IT University of Copenhagen

Master Thesis Project Report

Specification Matching for JML-annotated Java

Author:
Martin Hansen
311080-1113

Supervisors:
Joseph Kiniry
Josu Martinez

November 2010 — April 2011

Abstract

In 1997 Zaremski and Wing introduced definitions of specification matching of software components and implemented a system according to these definitions for components written in the ML programming language with specifications in the Larch/ML specification language [ZW97].

In this thesis we present a system for specification matching of JML-annotated Java methods. Our system is able to compare the JML specifications of two Java methods and decide if they match according to the definitions introduced by Zaremski and Wing in 1997 [ZW97].

The system is realized by translating JML-annotated Java into a many-sorted first-order logic defined by version 2 of the SMT-LIB standard. We define an object logic and rules for forming type predicates for JML-annotated Java.

The system is limited partly by the limitations of the underlying SMT-LIBv2 solver and partly by approximations in our object logic and rules for forming types predicates.

Contents

Chapter 1

Introduction

In his “No Silver Bullet”-paper from 1987, Brooks argues that complexity is at the essence of software engineering. Software engineering deals with the formulation of complex conceptual structures. In the natural sciences it is common to construct a simplified model of a complex phenomenon while still capturing the essence of the phenomenon in the model. But in software engineering it is often not possible to abstract away complexity without missing essential features of the domain in question. Software projects that fail to cope with complexity are prone to run over-time and over-budget, only to deliver unmaintainable and buggy solutions. Though it might not be the magic silver bullet, Brooks considers software reuse to be a promising approach to cope with the inherent complexity of software [Bro87].

This realization goes further back than 1987. Indeed the term Software Engineering is coined in 1967 to imply “the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering” [NRB69]. In response to this, at the NATO Software Engineering Conference in 1968, McIlroy presents his vision of Mass Produced Software Components [MBNR69]. McIlroy proposes that software production should be based on libraries of reusable components. Much in the same way as complex electronic components are assembled from a collection of simpler components in electronic engineering.

Drawing on this vision, over the last decade the field of study known as Component Based Software Engineering (CBSE) has emerged as an approach to constructing software that encourages software reuse. Despite the promises of this discipline to handle complexity, CBSE is not in widespread use [Hem05]. There are many reasons for the slow adoption, one of which is the problem of component retrieval: How does a software engineer find the components that match a given set of requirements or behavior?

Several authors have proposed specification matching as a formal approach to help overcome the challenge of retrieval [PP02, JC95, FS97, ZW97]. Specification matching is a way to determine if two software components are related by comparing their formal specifications. Thus, formal specifications are used to query a software library and extract components with the desired behavior.

The most comprehensive set of definitions on specification matching is given by Zaremski and Wing in 1997 [ZW97]. These authors also implement a system for specification matching of software components based on their definitions on specification matching. The software components considered by these authors are written in the programming language ML and specified in the Larch/ML specification language.

Despite it's potential within software reuse, a system like the one implemented by Zaremski and Wing has not been realized for a mainstream programming language such as C or Java.

The goal of this thesis is to implement a system for specification matching of Java methods. We use the Java Modeling Language (JML) as the specification language. The system must be able to compare the JML specifications of two Java methods and decide if they match according to the definitions introduced by Zaremski and Wing in 1997 [ZW97].

The outline of the thesis is as follows. In chapter 2 we introduce the necessary theories and technologies. The reader is assumed to be familiar with Java and first-order logic. Following the description of the methodology in chapter 3, we describe in chapter 4 how we implemented the system. In chapter 5 we describe the system tests we have performed. We close with related work in chapter 6 and summary and directions for future work in chapter 7.

Chapter 2

Background

2.1 JML

The Java Modeling Language (JML) is a formal specification language especially tailored for Java [LBR99, LC05]. Many aspects of Java can be formally specified using JML, including the interface and behavior of methods using pre- and postconditions. The syntax and sematics of pre- and postconditions in JML are inspired by the software development method Design by Contract (DbC). In DbC the pre- and postcondition for a method is known as it's *contract*. The contract is between the method and its client and states the rights and obligations of the method and its client when the method is called: The caller has the obligation to fulfill the methods precondition and the right to rely on the postcondition, in turn the method can rely on the caller to fulfill the precondition, but must guarantee the postcondition.

