Software Engineering
RWTH AACHEN UNIVERSITY

Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Software Engineering

# Strukturelle Analysen von MontiArc-Modellen mittels Z3-Solver

**Bachelorarbeit**

von

**Strodthoff, Nicolai**

**1. Prüfer: Prof. Dr. B. Rumpe**

**2. Prüfer: Prof. Dr. J.-P. Katoen**

**Betreuer: Dipl.-Math. Michael von Wenckstern**

Diese Arbeit wurde vorgelegt am Lehrstuhl für Software Engineering

Aachen, den 10. Oktober 2016

## Eidesstattliche Versicherung

_____________________________ _____________________________

Name, Vorname Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

____________________________________________________________

____________________________________________________________

____________________________________________________________

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

_____________________________ _____________________________

Ort, Datum Unterschrift

*Nichtzutreffendes bitte streichen

**Belehrung:**

**§ 156 StGB: Falsche Versicherung an Eides Statt**

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

**§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt**

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

_____________________________ _____________________________

Ort, Datum Unterschrift

## Kurzfassung

Diese Arbeit befasst sich mit der strukturellen Analyse von MontiArc-Modellen mit dem SMT-Solver Z3 von Microsoft. MontiArc ist eine Architecture Definition Language (ADL), die mit der MontiCore Language Workbench konstruiert wurde. Unter strukturellen Analysen wird die Durchführung von formalen Prüfungen auf Basis des Quelltextes verstanden. Es erfolgte die Erstellung von mehreren Repräsentationen des MontiArc-Modells in Z3. Des Weiteren wurde ein Codegenrator implementiert, der OCL-Ausdrücke nach Java übersetzt.

## Abstract

This thesis deals with the strucural analysis of MontiArc models using the SMT-Solver Z3 from Microsoft. MontiArc is a Architecture Definition Language (ADL) which was developed with the MontiCore Language Workbench. Structural analysis is the process of performing formal tests on the source code. Several Z3 representations of MontiArc-models were developed. Additionally a generator was implemented that converts OCL expressions to Java code.

# Inhaltsverzeichnis


**1 Einleitung** **1**

1.1 Motivation . . . 1

1.2 Ziele . . . 1

1.3 Aufbau der Arbeit . . . 1

**2 Stand der Technik** **3**

2.1 Bedingungen in Language Workbenches . . . 3

2.1.1 MontiCore . . . 3

2.1.2 JetBrains MPS . . . 4

2.1.3 XText . . . 5

2.2 OCL Auswertung . . . 6

2.2.1 DresdenOCL . . . 6

**3 Grundlagen** **7**

3.1 MontiArc . . . 7

3.2 OCL . . . 9

3.3 SMT . . . 10

**4 Strukturelle Analysen von MontiArc** **13**

4.1 Symboltabellen . . . 13

4.1.1 MontiCore Symboltabelle . . . 13

4.1.2 MontArc Symboltabelle . . . 15

4.2 Kontextbedingungen für MontiArc . . . 16

4.2.1 Namenseindeutigkeit (B1) . . . 16

4.2.2 Subkomponenten in qualifizierten Konnektorteilen (R5) . . . 16

4.2.3 Ports in qualifizierten Konnektorteilen (R6) . . . 17

4.3 Durchführung der strukturellen Analysen . . . 18

**5 Analysen mit Z3** **19**
5.1 Z3 Solver . . . . . . . . 19
5.2 SMT-LIB . . . . . . . . 19
5.3 MontiArc Modelle in Z3 . . . . . . . . 21
5.3.1 V1 . . . . . . . . 21
5.3.2 V2 . . . . . . . . 24
5.3.3 V3 . . . . . . . . 25
5.3.4 V4 . . . . . . . . 25
5.3.5 V5 . . . . . . . . 26
5.3.6 V6 . . . . . . . . 26
5.3.7 V7 . . . . . . . . 26
5.4 Evaluation . . . . . . . . 26

**6 Alternativen zum Z3 Solver** **29**
6.1 OCL2Java-Generator . . . . . . . . 29
6.1.1 Konzept . . . . . . . . 29
6.1.2 Integration in das MontiCore-Projekt . . . . . . . . 32
6.2 Qualitätssicherung . . . . . . . . 33
6.3 Leistungsvergleich von Java und Z3 . . . . . . . . 33

**7 Zusammenfassung und Ausblick** **35**

**Literaturverzeichnis** **37**

# Kapitel 1

# Einleitung

In diesem Kapitel soll die Motivation und das Ziel der Arbeit dargestellt werden. Zuletzt wird der Aufbau der Arbeit kurz beschrieben.

## 1.1 Motivation

Moderne Software wird immer komplexer. In einem modernen Auto befinden eine sehr hohe Anzahl an Steuergeräten, die mit einander kommunizieren und Funktionalitäten wie automatische Distanzregelung oder automatische Einparkenhilfe bereitstellen. Um die Entwicklung der Software für diese Steuergeräte so kostengünstig und fehlerfrei wie möglich zu gestalten, können Archtitectual Description Languages (ADL) eingesetzt werden. ADL ermöglichen die automatische Analyse der modellierten Systeme in den frühen Phasen des Entwicklungsprozesses. Diese Analysen können beispielsweise statische Codeanalysen sein, bei dem der Quelltext einer Reiher formaler Prüfungen unterzogen wird.

## 1.2 Ziele

MontiArc [Hab12] ist eine ADL, die mit Hilfe des MontiCore-Frameworks [Kra10] entwickelt wurde. Ziel dieser Arbeit ist es, die strukturellen Analysen von MontiArc-Modellen mit dem SMT-Solver Z3 durchzuführen. Die Object Constraint Language (OCL) soll dabei helfen, die gewollten Bedingungen zu modellieren. Es soll festgestellt werden, ob Z3 sich für die Auswertung eignet und wie die Bedingungen gemeinsam mit dem Modell in Z3 repräsentiert werden können.

## 1.3 Aufbau der Arbeit

In Kapitel 2 werden zunächst verwandte Arbeiten kurz beleuchtet. Dabei wird größenteils auf die Einbindung von Kontextbeindungen von verschiedenen *Language Workbenches* eingegangen.

In Kapitel 3 folgt eine Vorstellung der wichtigsten Grundlagen. Dafür wird zunächst die *Architecture Description Language* MontiArc an einem Beispiel erklärt. Danach kommt

eine Einführung in die textuelle Modellierungssprache OCL. Abschließend folgt ein kurzer Exkurs in die Aussagenlogik und Prädikatenlogik, um das Konzept von SMT zu erläutern.

Kapitel 4 beschäftigt sich mit Konzept von strukturellen Analysen von MontiArc. Das Kapitel beginnt mit einer Vorstellung von der Architektur von MontiCore Symboltabellen sowie der konkreten Implementation der MontiArc Symboltabelle. Danach werden einige ausgewählte Kontextbedingungen von MontiArc beschrieben. Zuletzt folgt eine kurze Beschreibung des vorgeschlagenen Analyseprozesses.

Kapitel 5 beschreibt die Umsetzung der strukturellen Analysen mit Hilfe des Z3 Solvers. Zu Beginn wird erklärt, was der Z3 Solver ist und danach folgt eine Vorstellung des verwendeten textuellen Inputformates *SMT-LIB*. Durch die Übersetzung der MontiArc Symboltabelle nach *SMT-LIB* sind unterschiedliche Repräsentationen entstanden, die alle beschrieben und nachträglich durch einen Leistungsvergleich bewertet werden.

In Kapitel 6 wird die Java-Umsetzung des Analyseprozesses vorgestellt. Dafür werden zunächst die Unterschiede und Gemeinsamkeiten dieser Implementation zu einer früheren Arbeit aufgezählt und beschrieben. Des Weiteren wird die Implementation in das MontiCore-Projekt und Qualitätssicherung erörtert. Zum Schluss folgt ein Leistungsvergleich zwischen den beiden Ansätze mit Z3 und Java.

Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick auf zukünftige Arbeiten.

# Kapitel 2

# Stand der Technik

Zunächst wird beschrieben, wie Bedingungen in gängigen Language Workbenches definiert werden können. Danach folgt eine kurze Vorstellung von DresdenOCL.

## 2.1 Bedingungen in Language Workbenches

### 2.1.1 MontiCore

Die MontiCore Language Workbench ermöglicht die Entwicklung und Realisierung von domänenspezifischen Sprachen (DSL - *domain specific language*) durch die textuelle Sprachdefinition mit dem MontiCore-Grammatikformat. Mit Hilfe der MontiCore-Grammatik kann der konkrete und abstrakte Syntax einer Sprache definiert werden [Kra10], was als Grundlage für die Dokumentation und Generierung von weiterer Infrastruktur dient. Zu dieser Infrastruktur zählt beispielsweise ein AST (*abstract syntax tree*) oder ein Parser für die spezifizierte Sprache.

Die MontiCore-Grammatik ist eine kontextfreie Grammatik (CFG - *context-free grammar*), was bedeutet, dass nur syntaktisch korrekte Ausdrücke festgelegt werden können. Kontextfreie Grammatiken sind jedoch nicht in der Lage alle möglichen Fehler zu beschreiben. Zum Beispiel kann nicht sichergestellt werden, dass eine Variable vor der Benutzung zuerst definiert wird. Für solche Bedingungen sind zusätzlich Kontextbedingungen (CoCo - *context condition*) nötig, welche die Anzahl an korrekten Modellen weiter einschränkt.

Die nötige Infrastruktur zum Überprüfen der Kontextbedingungen wird auch automatisch von MontiCore generiert. Für jede Sprache **`L`** wird eine Klasse **`L`**`CoCoChecker` generiert, welche jeweils eine Methode `addCoCo(AST`**`NT`**`CoCo c)` für jedes Nichtterminal-Symbol **`NT`** enhält. `addCoCo` dient zur Registrierung von benutzerdefinierten Kontextbedingungen, welche bei der Ausführung durch das Durchlaufen des AST's auf die korrespondieren AST-Knoten angewendet werden.

Die Implementation geschieht durch das Erstellen einer Klasse, welche das bereitgestellte **`L`**`AST`**`NT`**`CoCo` Interface erweitert und die Methode `check(AST`**`NT`**`)` implementiert. Jede Kontextbedingung sollte einen einzigartigen Fehlerkode besitzen, was das Schreiben von Tests vereinfacht. Des Weiteren soll eine Fehlermeldung ausgegeben werden, die Informationen über den Grund der unerfüllten Kontextbedingung enthält. Durch die Methode `get_SourcePositionStart()`, die jeder AST-Knoten besitzt, kann zusätzlich die Ausgabe der exakten Position im Quelltext erfolgen.

### 2.1.2 JetBrains MPS

JetBrains MPS ist eine Language Workbench, die sich durch die durchgehende Nutzung von AST's von anderen Produkten abhebt. Der AST einer Sprache wird direkt editiert, gespeichert und kompiliert. Dadurch kann das Definieren einer textuellen Grammatik und Konstruktion eines zugehörigen *Parsers* vermieden werden. Stattdessen kann die Manipulation der Knoten und Knotenbeziehungen direkt im bereitgestellten Editor erfolgen.

Die Struktur einer Sprache wird durch die Definition von *concepts* bestimmt. *Concepts* legen die Attribute, Kinder und Referenzen eines Knotens fest.

In Listing 2.1 ist das *concept* eines If-Then-Else Ausdrucks abgebildet.

```
1  concept IFStatement extends     Statement
2                      implements  IContainsStatementList
3                                  IDontSubstituteByDefault
4
5      instance can be root: false
6      scope: none
7
8      alias: if
9
10     properties:
11     forceOneLine    :   boolean
12     forceMultiLine  :   boolean
13
14     children:
15     condition       :   Expression[1]
16     ifFalseStatement:   Statement[0..1]
17     ifTrue          :   StatmentList[1]
18     elsifClauses    :   ElsIfClause[0..n]
19
20     references:
21     << ... >>
```

Listing 2.1: *Concept* für ein If-Then-Else Ausdruck [www16c].

