University of Applied Sciences Northwestern Switzerland
School of Engineering

IML: Enhancements for Robust Programming
Support for Design by Contract

Florian Lüscher, Matthias Brun
Fachhochschule Nordwestschweiz
<{florian.luescher,matthias.brun}@students.fhnw.ch>

18. Februar 2013

Obwohl in den vergangenen Jahrzehnten im Bereich des Softwareengineerings viel Aufwand in das Entwickeln von Technologien zur effizienteren Entwicklung robuster Software gesteckt wurde, gibt es noch immer wenige Programmiersprachen welche das Schreiben robuster Software beim Sprach-Design berücksichtigt haben. Eine der Ausnahmen ist die Sprache Eiffel, welche beispielsweise das Definieren von Pre-/Postconditions ermöglicht. Jedoch gibt es vermehrt auch Frameworks für etabliertere Sprachen, welche es erlauben, Elemente des Design-by-Contract anzuwenden und vom Compiler zu überprüfen. In der Sprache C# von Microsoft beispielsweise wird dies mittels *Code Contracts* ermöglicht, welche auch vom Compiler statisch analysiert werden [1]. Auf der Java Platform gibt es über den Community Process Anstrengungen, über Annotationen zusätzliche Metainformationen bereitzustellen, welche von einem externen Tool (z.B. FindBugs) statisch analysiert werden können und den Programmierer bei möglichen Fehlern warnen [2]. Jedoch wurde dies auch im aktuellen Compiler der Java Version 7 noch nicht implementiert. In diesem Bericht werden Erweiterungen der Unterrichtsprache IML beschrieben, welche durch das Einführen von Pre-/Postconditions das Schreiben robuster Software erleichtern sollen.

1 Ausgangslage

Als Grundlage wird das Sprachdesign von IML[3] dienen, wie es zur Verfügung gestellt wurde. In einer ersten Phase wurde ein Compiler für diesen Sprachumfang entwickelt, welcher danach für die in diesem Dokument beschriebenen Erweiterungen angepasst wurde. Dieser Compiler wurde auf Basis der Sprache Scala entwickelt, da diese einerseits funktionale Aspekte beim Programmieren ermöglicht, andererseits bereits Parser-Kombinatoren im Sprachumfang enthalten sind.

2 Spracherweiterungen

In diesem Abschnitt werden die geplanten Erweiterungen an der Sprache erläutert. Ein wichtiges Kriterium für unsere Erweiterungen war es, die Kompatibilität von zur Standard-IML beizubehalten. Es muss möglich sein, bereits vorhandene Programme auch mit dem erweiterten Compiler zu kompilieren und eine äquivalentes Programm zu erhalten. Dies macht es zudem leichter, bestehende IML Programme um pre-/postconditions zu erweitern. Die vollständige Grammatik der Sprache ist im Anhang A ersichtlich. Weitere funktionierende sowie nicht funktionierende Beispiele sind in den Anhängen B und C zu finden.

2.1 Pre-/Postconditions

Das definieren von pre- und postconditions sollte direkt beim Deklarieren der Prozedur oder der Funktion ermöglicht werden. So sind die Garantien für den Programmierer auf einen Blick ersichtlich und erleichtern so das Verständnis der Prozedur. In Listing 1 ist ein Beispiel einer Prozedur mit conditions ersichtlich. Das Label "dumbCondition"dient dazu, im Falle eines Fehlschlags der darauf folgenden condition hilfreiche Fehlermeldungen generieren zu können. In Folge des Fehlens eines String-Literals wird ein Ident verwendet. Als conditions können alle Expressions verwendet werden, welche zu einem boolean Typ ausgewertet werden. Mehr dazu wird im Kapitel 3 erläutert. Es wird ebenfalls erlaubt, eine leere Condition List zu definieren. Dies erlaubt es dem Programmierer zu zeigen, dass er an Pre-/Postconditions gedacht hat, die Routine aber beispielsweise für alle Eingaben funktioniert.

```
proc divide(in copy m:int32, in copy n:int32, out ref q:int32, out ref r:int32)
requires [n > 0, unnecessary(m, n): dumbCondition]
ensures [r >= 0]
{
    q init := 0;
    r init := m;

    while(r >= n) {
        q := q+1;
        r := r-n
    }
}
```