Being especially tailored for Java, JML defines pre- and postconditions directly in Java code. Consequently JML defines 1) an expression language which is an extension to the Java expression language and 2) a set of statements using the expressions. We will refer to such expressions and expression statements as JML expressions and JML statements respectively.

In JML a precondition can be expressed as a JML statement using the requires keyword

```
requires <JML expression>;
```

where the JML expression has (Java) type boolean. The statement is called a requires clause. Analogously, a postcondition is expressed as a JML statement using the ensures keyword

```
ensures <JML expression>;
```

The statement is called an ensures clause. We can use one or more of such JML statements to form a JML specification for a method directly in the Java code, as in the declaration of method `m` below

```
//@ requires a > 0 && b > 0;
//@ ensures \result == a || \result == b;
int m(int a, int b)
```

The JML statements are written inside Java comments beginning with an at-sign (`@`) or inside a multiline comment of the form /*@ @*/ possibly with @ allowed at the beginning of each line. In the example above the JML Expression in the requires clause is just a Java expression. In the ensures clause the result keyword stands for the return value of the method. The meaning of the specification composed of these two JML statements is that if the method `m` is called with both of its arguments greater than zero, then its return value will be equal to one of these arguments. In general the meaning of a JML method specification is that the precondition implies the postcondition.

Two ensures or requires clauses such as

```
//@ requires a > 0;
//@ requires b > 0;
```

are equivalent to

```
//@ requires a > 0 && b > 0;
```

If a JML method specification does not include a requires clause, by default the precondition is true. Likewise, if a JML method specification does not include an ensures clause, by default the postcondition is true.

Since runtime evaluation of the JML expression used in assertions such as these should not itself influence the behavior of any part of a program, these JML expressions are not allowed to have side effects.

By extending the Java expression language rather than defining its own specialized assertion language like OCL, it should be easy for Java programmers to pick up this specification language. In fact, JML is the most widely used specification language for Java, and JML is used many different tools [BCC+03]. However, since the Java expression language was not designed for formal specification it lacks some of the expressiveness that makes specialized assertion languages convenient. To make it suitable as a formal specification language, in addition to the result keyword introduced above, JML expressions extends the Java expression language by including constructs such as universal and existential quantifiers.

Despite the goal of being a simple language JML has evolved to include a lot of features some of which are not supported by all JML tools. To account for this, six different language levels are defined with level 0 being the most basic constituting “the heart of JML” [LPC+06] . The features in language level 0 are meant to be supported by all tools. A lot of JML features are excluded from this level. In this project we define a simpler language level as a subset of level 0. In addition to the requires, ensures and result keywords this subset also includes the old-expression. The JML expression `old(expr)` refers to the value of `expr` in the pre-state. In our subset `expr` is only allowed to be a field access. For instance if method `incr` is declared in the same class a field `i` of type `int` the meaning of the specification of `incr`, as given below, is that if `i` is less than 10, then `i` is incremented by one

```
//@ requires i < 10;
//@ ensures i == old(i) + 1;
void incr()
```

Additionally our subset of JML language level 0 includes

- the universal quantifier, `forall`,
- the existential quantifier, `exist`,
- the equivalence operator, `<==>`,
- the inequivalence operator, `<=!=>`,
- the forward implication operator, `==>`, and
- the reverse implication operator, `<==`.

2.2 Specification Matching

In this section we present the definitions of specification match predicates for methods[1] introduced by Zaremski and Wing [ZW97]. A *method specification match predicate* (referred to below simply as *match*) relates the formal specification of one method with the formal specification of another method. A formal method specification, S, consist of a *precondition*, S_{pre}, and a *postcondition*, S_{post}. The pre- and postconditions are expressed in first-order logic. A method specification S has an *interpretation*, S_{pred}, defined as $S_{pred} = S_{pre} \Rightarrow S_{post}$. A match has a name, M, and a symbol $match_M$. Given two methods `s` and `q` with specifications S and Q, respectively, if $match_M(S, Q)$ holds we may say that "`s` matches with `q` under M" or, equivalently, that "`q` is matched by `s` under M". The definitions of matches are divided into to two groups: *pre/post matches* and *predicate matches*.

