nah an die UML hält und viel davon wiederverwertet [14]. Dadurch kann das UTP leicht mit anderen UML-Erweiterungen benutzt werden, wie zum Beispiel MARTE oder SysML. MARTE ist ein UML-Profil für Echtzeitsysteme und eingebettete Systeme. Es erweitert die UML insbesondere um zeitliche Aspekte wie Performance und Scheduling sowie Ressourcenverwaltung[Vgl. 13, Kap. 1]. SysML ist eine auf der UML basierende Modellierungssprache für die Systementwicklung. Es benutzt einige UML Diagrammarten, definiert aber auch eigene wie zum Beispiel das Anforderungsdiagramm[9].

Mithilfe von UTP ist es möglich die Testausführungsumgebung, Testfälle, Testdaten und Testmanagement zu modellieren.

Die Testumgebung wird mit Komponentendiagrammen und Klassendiagrammen beschrieben. Die Klassendiagramme definieren die verwendeten Klassen und Schnittstellen des zu testenden Systems, der Testkomponenten, der Testkontexte und der Testausführungsumgebung. Testkomponenten sind die Testtreiber. Sie stimulieren das SUT und beobachten seine Reaktion. Testkomponenten bieten den konkreten Anschluss an das SUT. Ein Testkontext bündelt eine Menge von Testfällen. Die Testfälle sind Operationen eines Testkontextes die mit dem Stereotyp *«TestCase»* annotiert sind. Abbildung 1.1 verdeutlicht dies. Der Testkontext ist die übergeordnete Klasse. Sie beinhaltet sowohl das SUT als auch die

[5] http://utp.omg.org/

Testkomponenten und die Testfälle. Das SUT ist der Audiokonverter. Er hat zwei Ports jeweils einen für den Ein- und Ausgang. Die Testkomponente ist der *AudioConverterAdapter*, welcher die zu dem SUT passenden Ports hat. Die Testfälle des Testkontextes benutzen nun den *AudioConverterAdapter*, um mit dem SUT zu kommunizieren. Sowohl die Stimulation als auch das Beobachten des SUTs geschieht über die Testkomponente.

Testfälle können mit den aus der UML stammenden Interaktionsdiagrammen, Zustandsdiagrammen oder Aktivitätsdiagrammen beschrieben werden. UTP erweitert diese um Zeitkonzepte, Logaktionen, Zeitzonen und Ausnahmebehandlung.

Auch Testdaten können modelliert werden. Dazu bietet UTP mit dem *DataPool* die Möglichkeit Datenpartitionen zu definieren. Datenpartitionen in UTP werden auch Äquivalenzklassen genannt. Eine Äquivalenzklasse ist eine Menge von Eingabewerten, die bei dem SUT gleichartiges Verhalten auslösen sollen. Weiterhin bietet UTP die Möglichkeit Daten wieder zu verwenden. Dazu können Datenpartitionen importiert und dann modifiziert werden. Schließlich bietet UTP noch Wildcardliterale. Das Wildcardliteral „?“ steht für einen beliebigen Wert. Das Literal „*“ steht für einen beliebigen oder keinen Wert (*null* in Java). Für das Testmanagement bietet UTP die Möglichkeit Testziele zu definieren. Diese können mit Anforderungen, Anwendungsfällen oder SysML-*Requirements* verknüpft werden.

1.3 Fokus!MBT

Fokus!MBT[6] ist ein Werkzeug für modellbasiertes Softwaretesten auf Basis von UTP. Es wird am Fraunhofer Institut für offene Kommunikationssysteme (FOKUS)[7] entwickelt. Mit Fokus!MBT lassen sich nicht nur die verschiedenen Testmodelle aus UTP definieren. Darüber hinaus lassen sich auch Testdaten, Testfälle und Testcode generieren. Nach dem Testdurchlauf kann Fokus!MBT die Logs des benutzten Testausführungswerkzeugs in die Modelle importieren. Dazu bietet Fokus!MBT eine Anbindung an externe Werkzeuge wie zum Beispiel den Spec Explorer als Testfallgenerator für die Testfallgenerierung.

Fokus!MBT basiert auf Eclipse Papyrus. Eclipse Papyrus[8] eine Entwicklungsumgebung für modellbasierte Softwareentwicklung mithilfe des Eclipse Modeling Frameworks. Mo-

[6] http://www.fokusmbt.com/index.html
[7] http://www.fokus.fraunhofer.de/
[8] http://eclipse.org/papyrus/

delle können mit Papyrus in Form von Diagrammen oder textuell erstellt und bearbeitet werden. Papyrus liefert Diagrammeditoren für UML-Diagramme mit Unterstützung von MARTE und SysML. Außerdem unterstützt Papyrus UML-Profile, sodass auf Basis von Papyrus Diagramme und Modelle für eigene Sprachen entwickelt werden können. Fokus!MBT macht sich dies zunutze um Editoren für die Testmodelle mit dem UML Testing Profil bereitzustellen.

Fokus!MBT nutzt für alle Testartefakte UTP Modelle als einheitliches Format. Somit lassen sich Werkzeuge miteinander kombinieren, die unterschiedliche Ein- und Ausgabeformate haben. Die UTP Modelle werden durch Modelltransformation in die Eingabeformate der Werkzeuge umgewandelt und die Ergebnisartefakte in Modelle rücktransformiert.

1.4 Problembeschreibung

Damit Fokus!MBT die Testfallgeneratoren nutzen kann, muss es die Testentwurfsmodelle aus UTP in die Eingabemodellsprache der Testfallgeneratoren übersetzen. Die Eingabe vieler Testfallgeneratoren basiert auf Zustandsmaschinen mit Wächterausdrücken und Effektbeschreibungen. Testfallgeneratoren, die implizit oder explizit auf Zustandsmaschinen basieren, sind beispielsweise Graphwalker[9], Spec Explorer[10], Conformiq[11], OSMO[12], NModel[13], JSXM[14], JTorX[15], PyModel[16], TestOptimal[17], Uppaal CoVer[18] oder STG[19].

Ein Effekt ist ein ausführbares Stück Code. Es wird ausgeführt, wenn der Zustandsübergang, an dem der Effekt steht, ausgeführt wird. Effekte sind in imperativen Programmiersprachen eine Menge von Anweisungen und sind häufig nebeneffektbehaftet. Häufig ändern Sie den Zustand des Kontextobjektes der Zustandsmaschine.

[9] http://graphwalker.org/
[10] http://research.microsoft.com/en-us/projects/specexplorer/
[11] http://www.conformiq.com/
[12] https://github.com/mukatee/osmo
[13] http://www.codeplex.com/NModel
[14] http://www.jsxm.org/
[15] https://fmt.ewi.utwente.nl/redmine/projects/jtorx/wiki/
[16] http://staff.washington.edu/jon/pymodel/www/
[17] http://testoptimal.com/
[18] http://www.hessel.nu/CoVer/overview.php
[19] http://www.irisa.fr/prive/ployette/stg-doc/stg-web.html

Wächter sind logische Ausdrücke, die einschränken, welche Zustandsübergänge von einem Zustand aus genommen werden dürfen. Da sie Ausdrücke sind, sind sie eine Untermenge von den Anweisungen, die auch für Effektbeschreibungen benutzt werden. Jede hinreichend allgemeine Effektbeschreibungssprache muss in der Lage sein solche Ausdrücke zu definieren, da ansonsten keine boolschen Werte für Bedingungen und Variablen definiert werden könnten. Deswegen reicht es aus, sich in dieser Arbeit auf Anweisungen für Effektbeschreibungen zu konzentrieren.

Die Menge der Effekte und Wächter der Zustandsübergänge einer Zustandsmaschine bilden das konkrete Verhalten dieser Zustandsmaschine. Wächterausdrücke bilden zusammen mit den Zuständen und Zustandsübergängen den Kontrollfluss des Verhaltens der Zustandsmaschine. Über Effekte kann eine Zustandsmaschine Berechnungen ausführen, den Zustand seiner Attribute ändern und mit der Umwelt kommunizieren. Daher sind Effekte und Wächter von besonderer Bedeutung für modellbasiertes Softwaretesten.

Beim Testfallgenerator Graphwalker werden die Zustandsmaschinen in der Sprache GraphML[20] definiert. GraphML ist eine Graphbeschreibungssprache, die auf XML basiert. Die Wächterausdrücke und Transitionseffekte des Zustandsautomaten werden in Java verfasst. Zusätzlich können die Zustände Assertions haben, die testen, ob sich das SUT im angenommenen Zustand befindet.

Der Spec Explorer ist ein Testfallgenerator der Firma Microsoft. Beim Spec Explorer wird eine Zustandsmaschine entweder explizit durch Angeben der Transitionen oder implizit durch Angabe eines Modellprogramms definiert. Wächterausdrücke und Effekte werden dabei als *Regeln* in IL-basierten Sprachen wie C# definiert. Die *Intermediate Language* (IL) ist die Assemblersprache der .NET Runtime auf Windowsmaschinen. *Regeln* sind die Methoden einer Klasse des Modellprogramms. Sie sind die mit der [Rule]-Annotation versehen. Aus den *Regeln* werden dann sogenannte *Maschinen* erzeugt. Durch Exploration des Zustandsraumes der Maschinen des Modellprogramms wird ein Zustandsautomat erzeugt, aus dem Testfälle herausgeschnitten werden.

Ein weiterer Testfallgenerator ist Conformiq. Hier werden die Zustandsmaschinen grafisch definiert. Die grafische Notation entspricht in etwa den UML Zustandsdiagrammen. Die Effektbeschreibungen werden in der Conformiq Modeling Language (QML) geschrieben. QML ist eine objektorientierte Sprache, die stark von Java, C++ und C# beeinflusst

[20] http://graphml.graphdrawing.org/

wurde.

Mithilfe der UML Zustandsdiagramme können die Zustandsautomaten der Testfallgeneratoren generatorunabhängig spezifiziert werden. Problematisch ist jedoch die Beschreibung der Effekte und Wächter. Effekte können mit UML Mitteln über Interaktionsdiagramme und Aktivitätsdiagramme definiert werden. Dies ist jedoch unpraktisch, da für jeden Effekt und jeden Wächter ein Diagramm angefertigt werden müsste. Der gebräuchlichere Weg, um das Verhalten der Zustandsdiagramme zu beschreiben, ist der, die Trigger, Wächter und Effekte im Diagramm textuell an die Transitionen zu schreiben. Das Format, dass die OMG hierfür vorsieht, hat die Form: `Trigger [Wächter] /Effekt`. Die Wächter und Effekte können in einer beliebigen Programmiersprache oder auch natürlichsprachlich verfasst werden. Als textuelle Wächter- und Effektbeschreibungssprache kann hier die jeweilige Eingabesprache der Testfallgeneratoren verwendet werden. So könnte der Wächter- und Effektcode zur Verwendung des Spec Explorers in C# verfasst werden. Code in diesen Sprachen lässt sich dann über UML *OpaqueExpressions* textuell in das Testentwurfsmodell integrieren. *OpaqueExpressions* binden einen Ausdruck als reinen Text in ein Modell ein, ohne den Ausdruck im Modell auszuwerten. Auf diese Weise können beliebige Programmiersprachen oder natürliche Sprache für Ausdrücke verwendet werden. Nachteilig bei der direkten Verwendung der Eingabesprachen der Generatoren ist, dass jeder Testfallgenerator seine eigene Eingabesprache hat. Die Effektbeschreibungen würden dadurch plattformabhängig. Wenn der Testfallgenerator gewechselt wird oder mehrere verschiedene Testfallgeneratoren zurate gezogen werden sollen, dann muss für jeden Testfallgenerator die Effektbeschreibung neu geschrieben werden. Dies ist fehleranfällig und schlecht wartbar, da alle Effektbeschreibungen synchron gehalten werden müssen. Die Abstraktion von konkreten Werkzeugen, die UTP bietet, geht dabei verloren. Die unterschiedlichen Eingabesprachen der Testfallgeneratoren stellen ein Problem dar.

1.5 Aufgabenstellung

Für die Lösung dieses Problems gilt es eine textuelle Sprache für die Beschreibung von Effekten und Wächtern zu finden. Die Sprache sollte plattformunabhängig sein, damit sie leicht in die Eingabesprachen verschiedener Testfallgeneratoren übersetzt werden kann. Plattformabhängige Sprachen wie zum Beispiel Java haben starke Abhängigkeiten zu

ihren Laufzeitumgebungen und -bibliotheken. Idiomatischer Code in diesen Sprachen verfolgt spezifische Problemlösungsstrategien, die von der Laufzeitumgebung unterstützt wird. Daher kann es unter Umständen schwerfallen, Code einer plattformabhängigen Sprache in eine andere Sprache zu übersetzen. Plattformunabhängige Sprachen müssen high-leveliger und allgemeiner gehalten werden, damit sie für eine große Auswahl an Plattformen übersetzt werden können.

Die Effektbeschreibungssprache soll in Zustandsdiagrammen in UML Modellen verwendet werden können. Daher sollte sie Konzepte der UML verstehen und damit umgehen könne. Zu diesen Konzepten gehören zum Beispiel Klassen, Pakete, Operationen, Signale und Assoziationen. Wenn eine Effektbeschreibungssprache diese Konzepte nicht unterstützt, so schränkt dies die UML Elemente ein, die in diesem UML Modell verwendet werden können. Das erklärt sich daraus, dass Modellelemente die nicht in den Effekten des Zustandsdiagramms eines Testfallentwurfsmodells verwendet werden können, keinen Einfluss auf die generierten Testfälle haben, da sie nicht zum modellierten Zustand des SUTs beitragen.

Ein weiterer, nicht zu vernachlässigender Gesichtspunkt für die Wahl einer Sprache ist die Werkzeugunterstützung. Ein Editor mit Syntaxhervorhebung und Autovervollständigung trägt viel zum Komfort und der Effizienz bei der Benutzung einer Sprache bei. Das am weitesten verbreitete Werkzeug für die Softwareentwicklung und UML Modellierung ist Eclipse. Eclipse bildet auch die Grundlage für viele kommerzielle Werkzeuge im Bereich der Softwareentwicklung. Daher sind vor allem die im Eclipseumfeld vorhandenen Editoren von Interesse. Es wäre wünschenswert, wenn es Editoren in Eclipse für die gewählte Sprache gäbe.

Wenn eine solche textuelle Sprache ausgewählt wurde, dann muss eine konkrete Abbildung der plattformunabhängigen Sprache in die Eingabesprachen der Testfallgeneratoren gefunden werden. Da im Rahmen dieser Arbeit die Testfallgeneratoren Conformiq, Spec Explorer und Graphwalker betrachtet werden, sind die Eingabesprachen C#, Java und QML. Es ist vorteilhaft, wenn die plattformunabhängige Sprache ähnliche Konzepte wie die Eingabesprachen verfolgt, sodass die Übersetzung von Code in dieser Sprache in die Eingabesprachen der Testfallgeneratoren möglichst einfach ist.

1.6 Gliederung

In Kapitel 2 wird ein Überblick über die Eingabesprachen verschiedener Testfallgeneratoren gegeben und Gemeinsamkeiten und Unterschiede erläutert. In Kapitel 3 folgt ein Vergleich textueller, UML-basierter Sprachen im Hinblick auf die Eingabesprachen der Testfallgeneratoren. Es wird sich dort zeigen, dass nur Alf als plattformunabhängige Effektbeschreibungssprache in Betracht kommt.

Als Nächstes wird in Kapitel 4 gezeigt, wie Alf konkret auf die Eingabesprachen abgebildet werden kann. Die Implementierung dieser Abbildung und die Integration in Fokus!MBT werden in Kapitel 5 vorgestellt. An einer Fallstudie wird in Kapitel 6 der Einsatz von Alf als Effektbeschreibungssprache unter Benutzung eines Testfallgenerators demonstriert. Abschließend folgen das Fazit und ein Ausblick und offene Probleme.

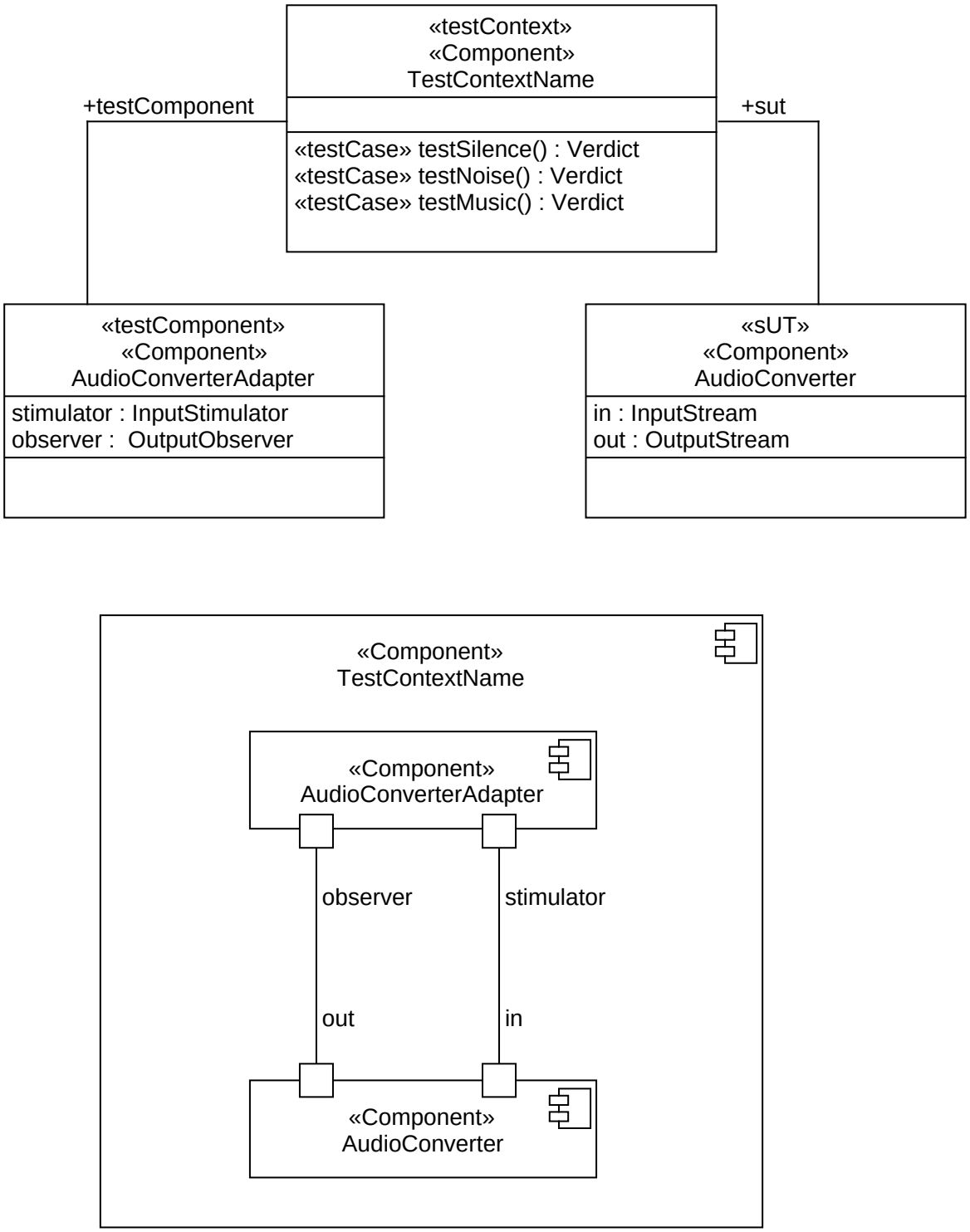


Abbildung 1.1: Anwendung von UTP am Beispiel eines Audiokonverters.

2 Die Eingabesprachen der Generatoren

In diesem Abschnitt werden exemplarisch drei verbreitete Testfallgeneratoren und deren Eingabesprachen vorgestellt. Die meisten anderen Testfallgeneratoren basieren auf Zustandsmaschinen und funktionieren ähnlich wie die drei hier vorgestellten.

Eine Sprache für die Effektbeschreibung sollte sich möglichst gut in die Eingabesprachen der Testfallgeneratoren übersetzen lassen. Dafür ist es sinnvoll, diese Sprachen miteinander zu vergleichen und Gemeinsamkeiten und Unterschiede herauszuarbeiten.

2.1 Conformiq Designer

Der Conformiq Designer ist ein Testfallgenerator der Firma Conformiq. Das Programm basiert auf Eclipse und ist alleinstehend oder als Plug-in verfügbar. Die Dokumentation zu diesem findet man im Evaluation Guide[2] und dem Handbuch[3]. Das Testentwurfsmodell wird mit UML Zustandsdiagrammen und einer Java-ähnlichen Effektbeschreibungssprache namens Conformiq Modeling Language (QML) geschrieben. Die Schnittstellen des SUTs werden in QML definiert. Ein Beispiel dafür ist in 2.1 zu sehen. Der `system`-Block definiert die Schnittstelle des SUTs. Er enthält ein oder mehrere Portdefinitionen. Ports sind Nachrichtenkanäle über die Nachrichten eines bestimmter Typen versendet werden können. Ein Port kann eingehend, ausgehend oder intern sein, wobei die Richtungsangaben aus Sicht des SUTs zu verstehen sind. Auf einem eingehenden Port empfängt das SUT Nachrichten und auf einem ausgehenden Port versendet das SUT Nachrichten. Interne Ports sind bidirektional. Sie verbinden verschiedene Threads des SUTs. Im Beispiel hat das SUT einen eingehenden Port namens *userIn* über den das SUT Nachrichten vom Typ *Login*, *Message* und *Logout* empfangen kann. Über den ausgehenden Port können vom SUT nur *Message*-Nachrichten verschickt werden.

Das modellierte Verhalten des Systems wird mithilfe von Zustandsdiagrammen beschrie-

```
system {
        Inbound userIn : Login, Message, Logout;
        Outbound userOut : Message;
}

record Login {
        String username;
        String password;
}

record Logout {
        String username;
}

record Message {
        String username;
        String message;
}
```

Abbildung 2.1: Ein Beispiel für die Schnittstellendefinition des SUTs in Conformiq.

ben. Aktionen, Effekte und Wächterausdrücke werden in QML angegeben. Die Zustände des Automaten können Entry- und Exitcode haben. Die darin definierten Aktionen werden beim Betreten beziehungsweise Verlassen des Zustandes ausgeführt. Außerdem gibt es zustandsinterne Transitionen. Transitionen können wie in der UML Auslöser, Wächter und Effekte haben. Der Auslöser eines Zustandsübergangs ist bei Conformiq entweder eine abgelaufene Zeitspanne oder eine empfangene Nachricht auf einem eingehenden Port des SUTs. Zusätzlich können den Transitionen Verweise auf Anforderungen mitgegeben werden, um bei den generierten Testfällen ein Maß für die Anforderungsüberdeckung zu haben. Die Wächter und Effekte können direkt im Diagramm an die Transition geschrieben werden. Alternativ können die Transitionen im Diagramm auf implementierende Funktionen in QML-Dateien verweisen.

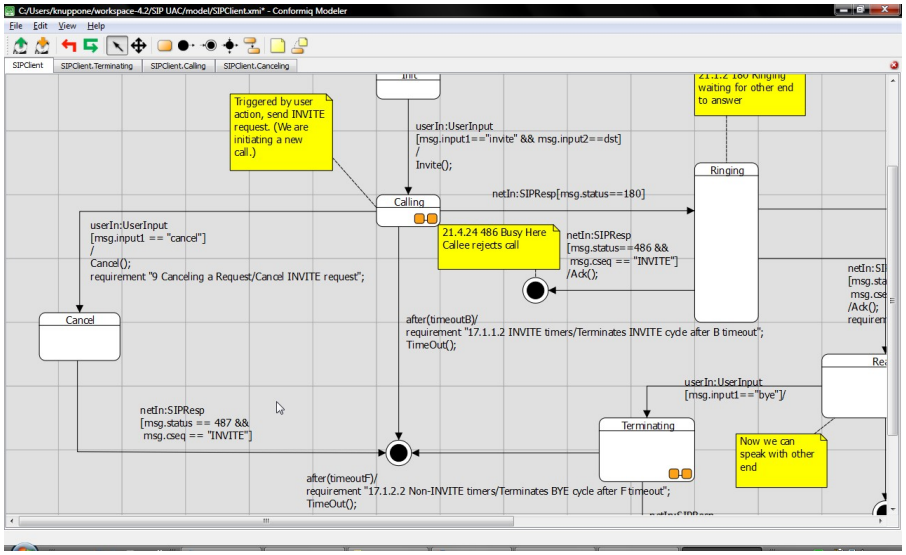


Abbildung 2.2: Beispiel eines Zustandsdiagramms im Conformiq Modeler.