MPS unterstützt das Prinzip der Vererbung. Ein *concept* kann von genau einem anderem *concept* erben sowie mehrere *concept interfaces* implementieren. Bei der Vererbung werden alle Attribute, Referenzen und Beziehungen zu anderen Knoten übernommen.

Ein Attribut (*property*) ist ein Wert, der im *concept* gespeichert ist. Jedes Attribut muss einen Typen haben. Als Typen existieren primitive Datentypen, wie Integer oder Boolean, Aufzählungstypen oder *constrained* Datentypen, wie Strings für die reguläre Ausdrücke gelten.

Des Weiteren beinhalten *concepts* Referenzen (*references*) zu anderen Knoten. Eine Referenz hat immer einen Namen, Typen und eine Kardinalität. Die Kardinalität bestimmt, wie viele Referenzen zu diesem Knoten bestehen und muss immer 0..1 (optionale Referenz) oder 1 (genau eine Referenz) betragen.

Durch die Angabe der Kinder (*children*) kann die Baumstruktur des AST's festgelegt werden. Kinder werden auch durch die Angabe durch Name, Typ und Kardinalität definiert. Bei der Kardinalität ist zusätzlich auch 0..n (beliebig viele) und 1..n (mindestens eine) erlaubt.

Mit Hilfe von *constraints* werden Bedingungen an die Beziehungen zwischen den Knoten und gültige Werte für Attribute definiert. Im Beispiel aus Listing 2.2 wird eine Bedingung an ein Attribut des `Constant`-Knoten gestellt. Dies erfolgt durch den `property` Befehl. Durch `set` und `set` können jeweils Methoden festgelegt werden, die aufgerufen werden, wenn das Attribut gelesen oder geschrieben wird. Durch `is valid` ist die Angabe einer Methode möglich, die die Korrektheit des Attributs überprüft.

```
1  concept constraints Constant {
2      can be child
3      (childConcept, node, link, parentNode, operationContext,
4                                                    scope)->boolean {
5              parentNode.isInstanceOf(Constants) &&
6              !(parentNode : Constants.constants.contains(node));
7      }
8
9      can be parent <none>
10
11     can be ancestor <none>
12
13     property {name}
14         get:<default>
15         set:<default>
16         is valid:{propertyValue, node, scope}->boolean {
17             propertyValue.length() > 5;
18         }
19 }
```

Listing 2.2: *Constraints* für den `Constant`-Knoten.

Mit Hilfe des `can be`-Ausdrucks kann die Einschränkung von Beziehungen zu anderen Knoten erfolgen. Die Einschränkung bestimmt, ob dieses *concept* als *child*, *parent*, *ancestor* eines anderen Knoten auftauchen darf.

### 2.1.3 XText

XText ist auch ein Framework für die Entwicklung von domänenspezifischen Sprachen. Wie MontiCore, nutzt XText eine textuelle Sprachdefinition in Form einer Grammatik zur Erstellung der nötigen Infrastruktur. Es wird als Teil vom *Eclipse Project* entwickelt und ist auch als *plug-in* für *eclipse* verfügbar.

Das Prüfen auf syntaktische Korrektheit durch den Parser und defekte Querverweise wird automatisch durchgeführt und als Fehler ausgegeben.

XText bietet die Möglichkeit der benutzerdefinierten Validation durch das Bereitstellen einer Java-Klasse (siehe Listing 2.3). Dort erfolgt die Erstellung von benutzerdefinierten Kontextbedingungen. Wie bei MontiCore, sollte eine deskriptive Fehlermeldung ausgegeben werden, falls die Bendingung nicht erfüllt ist. Der `check` Befehl hat einen optionalen Parameter, mit dem festgelegt wird, wann die Überprüfung der Bedingung erfolgt. Mit Parameter `FAST` wird die Überprüfung nach jeder Änderung einer Datei ausgeführt. Bei `NORMAL`, wenn die Datei gespeichert wird und bei `EXPENSIVE` nur dann, wenn die Validation durch einen expliziten Zugriff aus dem Menü aufgerufen wird.

```
class DomainmodelValidator extends AbstractDomainmodelValidator {

    @Check
    def void checkNameStartsWithCapital(Entity entity) {
        if (!Character.isUpperCase(entity.name.charAt(0))) {
            warning("Name should start with a capital",
                DomainmodelPackage.Literals.TYPE__NAME)
        }
    }
}
```

Listing 2.3: Benutzerdefinierte Bedingung in XText [www16e].

## 2.2 OCL Auswertung

### 2.2.1 DresdenOCL

DresdenOCL [www16a] ist ein *Toolkit*, welches die das Auswerten von OCL-Ausdrücken auf Modellen ermöglicht. Zu den unterstützten Sprachen für die Modelle zählen unter anderem Java, UML und EMF.

DresdenOCL besteht aus mehreren Werkzeugen. Der *OCLParser* transformiert OCL-Ausdrücke in einen abstrakten Syntaxbaum (AST). Mit dem *OCL2Java* kann der AST in Java-Quelltext umgewandelt werden. *OCLInjector4Java* nimmt den generierten Java-Quelltext und fügt diesen in das Modell ein, um eine Ausführung der Auswertung zu ermöglichen. Durch die *OCL2SQL* Klasse ist auch eine Generierung von SQL-Quelltext möglich.

Der Generierungsprozess von ausführbarem SQL- oder Java-Code wird durch die Benutzung eines *pivot models* ermöglicht [Bra07]. Ziel dieses Modells ist die Abstraktion von mehreren domänenspezifischen Sprachen, um OCL-Ausdrücke evaluieren zu können.

# Kapitel 3

# Grundlagen

## 3.1 MontiArc

MontiArc ist eine Architecture Descriotion Language (ADL), die zur Modellierung und Simulation von Software-Architekturen genutzt werden kann. Es wurde als textuelle domänenspezifische Sprache mithilfe vom MontiCore-Framework entwickelt. MontiArc beschreibt die einzelnen Komponenten und dessen Kommunikation eines (Software)-Systems. Als Komponente wird ein Element bezeichnet, welches Berechnungen anstellt und Daten speichert [HRR14]. Dabei kann es um ein beliebig großes Subsystem oder nur eine einzelne Funktion handeln. Des Weiteren besitzt eine Komponente eine vordefinierte Schnittstelle, welche für die Kommunikation mit der Umgebung oder anderen Komponenten genutzt wird.

Im Folgenden wird der Syntax und die Struktur von MontiArc-Modellen anhand eines automotiven Systems veranschaulicht. Das Beispiel wurde aus [Hab12] übernommen. Die *LightCtrl* Komponente steuert die Innenraumbeleuchtung eines Autos und hat folgendes Verhalten:

- Wenn der Alarm aktiv ist, blinkt das Licht in einem konfigurierbarem Intervall.
- Es schaltet das Licht ein, wenn der Schalter auf *on* steht.
- Es schaltet das Licht aus, wenn der Schalter auf *off* steht.
- Wenn der Schalter auf *door dependant* steht, dann bestimmt der Zustand der Tür das Verhalten. Wenn die Tür offen ist, dann ist das Licht an. Wenn die Tür geschlossen wird, dann schaltet sich das Licht nach einem konfigurierbarem Zeitraum aus.

Eine grafische Darstellung dieser Komponente ist in Abbildung 3.1 zu sehen.

Der Syntax von MontiArc wird ebenfalls am *LightCtrl* Beispiel vorgestellt, welches in Listing 3.1 zu sehen ist. Jede Komponente wird üblicherweise in einer eigenen Datei definiert und ist, wie in Java, Teil eines Paketes. Das Importieren von Datentypen und Komponenten wird duch den *import*-Befehl ermöglicht. Die Komponente *LightCtrl* besitzt einen Konfigurationsparameter *fadeOutTime* und die drei Subkomponente *AlarmCheck*, *DoorEval* und *Arbiter*. Die Arbiter Komponente wird im Gegensatz zu den anderen Komponenten nicht referenziert, sondern als innere Komponente in *LightCtrl* definiert (s. Zeile

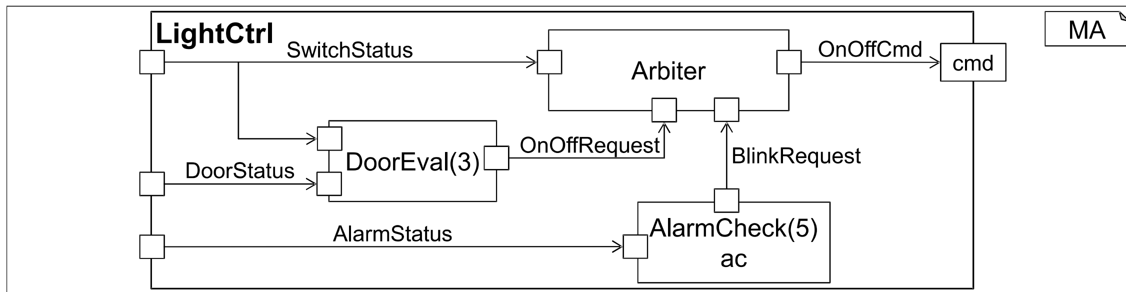


Abbildung 3.1: Komponentendefinition von *LightCtrl* [Hab12].

14-20). Durch den *autoinstantiate*-Befehl findet eine automatische Instantiierung der inneren Komponente *Arbiter* als Subkomponente statt. Die Schnittstelle einer Komponente wird durch die Ports festgelegt. Wird der optionale Name eines Ports ausgelassen, kann dieser implizit aus dem Port-Typen abgeleitet werden. Die eingehenden Ports von *LightCtrl* besitzen somit die Namen *switchStatus*, *alarmStatus* und *doorStatus*. Die implizite Namensvergabe funktioniert analog bei den Subkomponenten.

```
1  package ila;
2  import ila.signals.*;
3
4  component LightCtrl(int fadeOutTime) {
5      autoinstantiate on;
6      autoconnect port;
7
8      port
9          in SwitchStatus,
10         in AlarmStatus,
11         in DoorStatus,
12         out OnOffCmd cmd;
13
14     component Arbiter {
15         port
16             in SwitchStatus,
17             in BlinkRequest,
18             in OnOffRequest,
19             out OnOffCmd;
20     }
21
22     component AlarmCheck(fadeOutTime + 2) ac;
23     component DoorEval(fadeOutTime);
24
25     connect arbiter.onOffCmd -> cmd;
26 }
```

Listing 3.1: *LightCtrl* Beispiel.

Konnektoren verbinden immer einen sendenden Port mit einer beliebigen Anzahl an empfangenden Ports mit kompatiblen Datentyp. Die Ports sind genau dann kompatibel, wenn die Datentypen identisch sind oder der Datentyp des Empfängers ein Supertyp vom Datentyp des Senders ist. In Zeile 25 verbindet der Konnektor den ausgehenden Port *onOffCmd* von *arbiter* mit dem ausgehenden Port *cmd* von *LightCtrl*. Der *autoconnect*-Befehl in Zeile 6 führt zur automatischen Erzeugung von Konnektoren. Es existieren drei Strategien zur

automatischen Erzeugung:

- autoconnect port - Verbindet alle Ports mit dem gleichen Namen und kompatiblen Datentypen.
- autoconnect type - Verbindet Ports mit passenden Datentypen.
- autoconnect off - Ist der Standardwert und führt zu keiner automatischen Verbindung von Ports.

Im *LightCtrl* Beispiel würden die Konnektoren aus Listing 3.2 automatisch erzeugt werden.

```
1 connect switchStatus -> arbiter.switchStatus, doorEval.switchStatus;
2 connect alarmStatus -> ac.alarmStatus;
3 connect doorStatus -> doorEval.doorStatus;
4 connect ac.blinkRequest -> arbiter.blinkRequest;
5 connect doorEval.onOffRequest -> arbiter.onOffRequest;
```

Listing 3.2: Konnektionen, die durch *autoconnect* automatisch hergestellt werden.