[1]Zaremski and Wing use the term *function*. Their definition of this term covers constructs like C routines, Ada procedures and ML functions [ZW97]. Thus a Java method is encompassed by this definition of a function. In this section we will use the term method where Zaremski and Wing use the term function.

Generic pre/post match

Definition 1 is the definition of generic pre/post match. In this definition the relations $\Re_1$ and $\Re_2$ are either $\Leftrightarrow$ or $\Rightarrow$. The relation $\Re_1$ may be dropped in a given instantiation of definition 1. $\hat{S}$ is instantiated as either S_{post} or $(S_{pre} \wedge S_{post})$.

Definition 1.

$$match_{pre/post}(S, Q) = (Q_{pre} \Re_1 S_{pre}) \wedge (\hat{S} \Re_2 Q_{post})$$

Table 2.1 provides an overview of the possible instantiations of the definition of generic pre/post match.

Match name	Match symbol	$\Re_1$	$\Re_2$	$\hat{S}$
Exact pre/post	$match_{\text{E-pre/post}}$	$\Leftrightarrow$	$\Leftrightarrow$	S_{post}
Plug-in	$match_{\text{plug-in}}$	$\Rightarrow$	$\Rightarrow$	S_{post}
Plug-in post	$match_{\text{plug-in-post}}$	$*$	$\Rightarrow$	S_{post}
Guarded plug-in	$match_{\text{guarded-plug-in}}$	$\Rightarrow$	$\Rightarrow$	$S_{pre} \wedge S_{post}$
Guarded post	$match_{\text{guarded-post}}$	$*$	$\Rightarrow$	$S_{pre} \wedge S_{post}$

Table 2.1: Instantiations of the definition of generic pre/post match (definition 1). An asterisk ($*$) indicate that the relation $\Re_1$ is not instantiated.

Generic predicate match

Definition 2 is the definition of generic predicate match. In this definition $\Re$ is either $\Leftrightarrow$, $\Rightarrow$ or $\Leftarrow$.

Definition 2.

$$match_{pred}(S, Q) = (S_{pred} \Re Q_{pred})$$

Table 2.2 provides an overview of the possible instantiations of the definition of generic predicate match.

To illustrate the definitions we use the JML specifications of nine Java methods, `m0` through `m8`, defined in class `Q` (listing 2.1) and class `S` (listing 2.2).

2.2.1 Pre/post matches

Exact pre/post match

Exact pre/post match is an instance of generic pre/post match. Instantiating both $\Re_1$ and $\Re_2$ to $\Leftrightarrow$ and $\hat{S}$ to S_{post} in the definition of generic pre/post match (definition 1) yield the definition of exact pre/post match (definition 3).

Match name	Match symbol	$\Re$
Exact predicate	$match_{\text{E-pred}}$	$\Leftrightarrow$
Generalized	$match_{\text{gen-pred}}$	$\Rightarrow$
Specialized	$match_{\text{spcl-pred}}$	$\Leftarrow$

Table 2.2: Instantiations of the definition of generic predicate match (definition 2).

Listing 2.1: A JML-annotated Java class

```
class Q
{
    boolean a;
    boolean b;
    boolean c;

    //@ requires a & b;
    //@ ensures  b & c;
    void m0() {}
}
```

Definition 3.

$$match_{\textit{E-pre/post}}(S, Q) = (Q_{pre} \Leftrightarrow S_{pre}) \wedge (S_{post} \Leftrightarrow Q_{post})$$

Method `m0` (listing 2.1) is matched by method `m1` (listing 2.2) under exact pre/-post match. `m0` is not matched by any of the other methods of class `S` under exact pre/post match.

Plug-in match

Plug-in match is an instance of generic pre/post match. Instantiating both $\Re_1$ and $\Re_2$ to $\Rightarrow$ and $\hat{S}$ to S_{post} in the definition of generic pre/post match (definition 1) yield the definition of plug-in match (definition 4).