Abbildung 2.2 zeigt ein Beispiel eines Zustandsdiagrammes in Conformiq. Es ist dem Conformiq Evaluation Guide[2] entnommen. Das Beispiel zeigt das Zustandsdiagramm eines Voice-over-IP-Telefons. Zu sehen ist, dass die Zustandsdiagramme in Conformiq der UML-Notation folgen. Transitionen sind in der Form `Auslöser [Wächter] /Effekt` beschriftet. Zusätzlich können Transitionen durch das Schlüsselwort *requirement* an Anforderungen geknüpft werden.

QML ist eine Java-ähnliche Sprache mit starken Einflüssen von C# und C++. QML ist objektorientiert. Klassen funktionieren ähnlich wie in Java. Es gibt lediglich Einfachvererbung. Es können aber beliebig viele Interfaces implementiert werden. Allerdings haben Klassen immer die Sichtbarkeit *public*. Interfaces funktionieren genau wie in Java. Innere Klassen entsprechen den statischen, inneren Klassen in Java. Anders als in Java gibt es die Möglichkeit wie in C++ globale Funktionen und Variablen zu definieren. Es gibt keine Pakete wie in Java. Alle Typdeklarationen und globalen Funktionen befindet sich in einem globalen Namensraum. Die primitiven Datentypen entsprechen denen von Java. Allerdings ist die Größe der Typen *int*, *long*, *float* und *double* nicht explizit definiert. Strings und Arrays sind wie in Java Referenztypen.

Im Gegensatz zu Java gibt es Wertetypen in Form von *Records*. Sie ähneln den *structs* aus C#. *Records* werden für den Nachrichtenaustausch über Ports benötigt. Es sind nur

Records über Ports versendbar. Der Grund dafür ist, dass Nachrichten für das Versenden über Ports serialisiert werden müssen. *Records* sind darauf ausgelegt, serialisiert werden zu können. Deshalb sind nur primitive Typen, Strings, Arrays und andere Records als Felder von *Records* erlaubt. Felder können als optional gekennzeichnet werden. Auch Standardwerte für Felder sind möglich. *Records* können nicht von Klassen, sondern nur von anderen *Records* erben. Sie können auch keine Schnittstellen implementieren. Ähnlich wie in C++ gibt es Unions in QML. Allerdings sind diese *tagged unions*. Bei diesen ist es ein Laufzeitfehler, wenn nicht benutzte Felder gelesen werden. Ein weiterer Unterschied zu C++ ist, dass die Felder eines Unions in QML nicht den gleichen, überlappenden Speicherbereich benutzen, sondern die Felder nebeneinander im Speicher liegen.

Templates funktionieren in QML genau wie in C++. Für jede Instanziierung mit einem konkreten Typ wird der Template-Code für den konkreten Typ generiert. Des Weiteren bietet QML Operatorenüberladung wie in C++ und C# und *typedefs* wie in C++. Für primitive Typen gibt es keine *boxed* Variante wie in Java. Dafür können primitive Typen wie in C# *nullable* sein. Felder und Variablen, die als *nullable* gekennzeichnet sind, können auf null gesetzt werden, auch wenn der primitive Typ des Feldes oder der Variable nicht den gültigen Wert *null* hat.

Der Arbeitsablauf mit dem Conformiq sieht wie folgt aus. Als Erstes wird das zu testende System modelliert. Dazu gehören die Portdefinitionen im *system*-Block und der Zustandsautomat. Danach wird aus dem Modell mithilfe der definierten Testabdeckungseinstellungen eine Anzahl an Testfällen generiert. Diese können dann begutachtet werden. Dafür liefert der Conformiq Designer zahlreiche Statistiken über die Testabdeckung. Aus den Conformiq-internen Testfällen werden mithilfe von Scripting-Backends Stub-Code generiert. Conformiq liefert hier zahlreiche Backends, zum Beispiel für TTCN-3, Java, C++, Perl, Html oder Excel. Zum Schluss muss der Stub-Code noch für die Anbindung an das SUT implementiert werden.

2.2 Graphwalker

Der Graphwalker ist eine Java-Bibliothek für modellbasiertes Testen. Graphwalker ist Opensource. Die Idee hinter Graphwalker ist die, dass Graphwalker einen Graphen traversiert und dabei Methoden einer Java-Klasse aufruft. Diese Java-Klasse erbt dabei

von ModelApi und bildet zusammen mit dem Graphen das Testentwurfsmodell. Im Folgenden wird diese Klasse als die Modellklasse bezeichnet. Der Graph wird implizit als Zustandsautomat aufgefasst. Die Art der Traversierung des Graphen kann dabei festgelegt werden. So kann beispielsweise der Graph zufällig durchlaufen werden oder mittels des A*-Algorithmus der kürzeste Pfad zwischen zwei Knoten genommen werden.

Der Graph wird zur Laufzeit aus einer GraphML-Datei geladen. Ein einzelner Knoten mit nur einer ausgehenden Kante und dem Namen *Start* wird als Startzustand betrachtet. Knoten müssen mit einem Namen beschriftet sein. Die Namen müssen allerdings nicht eindeutig sein. Zwecks Wiederverwendbarkeit können mehrere Knoten denselben Namen haben. Zu jedem Knoten muss es eine Methode in der Modellklasse geben, die genauso heißt. Wenn der Graphwalker diesen Knoten durchläuft, wird die entsprechende Methode aufgerufen. Die Knoten entsprechen den Zuständen eines Zustandsautomaten. In ihnen soll das SUT einen bestimmten Zustand haben. Dementsprechend soll in den Methoden der Zustand des SUTs überprüft werden. Im Fehlerfall soll eine *Assertion* fehlschlagen. Diese wird vom Graphwalker aufgefangen und dem Aufrufer berichtet.

Die Kanten des Graphen entsprechen den Zustandsübergängen des Zustandsautomaten. Sie sind mit einem Namen beschriftet. Zusätzlich können Wächterausdrücke und Effekte in der Beschriftung der Kante angegeben werden. Sie werden in normaler Java-Syntax geschrieben und mittels Java BeanShell interpretiert. Zu jeder Kante gibt es eine Methode mit dem gleichen Namen in der Modellklasse. Der Zustandsübergang entspricht einer Aktion auf dem SUT. In der Methode soll das SUT stimuliert werden, um es in den Zustand zu bringen, der von dem nächsten Zustand angenommen und überprüft wird.

Abbildung 2.3 zeigt ein Beispiel eines Graphen, der vom Graphwalker als Zustandsautomat benutzt wird. Es ist Konvention, dass die Knotennamen mit „v_“ und die Kantennamen mit „e_“ beginnen. Die Kantenbeschriftung ist ähnlich wie bei UML Zustandsdiagrammen aufgebaut. Auf den Kantennamen folgt in eckigen Klammern der Wächterausdruck für die Zustandsübergänge. Hinter einem Schrägstrich folgt die Effektbeschreibung. Die dazugehörige Java-Klasse ist in Abbildung 2.4 zu sehen. Diese Java-Klasse bildet den Kontext für die Zustandsautomaten. Die Wächter und der Effektcode des Zustandsautomaten können die Methoden und Felder dieser Klasse benutzen. Zum Beispiel benutzt der Graph im Beispiel das Feld *LoginAttempts* der Klasse *ExampleModel*.

Wie der Graphwalker benutzt wird, ist alleine dem Benutzer überlassen. Graphwalker

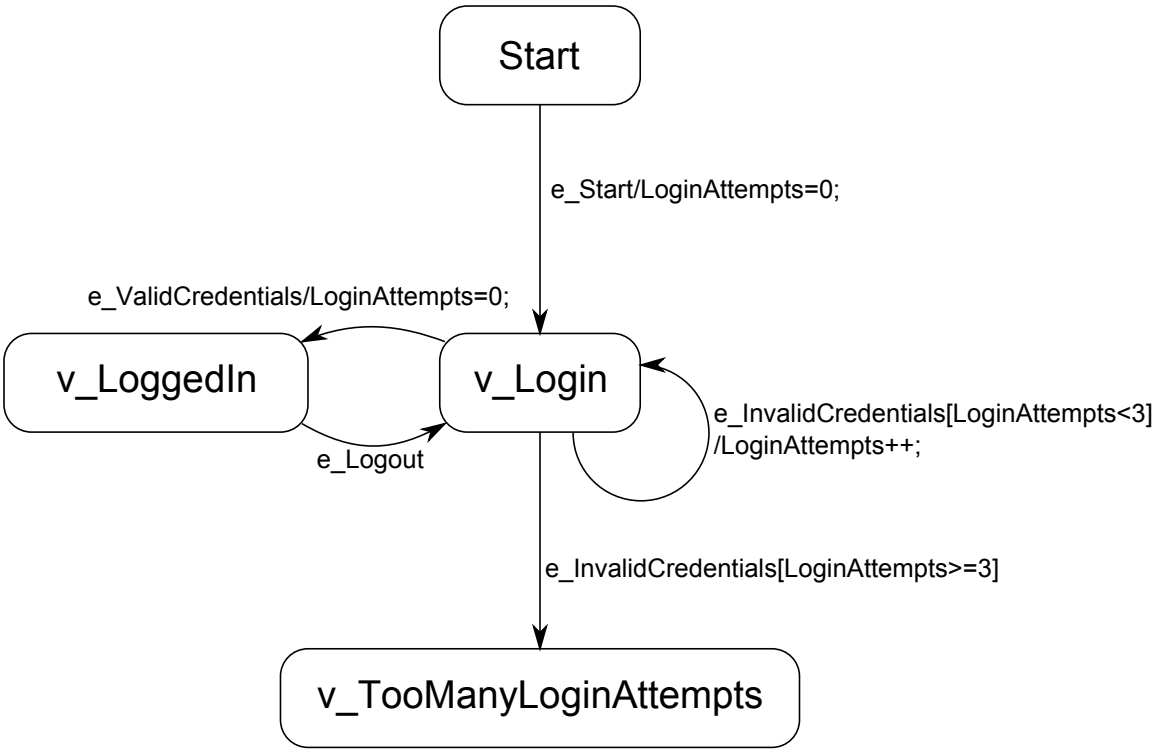


Abbildung 2.3: Beispiel eines Graphen für den Graphwalker.

ruft lediglich die Methoden der Modellklasse auf. Alternativ gibt der Graphwalker die Namen der Funktionen aus, die er aufrufen würde. Der Benutzer kann sich die Methodenaufrufe speichern oder tatsächlich das SUT stimulieren. Die Anbindung an das SUT muss der Benutzer auch selbst übernehmen. In den Beispielen zu Graphwalker wird zum Beispiel das Testframework Selenium[1] benutzt, um eine Webseite zu testen. Die Methoden der Modellklasse nutzen Selenium, um einen gewählten Webbrowser zu steuern und Informationen über die aktuelle Webseite auszulesen.

2.3 Spec Explorer

Der Spec Explorer ist ein von Microsoft entwickeltes Werkzeug für modellbasiertes Testen. Der Spec Explorer ist der Nachfolger des ebenfalls bei Microsoft entwickelten AsmL-T[4].

[1] http://www.seleniumhq.org/

```java
public class ExampleModel
        extends org.graphwalker.multipleModels.ModelAPI {

        int LoginAttempts;

        public ExampleModel(File model, boolean efsm,
                PathGenerator generator, boolean weight) {
                super(model, efsm, generator, weight);
        }

        public void e_Start() { }
        public void e_InvalidCredentials() { }
        public void e_ValidCredentials() { }
        public void e_Logout() { }

        public void v_Login() {
                Assert.assertTrue(LoginAttempts>=0
                        && LoginAttempts<=3);
        }
        public void v_LoggedIn() {
        }
        public void v_TooManyLoginAttempts() {
                Assert.assertTrue(LoginAttempts>=3);
        }
}
```

Abbildung 2.4: Die Java-Klasse zum Beispielgraphen für den Graphwalker.

AsmL-T basiert auf der high-level Spezifikationssprache AsmL[2], welche vom Spec Explorer ebenfalls unterstützt wird. Neben AsmL unterstützt der Spec Explorer auch weitere Sprachen, die auf der *Intermediate Language* (IL) der .Net Plattform basieren. Eine solche

[2] http://research.microsoft.com/en-us/projects/asml/

Sprache ist zum Beispiel C#. Der Spec Explorer ist sowohl alleinstehend benutzbar als auch als Erweiterung für Microsofts Visual Studio Entwicklungsumgebung. Die weiteren Informationen zu dem Spec Explorer in diesem Abschnitt sind der Dokumentation[15] entnommen. Zur Spezifikation des Verhaltens des SUTs wird erst ein Modellprogramm geschrieben. Das Modellprogramm kann in einer beliebigen IL-basierten Sprache oder AsmL geschrieben werden. Es beschreibt das abstrakte Verhalten des SUTs, das an der Schnittstelle beobachtet werden kann.

Mithilfe einer in Cord-Skript geschriebenen Konfiguration exploriert der Spec Explorer den Zustandsraum des Modellprogramms. Der explorierte Zustandsraum lässt sich als Graph in Visual Studio visualisieren. Aus dem Graphen werden dann, ebenfalls mithilfe der Konfiguration, Szenarien herausgeschnitten. Dieser Vorgang wird *Scenario Slicing* genannt. Aus diesen Szenarien werden die konkreten Testfälle für das ausgewählte Unittesting-Framework generiert. Es werden von Hause aus das in Visual Studio integrierte Unittesting-Framework VSTT und das eigenständige NUnit[3] unterstützt. Anstelle von Modellprogrammen lassen sich Szenarien im Cord-Skript auch direkt als eine Folge von Aktionen definieren. Weiterhin ist es möglich, mithilfe einer weiteren Visual Studio Erweiterung Szenarien aus UML Interaktionsdiagrammen zu generieren. Die generierten Testfälle testen, ob die von SUT erhaltenen Daten mit dem Zustand des Modellprogramms übereinstimmen.

Das SUT wird mithilfe von Regeln beschrieben. Regeln werden als Regelmethoden im Modellprogramm implementiert. Der Aufruf einer Regel entspricht dem Aufruf des SUTs. Eine Regel modelliert die Zustandsänderungen des SUTs beim Aufruf und die zu erwartenden Rückgabewerte der aufgerufenen Operation. Damit übernimmt das Modellprogramm die Rolle des Testorakels. Zusätzlich kann eine Regel definieren, wann sie aufgerufen werden darf und welche Anforderungen beim ihrem Aufruf abgedeckt werden. Die Regeln des Modellprogramms entsprechen der Schnittstelle des SUTs. Für jede Operation auf dem SUT gibt es eine Regel. Die Regelmethoden werden in C# mit der Annotation *Rule* versehen. Damit weiß der Spec Explorer, welche Methoden die Schnittstelle zum SUT definieren. Listing 2.5 zeigt ein Beispiel eines Modellprogramms. Diesem zufolge hat das SUT zwei Operationen *Add* und *ReadAndReset*, die durch die Regelmethoden spezifiziert werden. Die Bedingung `Condition.IsTrue(...)` schränkt ein, wann die Aktion ausgeführt werden kann. Sie ist äquivalent zu den Wächterausdrücken in Zustandsauto-

[3] http://www.nunit.org/

```
static class AccumulatorModelProgram
{
        static int accumulator;

        [Rule]
        static void Add(int x)
        {
                Condition.IsTrue(x > 0);
                accumulator += x;
        }

        [Rule]
        static int ReadAndReset()
        {
                Condition.IsTrue(accumulator > 0);
                int oldValue = accumulator;
                accumulator = 0;
                return oldValue;
        }
}
```

Abbildung 2.5: Beispiel für ein Modellprogramm in C#. Dieses ist dem Accumulator-Beispiel aus der Dokumentation des Spec Explorers entnommen [4].

maten.

Der für den Spec Explorer wichtigere Teil der Eingabe sind die Konfigurationen. Konfigurationen werden in der Skriptsprache Cord geschrieben. Cord-Skripte bestehen aus *config*-Elementen und Maschinen.

Ein *config*-Element ist eine Basiskonfiguration. Hier werden mithilfe von *switches* die Optionen für den Spec Explorer festgelegt. Es können zum Beispiel Terminierungsbedingungen für die Modellexploration festgelegt werden oder das zu verwendende Unittesting-

[4] http://msdn.microsoft.com/en-us/library/ee922545.aspx

```
config Main
{
action abstract static void AccumulatorModelProgram.Add(int x);
action abstract static int AccumulatorModelProgram.ReadAndReset();

switch StepBound = 1024;
switch TestClassBase = "vs";
switch GeneratedTestPath = "..\\TestSuite";
}
```

Abbildung 2.6: Beispiel für eine Konfiguration in einem Cord-Skript.

Framework bei der Generierung der Testfälle. In *config*-Elementen werden auch die Aktionen für den Zustandsautomaten festgelegt. Die definierten Aktionen sind die Auslöser für Zustandsübergänge des Zustandsautomaten. Es gibt drei Arten von Aktionen: *Call*-Aktionen sind Stimuli an das SUT. *Return*-Aktionen sind die synchronen Antworten des SUTs auf *Call*-Aktionen. *Event*-Aktionen sind asynchrone, spontane Nachrichten, die vom SUT ausgehen. Aktionen können entweder aus den Regeln des Modellprogramms generiert werden oder manuell festgelegt werden. Beim Import der Regeln aus dem Modellprogramm werden für jede Regel eine *Call*- und eine *Return*-Aktion erzeugt. Eine Konfiguration für das Modellprogramm aus Abbildung 2.5 könnte wie in Abbildung 2.6 aussehen. Die Konfiguration definiert einige Optionswerte und die Aktionen aus den beiden Regelmethoden. *TestClassBase* legt die Basisklasse für Testfälle des verwendeten Unit-Testing-Frameworks fest. Mit *GeneratedTestPath* wird der Ordner für die erstellten Testfälle angegeben. *StepBound* gibt die Anzahl der Schritte bei der Modellexploration an.

Maschinen definieren den konkreten Zustandsautomaten. Sie legen fest welche Aktionen wann aufgerufen werden können und welche Werte diese Aufrufe haben können. Dieses Verhalten kann auf vielfältige Art und Weise festgelegt werden. Aktionen können direkt mit ihren Parametern angegeben werden. Für nicht angegebene Parameter werden wenn möglich Werte generiert. Auf diese Weise können konkrete Szenarien abgebildet werden. Wie bei regulären Ausdrücken kann das Verhalten des Zustandsautomaten mit Opera-

```
machine ConstructedMachine() : Main
{
        construct model program where Scope
                = "AccumulatorModelProgram"
}

machine ExplicitScenarioMachine() : Main
{
        Add(_); (ReadAndReset(); Add(_) )* ReadAndReset()
}
```

Abbildung 2.7: Beispiel für zwei Maschinen in einem Cord-Skript

toren aus anderem Verhalten konstruiert werden. So gibt es zum Beispiel Wildcards für eine beliebige Aktion oder beliebig viele Aktionen. Wiederholungen können wie bei regulären Ausdrücken in Zahlen oder mithilfe der „*“-, „?“- und „+“-Operatoren angegeben werden. Dazu gibt es noch Operatoren, die zwei Verhaltensbeschreibungen zusammenführen wie zum Beispiel die parallele Komposition. Wichtig ist noch die Möglichkeit spezielles Verhalten zu festzulegen. Dazu gibt es das *construct* Verhalten, welches Verhalten aus anderem Verhalten generiert, indem es zum Beispiel Testfälle generiert, die Suchtiefe einschränkt, eine gewisse Anforderungsabdeckung erreicht oder nicht akzeptierende Pfade verwirft. Abbildung 2.7 zeigt Beispiele von zwei Maschinen. Die Maschine `ConstructedMachine` verwendet Modellexploration um aus den Aktionen, die in der Konfiguration aus Abbildung 2.6 definiert wurde, Testszenarien zu generieren. Die Maschine `ExplicitScenarioMachine` definiert Szenarien durch direkte Angabe der Aktionen.

Die Eingabe des Spec Explorers definiert implizit Zustandsautomaten, die durch Zustandsraumexploration erstellt werden. Die Effektbeschreibung wird in C# definiert. Die Wächter der Zustandsübergänge werden als boolsche Ausdrücke mithilfe von Annahmen wie `Condition.IsTrue(...)` ebenfalls in C# verfasst. Optionswerte für die Testfallgenerierung und die möglichen Zustandsübergänge werden in der Sprache Cord angegeben.

2.4 Gemeinsamkeiten und Unterschiede der Sprachen

Alle drei Testfallgeneratoren benutzen Zustandsautomaten mit jeweils einer anderen Sprache für die Wächter- und Effektbeschreibung von Zustandsübergängen. Bei dem Graphwalker und Conformiq wird der Zustandsautomat explizit durch Graphen beziehungsweise Zustandsdiagramme angegeben. Beim Spec Explorer ergibt sich der Zustandsautomat implizit durch Exploration des Zustandsraumes eines Modellprogramms.

Für die Wächter- und Effektbeschreibungen setzen die drei Testfallgeneratoren auch auf ähnliche Sprachen. Beim Graphwalker sind dies Java-Methoden und per BeanShell interpretiertes Java. Bei Conformiq wird das Java-ähnliche QML benutzt. Beim Spec Explorer kann im Prinzip jede IL-basierte Sprache benutzt werden, die Klassen, Methoden und Annotationen für die Regelbeschreibungen unterstützt. Dies schließt insbesondere C# mit ein. Für die Wächter- und Effektbeschreibung sind hier vor allem die Anweisungen und Ausdrücke der Sprachen und Standarddatentypen interessant.

Bei den Anweisungen und Ausdrücken unterscheiden sich alle drei Sprachen kaum. Sie haben alle ihren Ursprung in C und haben infolgedessen eine ähnliche C-artige Syntax. Die Sprachelemente wie Schleifen, Verzweigungen, Variablen und Operatoren sind so gut wie identisch in Syntax und Semantik bei den drei Sprachen.

Leichte syntaktische Unterschiede gibt es bei der Fallunterscheidung mittels *switch*. So erlaubt C# im Gegensatz zu Java kein *Fall-Through*. Allerdings kann ein Fallrumpf in C# mehrere Sprungmarken besitzen. Zusätzlich gibt es in C# einen Sprungbefehl zu einem anderen Fall mittels *goto case*. Des Weiteren können in einer Fallunterscheidung in C# keine neuen Variablen angelegt werden, in Java hingegen schon. Die Werte nicht angesprungener Variablendeklarationen in Java werden auf null gesetzt. Im Allgemeinen sind diese Unterschiede jedoch von geringer Bedeutung, da sie nur spezielle Fälle betreffen.

Ein weiterer Unterschied sind Lambda-Ausdrücke. In C# sind diese seit Version 3.0 vorhanden. In Java werden diese gerade erst mit Version 8 eingeführt. Zum Zeitpunkt dieser Arbeit hat Java 8 sich allerdings noch nicht durchgesetzt. Lambdas haben in C# und Java die gleiche Semantik und fast gleiche Syntax. Java benutzt den einfachen Pfeil ->, während C# den doppelten verwendet =>. In QML gibt es keine Lambda-Ausdrücke.

Unterschiede zwischen den Sprachen gibt es auch bei den Standarddatentypen. In C#

gibt es bei den primitiven Ganzzahltypen die Unterscheidung zwischen vorzeichenlosen und vorzeichenbehafteten Typen wie in C++. In Java und QML gibt es nur vorzeichenbehaftete Ganzzahltypen.