## 3.2 OCL

Die Object Constraint Language (OCL) ist eine textuelle Modellierungssprache, welche Bestandteil der Unified Modeling Language (UML) ist. OCL dient zur Definition von Bedingungen (Constraints) an ein modelliertes System. Die OCL ist an keine bekannte Programmiersprache angelehnt ist und weist somit einen ungewöhnlichen Syntax auf, was vielen Nutzern den Zugang erschwert. Aus diesem Grund wurde in [Rum02] ein neuer, an Java angelehnter, Syntax vorgeschlagen. Dieser Vorschlag wurde in [Rum11] erweitert und OCL/P genannt.

Die OCL bietet die Möglichkeit, Invarianten sowie Vor- und Nachbedingungen von Methoden (Methodenspezifikation) zu modellieren.

Invarianten werden genutzt, Bedingungen aufzustellen, die zu jedem Zeitpunkt vom System erfüllt werden sollen. Als Beispiel ist in Listing 3.3 eine Invariante dargestellt.

```
1 context Human human inv AgeNonNegative:
2     human.age >= 0;
```

Listing 3.3: Beispiel für eine OCL-Invariante.

Invarianten gelten immer für einen bestimmten Kontext, wie beispielsweise einer Klasse des Systems. Im Beispiel aus Listing 3.3 wird vorgeschrieben, dass das Attribute `age` der Klasse `Human` nicht-negativ ist. Zusätzlich besitzt die Invariante den optionalen Namen `AgeNonNegative`.

Methodenspezifikationen bestehen immer aus einer Vorbedingung und Nachbedingung. Falls die Vorbedingung erfüllt ist, kann die Methodenspezifikation überprüft werden, indem nach Ausführung der Methode die Nachbedingung interpretiert wird. Falls die Vorbedingung nicht erfüllt ist, kann keine Aussage über das System und Eigenschaft der Methode gemacht werden. In Listing 3.4 ist die Methodenspezifikation für die Methode

`marry(Human spouse)` aus Klasse `Human` zu sehen. Die Vorbedingung gibt an, dass die Methode nur dann ein definiertes Verhalten ausweist, wenn das `currentSpouse`-Attribut auf `null` gesetzt ist. Wenn das der Fall ist, beschreibt die Nachbedingung, dass die übergebe `Human`-Instanz, nach der Ausführung der Methode, mit dem `currentSpouse`-Attribut übereinstimmt. Auf gleicher Weise können auch Bedingungen an den Konstruktor gestellt werden.

```
1 context Human.marry(Human spouse)
2 pre: currentSpouse == null;
3 post: currentSpouse == spouse;
```

Listing 3.4: Beispiel für eine OCL-Methodenspezifikation.

## 3.3 SMT

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT - *satisfiability*) ist ein Entscheidungsproblem, bei dem gefragt wird, ob eine aussagenlogische Formel erfüllbar ist. SAT ist ein NP-vollständiges Problem, d.h. es konnte bisher noch nicht nachgewiesen werden, ob es in polynomieller Zeit lösbar wäre. Trotzdem existieren effiziente Verfahren (SAT-Solver), welche mit Hilfe von systematischen Suchverfahren und Heuristiken über die Erfüllbarkeit von vielen aussagenlogischen Formeln entscheiden.

Die Menge $A$ der aussagenlogischen Formeln ist wie folgt induktiv definiert:

- $0, 1 \in A$ (boolesche Konstanten).
- $\tau = \{X_0, X_1, X_2, ...\} \subseteq A$ (eine feste, abzählbar unendliche Menge an Aussagenvariablen).
- Wenn $\psi, \phi \in A$, dann sind auch $\neg\psi, (\psi \vee \phi), (\psi \wedge \phi), (\psi \rightarrow \phi) \in A$.

Eine Formel ist erfüllbar, wenn für für jede vorkommende Aussagenvariable eine Belegung (Interpretation) mit wahr (1) oder falsch (0) existiert, sodass die gesamte Formel wahr wird. Die Formel

$$\psi = (\neg X_0 \wedge ((X_1 \rightarrow X_2) \vee (X_0 \wedge X_1))) \tag{3.1}$$

ist beispielsweise durch die Belegung $X_0 \rightarrow 0$, $X_1 \rightarrow 1$, $X_2 \rightarrow 1$ erfüllbar. Eine Belegung, die eine Formel erfüllt, wird auch als Modell bezeichnet.

Manche Probleme lassen sich nur in ausdrucksstärkeren Logiken, wie zum Beispiel die Prädikatenlogik erster Stufe (FOL - *first-order logic*), ausdrücken [dMB09]. In der Prädikatenlogik werden Formeln durch Gleichheitszeichen, Relationssymbole, Funktionssymbole und Quantoren erweitert. Ein Modell für eine Formel der FOL muss zusätzlich eine Belegung der Funktionssymbole und Relationssymbole beinhalten.

Für viele Anwendungsbereiche ist jedoch nicht die generelle Erfüllbarkeit einer FOL-Formel entscheidend, sondern viel mehr die Erfüllbarkeit unter Berücksichtigung einer Theorie, welche bestimmte Funktions- und Relationssymbole festsetzt [BSST09]. Ist beispielsweise die Erfüllbarkeit der Formel 3.2 unter Berücksichtigung der Integer-Arithmetik

von Interesse, dann sind nur diejenigen Modelle relevant, bei denen $<$ die übliche Ordnungsrelation für Integer, $+$ die Addition von Integer und 13 eine Konstante ist.

$$(3 + y = x) \wedge (x < 13) \tag{3.2}$$

Ein SMT-Solver ist ein Werkzeug, welches über die Erfüllbarkeit von Formeln, denen eine oder mehrere Theorien zugrunde liegen, entscheidet.

# Kapitel 4

# Strukturelle Analysen von MontiArc

Die syntaktischen Fehler eines Modells werden bereits durch den Parser entdeckt und durch eine Fehlermeldung ausgegeben. Des Weiteren sollen auch kontextabhängige Bedingungen geprüft werden. Dies wurde in MontiCore bisher durch eine Implementierung in Java durchgeführt. In diesem Kapitel wird zunächst der Aufbau der Symboltabelle von MontiCore sowie MontiArc vorgestellt. Danach werden einige der Kontextbedingungen von MontiArc kurz erläutert. Zuletzt folgt eine Beschreibung, wie OCL genutzt werden kann, um die strukturelle Analyse durchzuführen.

## 4.1 Symboltabellen

MontiCore unterstützt die Implementation von Symboltabellen. Diese ermöglichen die vereinfachte Navigation eines Modells.

### 4.1.1 MontiCore Symboltabelle

Eine Symboltabelle ist eine Datenstruktur, die das effiziente Finden von Deklarationen, Typen, Signaturen, Implementationsdetails und mehr für einen gegebenen Namen ermöglicht. Die Symboltabelle besteht aus einem Baum an Scopes (Sichbarkeitsbereichen), die jeweils eine Liste an Symbolen enthalten. Ein Symbol enthält alle essentiellen Informationen eines Modellelements, wie beispielsweise Methodennamen, Modifier, Rückgabetypen oder Parametertypen. In Abbildung 4.1 ist die Struktur der Symboltabelle aus MontiCore als Klassendiagramm abgebildet.

Das Symbol Interface ist die Superklasse von allen Symbolen und beinhaltet Informationen, wie Symbolname, Paketname und der zugehörige Scope (*enclosing scope*). Ein Symbol kann auch eine optionale Referenz zum korrespondierenden AST-Knoten enthalten. Zusätzlich hat jedes Symbol ein soganänntes *Kind*, was angibt, zu welcher Art von Symbol (wie zum Beispiel Methoden oder Variable) es angehört.

Ein Scope repräsentiert eine logische Gruppierung an Symboldefinitionen und limitiert die Sichtbarkeit nach außen. Lokale Variablen in einer Methode gehören somit alle dem Sichtbarkeitsbereich der Methode an. Die Methode spannt den Sichtbarkeitsbereich auf.

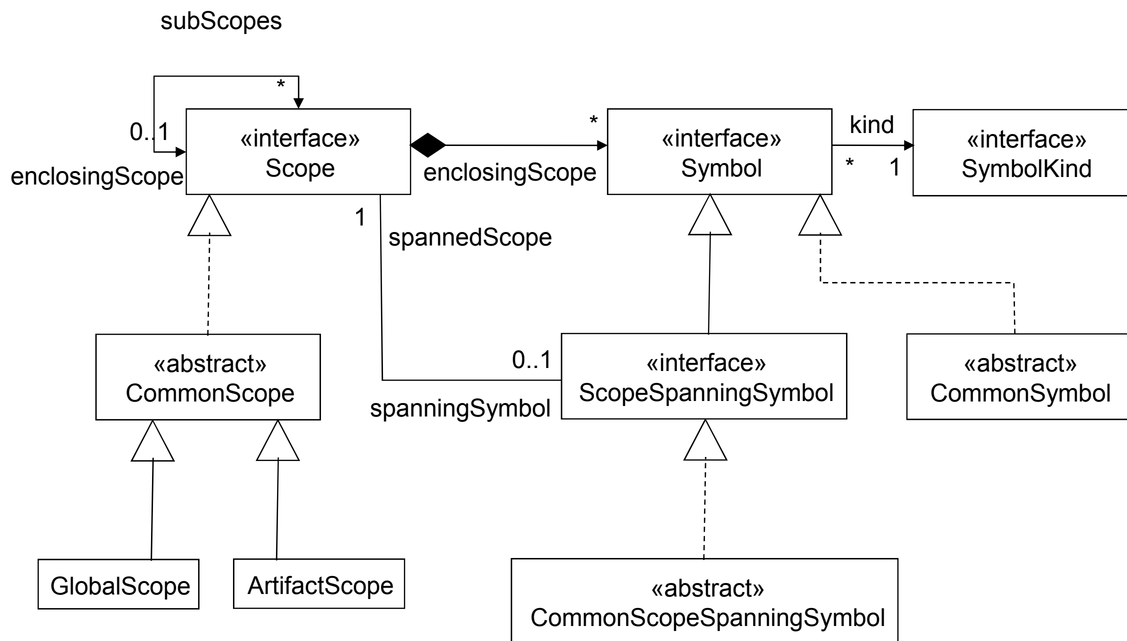


Abbildung 4.1: Klassendiagramm der Symboltablle von MontiCore.

Der *GlobalScope* ist die Wurzel der Scope Hierarchie. Danach folgen meist die *ArtifactScopes*, welche den Scope eines Software-Artefakts, wie z.B. eine Datei, repräsentieren. Der *ArtifactScope* beinhaltet unter anderem den Paketnamen und *import*-Ausdrücke.

Manche Symbole, wie beispielsweise ein Methodensymbol, spannen selbst einen *Scope* auf. Diese Art von Symbol kann durch das Implementieren des *ScopeSpanningSymbols* modelliert werden.

Die Klassen mit dem *Common*-Präfix sind Standardimplementationen für das jeweilige Interface. Der Nutzer sollte das *Symbol* Interface sollte nicht direkt implementieren, sondern stattdessen die jeweilge Standardimplementation erweitern und durch das Überschreiben von Methoden adaptieren.

Bei den Symbolen wird zwischen Symboldefinitionen und Symbolreferenzen unterschieden. Eine Symboldefinition kann nur genau einmal in einem Scope existieren. Symbolreferenzen hingegen, können jedoch mehrmals in unterschiedlichen Scopes auftauchen und referenzieren ein Symbol, das an einer anderen Position definiert wurde. Um herauszufinden, wo ein Symbol definiert wurde bzw. ob es überhaupt definiert wurde, muss ein *resolving mechanism* genutzt werden. Durch die Abgabe eines Namens und *Kinds* kann eine *Auflösung* des Symbols stattfinden, d.h. es wird zurückgegeben, falls es gefunden wird.