Listing 1: Beispiele von pre-/postconditions

2.1.1 Lexikalische Syntax

Es wurden die in Listing 2 ersichtlichen neuen Terminale eingeführt.

```
[           (Token: LBRACKET)
]           (Token: RBRACKET)
requires    (Token: REQUIRES)
ensures     (Token: ENSURES)
```

Listing 2: Liste neuer Terminalsymbole

2.1.2 Grammatikalische Syntax

Die Grammatik wurde um neue nichtterminal Symbole erweitert. Zudem wurden die NTS funDecl und procDecl erweitert, damit nun die conditions an die Deklaration gehängt werden können. Die angepassten Symbole sind in Listing 4 dokumentiert.

```
requires        ::= REQUIRES conditionList
ensures         ::= ENSURES  conditionList

conditionList   ::= LBRACKET [condition {COMMA condition}] RBRACKET
condition       ::= expr [COLON IDENT]
```

Listing 3: Neue nichtterminal Symbole

```
funDecl     ::= FUN IDENT paramList
                RETURNS storeDecl
```

```
                [GLOBAL globImpList]
                [LOCAL cpsDecl]
                [requires]
                [ensures]
                blockCmd

procDecl    ::= PROC IDENT paramList
                [GLOBAL globImpList]
                [LOCAL cpsDecl]
                [requires]
                [ensures]
                blockCmd
```

Listing 4: Angepasste nichtterminal Symbole

2.2 Zugriff auf pre-execution state

Der Zugriff auf die Werte der Variablen vor der Ausführung der Prozedur ist von entscheidender Bedeutung für das Definieren von post conditions. Nur so kann beispielsweise überprüft werden, dass sich das Vorzeichen einer Variable nicht geändert hat, oder das ein neuer Wert einer Variable im Vergleich zum Alten grösser geworden ist. Dies verlangt nach einem Konstrukt, welches es dem Programmierer erlaubt festzulegen, ob auf den Wert der Variablen nach der Ausführung oder vor der Ausführung der Prozedur zugegriffen werden soll.

Wir haben uns für das Definieren einer reservierten Funktion namens *old* entschieden. Dies hat den Vorteil, dass die Grammatik nicht komplexer wird. Jedoch muss ein neuer Kontext-Check eingeführt werden, welcher überprüft, dass kein Namenskonflikt mit einer existierenden Funktion auftritt. Zudem muss überprüft werden, ob *old* nur innerhalb von postconditions auftritt. Ein Beispiel ist in Listing 5 zu sehen. Diese Variante schränkt die Abwärtskompatibilität leider ein, da keine Programme verwendet werden können, welche eine Routine namens *old* definiert haben.

```
proc divide(in copy m:int32, in copy n:int32, out ref q:int32, out ref r:int32)
ensures [old(n) > r]
{
    q init := 0;
    r init := m;

    while(r >= n) {
        q := q+1;
        r := r-n
    }
}
```

Listing 5: Beispiel eines Zugriffs auf alten Zustand

2.2.1 Lexikalische Syntax

Es sind keine Änderungen notwendig.

2.2.2 Grammatikalische Syntax

Es sind keine Änderungen notwendig.

3 Kontext- und Typeinschränkungen

In diesem Kapitel werden die nötigen Kontext Checks beschrieben.

3.1 Kontext Einschränkungen

Für conditions gilt grundsätzlich, dass bei ihrer Ausführung kein State verändert werden darf. In IML wurde dies durch die saubere Trennung von Expressions und Commands, sowie die standard Kontext Checks bereits erreicht. Trotzdem sind einige weitere Einschränkungen nötig.

- Die Funktion old() darf nur innerhalb von postconditions aufgerufen werden.
- Es darf keine weitere Funktion mit dem Namen old deklariert werden.
- Das Label einer Condition muss innerhalb der Condition Lists der selben Routine einmalig sein.
- Expressions innerhalb der Conditions müssen einen boolschen Wert erzeugen.
- In Conditions darf kein rekursiver Aufruf an die Funktion zu der sie gehört stattfinden.

3.1.1 Speicherzugriff

- In Preconditions kann auf alle initialisierten Variablen und Konstanten zugegriffen werden, welche in der Parameterliste oder der Global Import List definiert wurden. Auf lokal deklarierte Variablen kann nicht zugegriffen werden. Da out-Parameter nicht initialisiert sind, stehen sie nicht zur Verfügung.
- In Postconditions kann auf alle Variablen zugegriffen werden, welche auch in Preconditions möglich sind.
- In Postconditions von Funktionen kann auf die Returnvariable zugegriffen werden.
- In Postconditions von Prozeduren kann auf out-Parameter zugegriffen werden.
- In einer Postcondition kann mit old() eine Expression im preexecution State ausgeführt werden. Für diese Expression gelten daher auch die Einschränkungen des preexecution state.