In Java und QML sind mehrdimensionale Arrays immer verzweigte Arrays. Ein verzweigtes, mehrdimensionales Array ist ein eindimensionales Array von Arrayobjekten der nächsten Dimension. Diese gibt es in C# auch. Zusätzlich gibt es hier auch rechteckige, homogene Arrays wie in C++. In mehrdimensionalen, rechteckigen Feldern liegen die einzelnen Dimensionen zeilenweise hintereinander im Speicher.

Bei den Standardcollectiontypen gibt es kaum Unterschiede. Java und C# bieten ähnliche Typen an. Unterschiede gibt es in nur der Benennung, den einzelnen Methoden und dem Aufbau der Klassenhierarchie. Zum Beispiel heißt die Java *Map* in C# *Dictionary*. In Java kann die Größe einer Collection über die Methode *size()* abgefragt werden, während dies bei C# die Eigenschaft *Count* ist. Ein Beispiel für den unterschiedlichen Aufbau der Klassenhierarchie ist die *Stack*-Klasse, die in Java von *Vector* abgeleitet ist und in C# keine Elternklasse hat. In QML gibt es nur den Java-*Vector*, *Queue*, *Stack* und die *HashTable*. QML hat damit weniger Collection-Datentypen.

Dies sind die größten Unterschiede der drei Sprachen im Bereich der Anweisungen und Ausdrücke. Auf Modul- und Klassenebene gibt es weitere signifikante Unterschiede. Diese sind aber nicht von großer Bedeutung, da sie für Wächterausdrücke und Effektbeschreibungen uninteressant sind. Alle drei Sprachen sind imperative Programmiersprachen, die ihre Wurzeln in C haben. Dementsprechend gibt es keine großen syntaktischen und semantischen Unterschiede. Dies erleichtert das Finden einer plattformunabhängigen, textuellen Sprache für Wächter- und Effektbeschreibungen.

3 Analyse und Bewertung UML-basierter Sprachen

Nach meinen Erkenntnissen gibt es drei von der OMG spezifizierte textuelle Sprachen für die Integration in die UML. Diese sind die *Action Language for foundational UML* (Alf), die *Object Constraint Language* (OCL) und die *Value Specification Language* (VSL) aus dem UML Profil MARTE. Im Folgenden werden diese drei Sprachen vorgestellt und anschließend verglichen. Dies erfolgt im Hinblick auf die Verwendung als Wächter- und Effektbeschreibungssprache.

3.1 Object Constraint Language

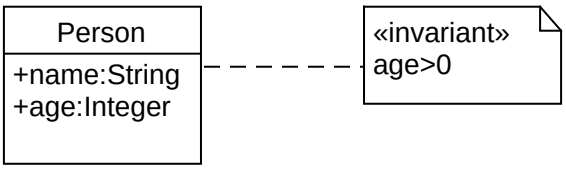


Abbildung 3.1: Beispiel für die Benutzung von OCL in einem Klassendiagramm zur Definition von Invarianten.

Die Object Constraint Language (OCL) der OMG ist eine Spezifikationssprache. Sie ist primär dazu gedacht, Randbedingungen und Abfragen in UML Modellen textuell zu definieren. Dieser Abschnitt beruft sich auf das Dokument des OCL-Standards als Quelle[8]. Auf Modellebene lässt sich eine OCL-Spezifikation als UML *OpaqueExpression* textuell in ein UML Modell einbinden. In UML Diagrammen können OCL Ausdrücke in

einem Kommentar angegeben werden. Abbildung 3.1 zeigt, wie eine Invariante für eine Klasse in einem Klassendiagramm spezifiziert werden kann.

OCL kann benutzt werden, um Objektinvarianten und Vor- und Nachbedingungen von Operationen zu definieren, um damit Codeverträge zu spezifizieren. In Zustandsdiagrammen können Wächterausdrücke in OCL definiert werden. Weiterhin kann OCL auch zur Festlegung von Initialwerten für Attribute genutzt werden. Schließlich kann mit OCL auch noch das Verhalten von Abfrageoperationen (Queries) und abgeleiteten Attributen festgelegt werden.

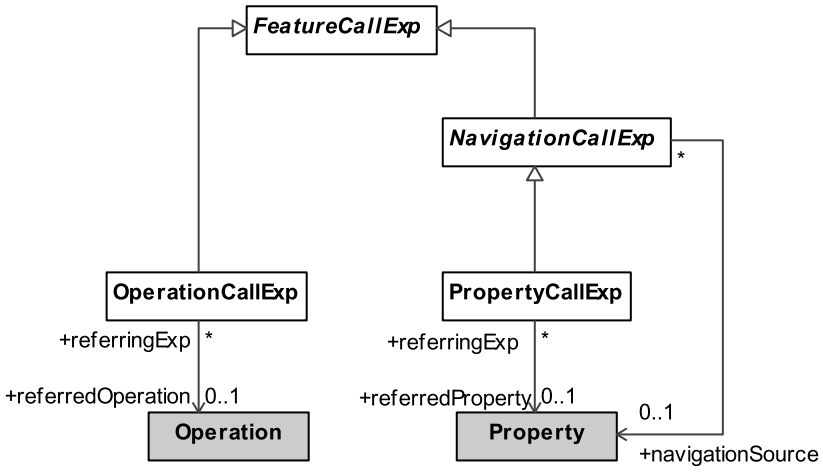
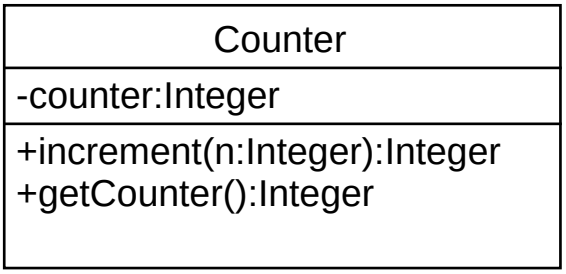


Abbildung 3.2: Auszug aus dem Metamodell der abstrakten Syntax von OCL. Die in OCL definierten Teile sind als weiße Boxen dargestellt. Die grauen Boxen stellen Elemente aus der UML dar. [Vgl. 8, Kap. 8.3.2]

Im OCL Standard[8] wird ein abstraktes Syntaxmodell für OCL-Ausdrücke definiert. Dieses abstrakte Syntaxmodell ist ein erweiterter abstrakter Syntaxbaum (AST). Der AST wurde um Referenzen in das umgebende UML Modell erweitert. Zum Beispiel verweist die *PropertyCallExpression* in OCL, die eine Eigenschaft einer Klasse abfragt, auf die UML *Property*, die aufgerufen werden soll (siehe Abbildung 3.2). Die weißen Modellelemente aus dem OCL Syntaxmodell verweisen auf die grauen Modellelemente des umgebenden UML Modells.

Ausdrücke müssen in OCL immer einen Kontext haben auf den sie Bezug nehmen. Bei Invarianten ist dies der dazugehörige UML *Classifier*. Bei Vorbedingungen, Nachbedin-




```
context Counter::increment(n : Integer ): Integer
pre : n > 0
post: result = self.counter
        and self.result = self.result@pre + n
```

Abbildung 3.3: Beispiel für Vor- und Nachbedingungen für Operationen.

gungen oder Operationsdefinitionen ist der Kontext die Operation. Bei Wertinitialisierungen und abgeleiteten Attributen ist der Kontext ein Attribut oder eine Assoziation. Mittels des *self* Schlüsselwortes kann in einem OCL Ausdruck auf eine Instanz des Kontextes beziehungsweise eine Instanz des beinhaltenden *Classifiers* zugegriffen werden. Bei Nachbedingungen kann mit dem Schlüsselwort *result* auf den Rückgabewert der Operation zugegriffen werden. Ebenfalls bei Nachbedingungen kann mit dem Postfix *@pre* an einem Attribut der Wert des Attributs vor der Ausführung der Operation benutzt werden. Abbildung 3.3 zeigt ein Beispiel für die Definition von Vor- und Nachbedingungen für Operationen.

Ein OCL-Ausdruck liefert immer genau einen Wert zurück. Bei Randbedingungen und Wächtern ist dies ein Wahrheitswert. Für Attribute und Operationen kann dies aber auch Werte anderer Datentypen sein. OCL-Ausdrücke haben keine Nebeneffekte. Sie dienen der reinen Spezifikation von Abfragen, Bedingungen und Werten. Es gibt keine Zuweisungen an Variablen des Kontextes. Deswegen lassen sich im Allgemeinen keine Operationsrümpfe in OCL spezifizieren. Hinzu kommt, dass OCL keinen Kontrollfluss mit Schleifen mit unbegrenzten Schleifendurchläufen erlaubt. Schleifen gibt es nur implizit durch das Iterieren über Collections. Für Verzweigungen gibt es If-Bedingungen der Form

if ... **then** ... **else** ... **endif**. OCL erlaubt das Aufrufen von anderen Operationen aus dem umgebenden UML Modell. Allerdings muss bei den aufgerufenen Operationen das Flag *isQuery* gesetzt sein. Dieses Flag garantiert, das die Operation nebeneffektfrei ist. Damit wird sichergestellt, dass keine Nebeneffekte durch Operationsaufrufe in einen OCL Ausdruck eingeführt werden.

OCL ist statisch typisiert. Jeder OCL Ausdruck hat einen Typ, der zur Übersetzungszeit festgestellt werden kann. OCL benutzt die in UML definierten Datentypen, fügt aber auch eigene hinzu. Die primitiven Datentypen sind wie in UML *String*, *Boolean*, *Real*, *Integer* und *UnlimitedNatural*. OCL kann auch mit UML Classifiern und Enumerations umgehen. Dazu definiert OCL die beiden Typen *OclInvalid* und *OclVoid*. *OclInvalid* ist der Untertyp aller anderen Typen außer *OclVoid* und hat als einzige Instanz den Wert *Invalid*. *OclVoid* ist ebenfalls der Untertyp aller anderen Typen mit Ausnahme von *OclInvalid* und hat als einzigen Wert das UML Literal *null*. OclVoid markiert das Fehlen eines Wertes, während OclInvalid der Rückgabetyp eines ungültigen Funktionsaufrufes ist.

Als Collectiontypen bietet OCL wie auch die UML die vier Typen *Sequence*, *Bag*, *Set* und *OrderedSet* an. Zusätzlich können mit OCL auch noch Tupel definiert werden. Werte dieser Typen können durch Angabe des Typs und den Werten in geschweiften Klammern dahinter definiert werden. Eine Liste lässt sich so folgendermaßen definieren: **Sequence** {1, 2, 3, 4, 5}

Primitiver Typ	Operationen
Integer	+, -, *, /, abs()
Real	+, -, *, /, floor()
UnlimitedNatural	+, *, /
Boolean	and, or, xor, not, implies
String	concat(), size(), substring()

Abbildung 3.4: Die Operationen auf primitiven Typen in OCL.

Tabelle 3.4 zeigt die in OCL definierten Operationen auf primitiven Typen. Hervorzuheben ist lediglich, dass es kein Modulo-Operator für die Ganzzahltypen gibt.

Wiederverwendbarkeit wird in OCL mit Let- oder Def-Ausdrücken erreicht. Let-Ausdrücke

```
context Person
def: income : Integer = self.job.salary->sum()
def: hasTitle(t : String) : Boolean = self.job->exists(title = t)
```

Abbildung 3.5: Beispiel für die Definition von *def*-Ausdrücken zur Wiederverwendbarkeit. (Aus [8])

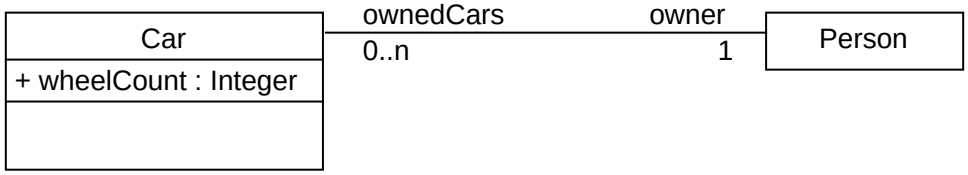
haben die Form: let a : **Integer** = 2∗2 in a+a Der Wert des OCL-Ausdrucks 2∗2 wird hier an den Namen *a* gebunden und kann in dem OCL Ausdruck hinter dem Schlüsselwort *in* beliebig benutzt werden. Let-Ausdrücke bieten Wiederverwendbarkeit innerhalb von Ausdrücken. Um OCL-Ausdrücke in anderen OCL-Ausdrücken benutzen zu können, gibt es die Definitionen mithilfe des Schlüsselwortes *def*. Listing 3.5 zeigt zwei Beispiele für Definitionen. Zum Beispiel berechnet *income* das Einkommen einer Person durch Aufsummierung der Gehälter aller Jobs der Person. Damit kann in anderen OCL-Ausdrücken das Einkommen einer Person abgefragt werden.

```
context Fruits
def fruitNameLengths:
        let str = Set {"apple", "pear", "tomato"} in
                str->collect( s|size(s))
```

Abbildung 3.6: Beispiel für die Operation *collect* angewendet auf eine Menge von Strings.

Da es in OCL keine Schleifen gibt, definiert OCL eine Reihe von Operationen für Collectiontypen. Sie werden mithilfe des Pfeil-Operators aufgerufen: collection −>operation(). Einigen Operationen werden Ausdrücke übergeben, die für jedes Element der Collection aufgerufen werden. So erzeugt der Ausdruck aus Abbildung 3.6 eine Collection mit den Längen der 3 Strings. Die Operation *collect* führt den Ausdruck, den es als Argument bekommt, für jedes Element der Liste aus und gibt eine Collection mit den Ergebnissen zurück.

Daneben kann OCL mit UML Signalen umgehen und es gibt die Möglichkeit Objekte zu anderen Typen zu casten. Des Weiteren beherrscht OCL die Objektnavigation in UML




```
context Person
inv wheelCount:
        let totalWheels = self.ownedCars.wheelCount in
        self.ownedCars->size() = 0 or totalWheels > 0
```

Abbildung 3.7: Beispiel für die Objektnavigation in OCL über Attribute und Assoziationen.

Modellen. Mithilfe des Punktoperators auf Objekten kann in OCL durch ein UML Modell navigiert werden. Dabei kann sowohl über Attribute als auch über Assoziationen navigiert werden. Bei der Navigation werden auch die Multiplizitäten berücksichtigt. Im Beispiel in Abbildung 3.7 navigiert die Invariante für die Klasse *Person* über eine Assoziation zu dem Attribut der Räderanzahl der Autos, die eine Person besitzt. Dabei wird die Multiplizität der Assoziation berücksichtigt, sodass der Ausdruck `self.ownedCars.wheelCount` den Typ `Integer[0..n]` hat.

OCL ist in der Modellierung mit UML weit verbreitet. Sie ist die Standardsprache für Randbedingungen in UML Modellen. Gängige Modellierungswerkzeuge wie zum Beispiel die Eclipse Modelings Tools[1] bieten deshalb eine ausgereifte OCL-Unterstützung mit einem Editor mit Syntaxhervorhebung und Autovervollständigung.

3.2 MARTE Value Specification Language

MARTE[2] ist ein UML Profil zur Modellierung von Echtzeitsystemen und eingebetteten Systemen. Die Quelle für diesen Abschnitt ist wieder der Standard[13]. MARTE steht

[1] http://eclipse.org/modeling/tools.php
[2] http://www.omgmarte.org/

für Modeling and Analysis of Real-Time Embedded Systems. Es dient der Spezifikation, Modellierung, Validierung und Verifikation solcher Systeme. Dafür ist es notwendig, in Modellen Ressourcenverbrauch und Zeit quantitativ angeben zu können. Dies sind sowohl für Echtzeitsysteme als auch für eingebettete Systeme häufig die maßgebenden Faktoren im Design des Systems. MARTE bietet dafür die Möglichkeit, UML Modellelemente mit ihren Zeit- und Ressourcennutzung zu annotieren.

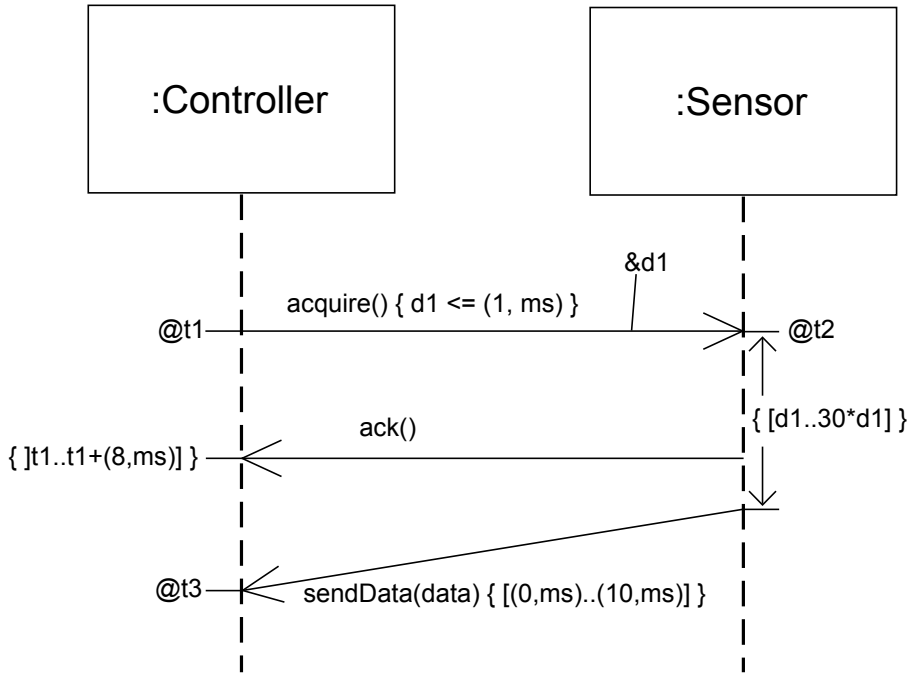


Abbildung 3.8: Beispiel für die Benutzung von VSL in einem Sequenzdiagramm zur Definition von Zeiten und Zeitintervallen.

In dem UML Profil werden außerdem weitere Erweiterungen (*Extension Units*) definiert. Eine solche Erweiterung ist VSL. VSL steht für Value Specification Language. Die VSL ist eine textuelle Sprache zur Definition von Werten in den Diagrammen, die von MARTE benutzt werden. So können zum Beispiel in Interaktionsdiagrammen die Zeiten bei den Ereignissen mit der VSL angegeben werden. In Abbildung 3.8 sind werden beispielsweise die Zeitbeschränkungen in den geschweiften Klammern mit der VSL definiert. Die VSL-Werte können direkt als Literale oder als Ausdrücke angegeben werden, die sich auf andere Variablen beziehen. VSL-Ausdrücke sind statisch typisiert. Die mittels VSL definierten Werte stehen in UML Modellen als *ValueSpecification* zur Verfügung und können

daher überall benutzt werden, wo konkrete Werte angegeben werden. *ValueSpecification*s in der UML Modellelemente, die Werte spezifieren. Dafür definiert die VSL wie bei OCL ein abstraktes Syntaxmetamodell.

Die VSL besteht aus 5 Paketen: *DataTypes*, *LiteralValues*, *Expressions*, *CompositeValues* und *TimeExpressions*.

Das *DataTypes* Paket definiert die von der VSL benutzten Datentypen. In der VSL werden nur Wertetypen wie UML *DataTypes* verwendet und keine Referenztypen wie UML *Class*. Der Grund dafür ist, dass Objekte alleine über ihre Werte identifiziert und verglichen werden können. Damit können problemlos alle Arten von Objektinstanzen textuell definiert werden. Bei Referenztypen wie UML Klassen fällt es schwer Objekte zu definieren, die zirkulär aufeinander verweisen. Werttypen können nicht aufeinander verweisen und können daher problemlos textuell definiert werden. Aus VSL heraus können nur Operationen und Verhalten ausgeführt werden, die nur *DataTypes* als Ein- und Ausgabeparameter haben. Basierend auf UML *DataTypes* definiert die VSL deswegen Stereotypen für eigene *DataTypes*. Es werden die primitiven Typen *Integer*, *Real*, *UnlimitedNatural*, *Boolean* und *String* aus der UML übernommen. Dazu kommt der MARTE-spezifische Datentyp *DateTime* für Zeit- und Datumsangaben. Die definierten *DataType*-Stereotypen sind *BoundedSubtype*, *IntervalType*, *CollectionType*, *TupleType* und *ChoiceType*. *BoundedSubtype* und *IntervalType* dienen beide zur Einschränkung des Wertebereiches eines anderen Typs. Der *CollectionType* ist in der VSL die Oberklasse aller Collectiontypen. Der *TupleType* definiert Tupel und wird unter anderem benutzt, um Zahlwerte mit der physikalischen Einheit anzugeben. Dies ist im Beispiel in Abbildung 3.8 zu sehen. Zeitangaben werden hier mit der Zeiteinheit Millisekunden versehen. Der *ChoiceType* entspricht einer *tagged union*. Immer genau eine UML *Property* eines *ChoiceType*s hat einen Wert. Es kann bestimmt werden, welche *Property* dies ist.

Das Paket *LiteralValues* beschreibt wie Literale für verschiedene primitive Typen und Aufzählungen (*Enumerations*) textuell angegeben werden können. Zu den primitiven Typen gehören die fünf in der UML definierten primitiven Typen *Integer*, *Real*, *UnlimitedNatural*, *Boolean* und *String*. Für Zeitwerte wie Uhrzeiten oder das Datum definiert die VSL DateTime-Literale. Für Standardwerte gibt es das *default*-Literal -. Schließlich gibt es noch das *null*-Literal.

Das *CompositeValues* Paket beschreibt, wie komplexere Daten beschrieben werden kön-

nen. Dazu zählen Intervalle, Tupel, Alternativen und UML Collections wie Sequences oder Sets. Intervalle werden in eckigen Klammern angegeben. Es lassen sich offene und geschlossene Intervallgrenzen definieren. Zum Beispiel ist [a .. b] ein geschlossenes und [1 .. 2[ein halboffenes Intervall. Im Beispiel aus Abbildung 3.8 werden geschlossene und halboffene Intervalle benutzt. Tupel werden in der üblichen Weise in runden Klammern mit Komma als Separator angegeben. Alternativ können auch die Einträge des Tupels namentlich angegeben werden: (value=2, unit=ms, clock=ck1). Collections werden in geschweiften Klammern angegeben. So definiert der Ausdruck {"apple", "pear", "tomato"} eine Collection mit drei Einträgen. Bei Alternativen des *ChoiceType* wird die ausgewählte Property genannt und die Werte für diese als Tupel angegeben: `periodic(period=10, jitter=0.1)`[13].

Das *TimeExpressions* Paket bietet eine textuelle Notation, um Zeiten anzugeben. Dabei gibt es zwei Typen von Zeitbeschreibungen. Zum einen gibt es Zeitpunkte von diskreten Events. Zum anderen werden Zeiten auch durch Zeitspannen beschrieben. Zeitspannen lassen sich durch ihre Länge oder durch Differenzen von zwei Zeitpunkten ausdrücken. Daneben bietet das Paket noch die Möglichkeit, *jitter*-Zeiten zu beschreiben. Diese sind zu erwartende Zeitabweichungen von einem Zeitsignal.

Während die bisher erwähnten Pakete nur Werte definieren, gibt es im *Expressions* Paket von VSL Verhaltensbeschreibungen. VSL Ausdrücke können dazu benutzt werden, Variablen zu initialisieren und Annahmen und Bedingungen zu definieren.

VSL bietet die Möglichkeit extern definiertes Verhalten zu benutzen. Jegliches UML *Behaviour* kann ausgeführt werden, welches als Eingabeparameter *DataTypes* besitzt. *Behaviour* ist die Bezeichnung von jeglichem ausführbarem Code in der UML. Des Weiteren kann auf Eigenschaften und Operationen aus dem umgebenden UML Modell zugegriffen werden. Für die primitiven Typen können die üblichen arithmetischen, logischen und relationalen Operationen aus der UML benutzt werden. In VSL können Variablen deklariert und benutzt werden. Die Variablen können dabei mit Typ, Richtung und Anfangswert deklariert werden. Die Variable wird dabei im umgebenden UML Kontext angelegt und kann damit von anderen Ausdrücken ebenfalls benutzt werden. Durch die Möglichkeit mithilfe der VSL Variablen anlegen zu können ist sie nicht nebeneffektfrei wie OCL. Allerdings sieht die VSL nicht vor, Variablen Werte zuzuweisen.