Die Implemenatation der Symbole muss durch den Nutzer erfolgen. Dafür können die abstrakten Klassen *CommonSymbol* und *CommonScopeSpanningSymbol* erweitert werden. Danach können beliebige benutzerdefinierte Attribute und Hilfsfunktionen hinzugefügt werden. Um eine Symboltabelle aus dem AST erstellen können, stellt MontiCore das *SymbolTabgleCreator* Interface zur Verfügung. Diese Klasse wird gemeinsam mit dem Visitor genutzt, um die Baumstruktur der Symboltabelle zu verwirklichen. Mit *CommonSymolTableCreator* steht wieder eine Standardimplementation zu Verfügung, die erweitert werden kann. Bei der Implementation des sprachenspezifischen *SymbolTableCreators* müssen die entsprechenden *visit*-Methoden überschrieben werden, um die benutzerdefinierten Attri-

bute in den Symbolen zu setzen.

### 4.1.2 MontArc Symboltabelle

Die wichtigsten Bestandteile der MontiArc Symboltabelle sind in Abbildung 4.2 dargestellt. Das *ComponentSymbol* erweitert das *CommonScopeSpanningSymbol*, da es einen eigenen Scope aufspannt, der alle Subkomponenten, Ports, Konnektoren und Konfigurationsparameter beinhaltet. *PortSymbol* und *ConnectorSymbol* erweitern jeweils das *CommonSymbol*.

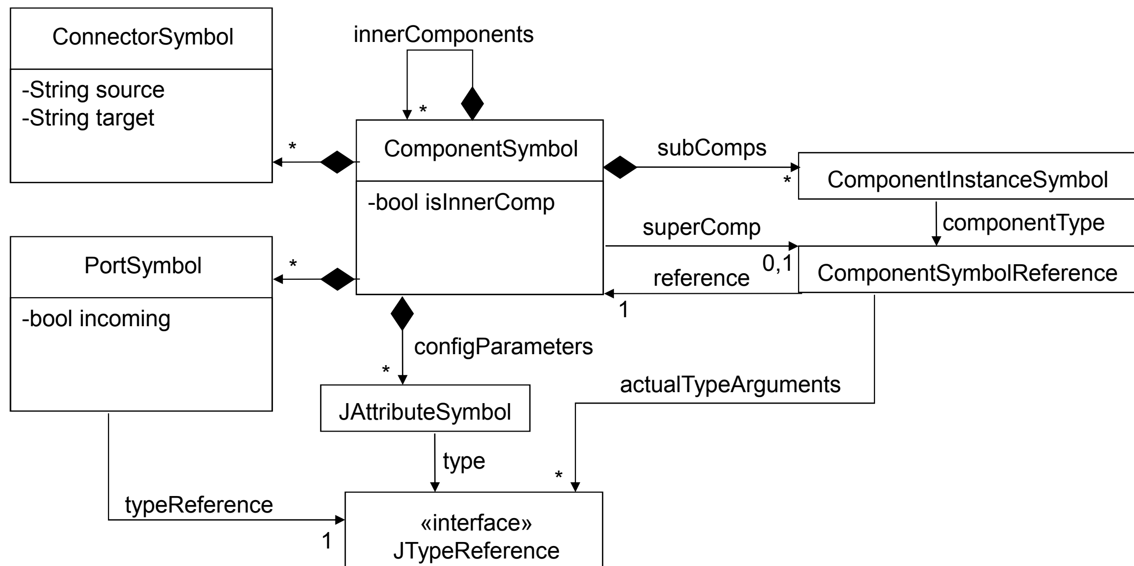


Abbildung 4.2: Klassendiagramm der Symboltablle von MontiArc.

Ein Reihe an Hilfsfunktionen erleichtern den Zugriff auf die Symbole, die innerhalb einer Komponente auftauchen. Zum Beispiel kann die `getOutgoingPorts()` Methode genutzt werden, um die Menge aller ausgehenden Ports zu bekommen. Der Quelltext dieser Methode ist in Listing 4.1 zu sehen. Die Funktionalität der Methode ist durch die Benutzung des *resolve mechanism* möglich. Zunächst wird die `resolveLocally` Methode auf den aufgespannten Scope der Komponente unter Angabe von `PortSymbol.KIND` aufgerufen, um eine *Collection* aller Portsymbolen, die sich in dieser Komponente befinden, zu bekommen. Danach findet eine Filterung statt, um nur die ausgehenden Ports zu erhalten.

```
1 public Collection<PortSymbol> getOutgoingPorts() {
2     return referencedComponent.orElse(this).getSpannedScope()
3         .<PortSymbol>resolveLocally(PortSymbol.KIND)
4         .stream()
5         .filter(p -> p.isOutgoing())
6         .collect(Collectors.toList());
7 }
```

Listing 4.1: Funktion, die alle ausgehenden Ports einer Komponente zurückgibt.

Subkomponenten werden durch *ComponentInstanceSymbol* repräsentiert, welche eine Referenz zu einer Komponentendefinition sowie eine Bezeichnung besitzen. Innere Komponenten sind diejenigen Komponenten, die innerhalb einer anderen Komponente definiert

werden. *JTypeReference* ist Teil des MontiCore-Frameworks und dient zur Referenzierung von Datentypen wie Integer oder Boolean.

Konnektoren besitzt die Namen des Ziel- und Quellports sowie Hilfsmethoden, um die Namen direkt zu Portsymbolen aufzulösen. Ports beinhalten die Information, ob es sich um einen eingehenden oder ausgehenden Port handelt. Des Weiteren existiert eine Referenz zum Datentyp des Ports.

## 4.2 Kontextbedingungen für MontiArc

Im Folgenden sollen einige ausgewählte Bedingungen aus [Hab12] kurz durch eine Beschreibung und Angabe eines invaliden Modells illustriert werden.

### 4.2.1 Namenseindeutigkeit (B1)

Um jedes Element einer Komponent eindeutig identifizieren zu können, müssen auch die Namen dieser Elemente eindeutig sein. Die Bedingung unterscheidet nicht zwischen verschiedenen Elementtypen, d.h. sie gilt für Namen von Ports, Subkomponenten, generischen Typparametern, Konfigurationsparametern und Constraints.

In Listing 4.2 ist eine MontiArc Komponente dargestellt, die die Context Condition zweimal verletzt. Zum einen hat der Konfigurationsparamter *a* den gleichen Namen wie ein eingehender Port (siehe Zeile 1 und 3). Des Weiteren wird der Name eines ausgehenden Ports *b* auch als Name für eine Subkomponentendeklaration genutzt (siehe Zeile 4 und 6).

```
1  component A(String a) {
2      port
3          in Integer a,
4          out String b;
5
6      component Inner b {
7          port
8              in Integer inInner,
9              out Integer outInner;
10     }
11 }
```

Listing 4.2: Verletzung der Kontextbedingung B1 durch die Mehrfachnutzung eines Namens in einer Komponente.

### 4.2.2 Subkomponenten in qualifizierten Konnektorteilen (R5)

Wenn die Quelle oder das Ziel eines Konnektors einen qualifizierten Namen hat, dann muss der erste Teil des Namens mit dem Namen einer Subkomponente übereinstimmen, die im gleichen Scope wie der Konnektor deklariert wurde.

In Listing 4.2 wird diese Kontextbedingung verletzt. Das Ziel des ersten Konnektors (siehe Zeile 14) ist qualifiziert mit *a*. Da jedoch auch eine Subkomponente mit dem Namen *a* im gleichen Scope deklariert wurde, ist der qualifizierte Name valide. Jedoch besitzt der Konnektor in Zeile 15 eine qualifizierte Quelle *b.aOut* zu der keine entsprechende Subkomponente existiert. Somit ist der Konnektor nicht valide.

```
component R5 {
    port
        in Integer dataIn,
        out Integer dataOut;

    component A {
        port
            in Integer aIn;
            out Integer aOut;
    }

    component A a;

    connect dataIn -> a.aIn;
    connect b.aOut -> dataOut;
}
```

Listing 4.3: Verletzung der Kontextbedingung R5.

### 4.2.3 Ports in qualifizierten Konnektorteilen (R6)

Laut Kontextbedingung R6 muss der zweite Teil eines qualifizierten Konnektorteils dem Namen eines Ports aus der referenzierten Komponente entsprechen. Die referenzierte Komponente geht aus dem ersten Teil des qualifizierten Namens hervor, welcher den Namen einer Subkomponente hat. Der Komponententyp dieser Subkomponente muss den entsprechenden Port besitzen.

In Listing 4.4 wird im ersten Konnektor der Port *dataIn* mit *aIn* von Subkomponente *a* verbunden. Wie in Zeile 8 zu sehen ist, besitzt der Komponententyp von *a* einen Port mit dem Namen *aIn*. Somit ist der Konnektor valide. Im zweiten Konnektor soll der Port *bOut* von *a* verbunden werden. Da die Komponente jedoch keinen Port mit dem Namen *bOut* besitzt, ist der Konnektor invalide.

```
component R6 {
    port
        in Integer dataIn,
        out Integer dataOut;

    component A {
        port
            in Integer aIn,
            out Integer aOut;
    }

    component A a;

    connect dataIn -> a.aIn;
    connect a.bOut -> dataOut;
}
```

Listing 4.4: Verletzung der Kontextbedingung R6.

## 4.3 Durchführung der strukturellen Analysen

Für die Durchführung der strukturellen Analysen von MontiArc-Modellen wird folgender Ablauf vorgeschlagen:

1. Erstellung der Bedingungen an das MontiArc-Modell durch OCL.
2. Übersetzung der Bedingungen und der Symboltabelle des Modells in eine Zielsprache.
3. Evaluation der Bedingungen in der Zielsprache.
4. Ausgabe der Ergebnisse.

Für den ersten Schritt benötigt der Nutzer Kenntnis über den Aufbau der MontiArc bzw. MontiCore Symboltabelle. Die Bedingung der Nameneindeutigkeit könnte beispielsweise mit dem OCL-Ausdruck, der in Listing 4.5 zu sehen ist, überprüft werden.

```
1 context ComponentSymbol cmp inv B1:
2   forall CommonSymbol sym1 in cmp.getSpannedScope().getSymbols():
3     forall CommonSymbol sym2 in cmp.getSpannedScope().getSymbols():
4         (sym1 != sym2) implies (sym1.name != sym2.name)
```

Listing 4.5: OCL-Ausdruck zum Überprüfen der Bedingung B1.

Danach müssen sowohl die Bedingungen, als auch die Symboltabelle des Modells in die Zielsprache übersetzt werden, damit eine Auswertung der OCL-Ausdrücke stattfinden kann. Ziel dieser Arbeit ist es, als Zielsprache SMT zu benutzen, damit eine Auswertung durch den Z3 Solver erfolgen kann. Nach der Auswertung der Kontextbedingungen in der Zielsprache müssen alle nötigen Ergebnisse gesammelt und ausgegeben werden.

Dieses Verfahren hat den Vorteil, dass beliebig viele Bedingungen hinzufügt werden können, ohne das MontiArc-Projekt zu verändern. Zu den Nachteilen zählt jedoch, dass die Struktur der Symboltabelle beim Erstellen der OCL-Ausdrücke beachten werden muss.

# Kapitel 5

# Analysen mit Z3

In diesem Kapitel wird zunächst der Z3 Solver und das SMT Standard vorgestellt. Danach folgt das Aufstellen von mehreren Repräsentation der Symboltabelle in SMT. Zum Schluss werden die verschiedenen Versionen durch einen Leistungsvergleich bewertet.

## 5.1 Z3 Solver

Z3 ist ein *open source* SMT-Solver von Microsoft Research, der von zahlreichen Projekten genutzt wird, um Programme auf Korrektheit zu prüfen, Fehler zu finden oder neue Testfälle zu erzeugen [DMB08]. Um Probleme zu beschreiben, akzeptiert Z3 textuelle Inputformate, wie *SMT-LIB*, *Simplify*, *DIMACS* und stellt eine API für *C*, *C++*, *.NET*, *Java*, *Python* bereit. Zu den unterstützten Theorien zählen unter anderem die Theorien der Bitvektoren, Arrays und nichtlinearen Arithmetik.