4 Code Generation

Die Zielplattform des generierten Codes ist die Java Virtual Machine [4]. Daher wird Java Bytecode generiert. Dies kann mit den selben Prinzipien angegangen werden, welche auch bei der Generierung von Maschinen Code oder Code für die im Unterricht zur Verfügung gestellte VM gelten. Da die JVM jedoch von Beginn an auf die Sprache Java zugeschnitten wurde, gibt es bei der Abbildung eines IML Programmes auf die Struktur eines JVM Programmes einige Spezialitäten, auf welche in diesem Abschnitt eingegangen wird. Für das Schreiben der *.class* Dateien wird die verbreitete Bibliothek ASM [5] verwendet.

4.1 Strukturierung

Die JVM ist wie die Sprache Java objektorientiert aufgebaut. Aus diesem Grund muss ein IML Programm auf die Klassenstruktur der JVM abgebildet werden. In diesem Abschnitt wird anhand des Beispielprogramms aus Listing 6 die Abbildung auf eine JVM Klasse erläutert. Dieses IML Programm wird zu einer Klasse, welcher der Klassendeklaration in Listing 7 entspricht. Dabei wurde folgende Abbildung gewählt:

- **Programm**: Ein IML Programm wird auf eine Klasse mit gleichem Namen abgebildet. Um die Kompatibilität mit andern JVM Sprachen zu gewährleisten werden die globalen Variablen im Konstruktor mit Defaultwerten initialisiert. Die Klasse wurde als final gekennzeichnet, um späteres überschreiben von Methoden zu verhindern.

- **Globale Variablen**: Globale Variablen werden zu privaten Feldern der Klasse des Programms.
- **Programm Body**: Die Commands des IML Programms werden zu einer neuen Funktion der Klasse, welche den Namen der Klasse hat, kompiliert. Der Vorteil im Gegensatz zum direkten kompilieren in die main Methode liegt darin, dass die globalen Variablen und Routinen nicht *static* sein müssen und so mehrere Instanzen des IML Programms in der selben VM aktiv sein können, ohne Zustand teilen zu müssen. Dies würde es auch erlauben, Bibliotheksfunktionen in IML zu entwickeln und in anderen JVM Sprachen zu verwenden.
- **Routinen**: Routinen werden zu Funktionen der Klasse. Wenn keine out-Parameter verwendet werden entsprechen diese Funktionen normalen JVM Funktionen und stehen ohne spezielle Konventionen an den Aufruf zur Verfügung. Wie out-Parameter abgebildet werden ist im nächsten Abschnitt erläutert.
- **Eintrittpunkt**: Falls die JVM eine Klasse ausführen will, so ruft sie die Funktion **main** auf. Daher wird ebenfalls eine solche generiert, welche eine neue Instanz der Klasse erzeugt und die Methode mit dem gleichen Namen der Klasse ausführt.

```
program outtest
global
    proc loadTwice(ref i: int32, out ref a: int32, out ref b: int32)
    requires[ i > 0 : exampleCondition ]
    ensures[ a = old(i), b = old(i) ]
    {
        a init := i;
        b init := i
    };
    a: int32;
    b: int32;
    var v: int32
{
    v init := 12;
    call loadTwice(v, a init, b init);
    ! a;
    ! b
}
```

Listing 6: Laden eines Int-Wertes in zwei Speicherstellen über out Parameter

```
public final class outtest {
    public void loadTwice(int, int[], int[]);
    public outtest();
    public void outtest();
    public static void main(java.lang.String[]);
}
```

Listing 7: Struktur der erzeugten Java-Klasse

4.2 Parameterübergabe

IML unterstützt ein sehr ausgefeiltes Handling von Parameterwerten. Die JVM jedoch erlaubt das Übergeben von Parametern lediglich *by-value*. Zudem ist es in IML möglich festzulegen, ob ein Parameter ein Eingabeparameter, Ausgabeparameter oder beides ist. Auch hier erlaubt die JVM lediglich Eingabeparameter. Um Parameter von IML vollständig abbilden zu können werden auch *ref* Parameter wie *copy* Parameter behandelt. *out*-Parameter können so jedoch nicht realisiert werden, da die JVM dafür Referenzen auf primitive Typen unterstützen müsste. Um nun das Schreiben von Informationen in den Speicherbereich des Aufrufers zu erlauben, könnte eine Hilfsklasse (z.B. IntRef) erzeugt werden. Dies würde jedoch eine Abhängigkeit zu dieser Klasse für den generierten Code und jeden, der diesen Benutzen möchte,