Definition 4.

$$match_{\textit{plug-in}}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge (S_{post} \Rightarrow Q_{post})$$

Method `m0` is matched by each of the methods `m1` and `m2` (listing 2.2) under plug-in match. `m0` is not matched by any of the other methods of class `S` under plug-in match.

Listing 2.2: Another JML-annotated Java class

```
class S
{
    boolean a;
    boolean b;
    boolean c;

    //@ requires b & a;
    //@ ensures  c & b;
    void m1() {}

    //@ requires a;
    //@ ensures  c & b;
    void m2() {}

    //@ requires a & c;
    //@ ensures  c & b;
    void m3() {}

    //@ requires b;
    //@ ensures  c;
    void m4() {}

    //@ requires c;
    //@ ensures  b;
    void m5() {}

    //@ ensures !(a & b) | (b & c);
    void m6() {}

    //@ ensures !(a & b | c) | (b & c);
    void m7() {}

    //@ ensures !(a & b) | b;
    void m8() {}
}
```

Plug-in postmatch

Plug-in postmatch is an instance of generic pre/post match. Instantiating $\Re_2$ to $\Rightarrow$ and $\hat{S}$ to S_{post} and dropping $\Re_1$ in the definition of generic pre/post match (definition 1) yield the definition of plug-in postmatch (definition 5).

Definition 5.

$$match_{\textit{plug-in-post}}(S, Q) = (S_{post} \Rightarrow Q_{post})$$

Method `m0` (listing 2.1) is matched by each of methods `m1`, `m2` and `m3` (listing 2.2) under plug-in postmatch. `m0` is not matched by any of the other methods of class `S` under plug-in postmatch.

Guarded plug-in match

Guarded plug-in match is an instance of generic pre/post match. Instantiating both $\Re_1$ and $\Re_2$ to $\Rightarrow$ and $\hat{S}$ to $S_{pre} \wedge S_{post}$ in the definition of generic pre/post match (definition 1) yield the definition of guarded plug-in match (definition 6).

Definition 6.

$$match_{\textit{guarded-plug-in}}(S, Q) = (Q_{pre} \Rightarrow S_{pre}) \wedge ((S_{pre} \wedge S_{post}) \Rightarrow Q_{post})$$

Method `m0` (listing 2.1) is matched by each of the methods `m1`, `m2` and `m4` (listing 2.2) under guarded plug-in match. `m0` is not matched by any of the other methods of class `S` under guarded plug-in match.

Guarded postmatch

Guarded post match is an instance of generic pre/post match. Instantiating $\Re_2$ to $\Rightarrow$ and $\hat{S}$ to $S_{pre} \wedge S_{post}$ and dropping $\Re_1$ in the definition of generic pre/post match (definition 1) yield the definition of guarded postmatch (definition 7).

Definition 7.

$$match_{\textit{guarded-post}}(S, Q) = (S_{pre} \wedge S_{post}) \Rightarrow Q_{post}$$

Method `m0` (listing 2.1) is matched by each of the methods `m1`, `m2`, `m3`, `m4` and `m5` (listing 2.2) under guarded postmatch. `m0` is not matched by any of the other methods of class `S` under guarded postmatch.

2.2.2 Predicate matches

Exact predicate match

Exact predicate match is an instance of generic predicate match. Instantiating $\Re$ to $\Leftrightarrow$ in the definition of generic predicate match (definition 2) yield the definition of exact predicate match (definition 8).

Definition 8.

$$match_{E\text{-}pred}(S, Q) = S_{pred} \Leftrightarrow Q_{pred}$$

Method `m0` (listing 2.1) is matched by each of the methods `m1` and `m6` (listing 2.2) under exact predicate match. `m0` is not matched by any of the other methods of class `S` under exact predicate match.

Generalized match

Generalized match is an instance of generic predicate match. Instantiating $\Re$ to $\Rightarrow$ in the definition of generic predicate match (definition 2) yield the definition of generalized match (definition 9).