## 5.2 SMT-LIB

Das Ziel des SMT-LIB 2.0 Standards [BST10] ist die Bereitstellung einer gemeinsamen Eingabe- und Ausgabesprache für SMT-Solver. Im Folgenden wird die Benutzung von SMT-LIB als Eingabesprache für Z3 erklärt.

Der Befehl `declare-const` ermöglicht das Deklarieren einer Konstante eines gegebenen Typs. Mit `declare-fun` kann eine Funktion deklariert werden. In Listing 5.2 wird zuerst der Integer $a$ und dann die Funktion *myfunc* deklariert, welche einen Integer sowie einen Boolean als Eingabe bekommt und einen Boolean zurückgibt.

```
1 (declare-const a Int)
2 (declare-fun myfunc (Int Bool) Bool)
```

In Z3 werden alle Formeln und Deklarationen auf einem Stack gespeichert. Mit dem Befehl `assert` kann eine neue Formel zum Stack hinzugefügt werden. Die Menge an Formeln auf dem Stack (*Assertions*) ist genau dann erfüllbar, wenn eine Interpretation der deklarierten Variablen und Funktionen existiert, welche alle Formeln wahr macht.

Die erste hinzugefügte Formel in Listing 5.2 sagt aus, dass die Konstante $a$ größer als 100 sein muss. Die zweite Formel gibt an, dass die Funktion *myfunc* mit den Eingaben $a$ und

```
(declare-const a Int)
(declare-fun myfunc (Int Bool) Bool)
(assert (> a 100))
(assert (= (myfunc a false) true))
(check-sat)
(get-model)
```

*false* den Wert *true* zurückgeben muss. Mit dem Befehl `check-sat` wird die Menge an Formeln auf dem Stack auf Erfüllbarkeit überpruft. Z3 gibt `sat` aus, wenn die Formelmenge erfüllbar ist und `unsat`, wenn sie nicht erfüllbar ist. Die Ausgabe `unknown` entsteht, falls die Erfüllbarkeit nicht bestimmt werden kann.

Wenn die Formeln erfüllbar sind, dann kann mit dem Befehl `get-model` eine Interpretation ausgegeben werden, die alle Formeln wahr macht und somit ein Modell ist. Für das Beispiel aus Listing 5.2 ergibt sich folgende Ausgabe:

```
sat
(model
    (define-fun a () Int 101)
    (define-fun myfunc ((x!1 Int) (x!2 Bool)) Bool
        (ite (and (= x!1 101) (= x!2 false)) true true)
    )
)
```

Eine Interpretation besteht immer aus der Angabe der Funktions- und Variablendefinitionen. Die Funtionsdefinition von *myfunc* basiert auf dem `ite`-Ausdruck (if-then-else) und gibt *true* zurück, wenn x!1 gleich 101 sowie x!2 gleich *false* ist. Ist dies nicht der Fall, wird trotzdem *true* zurückgegeben. Per Definition würde die Funktion also für alle beliebigen Eingaben immer *true* ausgeben.

Alle Deklarationen und *Assertions* werden auf einem globalen Stack abgelegt und sind an jeder Stelle zugreifbar. Durch die Befehle *push* und *pop* kann jedoch auch das Hinzufügen von Sichtbarkeitsbereichen (Scopes) erfolgen. Mit *push* wird ein solcher Scope erstellt. Der *pop* Befehl entfernt alle Deklarationen und *Assertions*, die zwischen dem letzten *push* und diesem *pop* hinzugefügt wurden.

SMT unterstützt die Definition von algebraischen Datentypen. Dazu zählen *Records* und *Scalars*, sowie komplexere Typen wie endliche Listen, Bäume oder andere rekursive Datentypen.

Ein *Record* ist ein Datentyp mit einem einzelnen Konstruktor, der beliebige Datentypen als Argumente bekommt. Im Folgenden Beispiel 5.2 wird ein *Pair* definiert, welches den Konstruktor *mk-pair* besitzt. Die Argumente können durch die Selektionsfunktionen *first* und *second* abgerufen werden.

```
(declare-datatypes (T1 T2) ((Pair (mk-pair (first T1) (second T2)))))
(declare-const p (Pair Int Int))
(assert (< (first p) 5))
(assert (> (second p) 20))
(check-sat)
(get-model)
```

Der *Scalar* Typ ist eine endliche Menge an Konstanten. `(declare-datatypes () ((S A B C)))` erzeugt den *Scalar* Typ $S$ mit den drei Werten $A$, $B$ und $C$.

## 5.3 MontiArc Modelle in Z3

Für das Prüfen eines MontiArc-Modells auf unerfüllte Kontextbedingungen, muss das Modell zuerst in SMT repräsentiert werden. Um eine geeignete und performante Repräsentation zu finden, wurden zunächst manuell mehrere Versionen (V1 bis V7) aufgestellt und die benötigte Zeit für die Auswertung von ausgewählten Kontextbedingungen gemessen. Für die erste Iteration existiert eine ausführliche Beschreibung der verwendeten Deklarationen, Funktionsdefinitionen und Kontextbedingungen. Für die darauffolgenden Versionen werden jeweils nur kurz die Veränderungen erläutert.

### 5.3.1 V1

#### Deklarationen

Zunächst findet die Deklaration der Datentypen statt, die für die Repräsentation der Symboltabelle benötigt werden. Jedem Symbol und Scope lässt sich ein Name sowie eine eindeutige Identifikation durch das Berechnen eines Hashwertes zuordnen. Symbole besitzen zusätzlich auch ein Kind. Dafür muss ein Durchlauf der gesamten Symboltabelle geschehen, um alle verwendeten Symbole und Scopes zu sammeln. Anhand dessen können alle verwendeten Namen, Hashwerte und Kinds als Aufzählungstyp (*scalar*) deklariert werden (siehe Listing 5.1).

```
1  (declare-datatypes () ((Scope_ID scope_871790326 scope_1893960929
2              scope_431506362 ... scope_invalid)))
3
4  (declare-datatypes () ((Symbol_ID sym_464676531 sym_726379593
5              sym_1593458942 ... sym_invalid)))
6
7  (declare-datatypes () ((Name n_switch3 n_switch2 n_switch1
8              n_Decelerator_pc n_cond ... n_invalid)))
9
10 (declare-datatypes () ((Kind ComponentKind PortKind
11             ComponentInstanceKind ConnectorKind ... InvalidKind)))
```

Listing 5.1: Deklaration der Name, Kind und ID Datentypen.

Danach folgt die Deklaration der Datentypen für die einzelnen Symbole von MontiArc sowie dem Scope Datentyp (siehe Listing 5.2). Es existiert der Symbol Datentyp, der alle Informationen eines Symbols, wie Name, Kind und ID enthält. Jede Unterklasse von Symbol beinhaltet nur die entsprechenden Attribute der Unterklasse und keine eigenen Informationen über den Namen oder Kind. So besitzt beispielsweise das *ConnectorSymbol* nur die Attribute *source* und *target*, welche als Liste von Namen gespeichert sind.

Da vorerst getestet werden soll, ob Z3 sich für die Analyse eignet, enthalten die vorgestellten Modelle nicht die vollständige Funktionalität der Symboltabelle. Es werden nur diejenigen Funktionen implementiert, die für die Auswertung von vereinzelten Kontextbedingungen nötig sind.

#### Funktionsdefinitionen

Mittels der `getSymbolFromId` Funktion sind Symbol ID's auf konkrete Instanzen von Symbolen abgebildet. Die Funktionsdefinition ist in Listing 5.3 abgebildet. Die Abbildung

```
(declare-datatypes () ((Scope (mk-scope
                     (name Name)
                     (id Scope_ID)))))

(declare-datatypes () ((Symbol (mk-symbol
                     (name Name) (kind Kind)
                     (id Symbol_ID)))))

(declare-datatypes () ((PortSymbol (mk-port
                     (incoming Bool) (type Name)
                     (id Symbol_ID)))))

(declare-datatypes () ((ComponentSymbol (mk-component
                     (isInner Bool) (id Symbol_ID)))))

(declare-datatypes () ((ComponentInstanceSymbol (mk-cmpins
                     (componentType Name) (id Symbol_ID)))))

(declare-datatypes () ((ConnectorSymbol (mk-connector
                     (source (List Name)) (target (List Name))
                     (id Symbol_ID)))))
```

Listing 5.2: Deklaration der Symboltypen.

wird durch die Benutzung des `ite`-Ausdrucks erreicht. Stimmt der Eingangsparameter mit keiner bekannten ID überein, dann besteht die Rückgabe aus dem invaliden Symbol.

```
(define-fun getSymbolFromId ((id Symbol_ID)) Symbol
  (ite (= id sym_464676531)
    (mk-symbol n_BrakeForce_pedal_pc PortKind sym_464676531)
  (ite (= id sym_726379593)
    (mk-symbol n_out1 PortKind sym_726379593)
  ...
    (mk-symbol n_invalid InvalidKind sym_invalid)
  )...)))
)
```

Listing 5.3: Abbildung von ID auf Symbol.

Eine analoge Funktion existiert auch für Abbildung von Scope ID's auf Scopes.

Der Zugriff auf die Attribute eines konkreten Symbols, wie beispielsweise einem *PortSymbol*, erfolgt die Funktionen `getXFromId`, wobei X für ein beliebiges MontiArc Symbol steht. Die entsprechende Funktion für die PortSymbole ist in Listing 5.4 zu sehen. Die Angabe einer ID, welche nicht zu einem *PortSymbol* gehört, führt zur Rückgabe eines invaliden *PortSymbols*.

Des Weiteren ist es möglich, die Scope ID anhand der ID eines *ComponentSymbols* durch *getScopeIdFromComponentId* zu erhalten. Die Baumstruktur der Symboltabelle ist durch die Funktionen `isSymbolOf` und `isSubScopeOf`, welche genau dann *true* ausgeben, wenn sich ein Symbol bzw. Scope in einem Scope befindet. Die Funktionsdefinition für `isSymbolOf` ist in Listing 5.5 dargestellt. `isSubScopeOf` weist einen analogen Aufbau auf.

```
(define-fun getPortSymbolFromId ((id Symbol_ID)) PortSymbol
  (ite (= id sym_464676531)
    (mk-port true n_Double sym_464676531)
  (ite (= id sym_119290689)
    (mk-port false n_Boolean sym_119290689)
  ...
  (mk-port false n_invalid sym_invalid)
  )...)))
)
```

Listing 5.4: Abbildung von ID auf Symbol.

```
(define-fun isSymbolOf ((sym Symbol_ID) (scope Scope_ID)) Bool
  (or
    (and (= sym sym_464676531) (= scope scope_527829831))
    (and (= sym sym_960733886) (= scope scope_504582810))
    (and (= sym sym_332873513) (= scope scope_527829831))
    ...
  )
)
```

Listing 5.5: Funktionsdefinition von isSymbolOf.

### Kontextbedingungen

Für jede Kontextbedingung existiert eine korrespondierende Funktionsdefinition. Die Funktion für das Überprüfen der Namenseindeutigkeit innerhalb einer Komponente (B1) ist in Listing 5.6 zu sehen. Als Eingabe erhält die Funktion die Scope ID des Scopes, welcher von der zu testenden Komponente aufgespannt wird. Mittels des Allquantors `forall` in Zeile 2 wird festgelegt, dass für alle Paare an Symbol ID's jeweils eine Implikation gilt. Die Implikation gibt an, dass wenn beide Symbole im Scope der zu testenden Komponente liegen und unterschiedlich sind, dann sollen die Namen der Symbole ungleich sein.

```
(define-fun checkCoCo_B1 ((scope_id Scope_ID)) Bool
  (forall ((sym_id_1 Symbol_ID) (sym_id_2 Symbol_ID))
    (=>
      (and
        (not (= sym_id_1 sym_id_2))
        (isSymbolOf sym_id_1 scope_id)
        (isSymbolOf sym_id_2 scope_id)
      )

      (not
        (=
            (name (getSymbolFromId sym_id_1))
            (name (getSymbolFromId sym_id_2))
        )
      )
    )
  )
)
```

Listing 5.6: Funktionsdefinition von der Kontextbedingung B1.