zur Folge haben. Dies kann umgangen werden, wenn der entsprechende Parameter in ein Array mit der Länge 1 umgewandelt wird. So können *out* Parameter auf der JVM elegant implementiert werden. Dies bedeutet für den Aufrufer jedoch mehr Aufwand, da er zunächst für jeden *out* Parameter ein neues Array erzeugen und den zu übergebenden Wert(im Falle von *inout*) hinein kopieren muss. Im Anschluss an den Methodenaufruf muss der Wert im Array wieder an die Speicherstelle des ursprünglichen Wertes kopiert werden. Für den Aufgerufenen fällt jedoch keinerlei Zusatzaufwand an, wie in Listing 8 zu sehen ist.

```
public void loadTwice(int, int[], int[]);
    flags: ACC_PUBLIC
    Code:
      stack=3, locals=4, args_size=4
        0: aload_2          // Laden des 2. Parameters auf Stack (Array Referenz)
        1: iconst_0         // Laden von 0 auf den Stack (Pos in Array)
        2: iload_1          // Laden des ersten Parameters auf Stack (zu speichernder Wert)
        3: iastore          // Speichern (int array store)
        4: aload_3
        5: iconst_0
        6: iload_1
        7: iastore
        8: return
```

Listing 8: Bytecode der loadTwice Prozedur ohne conditions

```
public void outtest();
    flags: ACC_PUBLIC
    Code:
      stack=4, locals=3, args_size=1
         0: aload_0
         1: bipush        12
        3: putfield      #24              // Field v:I
         6: iconst_1                      // Array Size 1
         7: newarray       int            // Array erzeugen
         9: astore_1                      // Speichern an lokalem Index 1
        10: iconst_1                      // Array Size 1
        11: newarray       int            // Array erzeugen
        13: astore_2                      // Speichern an lokalem Index 2
        14: aload_0                       // this - fuer Methoden Aufruf
        15: aload_0                       // this - fuer Laden der globalen Variable
        16: getfield      #24             // Globale Variable v laden
        19: aload_1                       // Array fuer a fuer Call laden
        20: aload_2                       // Array fuer b fuer Call laden
        21: invokevirtual #26             // Method loadTwice:(I[I[I)V call
        24: aload_0                       // this
        25: aload_1                       // Array fuer a laden
        26: iconst_0                      // Position in Array
        27: iaload                        // Laden des int aus Array auf den Stack
        28: putfield      #20             // Field a:I out Wert speichern
        31: aload_0                       // wie obiger Code
        32: aload_2
        33: iconst_0
        34: iaload
        35: putfield      #22             // Field b:I out Wert Speichern
        38: getstatic     #32             //Field java/lang/System.out:Ljava/io/PrintStream;
        41: aload_0
        42: getfield      #20             // Field a:I
        45: invokevirtual #38             // Method java/io/PrintStream.println:(I)V
        48: getstatic     #32             //Field java/lang/System.out:Ljava/io/PrintStream;
        51: aload_0
        52: getfield      #22             // Field b:I
        55: invokevirtual #38             // Method java/io/PrintStream.println:(I)V
```

```
       58: return
```

Listing 9: Bytecode des Aufrufs der loadTwice Prozedur

4.3 Pre-/Postconditions

Für das Umsetzen der Pre-/Postconditions kann gewöhnlicher Code generiert werden. Falls eine der Conditions jedoch fehlschlägt, muss die Programmausführung unterbrochen werden. Da Java bereits ein Konstrukt für Assertions unterstützt, gibt es bereits in der Standartbibliothek die *AssertionError* Exception. Diese kann auch für die Conditions in IML verwendet werden, wobei das Label der Condition, falls vorhanden, als Exception Message verwendet wird.

```
public void loadTwice(int, int[], int[]);
    flags: ACC_PUBLIC
    Code:
      stack=3, locals=4, args_size=4
         0: iload_1                      // i auf Stack laden
         1: bipush        0              // 0 auf Stack laden
         3: if_icmple     10             // conditional jump (less equals)
         6: iconst_1                     // true laden
         7: goto          11             // ans Ende springen
        10: iconst_0                     // false laden
        11: ifeq          17             // falls Ergebnis false zu AssertionError springen
        14: goto          27             // Weiter mit normalem Code
        17: new           #9             // class java/lang/AssertionError
        20: dup
        21: ldc           #11            // Laden der String Konstante "exampleCondition"
        23: invokespecial #15            // Method java/lang/AssertionError."<init>"
        26: athrow                       // Werfen der Exception
        27: aload_2
        28: iconst_0
        29: iload_1
        30: iastore
        31: aload_3
        32: iconst_0
        33: iload_1
        34: iastore
        35: return
```