Definition 9.

$$match_{gen\text{-}pred}(S, Q) = S_{pred} \Rightarrow Q_{pred}$$

Method `m0` (listing 2.1) is matched by each of the methods `m1` `m2`, `m4`, `m6` and `m7` (listing 2.2) under generalized match. `m0` is not matched by any of the other methods of class `S` under generalized match.

Specialized match

Specialized match is an instance of generic predicate match. Instantiating $\Re$ to $\Leftarrow$ in the definition of generic predicate match (definition 2) yield the definition of specialized match (definition 10).

Definition 10.

$$match_{spcl\text{-}pred}(S, Q) = Q_{pred} \Rightarrow S_{pred}$$

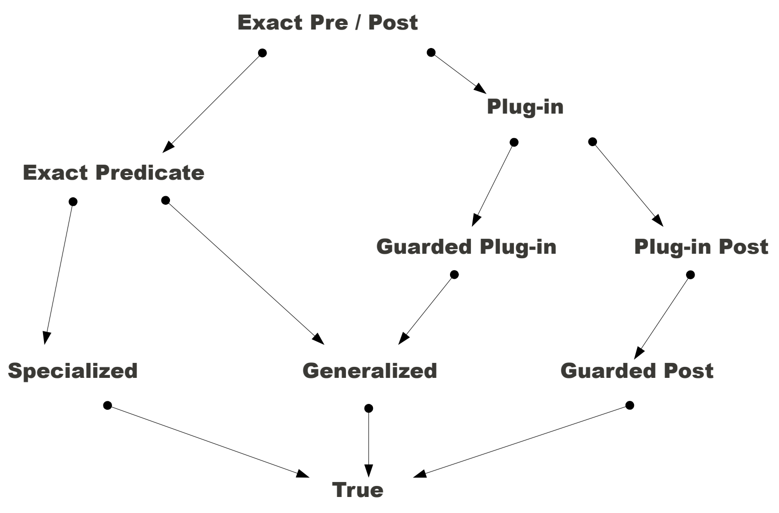
Method `m0` (listing 2.1) is matched by method `m1`, `m6` and `m8` (listing 2.2) under specialized match. `m0` is not matched by any of the other methods of class `S` under specialized match.

2.2.3 Match relations

The matches are related as shown in the lattice in figure 2.1. In this lattice an arrow from a match M_1 to a match M_2 indicates that $M_1(S, Q) \Rightarrow M_2(S, Q)$. (This is paraphrased as "M_1 is stronger than $M2$" or "$M2$ is more relaxed than $M1$".)

In table 2.3 we have summarized which of the methods of class `S` (listing 2.2) matches with `m0` (listing 2.1) from class `Q` under each of the matches presented above. We can validate the lattice in figure 2.1 using this summary. For instance,

Figure 2.1: Lattice of matches



from this summary we can see that each of the methods declared in class `S` that matches with `m0` (declared in class `Q`) under plug-in match also matches with `m0` under plug-in post, guarded plug-in, guarded post and generalized match. This is in agreement with the lattice in figure 2.1 where plug-in post, guarded plug-in, guarded post and generalized match have the upper bound plug-in match.

2.3 SMT-LIB

If a pure boolean formula, φ, such as

$A \wedge \neg B$

has an assignment of its variables (A and B) such that the formula is true it is said to be satisfiable. If the formula is satisfied for any assignment of its variables it is said to be valid. If a formula is not satisfiable it implies that its negation is valid. E.g., since $\neg(C \Leftrightarrow C)$ is not satisfiable its negation $\neg\neg(C \Leftrightarrow C)$ (which is equivalent to $C \Leftrightarrow C$) is valid. These problems are determined efficiently by SAT solvers.