### Prüfung auf Erfüllbarkeit

Um die Erfüllbarkeit einer Kontextbedingung zu prüfen, wird für jedes Symbol jeweils ein Stack angelegt und die zugehörige Funktion zur Menge der Formeln hinzugefügt. Der `echo` Befehl dient zur Ausgabe einer Nachricht, was in diesem Kontext dazu genutzt wird, die aktuelle Komponente und Kontextbedingung auszugeben. Danach erfolgt die Überprüfung auf Erfüllbarkeit durch den `check-sat` Ausdruck. Am Ende wird die Formel wieder entfernt.

```
1 (push)
2 (echo "Does Component VelocityControl satisfy context condition B1?")
3 (assert (checkCoCo_B1 scope_394785440))
4 (check-sat)
5 (pop)
```

Listing 5.7: Prüfung auf Erfüllbarkeit.

### 5.3.2 V2

#### Deklarationen

Die Datentypen für Symbole wurden vollständig entfernt. Die Deklarationen bestehen nur noch aus der Festlegung der verwendeten Scope ID's, Symbol ID's, Namen sowie Kinds.

#### Funktionsdefinitionen

Die Festlegung der Baumstruktur der Symboltabelle ist unverändert. Für den Zugriff auf die Attribute der einzelnen Symbole existieren separate Funktionsdefinitionen für jedes einzelne Attribut. So ist beispielsweise das Abrufen des Kinds durch die Funktion `getSymbolKindFromId`, welche in Listing 5.8 abgebildet ist, möglich.

```
1 (define-fun getSymbolKindFromId ((id Symbol_ID)) Kind
2   (ite (= id sym_464676531) PortKind
3   (ite (= id sym_1593458942) ConnectorKind
4   (ite (= id sym_1096485705) ComponentInstanceKind
5   (ite (= id sym_442199874) JavaTypeSymbolKind
6   ...
7   InvalidKind
8   )...)))
9 )
```

Listing 5.8: Zugriff auf den Kind eines Symbols.

#### Kontextbedingungen

Bei der Funktionsdefinition für Kontextbedingung B1 findet eine Ersetzung des Ausdrucks `(name (getSymbolFromId sym_id))` durch `(getSymbolNameFromId sym_id)` statt.

### 5.3.3 V3

#### Funktionsdefinitionen

Die Funktion isSymbolOf, welche zuvor für ein Symbol und Scope angegeben hat, ob sich das Symbol in diesem Scope befindet, bekommt einen veränderten Namen und Eingabe- bzw. Ausgabeparameter. Der neue Funktion getEnclosingScopeOf erhält eine Symbol ID und gibt die zugehörige Scope ID aus. Eine analoge Veränderung wurde bei isSuperScopeOf gemacht, wobei der neue Name getSuperScopeofScope ist und zu jeder Scope ID die entsprechende ID des *parent*-Scopes ausgibt. Für eine unbekannte ID als Eingabe erfolgt die Ausgabe von scope_invalid.

#### Kontextbedingungen

In Kontextbedingung B1 wurde der Ausdruck (isSymbolOf sym_id_1 scope_id) durch (= (getEnclosingScopeOfSymbol sym_id_1) scope_id) ersetzt.

### 5.3.4 V4

#### Kontextbedingungen

Bei der Nutzung der Implikation in den Kontextbedingungen wurden gewisse Teile der Antezedenz in einen ite Ausdruck ausgelagert. Damit soll erreicht werden, dass die Konsequenz nur dann ausgewertet wird, wenn gewisse Vorbedingungen gelten. In Kontextbedingung B1 wurde die Überprüfung der Zugehörigkeit der Symbole zum Scope ausgelagert (siehe Listing 5.9). Erst wenn beide Symbole zum angegebenen Scope gehören, wird die Implikation ausgewertet. Ansonsten muss der Ausdruck *true* sein, damit keine Unerfüllbarkeit aufgrund des Allquantors entsteht.

```
1  (define-fun checkCoCo_B1 ((scope_id Scope_ID)) Bool
2    (forall ((sym_id_1 Symbol_ID) (sym_id_2 Symbol_ID))
3      (ite (and (isSymbolOf sym_id_1 scope_id)
4                (isSymbolOf sym_id_2 scope_id))
5        (=>
6          (not (= sym_id_1 sym_id_2))
7
8          (not (= (getSymbolNameFromId sym_id_1)
9                  (getSymbolNameFromId sym_id_2)))
10       )
11       true
12     )
13   )
14 )
```

Listing 5.9: Auslagerung von Vorbedingungen.

### 5.3.5 V5

#### Deklarationen

Die Datentypen für die Scope sowie Symbol ID's wurden vollständig entfernt. Stattdessen kann eine Nutzung von Integer erfolgen, welche nativ von Z3 unterstützt sind. Alle Funktionssignaturen wurden entsprechend angepasst. Die Eigenschaft, dass Hashwerte nichtnegativ sind, wurde ausgenutzt, um die Werte `scope_invalid` und `symbol_invalid` durch -1 zu ersetzen. Die Definition der Kontextbedingungen bleibt großenteils unverändert. Durch die Kodierung geht jegliche Leserlichkeit des generierten SMT Quelltextes verloren.

### 5.3.6 V6

#### Deklarationen

Die restlichen Datentypen für die Namen und Kinds wurden auch entfernt. Stattdessen findet eine Kodierung statt, die alle verwendeten Namen und Kinds in Integer umwandelt. Für invalide Werte wird auch eine -1 zurückgeben. Die Kodierung ist eine einfache Nummerierung, die bei Null anfängt und von der Erfassung aller genutzten Symbole sowie Scopes der Symboltabelle abhängt. Die Eingabeparameter aller Funktionen akzeptieren ausschließlich Integer.

#### Kontextbedingungen

Die Nummerierung ermöglicht das Beschränken der Symbol ID in den Kontextbedingungen. Alle Quantoren werden auf Integer zwischen 0 und der maximalen Anzahl an Symbolen beschränkt.

### 5.3.7 V7

#### Kontextbedingungen

Das Auslagerung der Vorbedingungen aus den Implikationen wurde wieder rückgängig gemacht. Des Weiteren hat eine Entfernung der ID Beschränkungen in den Quantoren aus der vorigen Iteration stattgefunden.

## 5.4 Evaluation

Zur Bewertung der verschiedenen Versionen wurde die benötigte Zeit für das Ausführen alles Bedingungsüberprüfungen gemessen. Der Test beschränkt sich auf das Prüfen der Kontextbedingungen B1, R5 und R6. Des Weiteren existiert eine weitere Bedingung, die genau dann erfüllt ist, wenn das MontiArc-Modell keine identischen Komponentendefinitionen enthält. Die Teilkomponente *VelocityControl* eines Fahrerassistenzsystems dient als Testmodell. Die Symboltabelle des Modells umfasst insgesamt 59 Scopes und 126 Symbole. Das Testsystem besteht aus einem Intel Core i5-4670k Prozessor bei 3,4 Ghz und 8GB RAM.

Für alle Versionen wurde jeweils der Mittelwert aus drei Durchläufen erfasst und in Abbildung 5.1 festgehalten.

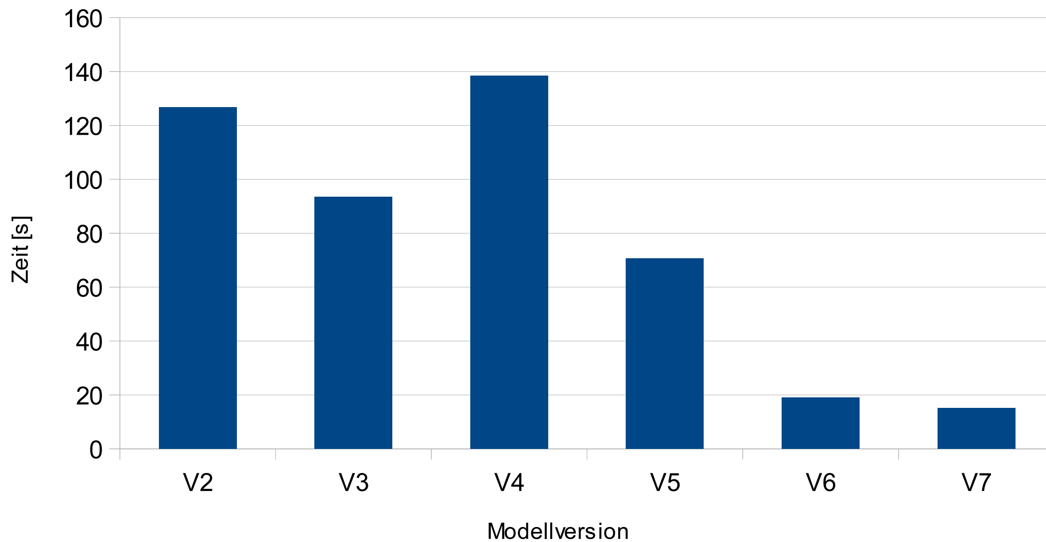


Abbildung 5.1: Benötigte Zeit der Modellversionen.

Die Testdaten für V1 konnten nicht erfasst werden, da die Testdurchführung nach einer Laufzeit von 30 Minuten noch nicht terminierte. Nach dem Entfernen der Datentypen sowie das direkte Abbilden von ID's auf Attribute hat die Auswertung nach ca. zwei Minuten terminiert.

Das Verändern der Signatur der Funktionen, die die Scope- und Symbolrelationen angegeben haben, hat zu einer weiteren Leistungssteigerung in V2 geführt.

Die Auslagerung der Vorbedingungen in V4 führte zu einer Verschlechterung der Leistung und wurde in V7 wieder entfernt. Selbst mit der Verschlechterung konnte jedoch der Wechsel von den ID Datentypen auf Integer in V5 eine Halbierung der benötigten Zeit verursachen.

In V6 konnten schließlich alle Datentypen durch Integer ersetzt werden. Dies war durch die Nummerierung aller Namen, Kinds, Symbol ID's und Scope ID's möglich. Des Weiteren wurden die Quantoren auf die gültigen ID's beschränkt. Diese Änderungen haben zu einem weiteren signifikanten Leistungsanstieg geführt.

Die letzte Verbesserung geschah in V7 durch das Entfernen der *ite*-Ausdrücke, welche in V4 eingeführt wurden. Die benötigte Zeit für die letzte Version beträgt 15,17 Sekunden.

Eine weitere signifikante Leistungssteigerung konnte durch das Anwenden von benutzerdefinierten Strategien in Z3 erzielt werden. Das Austauschen von *(check-sat)* durch *(check-sat-using (then simplify solve-eqs smt))* führt dazu, dass vor der Anwendung des SMT-Solvers zuerst eine Vereinfachung der Formelmenge stattfindet. Alle Tests wurden erneut mit der Vereinfachung durchgeführt und in Abbildung 5.2 dargestellt.