Listing 10: Bytecode der loadTwice Prozedur mit precondition

4.3.1 Speichern des Preexecution States

Der Preexecution State wird innerhalb der Methode als lokale Variable gespeichert, bevor die erste Anweisung des Routingen Bodies ausgeführt wird. Wenn nun die Postcondtitions ausgewertet werden, wird im Falle eines Calls der *old* Funktion der Wert an der entsprechenden Position der lokalen Variable geliefert. Die Position entspricht der Reihenfolge des Vorkommens in der Postcondition.

```
public void loadTwice(int, int[], int[]);
    flags: ACC_PUBLIC
    Code:
      stack=3, locals=5, args_size=4
         0: iload_1                 // Expression in old(i) -> i
         1: istore        4         // Speichern an lokalem Variablen Index 4
         3: aload_2                 // Normaler Code
         4: iconst_0
```

```
 5: iload_1
 6: iastore
 7: aload_3
 8: iconst_0
 9: iload_1
10: iastore
11: aload_2                // Array Referenz fuer a laden
12: iconst_0               // Position 0
13: iaload                 // Laden des aktuellen Wertes von a
14: iload         4        // Zuvor gespeicherter Wert aus Index 4 Laden
16: if_icmpne     23       // Falls Werte ungleich zu 23 springen
19: iconst_1               // true auf Stack laden
20: goto          24       // zu check springen
23: iconst_0               // false auf Stack laden
24: ifeq          30       // falls false zu AssertionError springen
27: goto          38       // Ans Ende des Checks springen
30: new           #9       // class java/lang/AssertionError
33: dup
34: invokespecial #13      // Method java/lang/AssertionError."<init>":()V
37: athrow
38: return
```

Listing 11: Bytecode der loadTwice Prozedur mit postcondition mit Zugriff auf old State. Damit der Code etwas übersichtlicher ist wurde nur die erste Postcondtition kompiliert.

5 Vergleich mit anderen Programmiersprachen

In diesem Abschnitt wird der gewählte Entwurf mit Implementierungen für andere Sprachen verglichen.

5.1 Java / C(++)

Die meist verwendeten Sprachen Java, sowie C(++) enthalten keine Unterstützung des Compilers zur Definition von pre-/postconditions auf Methoden. Jedoch gibt es die Möglichkeit über *Assertions* Bedingungen im Code zu definieren, welche zu einem Programmabbruch führen. Darüber hat ein Programmierer die Möglichkeit conditions zu formulieren. Diese können jedoch über weite Teile des Quellcodes verteilt sein und es ist für einen anderen Programmierer nicht sofort ersichtlich welche pre-/postconditions daraus abgeleitet werden können. Im speziellen Fall von Java ist es zudem so, dass die Ausführung von Assertions defaultmässig nicht aktiviert ist. Daher ist der Nutzen solcher Assertions leider nur gering. Da diese Sprachen jedoch sehr weit verbreitet sind haben sich zahlreiche Frameworks entwickelt, welche es erlauben zusätzliche Bedingungen an Methodenaufrufe zu knüpfen. Diese sind jedoch nicht im Compiler integriert und haben sich daher auch nicht weit verbreitet.

5.2 C# (.NET Framework)

Für die Common Language Runtime (CLR) von Microsoft gilt eigentlich dasselbe wie für die Sprachen Java und C(++). Es muss jedoch erwähnt werden, dass von Microsoft Research Tools entwickelt wurden, die in dieser Form in keiner der obengennanten Sprachen umgesetzt wurden. Das Framework *Code Contracts* ermöglicht es dem Programmierer über statische Methodenaufrufe conditions(oder eben contracts) zu formulieren (ähnlich wie Assertions). Danach ist es dem Framework möglich, daraus sowohl Laufzeitchecks als auch Dokumentationsdateien zu generieren. Viel interessanter ist jedoch die Möglichkeit einen statischen Checker zu benutzen, welcher Contracts bereits zur compile-time überprüfen kann [6]. Dies geht viel weiter als unsere Implementirung für IML, ist jedoch nicht direkt in der Sprache enthalten.