Für den Kontrollfluss gibt es nur die einfache Verzweigung mittels des ternären Operators

?:, wie er auch in Java oder C++ zu finden ist. Es gibt keine Fallunterscheidungen und keine Schleifen. Anders als in OCL gibt es aber auch keine Möglichkeit über Collections zu iterieren und einen Ausdruck mit jedem Element auszuführen. Die VSL ist weniger verbreitet als OCL, da es sich um eine Sprache für ein spezielles Profil handelt. OCL hingegen kann in jedem Einsatzgebiet von UML eingesetzt werden. Dementsprechend gibt es auch wenig Werkzeugunterstützung für MARTE VSL. Eclipse Papyrus bietet jedoch mit der Unterstützung für MARTE einen Editor für VSL.

3.3 Action Language for Foundational UML

Mit der Action Language for Foundational UML (Alf) bietet die OMG eine weitere textuelle Sprache an, um Ausdrücke zu definieren. Sofern nicht anders angegeben beruft sich dieser Abschnitt ebenfalls auf den Standard der OMG[6]. Die Syntax und Semantik von Alf ist stark von Java und C++ geprägt, fügt aber auch Konzepte aus der UML und OCL hinzu. So hat Alf bei Anweisungen, Ausdrücken und Kontrollflusselementen die typische C-artige Syntax mit geschweiften Klammern für Blöcke und Semikolon am Anweisungsende. Dafür kann Alf auch mit Signalen und Assoziationen der UML umgehen.

Alf ist eine Sprache, um mit der Foundational UML (fUML) textuell modellieren zu können. Alf lässt sich in fUML-Modelle übersetzen. Foundational UML ist eine Untermenge der UML mit wohldefinierter Ausführungssemantik[12]. Es dient dazu, UML Modelle mit Aktivitätsdiagrammen ausführbar zu machen. Dafür definiert es die konkrete, eindeutige Semantik und die Ausführungsumgebung, die das Modell ausführt.

In Foundational UML wird Verhalten nur über Verhaltensdiagramme beschrieben. Um ausführbare UML Modelle auch textuell definieren zu können, wurde Alf entwickelt. Alf-Code kann als UML *ValueSpecification*, UML *Action* oder UML *Behaviour* verwendet werden. Dazu kann der Alf-Code entweder als *OpaqueExpression*, *OpaqueAction* oder *OpaqueBehaviour* als unausgewerteter Text in das umgebende UML-Modell integriert werden oder zu fUML-Modellen übersetzt werden. Die OMG liefert dafür eine Abbildung von Alf Ausdrücken zu den fUML Aktivitätsdiagrammen. Damit kann Alf-Code in UML *Activities* oder UML *StructuredActivityNodes* übersetzt werden.

Wie OCL kann Alf-Code textuell in ein UML Modell integriert werden. Dies geschieht

zum Beispiel als *OpaqueExpression*. Der Alf-Standard definiert ebenfalls ein abstraktes Syntaxmetamodell. Ein abstraktes Syntaxmetamodell entspricht dem abstrakten Syntaxbaum, ist bei Alf aber um abgeleitete Eigenschaften angereichert. Diese lösen zum Beispiel Namen auf und geben eine Referenz auf Elemente des umgebenden UML Modells zurück.

```
Integer a = 5;
while( a>0 ) {
        a = a - 1;
}
```

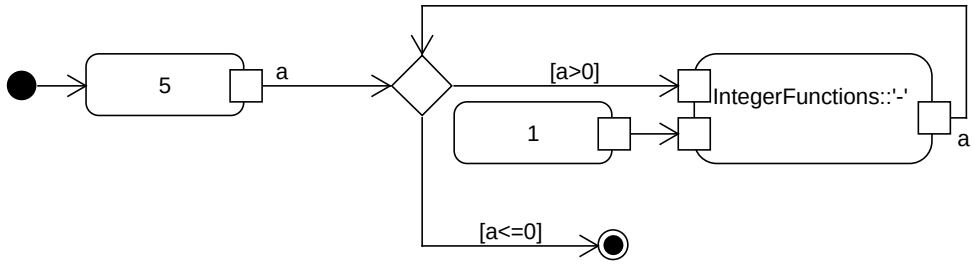


Abbildung 3.9: Beispiel der Übersetzung von Alf-Code in ein Aktivitätsdiagramm.

Die zweite Möglichkeit für das Einbinden von Alf-Code in ein UML-Modell ist das Übersetzen des Codes in eine fUML *Activity*. Dieser Weg hat den Nachteil, dass einige Konstrukte von Alf verloren gehen. Da UML Aktivitätsdiagramme auf Objektfluss basieren, gibt es zum Beispiel keine Variablen. Zuweisungen an Variablen im Alf-Code werden im Aktivitätsdiagramm aufgelöst durch einen *Fork* des Objektflusses von dem Ausdruck der rechten Zuweisungsseite zu allen Ausdrücken, die diese Variable benutzen. Das Beispiel in Abbildung 3.9 zeigt, wie sowohl die Schleife, als auch die Variable `a` bei der Übersetzung von Alf zu fUML Aktivitätsdiagramme verloren gehen.

Alf ist statisch typisiert. Bei der Übersetzung von Alf soll eine Typüberprüfung gemacht werden. Es werden die primitiven Typen *String*, *Boolean*, *Integer* und *UnlimitedNatural* der UML unterstützt, jedoch kein *Real*. Alf definiert keine Möglichkeit, mit Fließkomma-

zahlen zu arbeiten. Variablen und Parameter können in Alf mit Multiplizitäten versehen werden. Damit wird ausgedrückt, dass eine Variable mehrere Elemente des gleichen Typs haben kann. Multiplizitätsangaben in Alf entsprechen der Multiplizität `0..*` in UML-Notation. Typen mit Multiplizitätsangaben werden Sequenzen genannt. Sequenzen in Alf ähneln den Arraydeklarationen in Java oder C++, sowohl syntaktisch als auch semantisch. Allerdings können Sequenzen in Alf nicht mehrdimensional sein. Dies unterscheidet sie von den Arrays in Java oder C++. Collectiontypen wie Listen können verschachtelt werden. Damit ist es auch möglich, in Alf mit mehrdimensionalen Daten zu arbeiten.

Im Bereich der Anweisungen bietet Alf die meisten Konstrukte, die es auch in Java gibt. Lokale Variablen können deklariert und benutzt, Operationen auf Objekten aufgerufen und lesend und schreibend auf Eigenschaften von Objekten zugegriffen werden. Alf unterstützt die gleichen arithmetischen und logischen Operatoren wie Java. Objekte können mittels des Schlüsselwortes *new* und Angabe einer Konstruktoroperation neu erzeugt werden. Es gibt die einfache Verzweigung mit *if*, die Mehrfachverzweigung mit *switch* und *while*- und *for*-Schleifen in Alf.

```
Integer a = 4;
if(a % 3 == 0) {
        a *= 2;
} or if(a % 3 == 1) {
        a *= 3;
} or if(a % 3 == 2) {
        a *= 4;
}
```

Abbildung 3.10: Beispiel für das *if-or*-Konstrukt. Die drei Bedingungen werden parallel ausgeführt. Wenn eine davon wahr ist, wird der Rumpf dieser Bedingung ausgeführt.

Bei der einfachen Verzweigung mittels *if* können jedoch mehrere Bedingungen über das Schlüsselwort *or* verknüpft werden. In diesem Fall sollen alle Bedingungen parallel ausgewertet werden. Der Rumpf einer beliebigen erfüllten Bedingung wird dann ausgeführt. Wenn mehrere Bedingungen wahr sind, dann wird nichtdeterministisch eine ausgewählt

und deren Rumpf ausgeführt. Für diese nebenläufige Verzweigung ist in Abbildung 3.10 ein Beispiel zu sehen. Hier werden die drei Bedingungen `a % 3 == 0`, `a % 3 == 1` und `a % 3 == 2` parallel ausgewertet. Da `a` den Wert 4 hat, wird die zweite Bedingung ausgeführt und `a` bekommt den Wert 12.

Für gegenseitigen Ausschluss beim Zugriff auf Objekte gibt es den Lockingoperator *$*. Ähnlich wie das Schlüsselwort *synchronized* in Java wird damit das angegebene Objekt als Lock verwendet. Im Unterschied zu Java werden allerdings alle in dem Ausdruck verwendeten Objekte gelockt. Mit der *classify*-Anweisung kann der Typ von Objekten umgeändert werden. Dies entspricht der *ReclassifyObjectAction* [Vgl. 11, Kap. 11.3.39] der UML.

Alf kann auch mit UML Assoziationen umgehen. Auf binäre Assoziationen kann mithilfe des Punktoperators zugegriffen werden. Erstellt und gelöscht können binäre Assoziationen wie Attribute mithilfe von Zuweisungen. Für allgemeine Assoziationen gibt es in Alf noch sogenannte *Linkoperationen*. Diese müssen insbesondere dann verwendet werden, wenn die Assoziation nicht binär ist, oder die Enden der Assoziation nicht benannt sind. Mit *Linkoperationen* können Assoziationen erstellt oder gelöscht werden. Das Auslesen von Assoziationen hat die gleiche Syntax wie das Aufrufen statischer Funktionen der Assoziationsklasse. Zum Beispiel fragt der Alf-Ausdruck `Roster::player(team=>t, season=>y)`[Vgl. 6, Kap. 8.3.9] alle Spieler der Assoziation `Roster` ab, die im Team `t` in der Saison `y` gespielt haben.

Da Alf-Verhaltenscode eine textuelle Beschreibung von fUML Aktivitätsdiagrammen ist und die fUML Aktivitätsdiagramme zur Beschreibung des Verhaltens von aktiven Klassen verwendet werden kann, muss Alf auch Möglichkeiten zur Verwendung von Signalen besitzen. Aktive Klassen besitzen einen eigenen Thread um Code ausführen zu können. Signale werden von aktiven Klassen verwendet, um mit der Umwelt kommunizieren zu können. Mit der *accept*-Anweisung kann die Ausführung von Alf-Code solange angehalten werden bis bestimmte Signale empfangen werden. Syntaktisch werden Signale durch den Aufruf des Signalempfängers als Methode gesendet.

Alf übernimmt die Collectionoperationen aus OCL als Sequenzoperationen. Das OCL Beispiel aus Abbildung 3.6 für Collectionoperationen ist mit wenigen Änderungen auch gültiger Alf-Code, wie Abbildung 3.11 zeigt. Ebenfalls wie in OCL beherrscht Alf die Objektnavigation in UML Modellen über Attribute und Assoziationen.

```
let str : String[] = {"apple", "pear", "tomato"};
str->collect( s|size(s));
```

Abbildung 3.11: Beispiel für die Operation *collect* angewendet auf eine Menge von Strings in Alf.

Hauptsächlich dient Alf der Implementierung von *Activities* eines UML Modells. Daneben können mit Alf aber auch strukturelle Elemente eines Modells definiert werden. Alf erlaubt es Pakete, Klassen, *DataTypes*, Assoziationen, Aufzählungen, Signale zu definieren. Diese werden dann in die entsprechenden Modellelemente der UML übersetzt.

Alf baut an vielen Stellen auf dem UML Metamodell auf. Die primitiven Typen, die in Alf verfügbar sind, werden aus der UML importiert [Vgl. 10, Kap. 13.1]. Aus fUML wird das *Primitive Behaviour* importiert. Das ist das Paket für die grundlegenden Funktionen auf primitiven Typen, wie zum Beispiel die Länge eines Strings oder Typumwandlung von Zahltypen zu Strings. Der Alf-Standard definiert die Collectionklassen *Map*, *Queue*, *Deque*, *Set*, *OrderedSet*, *Bag* und *List* inklusive deren Operationen und Implementierungen.

Alf ist eine relativ neue Sprache. Die erste finale Version des Alf-Standards wurde 2012 veröffentlicht[1]. Daher gibt es bisher wenig Werkzeugunterstützung. Den einzigen Editor mit Syntaxhighlighting und Autovervollständigung für Alf gibt es in Papyrus. Dazu bietet die UML Ausführungsumgebung moliz[3] einen Compiler für Alf zu fUML. Des Weiteren gibt es eine Referenzimplementierung für Alf[4], die Alf zu fUML übersetzt und das Ergebnis mit einer fUML Referenzimplementierung ausführt. Dies berichten Micske et al.[5] für Opensource Werkzeuge. Aber auch bei kommerziellen Werkzeugen ist mir keine weitere Verbreitung von Alf bekannt.

3.4 Auswertung und Vergleich

Die drei vorgestellten Sprachen stammen alle von der OMG. Sie sind textuelle Sprachen, die für den Einsatz mit UML Modellen entworfen wurden und sich textuell in diese

[3] http://www.modelexecution.org/
[4] http://modeldriven.org/alf/

integrieren lassen. Aus Code in den drei Sprachen lässt sich jeweils ein abstraktes Syntaxmodell erstellen. Dieses verweist an den entsprechenden Stellen auf die referenzierten Elemente des umgebenden UML Modells.

Alle drei vorgestellten Sprachen können zur Beschreibung von Wächterausdrücken verwendet werden. Der Wächterausdruck einer Transition eines Zustandsdiagramms ist in der UML Superstructure ein *Constraint*-Objekt mit einer *ValueSpecification* für die eigentliche Spezifikation des Wächterausdrucks. Der Standard von allen drei Sprachen sieht vor, dass Ausdrücke als UML *ValueSpecification* benutzt werden können. Damit können Alf, VSL und OCL als für Wächterausdrücke benutzt werden. Im Fall von OCL ist die Benutzung von OCL zur Beschreibung von Wächterausdrücken ein Anwendungsfall, der explizit im Standard erwähnt wird[8, Kap. 12.11].

Der große Unterschied macht die Benutzung der Sprachen für Effektbeschreibungen von Zustandsübergängen. Effektbeschreibungen sind häufig nebeneffektbehaftet. Sie verändern den Zustand der Variablen des Modells oder kommunizieren mit der Umgebung. Beide Möglichkeiten rufen Nebeneffekte hervor.

Die Sprache OCL ist strikt nebeneffektfrei. Mit OCL können keine Variablen des Kontextes geändert werden. Es können keine Signale versendet werden und es können nur Operationen ausgeführt werden, die keine Nebeneffekte haben können. Daher ist OCL für die Verwendung als Effektbeschreibungssprache ungeeignet.

Die Value Specification Language schließt Nebeneffekte prinzipiell nicht aus. Mit dem Aufrufen von Funktionen mit Nebeneffekten könne eben solche erzeugt werden. Dies verschiebt das Problem jedoch nur. Die Funktionen müssen an anderer Stelle plattformunabhängig und textuell definiert werden. Die VSL unterstützt auch keine Zuweisung. Die Eigenschaften des Kontextes des Zustandsdiagramms können nicht direkt geändert werden.

Zuweisungen an Eigenschaften des Kontextes könnten in VSL über *CompositeTypes* nachgebildet werden. Dabei kann als Effektbeschreibung mittels VSL ein *CompositeType* benutzt werden. Die Felder dieses Typs sind dabei eine Untermenge der Eigenschaften des Kontextes des Zustandsdiagramms. Für jedes gesetzte Feld des angegebenen Typs wird die dazugehörige, gleichnamige Eigenschaft im Kontext geändert. Auf diese Weise könnten Änderungen am Zustand in VSL realisiert werden. Das Beispiel in Abbildung 3.12 veranschaulicht dies. Diese Variante eignet sich nur für die direkten Eigenschaften des

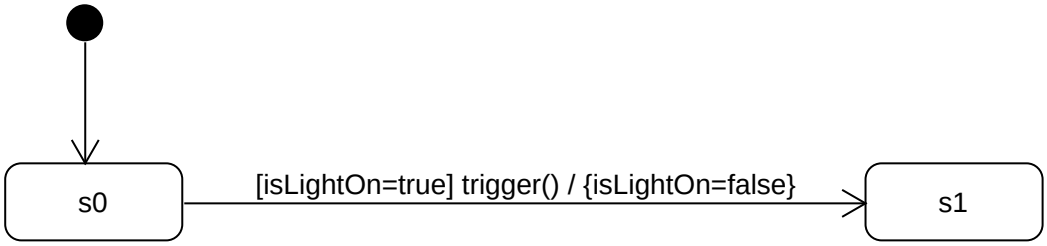


Abbildung 3.12: Beispiel für die Umgehung der Einschränkungen für Effektbeschreibungen von VSL. Die Zuweisung an den Kontext im Effektcode wird über eine Updatestruktur simuliert. Die Updatestruktur enthält eine Untermenge der Felder des Kontextes.

Kontextes und nicht für solche von Objekten, die nur vom Kontext referenziert werden. Diese Umgehung der Einschränkungen von VSL funktioniert aber auch nur in eingeschränkten Fällen, in denen der Effektcode lediglich aus Zuweisungen an den Kontext besteht. Hinzu kommt, dass diese Variante offiziell nicht vorgesehen ist und daher von den Werkzeugen nicht unterstützt wird.

Als einzige Sprache, die nebeneffektbehaftete Ausdrücke ausdrücklich erlaubt, bleibt Alf. Als UML *Actionlanguage* ist sie als plattformunabhängiger Ersatz für Sprachen wie Java gedacht. Mit ihr können uneingeschränkt Effekte beschrieben werden. Der größte Nachteil von Alf ist das Fehlen von Gleitkommazahlen. Sowohl OCL als auch die VSL haben eine Unterstützung für Gleitkommazahlen. Alf in etwa eine Obermenge von OCL Ausdrücken, da Alf die Sequenzoperationen von OCL übernimmt und auch die anderen Ausdrücke von OCL in Alf unterstützt werden. Bis auf die Angabe des Kontextes in OCL und syntaktischen Feinheiten wie Semikolons am Ende von Anweisungen in Alf gibt es nur geringe Unterschiede zwischen OCL- und Alf-Ausdrücken. Auf syntaktischer Ebene hat bietet OCL keinen Vorteil gegenüber Alf. Ein Vorteil den OCL gegenüber Alf hat ist die größere Verbreitung. OCL ist die Standardsprache zum Definieren von Randbedingungen in UML Modellen und wird von mehr Werkzeugen unterstützt als das relativ neue Alf. Günstigerweise ist Alf den gängigen General Purpose Languages Java, C++, und C# sehr ähnlich. Da Alf und Java, C++ und C# auf den gleichen Konzepten beruhen, sollte die Übersetzung von Alf in diese Sprachen einfach sein. Alf Code lässt sich damit auch

einfach in die Eingabesprachen der Testfallgeneratoren übersetzen.

Es gibt meines Wissens nach keine weiteren UML-basierten, textuellen Sprachen zur Effektbeschreibung. Als Alternativen bleiben somit nur noch Sprachen, die nicht auf den Konzepten der UML basieren. Solche Sprachen sind meist plattformabhängig und basieren auf einer Laufzeitumgebung mit deren Bibliotheken. Diese Bibliotheken können sehr groß sein, wie etwa die C# oder Java Standardbibliotheken. Um diese Sprachen im Kontext der UML verwenden zu können, müssten die Bibliotheken mittels UML nachgebildet werden, damit die Sprache wie gewohnt verwendet werden kann. Ein weiterer Nachteil ist der, dass nicht UML-basierte Sprachen nicht alle Konstrukte der UML beherrschen. Zum Beispiel werden Assoziationen und Signale von den meisten Sprachen nicht verstanden. Auch ist die Standardbibliothek der UML in solchen Sprachen nicht verfügbar. Daraus entstehen Inkonsistenzen in einem Modell zwischen den Diagrammen und dem Code in der Effektbeschreibungssprache. Wenn zum Beispiel Java als Effektbeschreibungssprache benutzt wird, dann wird eine Klassenvariable, die im Klassendiagramm als *Integer* angelegt ist im Effektcode als Typ *int* benutzt. Neben der namentlichen Inkonsistenz ist die Semantik beider Typen eine andere. Der UML Typ *Integer* hat keine festgelegte Bitbreite, während das Java *int* auf 32 Bit festgelegt ist. Ein Vorteil einer plattformabhängigen Sprache könnte sein, dass sie bei der Verwendung mit einem Testfallgenerator nicht übersetzt werden muss. Da die drei vorgestellten Testfallgeneratoren unterschiedliche Eingabesprachen haben, kommt dieser Vorteil nicht zum Tragen.

Ein Vorteil, den die VSL gegenüber Alf hat, ist die Unterstützung von Zeitmodellierung. Da in Testfällen auch Zeitbedingungen definiert werden oder auch mit Zeitzonen gearbeitet wird, wäre die Möglichkeit von VSL wünschenswert, Zeitliterale und Zeitausdrücke benutzen zu können. Diese können für das Konzept der *Timer* in UTP zur Definition von Timeouts benutzt werden. Der größte Nachteil von Alf gegenüber den anderen beiden Sprachen ist jedoch die Unterstützung des Gleitkommazahlentyps *Real* der UML.

Trotz dieses Nachteils ist Alf die geeignetste Wahl als plattformunabhängige, textuelle Sprache zur Beschreibung von sowohl Wächterausdrücken als auch den Effekten von Transitionen in UML Zustandsdiagrammen. In der VSL und OCL lassen sich nebeneffektbehaftete Effekte nicht oder nur schlecht beschreiben.

4 Abbildung von Alf zu Java, C# und QML

In diesem Abschnitt wird gezeigt, wie Alf Code für die Effekt- und Wächterbeschreibung in die Eingabesprachen der drei vorgestellten Testfallgeneratoren übersetzt werden kann. Die vier Sprachen Alf, Java, C# und QML ähneln sich in einigen Teilen stark. Es gibt aber auch syntaktische und semantische Unterschiede zwischen den Sprachen, sodass die Übersetzung nicht trivial ist.

Eine Einschränkung bei der vorgestellten Abbildung ist die, dass bei der Übersetzung von Alf die Verwendung als Effektbeschreibungssprache in Testfallentwurfsmodellen berücksichtigt wird. Es wird nur die Übersetzung der Ausdrücke und Anweisungen beschrieben. In Alf können auch strukturelle Elemente wie Klassen beschrieben werden. Im Kontext von Testfallentwurfsmodellen liegen die strukturellen Elemente aber bereits als UML Modelle vor, zum Beispiel als Klassendiagramme. Diese werden separat in die Eingabesprache der Testfallgeneratoren übersetzt. Die Übersetzung der UML Modelle des Testfallentwurfsmodells wird in dieser Arbeit nicht behandelt. Einige Alf-Konstrukte lassen sich auf Anweisungs- oder Ausdrucksebene nicht in Java, C# oder QML übersetzen. Um diese Konstrukte übersetzen zu können, muss auch die Übersetzung der Klassendiagramme angepasst werden. Die Übersetzung der Anweisungen und Ausdrücke von Alf in Java, C# und QML, sowie die Anpassung der Übersetzung der Klassendiagramme, werden im Folgenden erläutert.

4.1 Primitive Typen

Die primitiven Typen von Alf lassen sich relativ einfach übersetzen. Abbildung 4.1 zeigt die Übersetzung. Die Typen *Boolean* und *String* haben eine 1:1-Entsprechung in Java, C# und QML. Der Typ *Any* in Alf ist vergleichbar mit dem *Object*-Typ in Java und

Alf	Java	C#	QML
Integer	int	int	int
UnlimitedNatural	int	uint	int
Boolean	boolean	bool	boolean
String	String	string	String
Any	Object	object	Object / AnyRecord

Abbildung 4.1: Übersetzung der primitiven Typen von Alf.