Selbst nach der Anwendung der Strategien beträgt die Zeit 4,23 Sekunden in der letzten Versionsiteration. Weitere Tests haben ergeben, dass die benötigte Zeit exponentiell mit

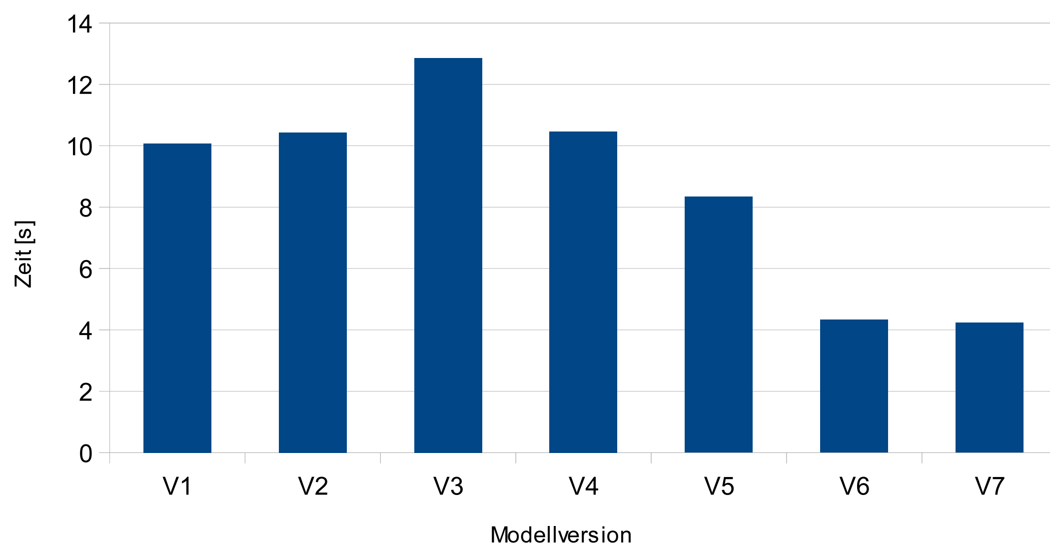


Abbildung 5.2: Benötigte Zeit der Modellversionen.

der Anzahl der Symbole ansteigt. Durch die fehlende Skalierbarkeit ist die Analyse mit Z3 nicht für große MontiArc-Modelle geeignet.

# Kapitel 6

# Alternativen zum Z3 Solver

Aufgrund der mangelhaften Skalierbarkeit der Analysen mit Z3 wurde im Rahmen dieser Arbeit ein Generator implementiert, der OCL-Ausdrücke nach Java übersetzt. Da die MontiArc-Symboltabelle bereits in Java vorliegt, muss das Modell nicht weiter übersetzt werden. In diesem Kapitel werden zunächst einige Designentscheidungen und Implementationen des *OCL2Java*-Generators erklärt. Danach kommt eine kurze Erörterung der angewendeten Qualitätssicherung. Abschließend folgt ein Leistungsvergleich der beiden Analyseverfahren mit Z3 und Java.

## 6.1 OCL2Java-Generator

Das MontiCore-Projekt enthält bereits einen Generator, der OCL zu Java transformiert und im Rahmen der Arbeit von Ulrich Helker [Hel10] entwickelt wurde. Da dieses Projekt jedoch von einer älteren MontiCore Version und OCL Grammatik abhängt, wurde in dieser Arbeit ein neuer Generator geschrieben, der auf einer neuen OCL Grammatik [Cel15] aufbaut. Basierend auf dem alten Generator wurden einige Konzepte übernommen oder verändert. Diese Unterschiede und Gemeinsamkeiten werden im Folgenden beschrieben.

### 6.1.1 Konzept

#### Logisches Lifting

Für den Fall, dass ein OCL Ausdruck nicht zu `true` oder `false` ausgewertet werden kann, muss dieser undefinierte Wert (`undef`) bei der Auswertung in Java separat behandelt werden. Dies kann mit dem booleschen Lifting, welches in [Rum11] definiert wurde, geschehen. Durch das Lifting wird ein undefinierter Wert wird immer als `false` interpretiert. Tabelle 6.1 zeigt alle Wahrheitswerte für die boolesche Disjunktion unter Berücksichtigung des undefinierten Wertes.

In Java gibt es nur zwei Situationen, in denen der Wert `undef` auftritt [Rum11]. Einerseits kann eine Exception geworfen werden und andererseits kann es sich um eine nicht-terminierende Auswertung handeln. Im zweiten Fall kann jedoch davon ausgegangen werden, dass aufgrund begrenzter Systemressourcen auch eine *Exception* geworfen

| a\|\|b | true | false | undef |
|---|---|---|---|
| true | true | true | true |
| false | true | false | false |
| undef | true | false | false |

Tabelle 6.1: Wahrheitswerte der booleschen Disjunktion nach Lifting.

wird. Deshalb wurde die Entscheidung, das Lifting durch `try-catch`-Blöcke zu realisieren, aus [Hel10] übernommen. Innerhalb des `try`-Blocks wird die Auswertung des Ausdrucks durchgeführt und das Ergebnis einer Hilfsvariable zugewiesen, welche außerhalb des `try-catch`-Blocks deklariert wurde. Falls der Ausdruck nicht ausgewertet werden kann, wird eine *Exception* geworfen und im `catch`-Block abgefangen. Dort findet das boolesche Lifting statt, indem die Hilfsvariable auf `false` gesetzt wird.

```
1 cmpSym.isInnerComponent || cmpSym.delayed
```

Listing 6.1: OCL Input.

```
1  boolean _oclInfix0LeftLifting = false;
2  try {
3      _oclInfix0LeftLifting = cmpSym.isInnerComponent;
4  } catch (Exception _oclInfix0LeftLiftingException) {
5      _oclInfix0LeftLifting = false;
6  }
7
8  boolean _oclInfix0RightLifting = false;
9  try {
10     _oclInfix0RightLifting = cmpSym.delayed;
11 } catch (Exception _oclInfix0RightLiftingException) {
12     _oclInfix0RightLifting = false;
13 }
14
15 ...
16
17 _oclInfix0LeftLifting || _oclInfix0RightLifting
```

Listing 6.2: Java Output.

#### Hilfsvariablenbenennung

Für die Benennung der Hilfsvariablen wird die *Singleton*-Klasse `OCLVariableNaming` vom Generator genutzt. Mit Hilfe der Methode `getName(ASTNode ast)` kann für jeden beliebigen AST-Knoten ein eindeutiger Variablenname erzeugt werden.

Bei der Initialisierung von `OCLVariableNaming` wird zunächst jeder AST-Klasse, welcher eine Variable zugewiesen werden kann, ein vordefinierter Präfix gegeben. Des Weiteren wird festgelegt, ob eine AST Klasse mehrfach auftauchen kann (wie beispielsweise geschachtelte `forall`-Ausdrücke) und somit eine Nummerierung als Suffix benötigt.

Beim Aufruf der `getName(ASTNode ast)` Methode wird zunächst überprüft, ob der gegebene AST-Knoten eine Nummerierung benötigt und ob diese bereits zugewiesen wurde. Diese Überprüfung wird durch eine *Map* ermöglicht, welche zu jedem Hashcode eines AST-Knotens die zugehörige Nummerierung speichert. Falls noch keine Nummer für den

gegebenen Knoten erzeugt wurde, wird ein globaler Zähler inkrementiert, welcher die Anzahl der bisher genutzten Variablen dieser AST-Klasse erfasst, und dem AST-Knoten zugewiesen.

Zuletzt wird die Hilfsvariable aus drei Teilen zusammengesetzt. Der erste Teil ist der String `_ocl`, um Konflikte mit reservierten Bezeichnern zu vermeiden und die Verständlichkeit des generierten Codes zu steigern. Danach folgt der Präfix der zugehörigen Klasse des Knotens. Zuletzt folgt die optionale Nummerierung.

Nach jeder erfolgreich generierten OCL-Datei wird `OCLVariableNaming` durch die Methode `public void reset()` zurückgesetzt, indem die globalen Zähler und die Zuordnung der Hashcodes zur Nummerierung zurückgesetzt werden.

### Pretransformationen

Bevor die Codegenerierung stattfindet, werden unterschiedliche Vortransformationen auf dem AST durchgeführt. Diese sorgen für eine Verringerung der Anzahl an AST-Knoten und vereinfachen die Implementation des Codegenerators.

Zunächst werden alle Infix-Ausdrücke zu einer Klasse `ASTOCLInfixExpression` zusammengefasst. Diese AST-Klasse beinhaltet jeweils einen OCL-Ausdrück für die rechte und linke Seite des Infix-Ausdrucks und einen Operator, welcher als String gespeichert wird. Dieser Schritt ist nötig, weil die neue OCL Grammatik zu einer Verschachtlung von unterschiedlichen Infixtypen geführt hat und somit den Generierungsprozess erschwert.

Danach folgen einige angepasste Vortransformationen aus der Implementation von Helker [Hel10]. Dazu zählen:

- Ersetzen des `if-then-else`-Ausdrucks durch den `?:`-Operator.
- Ersetzen des `any`-Ausdrucks durch die `any`-Containeroperation.
- Ersetzen des `isin`-Infix-Ausdrucks durch die `contains`-Containeroperation.
- Ersetzen des Äquivalenz-Operators. $a \Leftrightarrow b$ wird zu $((!a||b)\&\&(!b||a))$.
- Ersetzen des Implikation-Operators. a `implies` b wird zu $(!a||b)$.
- Vereinfachung von Quantoren mit mehrfachen Deklarationen zu geschachtelten Quantoren.

### *StringBuilder* versus *Template*

In [Hel10] wird die Template-Engine Freemarker [www16b] als Grundlage für die Codegenerierung genutzt. Freemarker ist als Java-Bibliothek in das MontiCore-Projekt eingebunden und wird u.a. für die Generierung der Infrastruktur (AST, Parser, Symboltabelle, usw.) von benutzerdefinierten Sprachen genutzt. In dieser Arbeit wurde sich jedoch für einen `StringBuilder` entschieden, welcher mithilfe von Visitoren für jeden AST-Knoten den entsprechenden Quelltext erzeugt. Dies bringt folgende Vorteile mit sich:

- Nutzung des bereits vorhandenen AST-Visitors.

- Erleichtertes Debugging.
- Vermeidung von verschachtelten und unübersichtlichen Template-Beziehungen.

Der Generierungsprozess beginnt mit dem Parsen des OCL-Modells und Erzeugen eines `StringBuilder` Objektes, welches zum Ende den vollständigen Java-Quelltext beinhaltet. Die Visitoren `OCL2JavaInplaceVisitor` und `OCL2JavaDeclarationVisitor` besuchen jeweils die AST-Knoten und schreiben den erzeugten Quelltext in den *StringBuilder*. Das Konzept und die Bezeichnung dieser Visitoren wurden von den Inplace- bzw. Declaration-Templates aus [Hel10] nachempfunden.

Durch das boolesche Lifting und der verschachtelten Struktur von OCL-Ausdrücken ist oftmals die Nutzung von Hilfsvariablen zur Zwischenspeicherung nötig. Aus diesem Grund muss für einige Ausdrücke zuerst der Quelltext für die Deklaration und Zuweisung von Hilfsvariablen generiert werden. Dies geschieht mit dem `OCL2JavaDeclarationVisitor`. Mit dem `OCL2JavaInplaceVisitor` wird der Quelltext erzeugt, indem die Hilfsvariablen schlussendlich genutzt werden. Der `OCLDeclarationNodesVisitor` durchläuft die Nachkommen des aktuellen AST-Knoten und gibt eine Liste an allen Knoten zurück, wie vor der Verwendung erst deklariert werden müssen. Jeder dieser Knoten akzeptiert den `OCL2JavaDeclarationVisitor`, was zum Aufbau der entsprechenden Deklarationen führt.

#### Prüfen von MontiArc Modellen

Die Klasse `CheckMontiArc` kann durch die Angabe von Startparametern genutzt werden, um ein MontiArc-Modell auf gegebene OCL-Ausdrücke zu prüfen. Als erster Parameter muss der Pfad des Modells angegeben werden. Danach folgt der Pfad des Ordners, in dem sich alle OCL-Dateien befinden, die getestet werden sollen.

Der Analyseprozess ist in mehrere Schritte eingeteilt. Zuerst wird das Modell eingelesen, zum AST transformiert und die entsprechende Symboltabelle erzeugt. Danach erfolgt das Generieren des Java-Quelltextes aus den OCL-Ausdrücken. Dieser Quelltext muss anschließend kompiliert werden. Am Ende findet der Aufruf der kompilierten Methoden auf der Symboltabelle statt. Eine Ausgabe in der Konsole gibt Auskunft über die Erfüllbarkeit jeder einzelnen Bedingung.