5.3 Eiffel

Die Sprache Eiffel enthält sehr ähnliche Implementierung wie die von uns für IML gewählte. Durch die objektorientierte Natur von Eiffel unterstützt sie zusätzlich Klasseninvarianten, wie auch die Vererbung von conditions. Den Zugriff auf pre-execution State von Variablen wir über das Keyword *old* ermöglicht.

5.4 D

Die Sprache D[7], welche C++ weiterentwickeln soll, ermöglicht es sowohl Klassen-Invarianten, als auch conditions auf der Ebene eines Statements zu definieren. Da auch diese Sprache objektorientiert ist, wurden Klasseninvarianten implementiert. Es ist jedoch nicht möglich innerhalb der Postcondition auf den Zustand von Variablen vor der Ausführung des bodys zuzugreifen.

```
in
{
    assert(x >= 0);
}
out (result)
{
    assert((result * result) <= x && (result+1) * (result+1) >= x);
}
body
{
    ... code ...
}
```

Listing 12: Beispiel in D

5.5 Ada

Ada erlaubt die Definition von pre-postconditions auf Ebene von Prozeduren/Funktionen und Subtypen seit der Version 2012. Subtypen sind dabei bereits eine Art von Einschränkung, welche vom Compiler als Typ aufgefasst werden und daher bereits bei der Kompilationszeit erkannt werden. Seit Ada 2012 ist es jedoch auch möglich, solche Subtypen mit dynamischen und statischen Prädikaten zu versehen. Dabei werden die dynamischen überprüft, während die statischen viele Überprüfungen bereits zur Kompilationszeit durchführen können. Auch Ada erlaubt den Zugriff auf Zustand vor der Ausführung der Prozedur.

```
procedure Update_Person (P : in out Person)
   with Post => P.Sex = P.Sex'Old
                and P.Birth_Date = P.Birth_Date'Old;

function Inc(X: Integer) return Integer
   with Pre  => X /= Integer'Last,
        Post => Inc'Result = X'Old+1;
```

Listing 13: Beispiel in Ada 2012

6 Zusammenarbeit

Das Programm zur Generierung der Parsetabelle ohne Erweiterungen wurde uns freundlicherweise vom Team 08 (Walther/Martin) zur Verfügung gestellt.

7 Arbeitsaufteilung

Der Grossteil des Codes sowie der Dokumentation wurde gemeinsam erarbeitet. Jedoch wurde der Initialisierungscheck grösstenteils von Matthias Brun implementiert, während sich Florian Lüscher mit der Code Generierung beschäftigte.

8 Erklärung

Hiermit bestätigen wir die Korrektheit der Angaben und das selbständige Erarbeiten dieser Arbeit sowie des Codes, sofern nichts anders deklariert wurde.

Florian Lüscher Matthias Brun

Literatur

[1] Microsoft Research. (2012) Code Contracts. [Online]. Available: http://research.microsoft.com/en-us/projects/contracts/
[2] Java Community Process. (2006) JSR 305: Annotations for Software Defect Detection. [Online]. Available: http://jcp.org/en/jsr/detail?id=305
[3] E. Lederer, “Slides IML V4.”
[4] Oracle Inc. (2012) The Java Virtual Machine Specification. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/
[5] OW2. (2012) ASM Bytecode Manipulation Framework. [Online]. Available: http://asm.ow2.org/
[6] MSDN Magazine. (2011, Aug.) Static Code Analysis and Code Contracts. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/hh335064.aspx
[7] (2012, Nov.) D Programming Language. [Online]. Available: http://dlang.org