C#. Er ist der Obertyp aller anderen Typen. In QML haben Objekte von Klassen und *Records* unterschiedliche Obertypen. Für Klassen ist der Wurzeltyp *Object*, für *Records* *AnyRecord*. Hier muss bei der Übersetzung des Alf-Typs *Any* ermittelt werden, ob einer Variable vom Typ *Any* Recordobjekte oder Klassenobjekte zugewiesen werden. Mithilfe des Typs des Initialisierungsausdrucks einer Variable vom Typ *Any* oder nachfolgender Zuweisungen an diese kann festgestellt werden, ob der Typ *Any* in *Object* oder *AnyRecord* übersetzt werden muss. Zuweisungen von Recordobjekten und Klassenobjekten an eine Variable vom Typ *Any* sind in QML nicht möglich.

Der Ganzzahltyp *Integer* in Alf hat keine festgelegte Bitbreite, der Standard besagt aber, dass eine Implementierung eine Bitbreite festlegen kann[12, Tabelle 9.1]. Es bietet sich an, in Java und C# den am meisten verbreiteten Typ *int* zu benutzen. Wenn größere Zahlen zu erwarten sind, oder mehr Schutz gegen Integerüberläufe erwünscht ist, kann auch der größere Ganzzahltyp *long* benutzt werden. Der Datentyp *int* in QML kann beliebig große Werte beinhalten und kann daher bedenkenlos für den Alf-Typ *Integer* verwendet werden.

Der Ganzzahltyp *UnlimitedNatural* ist vorzeichenlos und hat als besonderen Wert den Wert „unendlich“. Dafür gibt es in keiner der Sprachen Java, C# oder QML eine direkte Entsprechung. In C# gibt es vorzeichenlose Ganzzahltypen wie zum Beispiel *uint*. Bei Java und QML kann nur der vorzeichenbehaftete Typ *int* benutzt werden. Mit Java 8 wurden Methoden für das vorzeichenlose Rechnen mit den Ganzzahltypen eingeführt, aber keine vorzeichenlosen Ganzzahltypen an sich[17]. Den Wert „unendlich“ gibt es in keiner der drei Sprachen. Dieser Wert kann jedoch durch den größtmöglichen Integerwert ersetzt werden. Dies wäre in Java die Konstante `Integer.MAX_VALUE` und in C#

`UInt32.MaxValue`. In QML sind die arithmetischen Typen beliebig groß. Hier gibt es keinen größtmöglichen Wert. Als Ausweg kann eine Konstante gewählt werden, die größer ist als alle zu erwartenden Werte. Durch die Verwendung des größtmöglichen Wertes in Java und C#, gibt es einen Wert, der größer ist, als alle anderen Integerwerte. Allerdings darf nicht mit diesem Wert gerechnet werden, da sonst Integerüberläufe auftreten. Alf verwendet das Verhalten von fUML für *UnlimitedNatural* wieder, welches keine Arithmetik, sondern nur Vergleiche mit diesem Typen spezifiziert [12, Tabelle 9.6]. Daher kann auch in Alf nicht mit *UnlimitedNatural* gerechnet werden und es können keine Überläufe auftreten.

Der in der UML vorhandene Gleitkommazahlentyp *Real* fehlt in Alf, könnte aber in späteren Versionen hinzugefügt werden[6, Kap. 11.3].

4.2 Collectiontypen

Alf	Java	C#	QML
Set	Set	HashSet	-
OrderedSet	SortedSet	SortedSet	-
Bag	-	ConcurrentBag	-
List	List	List	Vector
Map	Map	Dictionary	HashTable
Queue	Queue	Queue	Queue
Deque	Deque	LinkedList	-
BitString	BitSet	BitArray	-

Abbildung 4.2: Übersetzung der Collectiontypen von Alf.

Alf unterstützt die Collectiontypen *BitString*, *Set*, *OrderedSet*, *Bag*, *Map*, *List*, *Queue* und *Dequeue*. Deren Übersetzung in Java, C# und QML ist in Abbildung 4.2 dargestellt.

Der Typ *Set* ist die ungeordnete Menge. In Java gibt es dafür das Interface *Set*, welches zum Beispiel durch *HashSet* oder *TreeSet* implementiert wird. C# bietet die Klasse *HashSet* für Mengen. Die geordnete Menge ist in Alf das *OrderedSet*. In Java gibt es dafür das Interface *SortedSet* welches ebenfalls durch das *TreeSet* implementiert wird. In

C# gibt es das *SortedSet*.

Die Datenstruktur *Bag* ist die ungeordnete Liste oder auch Multimenge. Für sie gibt es in der Standardbibliothek von Java keine Implementierung. In den Bibliotheken Google Guava[1] und Apache Commons Collection[2] gibt es Implementierungen. Soll diese nicht benutzt werden, dann bleibt nur die eigene Implementierung. In C# gibt es mit der Klasse *ConcurrentBag* eine Implementierung von Multimengen, die vor allem für nebenläufige Programmierung gedacht ist.

Die geordnete Liste gibt es in Alf als *List*. Auch in Java und C# heißt die Repräsentation dafür *List*. In Java ist dies aber ein Interface, welches zum Beispiel durch die Klassen *Vector*, *LinkedList* und *ArrayList* implementiert wird. Die Klasse *List* in C# entspricht der Java *ArrayList*. Sie basieren beide auf Arrays deren Länge verdoppelt wird, wenn sie voll sind.

Die Warteschlange heißt in Alf, Java und C# *Queue*. In Java verbirgt sich dahinter wie üblich ein Interface, dass zum Beispiel durch die *LinkedList* implementiert wird. Die *double ended queue* heißt in Alf und Java *Deque*. In Java ist es ein Interface, welches ebenfalls durch die *LinkedList* implementiert wird. In C# kann dafür gleich die *LinkedList* verwendet werden.

Die *Map* in Alf heißt in Java ebenfalls *Map* und in C# *Dictionary*. Der Typ *Map* in Java ist ein Interface, welches zum Beispiel durch die konkreten Klassen *HashMap* oder *TreeMap* implementiert wird. Das *Dictionary* in C# ist bereits eine konkrete Klasse.

Das *BitString* in Alf ist ein Feld fester Größe, deren Bits einzeln gesetzt werden können. Die drei Operationen sind *set* und *unset* für das Setzen und Löschen von einzelnen Bits und *length* für das Abfragen der Länge. In Java wird das *BitString* durch die Klasse *BitSet* implementiert. Hier heißen die Methoden *set*, *clear* und *length*. In C# implementiert das *BitArray* diese Datenstruktur. Für das Setzen und Löschen der Bits gibt es die Methode *Set* mit einem boolschen Parameter, der angibt ob, ein Bit gesetzt oder gelöscht werden soll. Die Länge lässt sich über die Eigenschaft *Length* herausfinden.

Die Collectiontypen von Alf sind größtenteils auch in Java und C# wiederzufinden. Die Klassen und deren Operationen heißen in den jeweiligen Sprachen nur anders. Dies bedeutet, dass bei der Übersetzung von Alf Code der Typ von Ausdrücken analysiert werden

[1] https://github.com/google/guava
[2] http://commons.apache.org/proper/commons-collections/

muss und gegebenenfalls die Operationen und Typen umgeschrieben werden müssen. Alternativ können die Typen in Java und C# mit den gleichen Bezeichnern nachimplementiert werden.

In QML gibt es nur für die Collectiontypen *Queue*, *List* und *Map* Entsprechungen. Diese heißen *Queue*, *Vector* und *HashTable*. Die anderen Collectiontypen müssen nachimplementiert werden.

4.3 Wertspezifikation und Objektnavigation

In Alf kann wie in Java, C# und QML mithilfe des Punktoperators über Objekte navigiert werden. Ein Ausdruck `a.b.c` liefert den Wert von `c` der im Objekt in `b` im Objekt `a` abgelegt ist. In den Eingabesprachen der Testfallgeneratoren kann ein Feld im Gegensatz zu Alf maximal ein Objekt beinhalten. Soll eine Sammlung von Objekten in einem Feld gespeichert werden, dann kann auf Collections oder Arrays zurückgegriffen werden. Trotzdem ist in einem Feld immer maximal ein Objekt gespeichert, das der Collection oder das des Arrays. Alf übernimmt das Konzept der Multiplizitäten aus der UML. Daher können in einem Feld mehrere Objekte enthalten sein. Bei der Objektnavigation werden die Multiplizitäten berücksichtigt. Es werden alle Objekte, die bei der Navigation angetroffen werden aufgesammelt. Dies ist syntaktischer Zucker für die Sequenzoperation *collect*. Der Ausdruck `a.b.c` entspricht in Alf dem Ausdruck `a->collect(o|o.b)->collect(o|o.c)`. Bei der Übersetzung von Alf-Code nach Java, C# und QML müssen die Multiplizitäten beachtet werden. Ein Alf-Ausdruck, der nur die einfache Multiplizität `[0..1]` enthält kann in Java und C# auch in die Punktnotation übersetzt werden. Sobald ein Ausdruck eine Sequenz ist, muss der Punktoperator in Java, C# und QML durch die Implementierung der *collect*-Operation ersetzt werden. Diese ist weiter unten in Abbildung 4.14 zu sehen.

Neue Objekte einer Klasse werden in Alf mit dem Schlüsselwort *new* erzeugt. Auf das Schlüsselwort *new* folgt die Angabe einer Klasse, eines *DataTypes* oder einer Operation, die mithilfe des Stereotypen *«constructor»* als Konstruktoroperation gekennzeichnet ist. Abbildung 4.3 zeigt die vier verschiedenen Arten der Objekterzeugung.

Wenn bei Klassen kein Konstruktor explizit angegeben ist, wird versucht, eine Konstruktoroperation der Klasse für die übergebenen Parameter zu finden: `new PointCls(1, 2)`

Diese Form der Objekterzeugung wird auch in Java und C# benutzt und kann daher ohne Änderung aus dem Alf-Code übernommen werden. In QML gibt es diese Variante bei Klassentypen. *Records* werden grundsätzlich nicht mit dem Schlüsselwort *new* erzeugt. Initialisierende Konstruktoren für *Records* müssen in statische Funktionen umgewandelt werden, welche das Objekt initialisieren.

Wenn in Alf eine Konstruktoroperation namentlich genannt wird, wie zum Beispiel `ConstructorOp` in `new PointCls::ConstructorOp(1, 2)`, dann kann in Java, C# und QML der Operationsname weggelassen werden und wie in diesen Sprachen üblich ein unbenannter Konstruktor aufgerufen werden. Dies funktioniert nur, wenn nicht zwei Konstruktoren die gleichen Funktionsparameter haben. Wenn zwei Konstruktoren die gleiche Funktionsparameter haben, können die Konstruktoroperationen in statische Methoden der Klasse umgewandelt werden, die ein initialisiertes Objekt der Klasse zurückliefern. Die Entscheidung ob eine statische Methode oder ein Konstruktor verwendet wird, muss schon bei der Übersetzung des UML Modells geschehen, da sie dort als Teil der Klasse generiert wird. Diese Entscheidung muss dann bei der Übersetzung von Alf-Code bekannt sein.

```
PointCls a = new PointCls::ConstructorOp(1, 2);
PointCls b = new PointCls(1, 2);
PointDt c = new PointDt(1, 2);
PointDt d = new PointDt(x => 1, y => 1);
```

Abbildung 4.3: Beispiel für die Möglichkeiten in Alf Objekte zu erstellen. `PointCls` ist hier eine Klasse. `PointDt` ist ein *DataType*

Wie in Abbildung 4.3, für die Variablen `c` und `d`, zu sehen, können die Attribute eines Objektes direkt initialisiert werden. Die Werte können entweder über die Position der Eigenschaft in der Klasse geschehen oder durch direkte Angabe des Attributsnamens gesetzt werden. In C# können Klassen und *Structs* mithilfe von geschweiften Klammern hinter dem Konstruktor initialisiert werden: `new PointDt() {x = 1, y = 2}`. In Java gibt es hierfür die sogenannte *double-brace initialization*. Dabei wird durch geschweifte Klammern hinter dem Konstruktor eine anonyme Unterklasse erzeugt, welches durch ein zweites Paar geschweifter Klammern initialisiert wird. Darin kann das Objekt initialisiert

werden: `new PointDt() {{setX(1); setY(2)}}`. Zu beachten ist, dass eine Unterklasse erzeugt wird. Dies kann in bestimmten Fällen unerwünscht sein, wenn zum Beispiel auf Typgleichheit überprüft wird. Als Alternative bieten sich Konstruktoren für die Initialisierung oder Hilfsfunktionen an. QML bietet keine gesonderte Syntax zum direkten Initialisieren von Objekten. Hier bietet es sich an Hilfsfunktionen zu generieren.

In Alf gibt es, wie in der UML auch, die Möglichkeit, mithilfe der Anweisung *classify* den Typ von Objekten zur Laufzeit zu ändern. Die Anweisung basiert auf dem Verständnis der UML von Objekten. In der UML sind Objekte generisch und haben *Slots* in denen Operationen und Eigenschaften enthalten sein können. Klassen sind Sätze von Operationen und Eigenschaften, die Objekten zur Laufzeit hinzugefügt werden können. Zur Laufzeit können die Klassen eines Objektes verändert werden. Es können Klassen mit den dazugehörigen *Slots* hinzugefügt und entfernt werden. Dieses Verhalten lässt sich in dynamisch typisierten Sprachen wie Javascript umsetzen, wo Objekte zur Laufzeit geändert werden können. In statisch typisierten Sprachen wie Java, C# oder QML ist dies nur schwer möglich. Im Allgemeinen ist es nicht möglich ein neues Objekt aus den Daten des alten Objektes zu erstellen und dieses zu benutzen, da andere Objekte noch auf das alte Objekt verweisen könnten.

Sequenzen können in Alf mit Werten in geschweiften Klammern initialisiert werden. Es lassen sich wie in `{1,2,3}` alle Werte angeben oder ein Wertebereich wie in `{1..3}`. Wertebereiche können in C# mittels LINQ erzeugt werden: `Linq.Range(1,3)`. LINQ ist eine Klassenbibliothek des .NET-Frameworks zur SQL-ähnlichen Abfrage von verschiedenen Datenquellen. Sowohl in Java als auch in QML beinhaltet die Standardbibliothek keine Methode zur Erzeugung eines Wertebereichs. Hier bietet es sich an, einfach eine Hilfsfunktion zu implementieren. Initialisierungen durch Angabe aller Werte können in Java durch *double-brace initialization* implementiert werden. In C# können Collections und Arrays wie bei der Objektinitialisierung mit einer Liste von Werten in geschweiften Klammern initialisiert werden: `new List<int> {1,2,3}`. In QML können Listen nicht direkt initialisiert werden. Auch hier kann eine Hilfsfunktion die Initialisierung übernehmen. Alle drei Eingabesprachen unterstützen die Angabe von Initialisierungswerten beim Erzeugen von Arrays: `new int[] {1,2,3}`. Bei Sequenzen muss der Typ der Sequenz in Java, C# und QML durch eine Liste oder ein Array ausgetauscht werden. Diese können dann wie beschrieben initialisiert werden.

Auf ein Element einer Sequenz kann in Alf wie bei Arrays in Java, C# und QML mit

eckigen Klammern zugegriffen werden. Wenn Sequenzen in Arrays übersetzt wurden, kann der Sequenzzugriff aus dem Alf-Code einfach übernommen werden. Wenn Sequenzen hingegen als Listen übersetzt wurden, dann muss der Sequenzzugriff in Java und QML durch die *get*-Methode ausgetauscht werden. Aus `a[i]` wird in Java und QML `a.get(i)`. C# unterstützt den Zugriff auf Elemente in einer Liste mit eckigen Klammern. Daher kann der Alf-Code in C# einfach übernommen werden.

UML Assoziationen werden in Alf syntaktisch auf zwei Arten unterstützt. Sie können wie Attribute verwendet werden oder mithilfe von Linkoperationen bearbeitet werden.

Binäre Assoziationen mit benannten Enden können in Alf wie Attribute benutzt werden. Es funktioniert die normale Zuweisung und der Punktoperator mit binären Assoziationen[6, Kap. 8.3.13, 8.8]. Assoziationen werden in Java und QML durch Attribute, mit oder ohne Getter- und Settermethoden, implementiert. In C# werden sie durch Eigenschaften ersetzt. Binäre Assoziationen können also in Alf, Java, C# und QML auf die gleiche Weise verwendet werden. Die Zuweisung an Assoziationen in Alf zum Erstellen oder Löschen einer Assoziation kann auch als gleich Zuweisung in Java, C# und QML übersetzt werden. Der Zugriff auf Assoziationen mittels des Punktoperators kann ebenfalls in Java, C# und QML übernommen werden.

Mithilfe von *Linkoperationen* können beliebige, auch n-äre, Assoziationen erstellt und gelöscht werden. Ist die von einer *Linkoperation* betroffene Assoziation binär und hat benannte Enden, dann kann die *Linkoperation* wie im vorigen Absatz beschrieben in die Syntax mit dem Zuweisungsoperator überführt werden. Diese kann dann in Java, C# und QML übersetzt werden. Nichtbinäre Assoziationen werden als Assoziationsklassen in Java, C# und QML implementiert. Über diese Klassen und deren Methoden können die Linkoperationen für n-äre Assoziationen in Java, C# und QML übersetzt werden. Ein Beispiel für eine ternäre Assoziation als Klasse in Java, C# und QML ist in Abbildung 4.4 dargestellt. Neue Assoziationen können mithilfe der statischen Methode `createAssoc` erstellt werden. Bestehende Assoziationen können mithilfe der statischen Methode `destroyAssoc` gelöscht werden. Die statische Methode `clearAssoc` löscht alle bestehenden Assoziationen. Diese drei Methoden entsprechen den drei *Linkoperationen* *createLink*, *destroyLink*, *clearAssoc* aus Alf. Die *Linkoperationen* haben auf diese Weise die gleiche Syntax in Alf wie in Java, C# und QML. Gelesen kann die Assoziation über die restlichen statischen Methoden. Auch hier ist die Syntax in Alf, Java, C# und QML die gleiche. Damit ist die Übersetzung von Assoziationen von Alf nach Java, C# und

Abbildung 4.4: Beispiel der Übersetzung einer ternären Assoziation nach Java, C# und QML. Entnommen aus [6, Kap. 8.3.9].

QML einfach.

4.4 Operationen

Die arithmetischen, logischen und Bitmanipulationsoperatoren übernimmt Alf aus Java. Abbildung 4.5 zeigt die Übereinstimmung der Operatoren. Die Operatoren können direkt von Alf-Code in Java übersetzt werden. Bei der Übersetzung von Alf-Code in C# gibt es nur einen Fall, indem es keine direkte Abbildung gibt. Dieser Fall ist der vorzeichenlose Shiftoperator >>> und die dazugehörige Zuweisungskombination >>>=. Diese Operatoren sind in Java und Alf notwendig, da es in diesen Sprachen keine vorzeichenlosen Ganzzahltypen gibt, mit denen gerechnet werden kann. Mit dem vorzeichenlosen Typ *UnlimitedNatural* in Alf kann nicht gerechnet werden. Da es aber vorzeichenlose Ganzzahltypen in C# gibt, gibt es hier keine Notwendigkeit für den vorzeichenlosen Shiftoperator. Der Operator kann nach C# übersetzt werden indem sichergestellt wird, dass der linke Operand vorzeichenlos ist. Dies kann durch Casten des Operanden erreicht werden:

Alf	Java	C#	QML
`--`, `++`	`--`, `++`	`--`, `++`	`--`, `++`
`!`, `~`	`!`, `~`	`!`, `~`	`!`, `~`
`+`, `-`, `*`, `/`, `%`	`+`, `-`, `*`, `/`, `%`	`+`, `-`, `*`, `/`, `%`	`+`, `-`, `*`, `/`, `%`
`&&`, `||`	`&&`, `||`	`&&`, `||`	`&&`, `||`
`^`, `|`, `&`	`^`, `|`, `& &`	`^`, `|`, `&`	
`<`, `>`, `<=`, `>=`	`<`, `>`, `<=`, `>=`	`<`, `>`, `<=`, `>=`	`<`, `>`, `<=`, `>=`
`<<`, `>>`, `>>>`	`<<`, `>>`, `>>>`	`<<`, `>>`, `(((uint)i)>>j)`	
`+=`, `-=`, `/=`, `*=`, `%=`	`+=`, `-=`, `/=`, `*=`, `%=`	`+=`, `-=`, `/=`, `*=`, `%=`	`+=`, `-=`, `/=`, `*=`, `%=`
`|=`, `^=`, `&=`	`|=`, `^=`, `& &=`	`|=`, `^=`, `&=`	
`<<=`, `>>=`, `>>>=`	`<<=`, `>>=`, `>>>=`	`<<=`, `>>=`, `(i=((uint)i)>>j)`	

Abbildung 4.5: Übersetzung der arithmetischen, logischen und relationalen Operatoren.

`((uint)i)>>j`. QML bietet keine Bitmanipulations- und Shiftoperatoren. Die Shiftoperatoren können durch simple Multiplikation und Division von Zweierpotenzen ersetzt werden. Der Ausdruck `x<<3` lässt sich durch `x*8` ersetzen, da eine Linksverschiebung um drei Bits einer Multiplikation mit $2^3 = 8$ entspricht. Für die Bitmanipulationsoperatoren gibt es keine einfache Übersetzung. Hier hilft nur eine Hilfsfunktion.

Sprache	Typvergleich
Alf	`obj hastype MyType`
Java	`obj.getClass() .equals( Class.forName("MyType") )`
C#	`obj.GetType() == typeof(Typ)`

Abbildung 4.6: Übersetzung des *hastype*-Operators von Alf nach Java und C#.

In Alf kann der Typ eines Objektes mit den beiden Operatoren *instanceof* und *hastype* abgefragt werden. Mit dem Operator *hastype* kann getestet werden, ob ein Objekt genau den abgefragten Typ hat: `obj hastype MyType`. Hier liefert `hastype` den Wert wahr zurück, wenn `obj` genau den Typ `MyType` hat. Bei von `MyType` abgeleiteten Typen wertet *hastype* zu *falsch* aus. Der Operator *instanceof* testet ob, ein Objekt wie ein Objekt des angegebenen Typs behandelt werden darf. Der Ausdruck `obj instanceof MyType` testet ob `obj` den Typ `MyType` oder den Typ einer Unterklasse von `MyType` hat. Für den *instan-*

ceof-Operator in Alf gibt es in Java und QML ebenfalls den Operator *instanceof*. In C# heißt der Operator *is*. Für den *hastype*-Operator gibt es in keiner der drei Sprachen der Testfallgeneratoren eine direkte Abbildung. Das Verhalten kann in Java und C# emuliert werden, indem die Klasse des Objektes mit dem zu testenden Typ auf Gleichheit getestet wird. Die Implementierung dazu ist in Abbildung 4.6 dargestellt. In QML kann der Typ eines Objektes nicht abgefragt werden. Hier kann der *hastype*-Operator implementiert werden, wenn bei der Generierung der Datentypen aus dem UML Modell eine Methode generiert wird, die den Namen des Typs als String zurückliefert. Damit kann der *hastype*-Operator durch einen Stringvergleich implementiert werden.

Der Isolationsoperator *$expr* in Alf isoliert sämtliche bei der Auswertung des Ausdrucks vorkommenden Objekte von anderen Threads. Kein anderer Thread kann in die in dem Ausdruck vorkommenden Objekte schreiben solange der Ausdruck noch ausgewertet wird. In Java, C# und QML ist es nicht einfach möglich alle an der Auswertung eines Ausdrucks beteiligten Objekte zu locken. Da alle Objekte zu Beginn der Ausdrucksauswertung bekannt sein müssen, reicht es nicht aus Objekte bei Bedarf zu locken. Es kann auch keine Liste aller Objekte aufgestellt werden, die an der Auswertung eines Ausdrucks beteiligt sind. Die Frage, ob ein Objekt beteiligt ist, ist unentscheidbar. Eine Lösung für das Problem könnte darin bestehen, alle anderen Threads zu blockieren, wenn ein isolierter Ausdruck ausgewertet werden soll. Dazu kann ein Semaphor benutzt werden, das ein Ticket für jeden Thread enthält. Jeder Thread erwirb ein Ticket für jeden Ausdruck, der nicht isoliert werden soll. Bei der Auswertung eines isolierten Ausdrucks muss ein Thread alle Tickets reservieren, um alle anderen Threads zu stoppen. In der Praxis verwendet aber weder der Graphwalker noch der Spec Explorer Threads. In Conformiq teilen sich Threads keinen gemeinsamen Speicher. Die Kommunikation zwischen Threads findet über den Austausch von Nachrichten statt. Daher muss der Isolationsoperator im Kontext der Testfallgenerierung nicht übersetzt werden.