### 6.1.2 Integration in das MontiCore-Projekt

#### Codegenerierung

Die Implementation ist im *montiarc4-features* Projekt [www16d] aufzufinden.

Im Quellcodeverzeichnis `main/java` des Projektes `features.verification.ocl2java` befindet sich das Paket `de.monticore.lang.ocl.codegen`. Die Paketstruktur sieht wie folgt aus:

- `de.monticore.lang.ocl.codegen.substitutions`
  enthält die Visitoren, die zur Substitution von AST-Knoten genutzt werden und von `SubstitutionVisitor` erben.

- `de.monticore.lang.ocl.codegen.modifications`
  enthält die Visitioren, die zur Modifikation der AST-Knoten genutzt werden.
- `de.monticore.lang.ocl.codegen.visitors`
  enthält die Visitoren zum Aufbau des Quelltextes mithilfe eines *StringBuilders*.
- `de.monticore.lang.ocl.codegen`
  enthält `OCL2Java`, welche die Transformationen durchführt und den Generator startet sowie Hilfsklassen wie `OCLVariablenNaming` und `OCLHelper`.

**Tests**

Die Tests sind im Quellcodeverzeichnis `src/test/java` zu finden und wurden mit JUnit realisiert.

- `OCLCoCoTest` testet die Codegenerierung von MontiArc's Kontextbedingungen.
- `OCLInvariantTest` testet die Codegenerierung von OCL Invarianten.

## 6.2 Qualitätssicherung

Für die Qualitätssicherung des Generators wurden Tests mit dem JUnit-Framework erstellt. Alle benötigten Testressourcen befinden sich im Verzeichnis `test/resources/`.

Der Testvorgang für die Korrektheit der Generatorausgabe ist wie folgt aufgebaut. Die zu testenden OCL-Ausdrücke befinden sich alle in `ocl/codegen/input`. Die erwarteten Ausgaben des Generators liegen in `ocl/codegen/desired`. Nach dem Generierungsprozess wird der resultierende Java-Quelltext genutzt, um einen AST daraus aufzubauen. Das Gleiche wird mit der erwarteten Ausgabe, die auch als Java-Quelltext vorliegt, gemacht. Durch den Aufruf der `deepEquals` Methode können zwei AST's auf Gleichheit überprüft werden. Wenn keine Gleichheit vorliegt, schlägt der Test fehlt.

Des Weiteren existieren im Verzeichnis `ma` jeweils ein invalides und valides Modell zu jeder Kontextbedingung. Durch das Generieren des entsprechenden OCL-Ausdrücke wird jeweils die Korrektheit der Bedingung getestet. Bei invaliden Modellen darf die Bedingung nicht erfüllt sein während sie bei validen Modell erfüllt ist.

## 6.3 Leistungsvergleich von Java und Z3

Um die Skalierbarkeit des Analyseverfahrens mit Java zu erfassen, wurden von mehreren MontiArc Modelle mit unterschiedlichen Größen die benötigte Zeit zur Auswertung erfasst. Für die gleichen Modelle wurde zusätzlich die benötigte Zeit der Modellversion V7 mit *simplify* in Z3 gemessen. Das Erfassen der Messdaten erfolgte wie in Kapitel 5 durch die Berechnung des Mittelwertes aus drei Durchgängen. Die Ergebnisse für Kontextbedingung B1 (Namenseindeutigkeit) sind in Abbildung 6.1 zu sehen.

Auf der x-Achse ist jeweils die Modellgröße zu sehen, welche durch die Anzahl an Symbolen der Symboltabelle repräsentiert wird. Die benötigte Zeit wurde in Sekunden gemessen und

in einer logarithmischen Darstellung auf der y-Achse abgebildet. Für alle Modelle bleibt die Ausführungszeit in Java unter 0,01 Sekunden während sie für den Z3 Solver exponentiell ansteigt.

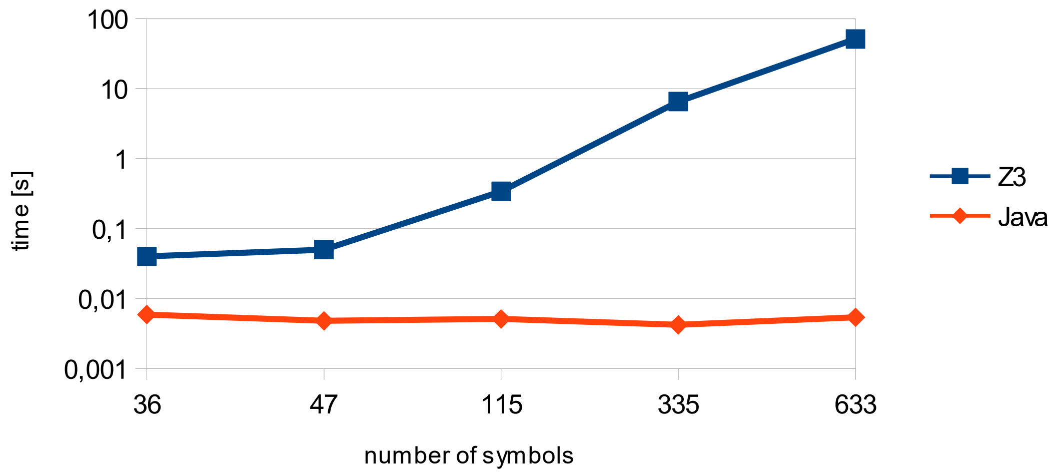


Abbildung 6.1: Benötigte Zeit für Kontextbedingung B1 in Java und Z3.

Für die Erkennung von identischen Komponentendefinitionen (siehe Abbildung 6.3) ergibt sich ein ähnlicher Verlauf der Testdaten.

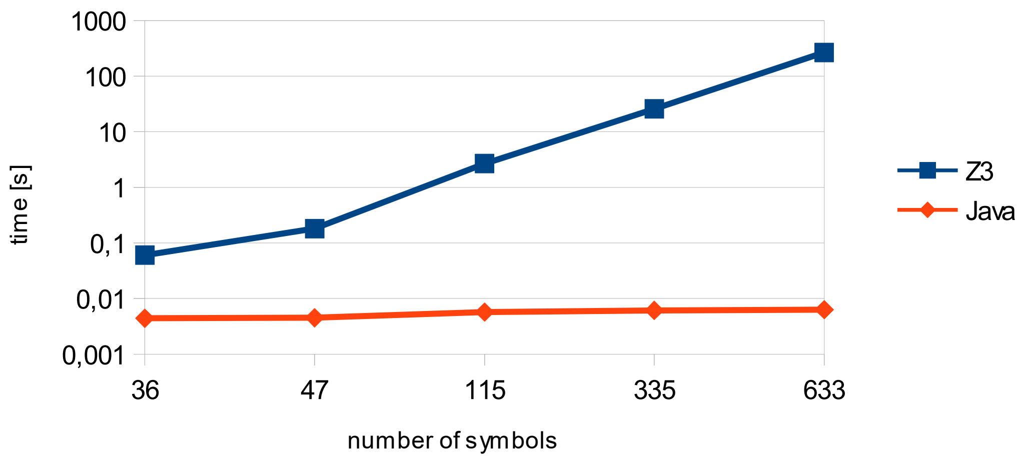


Abbildung 6.2: Benötigte Zeit für die Erkennung von identischen Komponentendefinitionen in Java und Z3.

Aus den Tests folgt, dass die Z3-Modelle aus Kapitel 5 nicht für die strukturelle Analyse geeignet sind. Die alternative Java-Implementation hingegen bleibt auch bei größeren Modellen performant.

# Kapitel 7

# Zusammenfassung und Ausblick

Ziel dieser Arbeit war die strukturelle Analyse von MontiArc-Modellen mittels des Z3 Solvers. Nach einer Einführung in die Grundlagen von OCL, SMT und MontiArc wurde zunächst ein Ansatz vorgestellt, der die OCL nutzt, um Bedingungen an MontiArc-Modelle zu stellen. Dies wurde erreicht, indem die Bedingungen vom Nutzer als OCL-Ausdrücke implementiert und danach zusammen mit der Symboltabelle des Modells in eine Zielsprache übersetzt werden, wo die Auswertung der OCL-Ausdrücke stattfindet.

Dann wurde untersucht, ob SMT-LIB in Verbindung mit Z3 sich für diese Auswertung eignet. Mehrere Repräsentationen der Symboltabelle in SMT-LIB wurden erstellt und anschließend anhand von Leistungsvergleichen bewertet. Die Evaluation hat ergeben, dass die entwickelten SMT-Repräsentationen mit Z3 selbst für kleine Modelle zu langsam waren. Eine gute Skalierbarkeit konnte nicht erzielt werden.

Deshalb wurde danach ein alternativer Ansatz vorgestellt, bei dem nur die OCL-Ausdrücke nach Java übersetzt und auf der bereits vorhandenen Symboltabelle ausgewertet werden. Für die Übersetzung nach Java, wurde der *OCL2Java*-Generator in das MontiCore-Projekt implementiert. Der Generator basiert auf der Arbeit von [Hel10] und wurde entsprechend an die neue OCL Grammatik und MontiCore Version angepasst. Jedoch fehlen einige Funktionalitäten wie beispielsweise die Unterstützung von Methodenspezifikationen, einige Container-Funktionen und Operatoren wie *defined* oder iterate. Zukünftige Arbeiten könnten den neuen Generator um die fehlenden Funktionen ergänzen.

Die Benutzung von Java hat zu einer deutlich schnelleren Auswertung im Vergleich zu Z3 geführt. In dieser Arbeit konnte jedoch keine performante Repräsentation der Symboltabelle als SMT-LIB erstellt werden. Für zukünftigen Arbeiten wäre das Entwickeln einer neuen und besseren Kodierung der Symbolbeziehungen möglich.

# Literaturverzeichnis


[Bra07] Matthias Braeuer. *Design and Prototypical Implementation of a Pivot Model as Exchange Format for Models and Metamodels in a QVT/OCL Development Environment*. Großer beleg, Technische Universitaet Dresden, 2007.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 26, pages 825–885. IOS Press, February 2009.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010. Available at `www.SMT-LIB.org`.

[Cel15] Abdullah Celik. *Implementierung der Object Constraint Language (OCL) mit der MontiCore Language Workbench*. Bachelorarbeit, RWTH Aachen, 2015.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[dMB09] Leonardo de Moura and Nikolaj Bjørner. *Satisfiability Modulo Theories: An Appetizer*, pages 23–36. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[Hab12] Arne Haber. *MontiArc - Architectural Modeling and Simulation of Interactive Distributed Systems*. Dissertation, RWTH Aachen, 2012.

[Hel10] Urlich Helker. Integration der Object Constraint Language in die UML/P. Diplomarbeit, RWTH Aachen, 2010.

[HRR14] Arne Haber, Jan Oliver Ringert, and Bernhard Rumpe. Montiarc - architectural modeling of interactive distributed and cyber-physical systems. *CoRR*, abs/1409.6578, 2014.

[Kra10] Holger Krahn. *MontiCore: Agile Entwicklung von domänenspezifischen Sprachen im Software-Engineering*. Dissertation, RWTH Aachen, 2010.

[Rum02] Bernhard Rumpe. *≪Java≫OCL Based on New Presentation of the OCL-Syntax*, pages 189–212. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[Rum11] Bernhard Rumpe. *Modellierung mit UML*, volume 2nd Edition. Springer, 2011.

[www16a] DresdenOCL http://www.dresden-ocl.org/index.php/DresdenOCL, september 2016.

[www16b] Freemarker Dokumentation http://freemarker.org/docs/, september 2016.

[www16c] https://confluence.jetbrains.com/display/MPSD34/Structure MPS Dokumentation, september 2016.

[www16d] https://git.rwth-aachen.de/vonwenckstern/montiarc4-features montiarc4-features Projekt, september 2016.

[www16e] https://www.eclipse.org/Xtext/documentation/303_runtime_concepts.html XText Dokumentation, september 2016.