A Vollständige Grammatik

```
program     ::= PROGRAM IDENT [GLOBAL cpsDecl] blockCmd

decl        ::= storeDecl
              | funDecl
              | procDecl

storeDecl   ::= [CHANGEMODE] IDENT COLON TYPE

funDecl     ::= FUN IDENT paramList
                RETURNS storeDecl
                [GLOBAL globImpList]
                [LOCAL cpsDecl]
                [requires]
                [ensures]
                blockCmd

procDecl    ::= PROC IDENT paramList
                [GLOBAL globImpList]
                [LOCAL cpsDecl]
                [requires]
                [ensures]
                blockCmd

cpdDecl     ::= decl {SEMICOLON decl}

paramList   ::= LPAREN [param {COMMA param}] RPAREN
param       ::= [FLOWMODE] [MECHMODE] storeDecl
globImpList ::= globImp {COMMA globImp}
globImp     ::= [FLOWMODE] [CHANGEMODE] IDENT

cmd         ::= SKIP
              | expr BECOMES expr
              | IF LPAREN expr RPAREN blockCmd ELSE blockCmd
              | WHILE LPAREN expr RPAREN blockCmd
              | CALL IDENT exprList [INIT globInitList]
              | QUESTMARK expr
              | EXCLAMARK expr

blockCmd    ::= LBRACE cmd {SEMICOLON cmd} RBRACE
globInitList::= IDENT {COMMA IDENT}

expr        ::= term1 {BOOLOPR term1}
term1       ::= term2 [RELOPR term2]
term2       ::= term3 {ADDOPR term3}
term3       ::= factor {MULTOPR factor}

factor      ::= LITERAL
              | IDENT
              | IDENT INIT
              | IDENT exprList
              | monadicOpr factor
              | LPAREN expr RPAREN

exprList    ::= LPAREN [expr {COMMA expr}] RPAREN
monadicOpr  ::= NOT | ADDOPR

requires        ::= REQUIRES conditionList
ensures         ::= ENSURES  conditionList

conditionList   ::= LBRACKET [condition {COMMA condition}] RBRACKET
```

```
condition       ::= expr [COLON IDENT]
```

B Funktionierende Beispiele

```
proc divide(in copy m:int32, in copy n:int32, out ref q:int32, out ref r:int32)
requires [n > 0]
ensures [r >= 0]
{
    q init := 0;
    r init := m;

    while (r >= n) {
        q := q + 1;
        r := r - n
    }
}
```

Listing 14: Pre-/postconditions in Prozeduren

```
proc incrementPositive(inout var a: int32)
requires [ a > 0 : ParameterShouldBePositive]
ensures  [ a = old(a + 1), a > 0 : ResultShouldBePositive]
{
        a := a + 1
}
```

Listing 15: Pre-/postconditions in Prozeduren mit old Funktion

```
fun gcd(in copy var a:int32, in copy var b:int32) returns var r:int32
requires [a > 0, b > 0]
ensures [r > 0]
{
    r init := 0;

    if (a = 0) {
        r := b
    } else {
        while (b /= 0) {
            if (a > b) {
                a := a - b
            } else {
                b := b - a
            }
        };
        r := a
    }
}
```

Listing 16: Pre-/postconditions in Funktionen mit mehreren Conditions

```
fun multiply(in copy m:int32, in copy n:int32) returns var r:int32
local var i:int32
requires [n > 0]
{
    i init := 1;
    r init := m;

    while (i < n) {
        i := i + 1;
```

```
        r := r + m
    }
}
```

Listing 17: Pre-/postconditions in Funktionen

```
fun sqrt(s:int32)
returns r: int32
local   var a: int32
requires [s >= 0 : pre]
ensures [ r * r <= s, r <= s]
{
    a init := 1;
    while (a * a <= s) {
        a := a + 1
    };
    r init := a - 1
}
```

Listing 18: Berechnen einer ganzzahligen Quadratwurzel. Verwendung mehrerer Conditions

```
proc swapAndIncrement(inout var a: int32, inout var b: int32)
local
    var tmp: int32
requires []
ensures  [ a = old(b + 1), b = old(a + 1) ]
{
    tmp init := a + 1;
    a := b + 1;
    b := tmp
}
```

Listing 19: Evaluieren einer Expression im preexecution State

```
fun multiply(in copy m:int32, in copy n:int32) returns var r:int32
local var i:int32
requires [isPositive(n): positive]
{
    i init := 1;
    r init := m;

    while (i < n) {
        i := i + 1;
        r := r + m
    }
}

fun isPositive(in copy n:int32) returns r:bool
{
    r init := n > 0
}
```

Listing 20: Pre-/postconditions mit Funktion in der Condition List und Label

```
proc twoTimes(in copy m:int32, out ref r:int32)
ensures [old(m) = (r - m)]
{
    r init := m + m
}
```