Alf verwendet wie alle C-artigen Programmiersprachen das `==` für den Vergleich auf Gleichheit und `!=` für den Vergleich auf Ungleichheit. Bei der Äquivalenz von Daten wird in Alf zwischen Referenztypen und Wertetypen unterschieden. Referenztypen wie die UML *Class* werden auf Referenzgleichheit geprüft. Es wird getestet ob zwei Objektreferenzen auf dasselbe Objekt verweisen. In Java, C# und QML werden Objekte von Klassen ebenfalls auf Referenzgleichheit geprüft. Wertetypen wie der UML *DataType*, Strings und andere primitive Typen werden auf Wertegleichheit geprüft. Bei der Wertgleichheit wird

	Alf	Java	C#	QML
Primitive Typen (Wertevergleich)	==	==	==	==
Strings (Wertevergleich)	==	`str.Equals(otherStr)`	==	==
DataTypes (Wertevergleich)	==	`obj.Equals(otherObj)`	==	==
Klassen (Referenzvergleich)	==	==	==	==

Abbildung 4.7: Die Übersetzung des Gleichheitsoperators.

der Inhalt der Objekte miteinander verglichen. Bei primitiven Typen verhalten sich Java, C# und QML gleich. Hier werden immer die Werte verglichen. Strings sind in Java Objekte und werden auf Referenzgleichheit geprüft, wenn der Gleichheitsoperator verwendet wird. Um den Stringinhalt zu vergleichen, muss die Methode *equals()* aufgerufen werden: `string.equals(otherString)`. In C# überschreibt die Stringklasse den Gleichheitsoperator. In QML sind Strings ebenfalls mit dem Gleichheitsoperator vergleichbar. Daher können in C# und QML zum Vergleichen von Strings der Gleichheitsoperator verwendet werden. *DataTypes* werden in Alf auf Wertgleichheit geprüft. Dabei werden von zwei Objekten des gleichen *DataTypes* die einzelnen Felder miteinander verglichen. In C# entsprechen die *structs* dem *DataType*. *Structs* werden auch auf Referenzgleichheit geprüft. In QML gibt es hierfür die *Records*, welche auf Wertgleichheit geprüft werden. Java unterstützt das Konzept der Werttypen nicht von Haus aus. Durch Überschreiben der Methode *equals()* kann aber auch ein Wertvergleich für Klassen implementiert werden. Dies muss bei der Übersetzung des *DataTypes* aus dem UML Modell in die Java Klasse geschehen.

Mit Alf können auch UML Signale gesendet und empfangen werden. Das Senden geschieht durch den Aufruf des Signalempfängers auf dem Empfängerobjekt wie bei einem Operationsaufruf. Signale können über die *accept*-Anweisung empfangen werden. Java und C# unterstützen Signale nicht von Haus aus. Über Bibliotheken lässt sich die Funktionalität nachrüsten. Die Übersetzung der Alf-Konstrukte für die Signalbehandlung ist daher von diesen Bibliotheken abhängig. Eine Übersetzung in Funktionsaufrufe ist jedoch nicht schwer. In QML gibt es *Ports*, die über die Nachrichten verschickt werden können. Sie entsprechen den UML Signalen. Allerdings können nur *Records* über *Ports* versandt werden. Die *accept*-Anweisung in Alf zum Empfangen von Nachrichten über ein Signal wird in QML übersetzt in den Aufruf der blockierenden Methode *receive* auf dem

Portobjekt.

4.5 Kontrollfluss

Die Kontrollflussstrukturen in Alf sind die üblichen Einfachverzweigungen (if), Mehrfachverzweigungen (switch), While- und For-Schleifen, die es auch in Java, C# und QML gibt. Allerdings haben diese Strukturen in Alf teilweise andere Syntax und Semantik.

```
int a = 4;
if(a % 3 == 0) {
        a *= 2;
} else if(a % 3 == 1) {
        a *= 3;
} else if(a % 3 == 2) {
        a *= 4;
}
```

Abbildung 4.8: Beispiel der Übersetzung einer *if-or*-Verzweigung von Alf zu Java, C# und QML. Das *or* wurde durch *else* ersetzt.

Für die *if*-Verzweigung gibt es in Alf das *if-else*-Konstrukt, welches es auch in Java, C# und QML gibt. Dieses lässt sich ohne Veränderung in Java, C# und QML übernehmen. Zusätzlich gibt es in Alf aber auch noch die nebenläufige Verzweigung, bei der mehrere Bedingungen über das Schlüsselwort *or* verknüpft werden. Alle nebenläufig verknüpften Bedingungen werden parallel ausgewertet und ein beliebiger Zweig, dessen Bedingung zu wahr ausgewertet wurde, wird ausgeführt. Die Alf-Spezifikation legt nicht fest, wie ein Zweig ausgewertet wird. Daher kann immer der erste Zweig ausgeführt werden, dessen Bedingung wahr ist. Damit kann bei der Übersetzung das Schlüsselwort *or* durch ein *else* ausgetauscht werden. Die Übersetzung des Beispiels aus Abbildung 3.10 nach Java, C# und QML ist in Abbildung 4.8 zu sehen.

Die Mehrfachverzweigung mittels *switch* funktioniert in Alf ähnlich wie in Java, C# und QML. Es gibt kein *fall-through*, aber auch keine *break*-Anweisung. Mehrere Fälle können zusammengefasst werden, um denselben Codeblock zu benutzen. Außerdem gibt es den

default-Fall, der benutzt wird, wenn kein anderer Fall zutrifft. Die Mehrfachverzweigung ist in Java, C# und QML mächtiger. In Java und QML gibt es den *fall-through* und in C# gibt es ein *goto*, um zu anderen Fallmarken zu springen. Da hier aber nur die Übersetzung von Alf in Java, C# und QML und nicht in die entgegengesetzte Richtung von Interesse ist, sind auch diese Unterschiede nicht von Bedeutung. Bei der Übersetzung von Alf in Java, C# und QML muss lediglich am Ende jedes Codeblocks eine *break*-Anweisung eingeführt werden.

While- und *do-while*-Schleifen haben in Alf die gleiche Syntax und Semantik wie in Java, C# und QML. Sie können einfach übernommen werden. Bei den *for*-Schleifen gibt es in Alf zwei Varianten. Bei wertebereichsbasierten *for*-Schleifen wird ein Wertebereich definiert, der von der Schleifenvariable durchiteriert wird:

```
for(i : 1..10).
```

Diese Art der Schleifen kann folgendermaßen nach Java, C# und QML übersetzt werden:

```
for(int i=1; i<=10; ++i).
```

Die zweite Art von *for*-Schleifen ist das Iterieren über eine Sequenz. Für die einfache *foreach*-Schleife übernimmt Alf die Syntax und Semantik aus Java:

```
for( MyType i : myTypeSeq ),
```

wobei `myTypeSeq` eine Sequenz von `MyType`-Objekten ist. Die Übersetzung von Alf nach Java entfällt durch die gleiche Syntax. C# verwendet eine ähnliche Syntax:

```
foreach( MyType i in myTypeSeq ).
```

QML bietet nur die dreiteilige *for*-Schleife `for(;;)`. Die Übersetzung der *foreach*-Schleife von Alf nach QML hängt von der Übersetzung der Sequenz ab. Um über Arrays zu iterieren, bietet sich die klassische, indexbasierte *for*-Schleife an:

```
for(int i=0; i<myTypeSeq.length; ++i) {MyType t = myTypeSeq[i]; ... }
```

Wird die Sequenz in den Listentyp *Vector* in QML übersetzt, kann ein Iterator zur Implementierung der *foreach*-Schleife eingesetzt werden:

```
for(Enumeration<MyType> i_it =  myTypeSeq.elements();
        i_it.hasMoreElements();)
{
        MyType i = i_it.nextElement();
}


Integer[] a = {1,2,3,4,5,6};
Integer[] b = {4,5,6};
for(Integer i : a, Integer j : b)
{
        StandardOutputChannel.writeInteger(i+j);
}
```

Abbildung 4.9: Beispiel einer *for*-Schleife mit mehreren Sequenzen

```
int[] a = {1,2,3,4,5,6};
int[] b = {4,5,6};
IEnumerator<int> it_b = a.AsEnumerable().GetEnumerator();
for(int i in a)
{
        int j = it_b.MoveNext() ? it.Current : default(int);

        Console.Write(i+j);
}
```

Abbildung 4.10: Beispiel der Übersetzung einer *for*-Schleife mit mehreren Sequenzen von Alf in C#

In Alf kann über mehrere Sequenzen gleichzeitig iteriert werden. Ein Beispiel ist in Abbildung 4.9 dargestellt. Dabei gibt es für jede Sequenz eine Schleifenvariable. Nach jedem Schleifendurchlauf erhalten alle Schleifenvariablen das nächste Element ihrer Sequenz. Haben die Sequenzen nicht die gleiche Länge, dann entscheidet die Länge der ersten Se-

```
int[] a = new int[] {1,2,3,4,5,6};
int[] b = new int[] {4,5,6};
Enumeration<int> b_it = arrayToList<int>(b).elements();
for(Enumeration<int> a_it =  arrayToList<int>(a).elements();
        a_it.hasMoreElements();)
{
        int a = a_it.nextElement();
        int j = b_it.hasMoreElements() ? b_it.nextElement() : 0;

        trace(i+j);
}
```

Abbildung 4.11: Beispiel der Übersetzung einer *for*-Schleife mit mehreren Sequenzen von Alf in QML

```
int[] a = new int[] {1,2,3,4,5,6};
int[] b = new int[] {4,5,6};
Iterator<Integer> b_it = Arrays.asList(b).iterator();
for(int i : a)
{
        int j = b_it.hasNext() ? b_it.next() : 0;

        System.out.write(i+j);
}
```

Abbildung 4.12: Beispiel der Übersetzung einer *for*-Schleife mit mehreren Sequenzen von Alf in Java

quenz über die Anzahl der Schleifendurchläufe. Im Beispiel wird die Schleife sechs Mal durchlaufen. Ab der vierten Iteration ist die Schleifenvariable `j` leer. Mithilfe von zusätzlichen Iteratoren lässt sich dieses Verhalten auch in Java und C# implementieren. Die Implementierungen für C#, Java und QML zum Beispiel aus Abbildung 4.9 sind in den

Abbildungen 4.10, 4.12 und 4.11 zu sehen.

4.6 Variablen und Funktionsaufrufe

Um Variablen in Alf anzulegen, gibt es zwei Möglichkeiten. Es gibt die aus C-artigen Sprachen bekannte Syntax, bei der der Typ gefolgt vom Variablennamen angegeben wird:

```
Integer i = 12;
```

Andererseits können wie in OCL auch Variablen mithilfe des Schlüsselwortes *let* angelegt werden:

```
let i : Integer = 12;
```

Beide Varianten sind semantisch äquivalent. In Java, C# und QML wird die erstgenannte Variante verwendet. Bei der Übersetzung muss nur die Syntax geändert werden.

```
static List<String> collectHelper(List<Person> persons)
{
        List<String> res = new List<String>();
        for(Person p : persons)
        {
                res.add(p.name);
        }
        return res;
}
```

Abbildung 4.13: Beispiel der Übersetzung der Sequenzoperation *collect* aus Alf in eine Hilfsmethode in Java.

Alf definiert eine Menge von Sequenzoperationen, die auf Sequenzen als Ganzes arbeiten. Sowohl Java als auch QML besitzen auf sprachlicher Ebene und mithilfe der Standardbibliothek keine Möglichkeiten die Sequenzoperationen einfach nachzubilden. Die gängige

Methode in Java und QML mit Collections oder Arrays zu arbeiten ist es, mit einer *for*-Schleife über diese zu iterieren. Schleifen sind aber Anweisungen und keine Ausdrücke und lassen sich somit nicht in Ausdrücken verwenden. In der Java-Version 8 können Sequenzoperationen mit der neu eingeführten *Streams Api* und Lambdaausdrücken nachgebildet werden. In QML und in Java, sofern nicht Java 8 verwendet wird, müssen die Sequenzoperationen durch Hilfsmethoden implementiert werden. Zum Beispiel kann mithilfe der Hilfsmethode, die in Abbildung 4.13 dargestellt ist, der Alf-Ausdruck

```
persons->collect p (p.name)
```

in Java folgendermaßen übersetzt werden:

```
collectHelper(persons)
```

Alf	C#
seq->reduce Sum	seq.Aggregrate((old,next) => old.Sum(next))
seq->select x (x>1)	seq.Where(x => (x>1))
seq->reject x (x>1)	seq.Where(x => !(x>1))
seq->collect x (x.ToString())	seq.Select(x => x.ToString())
seq->iterate x (x.ToString())	seq.Select(x => x.ToString())
seq->forAll x (x>1)	seq.All(x => (x>1))
seq->exists x (x>1)	seq.Any(x => (x>1))
seq->one x (x>1)	seq.GroupBy(x => (x>1)).Any(x => x.Key && (x.Count() == 1))
seq->isUnique x (x % 10)	seq.Select(x => x % 10).GroupBy(x => x) .All(x => x.Count == 1)

Abbildung 4.14: Übersetzung der Sequenzoperation von Alf nach C#.

In C# gibt es die Bibliothek LINQ mit der die Sequenzoperationen implementiert werden können. Abbildung 4.14 zeigt, wie die einzelnen Sequenzoperationen mithilfe von LINQ in C# übersetzt werden können. Die Operation *reduce* reduziert eine Sequenz mithilfe einer angegebenen Operation auf einen Wert. Die C#-Implementierung nutzt hierfür die Methode *Aggregate* aus LINQ. Problematisch bei der Implementierung ist der Fall,

in dem die Operation ein Operator wie zum Beispiel + ist, und keine Methode. Dieser Fall muss abgefangen und entsprechend behandelt werden. Die Übersetzungen für die Operationen *select* bis *exists* sind relativ einfach. Interessant ist die Implementierung der Operation *one*. Hier wird die Sequenz nach Elementen, die die gegebene Bedingung erfüllen, und Elementen, die die gegebene Bedingung nicht erfüllen, gruppiert. Anschließend wird geprüft, ob es eine Gruppe gibt, die genau ein Element enthält und die gegebene Bedingung erfüllt. Die Operation *isUnique* prüft, ob der Schlüssel für jedes Element der Sequenz maximal einmal vorkommt. Die C#-Implementierung gruppiert die Sequenz nach den Schlüsseln und überprüft, ob jede Gruppe genau ein Element enthält.

In der UML hat jede Klasse eine statische Methode *allInstances()*, die eine Liste aller Instanzen der Klasse zurückgibt. Diese Methode gibt es auch in Alf, nicht jedoch in Java, C# oder QML. Eine Implementierung für die Sprachen Java und C# ist jedoch nicht kompliziert. In allen Konstruktoren der Klasse muss eine *WeakReference* des neuen Objekts in einer statischen Liste in der Klasse gespeichert werden. *WeakReferences* in Java und C# sind Objektreferenzen, die von der Garbage Collection nicht berücksichtigt werden. Die Garbage Collection kann Objekte freigeben, auf die noch *WeakReferences* verweisen. Daher werden Objekte nicht durch die Liste aller Instanzen einer Klasse am Leben gehalten. Die Implementierung von *allInstances()* muss dann nur diese Liste zurückgeben. Die Implementierung für C# ist in Abbildung 4.15 dargestellt. In QML gibt es keine *WeakReferences*. Daher lässt sich die Methode *allInstances()* nicht einfach für Conformiq übersetzen.

In diesem Abschnitt wurde eine mögliche Abbildung von Alf-Code in die Eingabesprachen der drei Testfallgeneratoren vorgestellt. Gezeigt wurde die Abbildung der Ausdrücke und Anweisungen von Alf. Diese sind für die Verwendung als Wächter- und Effektbeschreibungssprache von Interesse. Die Alf-Syntax zur Definition von strukturellen Elementen wie Klassen ist für den Awnedungsfall als Wächter- und Effektbeschreibung uninteressant. Wächterausdrücke sind boolsche Ausdrücke. Sie bestehen meistens aus logischen und arithmetischen Operationen. Diese lassen sich fast ohne Änderung von Alf in die Eingabesprachen der Testfallgeneratoren übersetzen. Hilfreich sind auch Sequenzoperationen um Aussagen über den Inhalt von Sequenzen treffen zu können. Die Übersetzung dieser ist nicht trivial. In C# können die Sequenzoperationen mit LINQ-Ausdrücken realisiert werden. In Java und QML müssen hierfür Hilfsfunktionen generiert werden. Effektcode besteht wie der Rumpf von Methoden aus einer Anweisungssequenz. Haupt-

```
public class AllInstancesExample
{
        static List<WeakReference> instances
                = new List<WeakReference>();

        public AllInstancesExample()
        {
                addInstance(this);
        }

        private addInstance(AllInstancesExample instance)
        {
                // remove references to collected instances
                instances = instances.Where(x => !x.IsAlive).ToList();
                // add new instance
                instances.Add(instance);
        }

        public static List<AllInstancesExample> allInstances()
        {
                return instances
                .Where(r => r.IsAlive)
                .Select(r => (AllInstancesExample)r.Target )
                .ToList();
        }
}
```

Abbildung 4.15: Implementierung der Methode `allInstances` in C#

sächlich bestehen diese aus Kontrollstrukturen und Zuweisungen an Variablen und Attribute. Sowohl die Kontrollstrukturen als auch die Zuweisungen sind leicht von Alf in die Eingabesprachen der Testfallgeneratoren übersetzbar. Bei den Kontrollstrukturen ist die *foreach*-Schleife mit mehreren Variablen das aufwendigste Konstrukt. Aber auch dieses

kann mithilfe von Iteratoren in alle drei Sprachen übersetzt werden.

Insgesamt zeigt sich, dass der Großteil der Alfkonstrukte leicht übersetzbar ist. Dies liegt vor allem daran, dass Alf auf Basis von Java und der UML entworfen wurde. Zusätzlich ähneln sich auch die Sprachen Java, C# und QML stark. Probleme bei der Übersetzung bereiten vor allem die UML-spezifischen Sprachkonstrukte wie die *classify*-Anweisung oder der Isolationsoperator. Diese Konstrukte sind schlecht in Java, C# und QML übersetzbar, weil sie in diesen Sprachen nicht unterstützt werden. Zur Übersetzung einiger weiterer Alf-Konstrukte müssen auf Klassenebene Anpassungen gemacht werden. Darunter fallen zum Beispiel statische Funktionen zur Objekterzeugung oder Hilfsfunktionen. Es zeigt sich weiterhin, dass C# die einfachsten Übersetzungen ermöglicht. Hier erweist sich die Klassenbibliothek LINQ als besonders hilfreich. QML hingegen ist gegenüber Java und C# sehr viel eingeschränkter. Dementsprechend fällt hier die Übersetzung auch nicht so leicht.

5 Integration in Fokus!MBT

Zur Demonstration der Praxistauglichkeit der Abbildung von Alf in die Eingabesprachen der Testfallgeneratoren wurde im Rahmen dieser Arbeit ein Alf-zu-C#-Übersetzer für Fokus!MBT entwickelt. Fokus!MBT besitzt bereits eine Anbindung an den Spec Explorer zur Testfallgenerierung aus Testfallentwurfsmodellen. Vor dieser Arbeit wurde der Wächter- und Effektcode in den Zustandsmaschinen in C# geschrieben. Wenn ein anderer Testfallgenerator integriert werden sollte, dann müsste dieser Code für den neuen Testfallgenerator neu geschrieben werden.

Um den Spec Explorer zur Testfallgenerierung zu benutzen, generiert Fokus!MBT die *Cord*-Konfiguration und das Modellprogramm in C# aus dem UML Modell und den Optionen für den Spec Explorer. Mit der Konfiguration und dem Modellprogramm wird der Spec Explorer ausgeführt. Dieser schreibt die durch Modellexploration und anschließendem *scenario slicing* generierten Testfälle in eine .seexpl-Datei. Aus dieser importiert Fokus!MBT die Testfälle und fügt sie in das UML Modell ein.

Der Alf-zu-C#-Übersetzer wurde als eigenständige Komponente für Fokus!MBT entwickelt. Sie klinkt sich bei der Generierung der Eingabe für den Spec Explorer ein und übersetzt die Wächter und Effekte des Zustandsdiagramms des Testfallentwurfsmodells von Alf in C#. Die Übersetzung des restlichen UML Modells muss dabei nicht geändert werden. Nur der C#-Code muss manuell in Alf-Code übersetzt werden.

Im Folgenden wird diese Übersetzungskomponente vorgestellt.

Die Komponente des Alf-zu-C#-Übersetzers wurde als Eclipse Plug-in entwickelt. Fokus!MBT basiert als Erweiterung von Papyrus auf der Eclipse Rich Client Platform (Eclipse RCP). Bei dieser werden Softwarekomponenten als Plug-ins implementiert, welche zur Laufzeit geladen werden. Eine mit der Eclipse RCP erstellten Anwendung besteht lediglich aus einer Sammlung von Plug-ins für die Plattform. Dies hat den Vorteil, das die Anwendung leicht erweiterbar und konfigurierbar ist. Fokus!MBT selber besteht aus

über 50 Plug-ins. Komponenten einer Eclipseanwendung können über Erweiterungspunkte (*extension point*) auf andere Komponenten zugreifen. Erweiterungspunkte dienen als Schnittstelle zwischen Komponenten. Ein Erweiterungspunkt besteht aus einem zu implementierenden Java Interface und einer Menge von zusätzlichen Attributen, die dazu verwendet werden können, um aus mehreren Implementierungen eine auszuwählen. Sie werden durch ein XML Schema definiert. Erweiterungspunkte werden von der Komponente, die den Erweiterungspunkt benutzt, bei der *Extension Registry* registriert. Andere Komponenten können dann ihre Implementierung des Erweiterungspunktes bei der *Extension Registry* eintragen. Anschließend können für einen Erweiterungspunkt alle Implementierungen bei der *Extension Registry* erfragt werden.

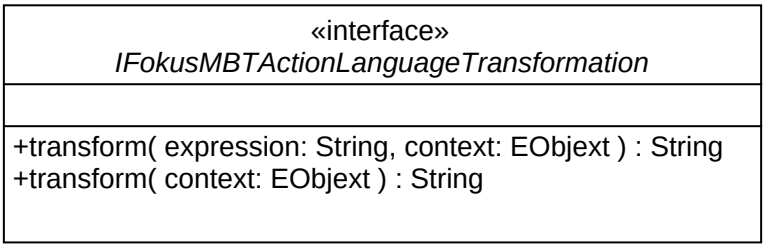


Abbildung 5.1: Die Schnittstelle für Übersetzer.

Für die Übersetzungskomponente wurde ein Erweiterungspunkt angelegt. Dieser Erweiterungspunkt vermittelt Text-zu-Text-Übersetzer für verschiedene Sprachen an Komponenten. Implementierungen des Erweiterungspunktes müssen neben einer Id und ihrem Namen noch die Quell- und Zielsprache der Übersetzung und die Klasse angeben, die den Übersetzer implementiert. Dadurch ist die Schnittstelle nicht auf Alf als Quellsprache und C# als Zielsprache festgelegt. Mit derselben Schnittstelle können auch Übersetzungen zwischen anderen Sprachen verwendet werden. Die Java Schnittstelle, die ein Übersetzer implementieren muss, ist in Abbildung 5.1 dargestellt. Die Methode `transform` der Schnittstelle `IFokusMBTActionLanguageTransformation` übersetzt einen Quelltext einer Sprache in eine andere. Der zu übersetzende Code kann auch nur aus Ausdrücken oder Anweisungen bestehen und muss kein syntaktisch vollständiges Programm sein. Dies ist zum Beispiel bei der Übersetzung von Wächter- und Effektcode für die Testfallgenerierung der Fall. Damit bei der Übersetzung Klassen, Attribute und Methoden aus

dem Kontext des Quelltextes benutzt werden können, wird zusätzlich zum Quelltext der Kontext mit übergeben. Im Falle der Übersetzung von Wächter- und Effektcode von Zustandsdiagrammen ist der Kontext die dazugehörige Transition. Über diese stehen auch das Zustandsdiagramm und andere Elemente des Testfallentwurfsmodells zur Verfügung. Die Klasse *EObject* ist die Basisklasse für alle Klassen, die das Eclipse Modeling Framework benutzen.