Listing 21: Pre-/postconditions mit old Funktion und Zugriff auf out Parameter

```
fun fib(i: int32)
    returns r: int32
    requires [ isPos(i): InputHasToBePositive ]
    ensures [ r >= old(i-1) ]
{
    if(i = 0) {
         init := 0
    } else {
         if(i = 1){
             r init := 1
         } else {
             r init := fib(i - 1) + fib(i - 2)
         }
    }
}
```

Listing 22: Pre-/postconditions mit old Funktion und Zugriff auf Return Wert

C Nicht funktionierende Beispiele

```
proc divide(in copy m:int32, in copy n:int32, out ref q:int32, out ref r:int32)
requires [n := 0]
ensures [r := 12 + 1]
{
    q init := 0;
    r init := m;

    while (r >= n) {
        q := q + 1;
        r := r - n
    }
}
```

Listing 23: Fehler: Wert einer Variable in der pre-/postcondition ändern

```
fun gcd(in copy a:int32, in copy b:int32) returns r:int32
requires [a > 0, b > 0]
ensures [old(x) > 0]
{
    r init := 0;

    if (a = 0) {
        r := b
    } else {
        while (b /= 0) {
            if (a > b) {
                a := a - b
            } else {
                b := b - a
            }
        };
        r := a
    }
}
```

Listing 24: Fehler: Zugriff auf nicht vorhandene Variable in der old Funktion

```
fun multiply(in copy m:int32, in copy n:int32) returns r:int32
requires [n + 0]
```

```
{
    i init := 0;
    r init := m;

    while (i < n) {
        i := i + 1;
        r := r + m
    }
}
```

Listing 25: Fehler: Eine nicht Boolsche Expression in der Condition List

```
fun multiply(in copy m:int32, in copy n:int32) returns r:int32
requires [notBool(n): positive]
{
    i init := 0;
    r init := m;

    while (i < n) {
        i := i + 1;
        r := r + m
    }
}

fun notBool(in copy n:int32) returns r:int32
{
    r init := 10
}
```

Listing 26: Fehler: Eine Funktion in der Condition List welche keinen boolschen Wert zurückliefert

```
proc twoTimes(in copy m:int32, out ref r:int32)
ensures [g > 10]
{
    r init := m + m
};

var g:int32
```

Listing 27: Fehler: Beispiel mit Zugriff auf globale Variable welche nicht in der globalImportList importiert wurde

```
proc twoTimes(in copy m:int32, out ref r:int32)
requires [r > 10]
{
    r init := m + m
}
```

Listing 28: Fehler: Zugriff auf out Parameter in precondition.

```
proc twoTimes(in copy m:int32, out ref r:int32)
requires [m > 10: cond, m > 11: cond2]
ensures [ m > 20: cond ]
{
    r init := m + m
}
```

Listing 29: Fehler: Gleiche Labels in Conditions derselben Routine.

```
proc swapAndIncrement(inout var a: int32, inout var b: int32)
local
```

```
    var tmp: int32
requires [ ]
ensures  [ a = old(old(b) + 1), b = old(old(a) + 1) ]
{
    tmp init := a + 1;
    a := b + 1;
    b := tmp
}
```

Listing 30: Fehler: Auf die Funktion old darf im preexecution State nicht zugegriffen werden.

```
fun sqrt(s:int32)
returns r: int32
local
    var a: int32
requires [s >= 0 : pre]
ensures [ r * r <= s, r <= s, old(r) <= s]
{
    a init := 1;
    while (a * a <= s) {
        a := a + 1
    };
    r init := a - 1
}
```

Listing 31: Fehler: Zugriff auf uninitialisierter Wert in preexecution State mittels old

```
fun sqrt(s:int32)
returns r: int32
local
    var a: int32
requires [old(s) >= 0 : pre]
ensures [ r * r <= s, r <= s]
{
    a init := 1;
    while (a * a <= s) {
        a := a + 1
    };
    r init := a - 1
}
```

Listing 32: Fehler: Zugriff auf Funktion old in precondition

```
program rec
global
    fun inc(i: int32)
    returns r: int32
    ensures [r = old(inc(i))]
    {
        r init := i + 1
    };
    a: int32
{
    ? a init;
    ! inc(a)
}
```

Listing 33: Fehler: Rekursiver Aufruf in condition.

```
program rec
global
```

```
        fun inc(i: int32) returns r: int32
        ensures [r - 1 = i]
        {
                r init := i + 1
        };
        fun old(a: int32) returns r: int32
        {
                r init := a - 1
        }
{       ! inc(12)    }
```

Listing 34: Fehler: Namenskonflikt mit reservierter Funktion old.