Die Übersetzung läuft wie bei Compilern üblich in drei Schritten ab:

1. Parsen des Quelltextes in eine Zwischenrepräsentation
2. Weiterverarbeitung der Zwischenrepräsentation
3. Generieren des Zielsprachencodes aus der Zwischenrepräsentation

Die Implementierung der einzelnen Schritte wird im Folgenden beschrieben.

```
Test :
        ('testExpression' expression += Expression)*
        ('testAssignmentExpression'
        assignExpression += AssignmentCompletion)*
        ('testStatement' statements += Statement)*
        ('testBlock' block = Block);
        //('testStatementSequence'
        //        statement += StatementSequence)* ;
```

Abbildung 5.2: Auszug der ersten Regel der Grammatik des Alf-Parsers von Papyrus.

Für das Parsen des Alf-Quellcodes wird der Alf-Parser aus Papyrus wiederverwendet. Papyrus hat eine auf Xtext basierende Alf-Unterstützung mit einem Editor mit Syntaxhighlighting und Autovervollständigung. Xtext[1] ist ein Eclipse-basiertes Framework, um aus Grammatiken Parser und Editoren für domänenspezifische Sprachen zu generieren. Der Alf-Parser aus Papyrus kann auch alleinstehend verwendet werden. Er wurde für den Alf-zu-C#-Übersetzer benutzt, um sich doppelte Arbeit zu ersparen und von späteren Verbesserungen des Parsers zu profitieren. Nebenbei ist der Alf-Parser der gleiche, der auch für das Syntaxhighlighting im Editor verwendet wird. Daher treten keine Un-

[1] https://eclipse.org/Xtext/

stimmigkeiten zwischen dem Editor und dem Alf-zu-C#-Übersetzer auf. Der Alf-Parser befand sich jedoch zum Zeitpunkt dieser Arbeit noch in der Entwicklung. Dies zeigte sich daran, dass es einige Sicherungskopien der Grammatik im Projekt des Parsers befanden, die Grammatik viele alternative, auskommentierte Regeldefinitionen beinhaltete und dass die erste Regel der Grammatik des Parsers `Test` hieß. Diese Regel ist in Abbildung 5.2 dargestellt. Die erste Regel einer XText-Grammatik ist die, die standardmäßig zum Parsen benutzt wird. Der „Test“-Regel zufolge muss jeder Alf-Code einen `testBlock` haben und kann weitere `testExpressions`, `testAssignmentExpressions` oder `testStatement` enthalten. Die Zeilen, die mit `//` anfangen, stellen Kommentare dar.

Effektcode besteht aus einer Menge von Anweisungen. Um diese parsen zu können, muss der Alf-Code in einen `testBlock` eingebettet werden:

```
textBlock { Code },
```

wobei an die Stelle von `Code` der Effektcode eingefügt wird. Einzelne Ausdrücke, wie sie bei Wächtern verwendet werden, können geparst werden, wenn das Schlüsselwort `testExpression` vorangestellt wird:

```
testExpression Code textBlock { }
```

Auch hier wird `Code` durch den Wächterausdruck ersetzt.

Die Schnittstelle `IFokusMBTActionLanguageTransformation` wurde ungünstigerweise so entworfen, dass bei einem Aufruf der Übersetzungskomponente mit Alf-Code und dem Kontext nicht ersichtlich ist, ob der übergebene Alf-Code Effektcode oder Wächtercode darstellt. Daher kann nicht auf einfache Weise festgestellt werden, ob der Code einen Ausdruck oder eine Menge von Anweisungen darstellt. Aus diesem Grund versucht die Übersetzungskomponente, den übergebenen Alf-Code sowohl als Ausdruck als auch als eine Menge von Anweisungen zu parsen. Bei einem der beiden Parsevorgänge tritt notwendigerweise ein Fehler auf, da Alf-Code nicht gleichzeitig ein gültiger Ausdruck und eine gültige Anweisungssequenz sein kann. Um zu ermitteln, ob der übergebene Alf-Code Ausdrücke oder Anweisungen darstellt, werden sowohl die Anzahl der Parserfehler als auch die Länge der C#-Übersetzung betrachtet. Die Heuristik ist die, dass bei weniger Parserfehlern oder längerem C#-Code wahrscheinlich mehr Alf-Code übersetzt werden konnte. Im Idealfall treten bei genau einem der beiden Parsevorgänge keine Parserfehler auf.

Die Grammatik des Alf-Parsers hat das Problem, dass sie unvollständig ist. Mit dieser Grammatik lassen sich zum Beispiel keine Variablen vom Typ `List<String>` anlegen. Der Typ einer Variable darf in dieser Grammatik nur ein einfacher Identifier sein und kann daher keinen Typparameter haben. Dieser Fehler ist den Entwicklern bereits bekannt.

Der Parser liefert als Ergebnis den geparsten Baum der konkreten Syntax zurück. Nach dem Parsen wird dieser Syntaxbaum zunächst weiterverarbeitet. Dazu wird der Baum traversiert und einige Ersetzungen getätigt. Es werden die primitiven Typen in Variablen- und Schleifenvariablendeklarationen ersetzt. Die Abbildung wurde oben in Abbildung 4.1 gezeigt. Daneben werden die Collectiontypen von Alf in die von C# übersetzt. Die Abbildung hierzu ist in 4.2 dargestellt. Letztlich müssen die globalen Funktionen der Alf Standardbibliothek ersetzt werden. Da es in C# wie auch in Java keine globalen Funktionen gibt, wurde eine Helferklasse in C# mit dem Namen `AlfWrapper` entwickelt. Diese implementiert die globalen Funktionen von Alf. Die C#-Klasse, in die der erzeugte Code eingefügt wird, muss den `AlfWrapper` importieren, um diesen nutzen zu können.

```
public static void add<T>(this ICollection<T> seq, T elem)
{
        seq.Add(elem);
}
```

Abbildung 5.3: Beispiel der Übersetzung von Methodennamen von Alf nach C# mittels Erweiterungsmethoden.

Die Helferklasse übernimmt auch die Übersetzung der Methodennamen der Collectionklassen aus Alf. Wie in Java üblich sind die Methodennamen in Alf in Camel Case mit kleinem Anfangsbuchstaben geschrieben. Im Gegensatz dazu verwendet C# das Pascal Case mit großem Anfangsbuchstaben. Damit stimmen die Methodennamen von Alf-Klassen nicht mit denen aus C# überein. Bei der Weiterverarbeitung des Syntaxbaums können die Methodennamen von Collectiontypen durch die C#-Äquivalente ersetzt werden. Dies bedeutet aber, dass für jeden Methodenaufruf einer Funktion mit dem Namen einer Collectionklassenmethode eine Typüberprüfung des Objektes, auf dem die Methode aufgerufen wird, durchgeführt werden muss. Die Operation ist relativ aufwendig. Die einfachere Methode ist die, dass die Methodennamen der Alf-Collectiontypen nicht über-

setzt werden. Stattdessen werden die Alf-Collectionklassenmethoden zu den Methoden der C#-Collections hinzugefügt. C# bietet dafür das Konzept der Erweiterungsmethoden. Diese ermöglichen es Klassen nachträglich Methoden hinzuzufügen, welche allerdings nur auf die öffentlich verfügbaren Bestandteile der Klasse zugreifen dürfen. Diese Einschränkung ist jedoch unwesentlich für die Verwendung von Erweiterungsmethoden zur Übersetzung von Alf-Collectionklassenmethodennamen. Am Beispiel der Methode `add` ist die dazugehörige Erweiterungsmethode in Abbildung 5.3 dargestellt. Das Schlüsselwort `this` am ersten Methodenparameter markiert diese Methode als Erweiterungsmethode. Diese Methode bildet den Methodennamen `add` in Camel Case auf den Methodennamen `Add` in Pascal Case ab.

```
[template public
        generateElement(switch : SwitchStatement)
        post (trim()) ]

switch ([switch.expression.generateElement() /])
{
        [for(sc : SwitchClause | switch.switchClause) ]
                [sc.generateElement() /]
                break;
        [/for]

        [if (switch.defaultClause->size() > 0)]
                default:
                [switch.defaultClause.statementSequence
                        .generateElement() /]
                break;
        [/if]
}
[/template]
```

Abbildung 5.4: Acceleo-Template am Beispiel der *switch*-Anweisung.

Abschließend wird der Syntaxbaum mithilfe von Modell-zu-Text-Transformation in C#

übersetzt. Als Modell-zu-Text-Transformator kommt das ebenfalls Eclipse-basierte Acceleo[2] zum Einsatz. Acceleo basiert auf der *MOF Model to Text Transformation Language* (MTL). Diese Sprache ist dazu gedacht, bei der modellgetriebenen Softwareentwicklung aus UML Modellen Code zu generieren [7]. Die Sprache basiert auf Templates. Per Patternmatching über dem zu transformierenden Modellelement oder Objekt wird ein passendes Template ausgewählt. Ein Beispiel für ein Template ist in Abbildung 5.4 dargestellt. Es übersetzt eine *switch*-Anweisung aus dem Syntaxbaum in C# Code. Anweisungen an Acceleo werden in eckige Klammern geschrieben. So definiert die Anweisung `[template public generateElement(switch : SwitchStatement) post (trim()) ]` ein Template mit dem Namen `generateElement`, welches auf `SwitchStatement`s matcht. Wie in Templatesprachen üblich gibt es auch in MTL Kontrollflussanweisungen wie *if*-Verzweigungen oder Schleifen und Hilfsdefinitionen mittels `[let]`. Die Anweisung `[for(sc : SwitchClause | switch.switchClause) ][/for]` definiert zum Beispiel eine *for*-Schleife über alle Fallklauseln der *switch*-Anweisung. Ein Template wird wie im Beispiel bei `[sc.generateElement() /]` als Methodenaufruf auf einem Objekt realisiert.

Die im Rahmen dieser Arbeit erstellte Übersetzungskomponente von Alf nach C# implementiert die Übersetzung zum großen Teil. Es gibt jedoch einige Einschränkungen. Der Grund für die Einschränkungen ist der, dass keine Typüberprüfung stattfindet und die Typen von Ausdrücken damit nicht bekannt sind. Der Syntaxbaum des Alf-Parsers bietet keine Möglichkeit, ihn mit Typannotationen zu versehen. Daher ist die Übersetzung von Alf-Code rein syntaktischer Natur. Dies reicht jedoch in den meisten Fällen aus. Für eine vollständige Übersetzung muss eine Typannotation stattfinden.

Eine Einschränkung ist zum Beispiel die, dass die Angabe von Rollen bei Typ- und Funktionsparametern nicht unterstützt wird. Bei Funktionsargumenten kann so zum Beispiel der Parameter angegeben werden. Dadurch können einer Funktion die Argumente in beliebiger Reihenfolge übergeben werden. So wurde im Beispiel `nameList.replaceAt( element => "Peter", index=> 5)` der Index nach dem neuen Element übergeben. Um diesen Code richtig in C# übersetzen zu können und die richtige Reihenfolge der Argumente wiederherzustellen, muss der Typ und die Parameter der Funktion bekannt sein. Hierfür muss die Typannotation stattfinden. Gleiches gilt für die Angabe der Typparameter für generische Typen.

[2] https://eclipse.org/acceleo/

Eine weitere Einschränkung ist die, dass die Objektnavigation mit dem Punktoperator nicht über Sequenzen funktioniert. Dazu müsste für jeden Teilausdruck des Alf-Codes die Multiplizität bekannt sein. Da die Navigation über Sequenzen mit dem Punktoperator nur syntaktischer Zucker für die Sequenzoperation *collect* ist, bedeutet diese Einschränkung kein Verlust der Mächtigkeit von Alf.

Auch das *Reclassify* wird vom Alf-zu-C#-Parser nicht unterstützt. Der Grund dafür ist, dass das Konzept in C# nicht unterstützt wird. Das Verhalten der *Reclassify*-Anweisung kann durch Konstruktoren für jede Oberklasse nachgebildet werden. Dies muss jedoch beim Übersetzen des UML Modells geschehen. Hinzu kommt, dass es wenige Anwendungsfälle für dieses Verhalten gibt.

Das *accept*-Statement zum Empfangen von Signalen wird nicht unterstützt. Beim Spec Explorer werden für Testentwurfsmodelle keine Threads oder verschiedene miteinander kommunizierende Komponenten verwendet. Daher müssen auch keine Signale zum Nachrichtenaustausch verwendet werden.

Linkoperationen aus Alf werden auch nicht von der erstellten Ersetzungskomponente unterstützt. Der Grund ist der, dass bei der Übersetzung der UML Modelle keine n-ären Assoziationen berücksichtigt werden. Daher können diese in Fokus!MBT im Kontext der Testfallgenerierung nicht benutzt werden. Binäre Assoziationen werden unterstützt. Diese sind als Eigenschaften in C# implementiert. Binäre Assoziationen können in Alf aber auch mithilfe der Zuweisung und des Punktoperators verwendet werden. Daher sind die Linkoperationen nicht weiter notwendig.

Der vorgestellte Alf-zu-C#-Übersetzer ist jedoch für die meisten Fälle ausreichend. Im folgenden Abschnitt wird dies anhand eines Beispiels veranschaulicht.

6 Fallstudie

In diesem Abschnitt wird ein Beispiel für die Benutzung von Alf in Testfallentwurfsmodellen mit Fokus!MBT vorgestellt. Das Beispiel verwendet den Spec Explorer zur Testfallgenerierung. Das Beispiel ist eines der Vorführbeispiele in Fokus!MBT. In dem Beispiel sind die Effekte und Wächter in C# geschrieben. Im Rahmen dieser Arbeit wird gezeigt, wie der C#-Code in Alf-Code umgewandelt werden kann und wie der Alf-Code dann mithilfe der vorgestellten Alf-zu-C#-Übersetzungskomponente zurück in das C# für den Spec Explorer übersetzt werden kann.

Das Beispiel modelliert einen Log-in-Server. Um sich einzuloggen, muss der Benutzer seinen Benutzernamen und sein Passwort eingeben. Stimmen Benutzername und Passwort überein, ist der Benutzer eingeloggt. Stimmen Benutzername und Passwort nicht überein, so schlägt der Einlogversuch fehl. Jeder Benutzer hat drei Einlogversuche. Nach dem dritten fehlgeschlagenen Einlogversuch wird der Benutzer gebannt und kann sich nicht mehr einloggen.

Die Testkonfiguration ist in Abbildung 6.1 zu sehen. Der Testkontext beinhaltet den Log-in-Server und eine Testkomponente. Der Log-in-Server ist das SUT. Die Testkomponente ist der Treiber, der die Einloganfragen an den Server sendet. Beide Komponenten, der Server und der Treiber, sind über ihre Ports miteinander verbunden. Der Typ des Ports ist das Interface `LoginInterface`, welches in Abbildung 6.2 dargestellt ist. Die Typen des Eingabeparameters und des Rückgabewertes der Operation `login` sind in Abbildung 6.3 dargestellt. Dadurch wird beschrieben, dass der Log-in-Server eine Operation `login` hat, die einen Benutzernamen und ein Passwort als Eingabe erwartet. Der Rückgabewert der Operation sagt aus, ob der Benutzer erfolgreich eingeloggt oder gebannt ist, oder ob der Benutzer einen falschen Benutzernamen oder ein falsches Passwort eingegeben hat.

Das Testfallentwurfsmodell ist ein Zustandsdiagramm, die das oben beschriebene Verhalten des Log-in-Servers modelliert. Es ist in Abbildung 6.4 dargestellt. Neben dem Start-

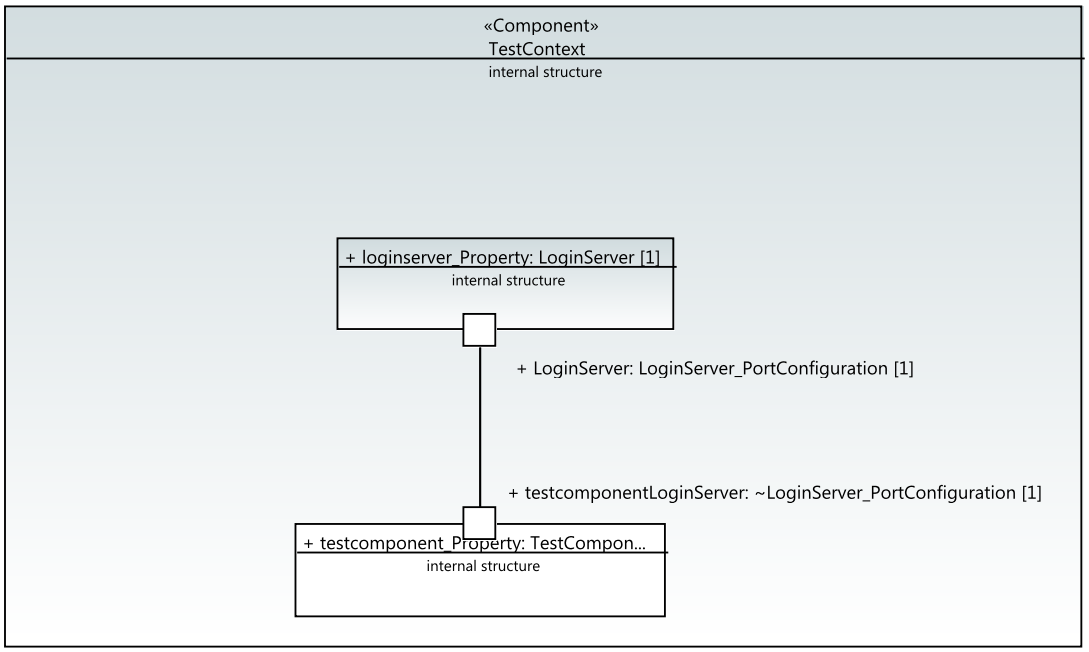


Abbildung 6.1: Die Testkonfiguration.

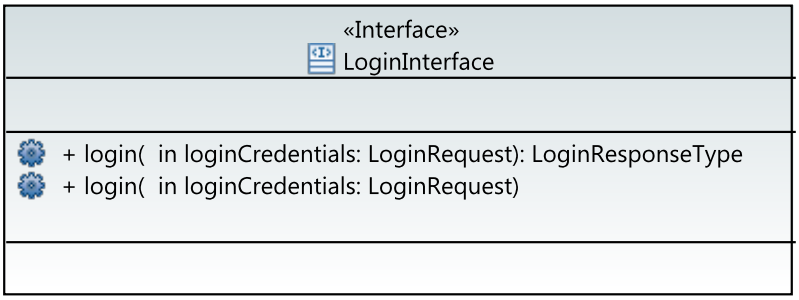


Abbildung 6.2: Das Logininterface ist die Schnittstelle des Log-in-Servers.

und Endzustand gibt es drei relevante Zustände. Das System befindet sich im Zustand `user_logged_out`, solange der Benutzer noch nicht eingeloggt ist und er noch Einlogversuche frei hat. Wenn der Benutzer sich versucht einzuloggen, dann wird eine von vier möglichen Transitionen genommen:

1. Der Benutzername ist falsch.

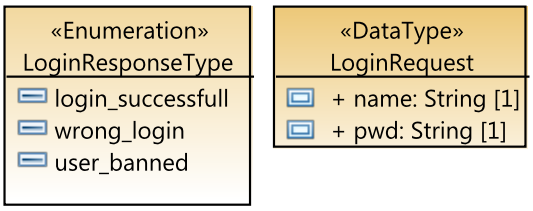


Abbildung 6.3: Die Typen für Parameter und Rückgabewerte der Operation `login` der Schnittstelle des SUTs.

2. Das Passwort ist falsch und der Benutzer hat noch Einlogversuche frei.
3. Die Einlogdaten sind korrekt und der Benutzer hat noch Einlogversuche frei .
4. Der Benutzer hat keine Einlogversuche mehr.

Im Fall 1 und 2 bleibt das System im Zustand `user_logged_out`. Im Fall 3 geht das System in den Zustand `user_logged_in` über und im Fall 4 geht das System in den Zustand `user_banned`.

Das ganze Verhalten des Modells wird über die vier Transitionen beschrieben. Sie werden alle durch die Aktion `login` ausgelöst. Diese markiert das Aufrufen der Operation `login` des Log-in-Servers. Die Effekte und Wächter lassen sich in Papyrus in verschiedenen Sprachen angeben. So kann eine Transition eine natürlichsprachliche Effektbeschreibung haben und daneben eine Effektbeschreibung in einer Programmiersprache. Die vier Transitionen sind mit ihrem Wächter- und Effektcode in den Tabellen 6.5 bis 6.8 zu sehen. Im Code ist `loginCredentials` der Eingabeparameter, wie er auch in Abbildung 6.2 abgebildet ist. Die Variable `result` ist der Rückgabewert. Die statische Methode `Condition.IsTrue()` ist die Übersetzung des Wächters für den Spec Explorer. Diese Anweisung sagt dem Spec Explorer, dass die dazugehörige Regelmethode nur ausgeführt werden kann, wenn die Bedingung wahr ist. Diese Semantik entspricht dem Wächterausdruck einer Transition in einer Zustandsmaschine.

Die Anzahl der Einlogversuche jedes Benutzers wird in der Variable `idAttemptsRegister` gespeichert. `idAttemptsRegister` ist ein Attribut der Kontextklasse des Zustandsdiagramms des Testfallentwurfsmodells. `idAttemptsRegister` ist eine Sequenz vom Typ `IDLoginAttempt`. `IDLoginAttempt` wiederum ist ein UML *DataType* mit dem Benutzer-

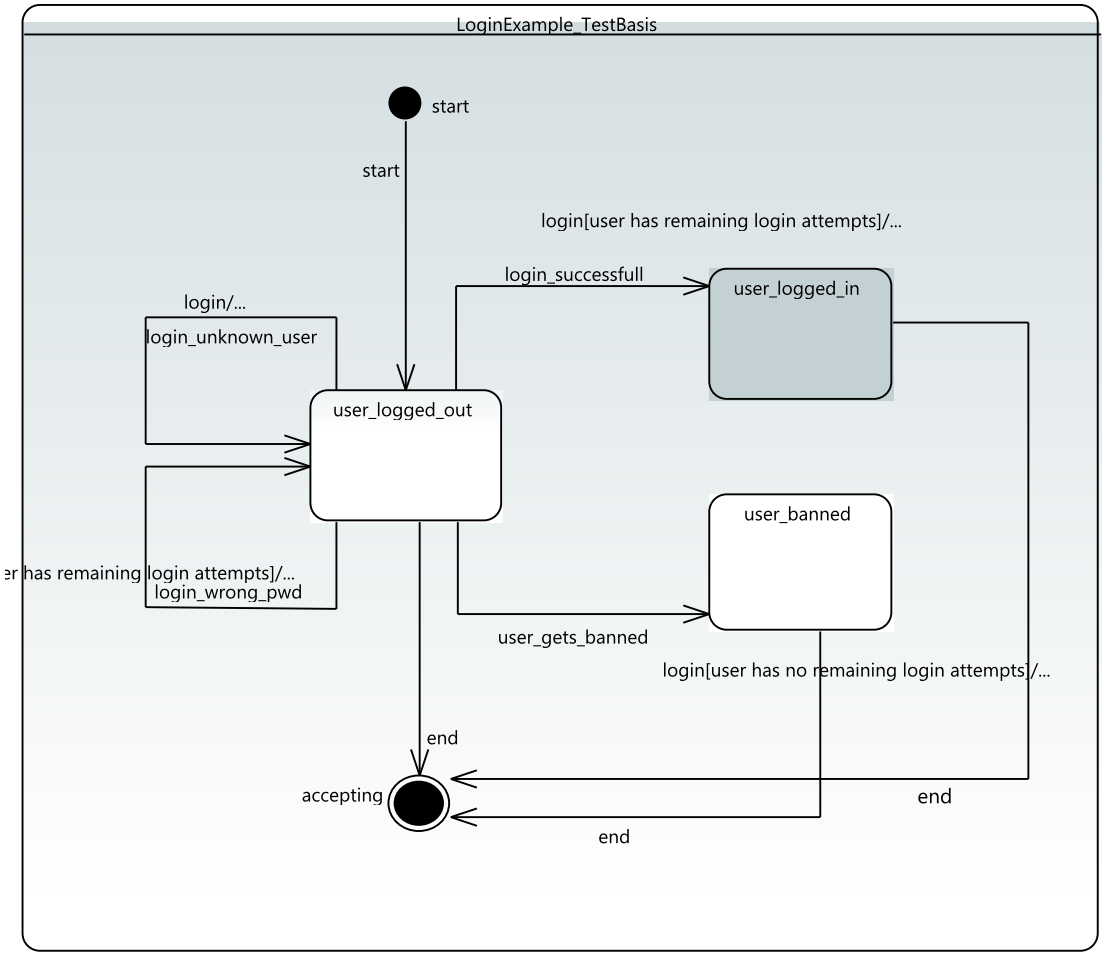


Abbildung 6.4: Das Zustandsdiagramm des Log-in-Beispiels.

namen als String und der Anzahl an Einlogversuchen dieses Benutzers seit dem letzten erfolgreichen Einlogversuch. Auf C#-Seite ist die Sequenz als *List*-Datentyp des .NET-Frameworks implementiert.

Der Wächter- und Effektcode verwendet zwei Hilfsmethoden `hasRemainingAttempt` und `incrementLoginAttempt`. `hasRemainingAttempt` prüft, ob ein Benutzer noch Einlogversuche frei hat. Die Signatur der Methode ist

```
+ hasRemainingAttempt(loginRequest : LoginRequest) : Boolean
```

Die Methode iteriert über alle Benutzer und setzt das Ergebnis auf *false*, wenn der Eintrag

Name	`login_unknown_user`
Anfangsknoten	`user_logged_out`
Endknoten	`user_logged_out`
Auslöser	`login`
Wächter	-
Effekt	`result = LoginResponseType.wrong_login;`

Abbildung 6.5: Die Transition `login_unknown_user`.

Name	`login_wrong_pwd`
Anfangsknoten	`user_logged_out`
Endknoten	`user_logged_out`
Auslöser	`login`
Wächter	`Condition.IsTrue(hasRemainingAttempt(loginCredentials));`
Effekt	`incrementLoginAttempt(loginCredentials);` `result = LoginResponseType.wrong_login;`

Abbildung 6.6: Die Transition `login_wrong_pwd`.

Name	`login_successfull`
Anfangsknoten	`user_logged_out`
Endknoten	`user_logged_in`
Auslöser	`login`
Wächter	`Condition.IsTrue(hasRemainingAttempt(loginCredentials));`
Effekt	`result = LoginResponseType.login_successfull;`

Abbildung 6.7: Die Transition `login_successfull`.

des Benutzers gefunden wurde und dieser bereits mehr als zwei erfolglose Einlogversuche getätigt hat. Die Implementierung hierfür ist in Abbildung 6.9 zu sehen.

Die Methode `incrementLoginAttempt` erhöht die Anzahl der Einlogversuche eines Be-

Name	`user_gets_banned`
Anfangsknoten	`user_logged_out`
Endknoten	`user_banned`
Auslöser	`login`
Wächter	`Condition.IsTrue(!hasRemainingAttempt(loginCredentials));`
Effekt	`result = LoginResponseType.wrong_login;`

Abbildung 6.8: Die Transition `user_gets_banned`.

```
returnValue = true;
foreach (IDLoginAttempt att in idAttemptsRegister)
{
        if (att.id == loginRequest.name && att.attempts > 2)
        {
                returnValue = false;
                break;
        }
}
```

Abbildung 6.9: Implementierung der Hilfsmethode `hasRemainingAttempt` in C#

nutzers um eins. Sie wird aufgerufen, wenn ein Benutzer versucht sich mit einem falschen Passwort einzuloggen. Die Signatur der Methode ist

```
+ incrementLoginAttempt(loginAttempt : LoginRequest) : Integer
```

Die Implementierung dieser Hilfsmethode ist in Abbildung 6.10 zu sehen.

In dem Beispiel gibt es drei Datenpartitionen für die Eingabeparameter der Operation `login`. Diese sind in Tabelle 6.11 dargestellt. Mit diesen drei Datensätzen können alle Transitionen der Zustandsmaschine des Log-in-Systems erreicht werden.

Im Folgenden wird die Übersetzung der Wächterausdrücke, Effekte und Hilfsmethoden am Beispiel der Transition `login_wrong_pwd` und der Hilfsmethode `incrementLoginAttempt`

```
returnValue = -1;
bool usrExists = false;
foreach (IDLoginAttempt att in idAttemptsRegister)
{
        if (att.id == loginAttempt.name)
        {
                usrExists = true;
                att.attempts += 1;
                returnValue = att.attempts;
                break;
        }
}

if (!usrExists)
{
        IDLoginAttempt newAtt = new IDLoginAttempt();
        newAtt.attempts = 1;
        newAtt.id = loginAttempt.name;
        idAttemptsRegister.Add(newAtt);
        returnValue = newAtt.attempts;
}
```

Abbildung 6.10: Implementierung der Hilfsmethode `incrementLoginAttempt` in C#

Name der Partition	Benutzername	Passwort	Erwarteter Rückgabewert
DP_LoginRequest_Successfull	known1	right	login_successfull
DP_LoginRequest_WrongPW	known1	wrong	wrong_login
DP_LoginRequest_Unknown	unknown	*	wrong_login

Abbildung 6.11: Die Datenpartitionen der Eingabeparameter der Operation `login` des Login-Beispiels.

vorgestellt. Die Übersetzung der restlichen Transitionen und Hilfsmethoden ist analog.

Der Wächterausdruck der Transition `login_wrong_pwd` ist:

```
Condition.IsTrue(hasRemainingAttempt(loginCredentials));
```

Bei der Integration des Alf-zu-C#-Übersetzers wurde der Aufruf der statischen Methode `Condition.IsTrue()` in die Übersetzung der Transition ausgelagert. Der Aufruf dieser Funktion gehört nicht zum Wächterausdruck, sondern ist ein Implementierungsdetail von Wächterausdrücken im Spec Explorer. Daher sollten Wächterausdrücke frei von diesem Aufruf sein. Nebenbei führt das Aufrufen dieser Funktion in jedem Wächterausdruck zu viel unnötigem Code. Wie in den Tabellen 6.6 bis 6.8 zu sehen ist, wird die Methode in jedem Wächterausdruck benutzt. Durch die Auslagerung des `Condition.IsTrue()`-Aufrufs in die Übersetzung reduziert sich der C#-Code des Wächterausdrucks auf:

```
hasRemainingAttempt(loginCredentials)
```

Dieser Code ist gleichzeitig gültiger Alf-Code.

Der Effektcode der Transition `login_wrong_pwd` ist:

```
incrementLoginAttempt(loginCredentials);
result = LoginResponseType.wrong_login;
```

Hier muss keine Änderung vorgenommen werden. Der Code ist gültiger Alf-Code.

Bei der Übersetzung der Hilfsmethode `incrementLoginAttempt` müssen nur wenige Änderungen gemacht werden. Der primitive Typ *bool* in C# heißt in Alf *Boolean*. Bei der *foreach*-Schleife muss das Schlüsselwort *in*, welches in C# verwendet wird, durch einen Doppelpunkt ersetzt werden. Letztlich heißt die Operation um ein Element an eine Sequenz anzufügen in Alf *add*, während sie in C# großgeschrieben *Add* heißt. Die Implementierung dieser Methode in Alf ist in Abbildung 6.12 noch mal abgebildet.

Bei der Generierung der Testfälle mit dem Spec Explorer aus Fokus!MBT heraus werden als Erstes das Modellprogramm und die Konfiguration erzeugt. Hierbei wird das Testfallentwurfsmodell, die definierten Typen, die Datenpartitionen und Einstellungen für den Spec Explorer verwendet. In diesem Schritt wird der Alf-zu-C#-Übersetzer verwendet um aus dem Alf-Code, C#-Code zu erzeugen. Dabei stellt sich heraus, dass sowohl bei der Benutzung der Übersetzungskomponente als auch ohne diese, semantisch gleiche Modellprogramme und Konfigurationen generiert werden. Unterschiede gibt es nur bei der

```
returnValue = -1;
Boolean usrExists = false;
foreach (IDLoginAttempt att : idAttemptsRegister)
{
        if (att.id == loginAttempt.name)
        {
                usrExists = true;
                att.attempts += 1;
                returnValue = att.attempts;
                break;
        }
}

if (!usrExists)
{
        IDLoginAttempt newAtt = new IDLoginAttempt();
        newAtt.attempts = 1;
        newAtt.id = loginAttempt.name;
        idAttemptsRegister.add(newAtt);
        returnValue = newAtt.attempts;
}
```

Abbildung 6.12: Übersetzung der Hilfsmethode `incrementLoginAttempt` von C# in Alf

Codeformatierung und auf syntaktischer Ebene. Der aus Alf-Code generierte C#-Code hat weder Leerzeilen noch Einrückung. Der von Hand geschriebene C#-Code, welcher vor der Alf-Integration im Beispiel verwendet wurde, ist dagegen zur besseren Lesbarkeit durch Einrückung und Leerzeilen und -zeichen formatiert. Auf der syntaktischen Ebene bestehen die Unterschiede aus der Verwendung von dem syntaktischen Zucker den C# bietet. Zum Beispiel kann in C# bei der Variablendeklaration das Schlüsselwort *var* verwendet werden anstelle des Variablentyps, wenn aus dem Initialisierungsausdruck der Typ der Variable hervorgeht. Dies wird bei der Generierung des C#-Codes aus Alf ausgenutzt.

```
public static int incrementLoginAttempt
        (ModelProgram.LoginRequest loginAttempt)
{
        int returnValue;
        returnValue = -1;
        bool usrExists = false;
        foreach( var att in idAttemptsRegister )
        {
                if ( att.id==loginAttempt.name ) {
                        usrExists = true;
                        att.attempts += 1;
                        returnValue = att.attempts;
                        break;
                }
        }
        if ( !usrExists ) {
                IDLoginAttempt newAtt = new IDLoginAttempt ();
                newAtt.attempts = 1;
                newAtt.id = loginAttempt.name;
                idAttemptsRegister.Add( newAtt );
                returnValue = newAtt.attempts;
        }
        return returnValue;
}
```

Abbildung 6.13: Die Übersetzung des Hilfsmethode `incrementLoginAttempt` im Modellprogramm.

Die von Alf in C# übersetzte Hilfsmethode `incrementLoginAttempt` ist in Abbildung 6.13 zu sehen. Zu sehen ist neben Methodenrumpf der Funktion zusätzlich die implementierungsspezifische Signatur der Methode. Dieser generierte C#-Code unterscheidet sich von dem Code aus Abbildung 6.10, nur durch die Formatierung und durch die Verwendung des Schlüsselwortes *var* im Kopf der *foreach*-Schleife. Dadurch werden durch die

Verwendung von Alf als Wächter- und Effektbeschreibungssprache keine anderen Testfälle erzeugt, als bei der direkten Verwendung von C#. Ein Vergleich der erzeugten Testfälle bestätigt dies.

Abbildung 6.14: Ausschnitt aus der Grafik für die erzeugten Testszenarien im Spec Explorer Plug-in für Visual Studio.

Mit der Konfiguration und dem Modellprogramm wird dann der Spec Explorer ausgeführt. Dieser exploriert den Zustandsraum des Modellprogramms und schneidet aus dem Zustandsgraphen Testfälle heraus. Im Fall des Log-in-Systems erzeugt der Spec Explorer vier Testfälle. Ohne die Verwendung des Alf-zu-C#-Übersetzers und mit C# als Wächter und Effektbeschreibungssprache werden die gleichen vier Testfälle generiert. Ein Testfall davon ist in Abbildung 6.14 dargestellt. Die Abbildung ist ein Ausschnitt aus der Grafik der Testszenarien, die das Visual Studio Plug-in des Spec Explorers anzeigt. In dem abgebildeten Testfall versucht sich ein Benutzer erst mit falschem Benutzernamen (login_unknown_user), dann mit falschem Passwort (login_wrong_pwd) und wieder mit falschem Benutzernamen einzuloggen. Der Rückgabewert ist in allen drei Fällen `wrong_login`. Im vierten Schritt loggt sich der Benutzer dann mit den richtigen Daten ein und erhält als Rückgabewert den Wert `login_successfull`. Dann ist der Benutzer

eingeloggt und die Zustandsmaschine geht in den Endzustand über.

Die vom Spec Explorer erzeugten Testfälle werden am Ende in das Testmodell in Fokus!MBT importiert. In Fokus!MBT werden die Testfälle als Sequenzdiagramme mit der Kommunikation zwischen SUT und Testkomponente dargestellt.

Das Log-in-Beispiel zeigt, das eine Überführung von bestehendem plattformspezifischen C#-Code in Alf, relativ einfach ist. Syntaktisch mussten nicht viele Änderungen durchgeführt werden, was auch damit zusammenhängt, dass die Syntax beider Sprachen ihren Ursprung in der Sprache C haben. Der Alf Code ist zudem frei von Implementierungsdetails des Spec Explorers wie den `Condition.IsTrue()`-Aufrufen. Durch die Verwendung von Alf anstelle von C# fand keine Änderung an den generierten Testfällen statt.

7 Auswertung

In dieser Arbeit wurden als Erstes die Eingabesprachen von drei verbreiteten Testfallgeneratoren verglichen. Es stellt sich heraus, dass die Testfallgeneratoren auf Zustandsmaschinen beruhen. Die Eingabesprachen für die Wächter- und Effektbeschreibung aller drei Generatoren basieren auf Java oder C#, wobei QML von Java abstammt. Die Sprachen Java und C# ähneln sich ihrerseits wieder stark. Dies erleichtert das Finden einer plattformunabhängigen Sprache, welche sich leicht in die Eingabesprachen der Generatoren übersetzen lässt.

Weiterhin wurden in dieser Arbeit verschiedene textuelle Sprachen der UML hinsichtlich der Fragestellung untersucht, inwieweit sich diese für die Wächter- und Effektbeschreibung in Testfallentwurfsmodellen eignen. Es stellt sich heraus, dass die drei untersuchten Sprachen verschiedene Anwendungsbereiche innerhalb der UML haben. OCL dient hauptsächlich dem Spezifizieren von Randbedingungen und ist strikt nebeneffektfrei. MARTE VSL dient zum textuellen Spezifizieren von Werten, wie zum Beispiel Zeitwerten. Alf ist als *Actionlanguage* die textuelle Sprache innerhalb der UML zur Beschreibung von Verhalten. Damit ist Alf genau die Sprache, die für die Wächter- und Effektbeschreibung zu verwenden ist. Da die spezifischen OCL Konstrukte wie Sequenzoperationen auch in Alf vorhanden sind, bietet OCL kaum einen Mehrwert gegenüber Alf. Die VSL bietet die Möglichkeit, Zeiten und andere Werte mit physikalischen Einheiten anzugeben. Insbesondere das Angeben von Zeiten kann im Kontext des modellbasierten Softwaretestens von Interesse sein, um zum Beispiel erwartete Laufzeiten und Timeouts definieren zu können. Dies ist ein Vorteil von VSL gegenüber Alf. Der große Nachteil von Alf ist das Fehlen von Gleitkommazahlen. Die Unterstützung für Gleitkommazahlen in Alf kann aber in zukünftigen Versionen des Alf-Standards noch hinzugefügt werden.

Günstigerweise ist Alf stark an Java angelehnt. Die Übersetzung von Alf-Code in Java, C# oder QML ist damit sehr einfach. Die meisten Sprachbestandteile lassen sich eins zu eins aus Alf-Code in Java, C# oder QML übernehmen. Einige wenige UML spezifische

Konstrukte erfordern jedoch eine spezielle Übersetzung oder sind schwer bis gar nicht übersetzbar. Für diese Konstrukte stellt sich die Frage der Praxisrelevanz. Wenn die Konstrukte ohnehin selten verwendet werden, dann ist es kein großes Problem, diese nicht übersetzen zu können.

Anschließend wurde in Fokus!MBT die Alf-zu-C#-Übersetzung für den Spec Explorer implementiert. Im Rahmen dieser Arbeit wurde dabei der Alf-Parser aus Papyrus wiederverwendet. Dieser befindet sich aber zum Zeitpunkt dieser Arbeit noch in der Entwicklung und ist daher nicht vollständig. Der Parsebaum, den der Parser liefert, ist nicht mit Typannotationen versehen. Au diesem Grund konnte im Rahmen dieser Arbeit kein vollständiger Alf-zu-C#-Übersetzer entwickelt werden. Eine komplette Übersetzung von Alf nach C# ist jedoch wie beschrieben möglich, wenn ein Übersetzer mit Typannotationen verwendet wird.

In der Fallstudie zeigte sich, dass tatsächlich wenige Änderungen gemacht werden müssen, um statt C# Alf zu verwenden. Der implementierte Übersetzer ist hierfür ausreichend.

Die Fragestellung, die diese Arbeit zu beantworten versucht, ist, inwieweit sich UML-basierte textuelle Sprachen dazu eignen plattformunabhängig Wächter und Effekte für Zustandsmaschinen zu beschreiben, um diese für die Testfallgenerierung verwenden zu können. Es hat sich gezeigt, dass sich Alf sehr gut als eine plattformunabhängige Sprache verwenden lässt. Dabei ist vor allem von Bedeutung, dass Alf den Eingabesprachen der Testfallgeneratoren sehr ähnlich ist. Der Nachteil, eine andere Sprache zu verwenden als der Testfallgenerator und wegen dieser allgemeineren Sprache an Ausdrucksstärke zu verlieren, fällt nicht ins Gewicht. Der große Vorteil der plattformunabhängigen Sprache hingegen ist die Abstraktion von der technischen Umsetzung des konkreten Testfallgenerators und die Wiederverwendbarkeit von Wächter- und Effektcode für verschiedene Testfallgeneratoren.

7.1 Ausblick

Alf kann nicht nur für die Übersetzung von Wächter- und Effektcode in Zustandsdiagrammen verwendet werden. Auch andere Modellbestandteile können Alf benutzen. Zum Beispiel können Testfallentwurfsdiagramme nicht nur Zustandsdiagramme, sondern auch Sequenzdiagramme sein. Mit Sequenzdiagrammen können spezifischere Testszenarien ange-

geben werden. UML Sequenzdiagramme veranschaulichen die Kommunikation zwischen Objekten mit Nachrichten, um Verhalten auf den Objekten aufzurufen. Um das Verhalten, die sogenannte Aktivierung, zu beschreiben, könnte Alf ebenfalls eingesetzt werden.

Im Folgenden werden noch ein paar Probleme erwähnt, die sich bei der Verwendung von Alf als Wächter- und Effektbeschreibungssprache auftaten.

Bei der Entwicklung des Übersetzers in Papyrus zeigten sich einige Probleme bei der Alf-Unterstützung. Der verwendete Alf-Parser ist wie schon erwähnt nicht vollständig. Die Autovervollständigung und Syntaxhighlighting für Alf funktioniert nur, wenn Alf-Code als .alf-Datei editiert wird. Alf-Editoren, die in Dialoge integriert sind, bieten weder Autovervollständigung noch Syntaxhighlighting, hinterlegten aber jeglichen Alf-Code rot, um anzuzeigen, dass dieser ungültig ist. Da die Editoren für die Wächter- und Effektbeschreibungen in Dialoge integriert sind, betrifft dieses Problem die Verwendung von Alf in Testfallentwurfsmodellen. Hier muss die Alf-Unterstützung in Papyrus verbessert werden.

Ein Problem, dass bei Alf in Verbindung mit UML Modellen auftritt, sind assoziative Arrays (*Dictionaries*). In der UML werden qualifizierte Assoziationen für die Modellierung von *Dictionaries* verwendet. Qualifizierte Assoziationen werden in Alf aber nicht unterstützt. Stattdessen gibt es in der Standardbibliothek von Alf die Klasse *Dictionary*. Diese Tatsache erzeugt ein Inkonsistenzen zwischen UML Modellen und Alf. Es muss also in UML Modellen statt der üblichen qualifizierten Assoziationen die Alf-Klasse *Dictionary* importiert und verwendet werden.

In der Standardbibliothek von Alf gibt es auch einige Inkonsistenzen. Zum Beispiel besagt der Standard, dass die Operation `add` der Klasse `List<T>` immer den Wert *true* zurückgibt [Vgl. 6, Kap. 11.7.5]. In der Implementierung hingegen gibt die gleiche Operation nur dann den Wert *true* zurück, wenn der hinzugefügte Wert vorher noch nicht in der Liste vorhanden ist [Vgl. 6, Kap. B.4.8, B.4.3]. Diese Differenz kann im Zweifelsfall zu fehlerhafter Programmausführung führen.

Literatur

[1] Jordi Cabot. *The New Executable UML Standards: fUML and Alf*. 19. Jan. 2011. URL: http://modeling-languages.com/new-executable-uml-standards-fuml-and-alf/ (besucht am 07. 04. 2015).

[2] *Designer 4.2 Evaluation Guide*. Conformiq.

[3] *Designer 4.4 Manual*. Conformiq.

[4] Wolfgang Grieskamp. *The Spec Explorer Story*. 28. Okt. 2009. URL: http://blogs.msdn.com/b/wrwg/archive/2009/10/28/the-spec-explorer-story.aspx (besucht am 13. 04. 2015).

[5] Zoltán Micske u. a. "On Open Source Tools for Behavioral Modeling and Analysis with fUML and Alf". In: *Proceedings of the 1st Workshop on Open Source Software for Model Driven Engineering (OSS4MDE 2014) co-located with 17th International Conference on Model Driven Engineering Languages and Systems (MODELS 2014)*. Valencia, Spain: CEUR-WS.org, Sep. 2014, S. 31–41. URL: http://ceur-ws.org/Vol-1290/paper3.pdf.

[6] OMG. *Action Language for Foundational UML (Alf), Concrete Syntax for a UML Action Language, Version 1.0.1*. Object Management Group, Sep. 2013. URL: http://www.omg.org/spec/ALF/1.0.1/.

[7] OMG. *MOF Model to Text Transformation Language, v1.0*. Object Management Group, Jan. 2008. URL: http://www.omg.org/spec/MOFM2T/1.0/.

[8] OMG. *Object Constraint Language, Version 2.4*. Object Management Group, Feb. 2014. URL: http://www.omg.org/spec/OCL/2.4.

[9] OMG. *OMG Systems Modeling Language (OMG SysML), Version 1.3 with change bars*. Object Management Group, Juni 2012.

[10] OMG. *OMG Unified Modeling Language (OMG UML), Infrastructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1.

[11] OMG. *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*. Object Management Group, Aug. 2011. URL: http://www.omg.org/spec/UML/2.4.1.

[12] OMG. *Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1*. Object Management Group, Aug. 2013. URL: http://www.omg.org/spec/FUML/1.1/.

[13] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Version 1.1*. Object Management Group, Juni 2011.

[14] OMG. *UML Testing Profile (UTP), Version 1.2*. Object Management Group, Apr. 2013. URL: http://www.omg.org/spec/UTP/1.2.

[15] *Spec Explorer*. Microsoft Developer Network. URL: https://msdn.microsoft.com/en-us/library/ee620411.aspx (besucht am 13. 04. 2015).

[16] M.-F. Wendland. "Abstractions on test design techniques". In: *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*. Sep. 2014, S. 1575–1584. DOI: 10.15439/2014F316.

[17] *What's New in JDK 8*. Oracle. URL: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html (besucht am 14. 04. 2015).