If we want to check the satisfiability of formulas such as

$i < f(a) \wedge a[i] = g(x + 99.9)$

that include integers, uninterpreted functions and arrays, we cannot rely on SAT solvers. To cope with such formulas SAT solvers must be extended with several

	`m1`	`m2`	`m3`	`m4`	`m5`	`m6`	`m7`	`m8`
Exact pre/post	x							
Plug-in	x	x						
Plug-in post	x	x	x					
Guarded plug-in	x	x		x				
Guarded post	x	x	x	x	x			
Exact predicate	x					x		
Generalized	x	x		x		x	x	
Specialized	x					x		x

Table 2.3: Summary of which methods from class `S` matches with method `m0` under different matches. An x indicates that `m0` is matched by the method in the column descriptor under the match given in the row descriptor.

concepts. First of all an underlying many-sorted logic (as opposed to a single sorted boolean logic) is needed, since besides the boolean sort, sorts such as integer and array may be needed as in the formula above. Also, theories that describe how to interpret symbols such as '$<$' and '$=$' are needed. For instance the meaning of '$=$' in $a[i] = g(x + 99.9)$ depends on the (common) sort of $a[i]$ and $g(x+99.9)$. Such extensions to, or generalizations of, SAT solvers constitute the Satisfiability Modulus Theories (SMT) solvers. Within SMT the satisfiability of a formula is determined with respect to, or modulo, some theories, e.g. the theories of integers, arrays and uninterpreted functions.

The SMT-LIB initiative has standardized an underlying multisorted first-order logic; a language for defining theories and logics; and a command language for interacting with SMT solvers. Below we describe parts of version 2 of the SMT-LIB standard (SMT-LIBv2) [BST10, Cok11a].

In addition to the Core boolean theory, the theories defined in the SMT-LIB standard include a theory of integers, a theory of real numbers and a theory of arrays all of which implicitly include the boolean theory. Each theory may declare some sorts, and some functions with those sorts as parameter and return sorts. The Core theory, for example, defines the sort *Bool* and among other functions uses that sort in $not : Bool \mapsto Bool$ and in the implies operator `=>` : $Bool \times Bool \mapsto Bool$. While the SMT-LIB standard defines a language for specifying theories and a catalogue of such specifications, it is totally up to the solver to "implement" these specifications.

The theory Reals_Ints define the sorts *Int* an *Real*. The sort *Int* represent the integers, $\mathbb{Z}$. The sort *Real* represent the reals, $\mathbb{R}$. The operators $+$, $-$, $*$, $/$, $<$, $>$, $\leq$ and $\geq$ are defined for both sorts, e.g. $+ : Int \times Int \mapsto Int$ and $+ : Real \times Real \mapsto Real$. The theory ArraysEx defines a sort Array that is parameterized with two parameters. The first parameter is the index sort the second parameter is the value sort. The theory includes the function select that selects values from the array

$select : ArraySort\ IndexSort \mapsto ValueSort$

A logic in the SMT-LIB standard is specified informally by one or more theories and a stipulation of which expressions are allowed. Logics are specified in the SMT-LIB standard completely informally, thus it is completely up to any given solver wishing to comply with the standard to "implement" these specifications.

The AUFLIRA logic includes the theories ArraysEx and Reals_Ints. Only linear arithmetic is allowed in this logic. Array sort may only have index sort Int and value sort either *Real* or Array sort with index sort Int as value sort *Real*. The AUFNIRA logic includes the theories ArraysEx and Reals_Ints. There are no restrictions on the sort parameters of sort Array in this logic.

Finally the SMT-LIB initiative standardizes a language for interaction with SMT solvers. The language consists of commands. A sequence of commands is called a script. In the example script in listing 2.3 below, the commands we use in this project are seen in action. It does one out of several possible encodings of the formula above and asks the solver to decide satisfiability.

Listing 2.3: An example of an smt script.

```
(set-logic AUFNIRA)
(declare-sort MySort 0)
(declare-fun i () Int)
(declare-fun f ((Array Int MySort)) Int)
(declare-fun a () (Array Int MySort))
(declare-fun g (Real) MySort)
(declare-fun x () Real)

(define-fun formula () Bool
  (and
    (< i (f a))
    (= (select a i) (g (+ x 99.9)))
  )
)

(assert formula)
(check-sat)
```

The first command, `(set-logic AUFNIRA)`, instructs the solver to use the AUFNIRA logic. The next command `(declare-sort MySort 0)` declares the sort $MySort$ with arity 0. Sorts can have arity greater than 0. A list sort, for instance, declared as `(declare-sort List 1)` could be instantiated as `(List MySort)` — the list of *MySort*'s. However, *MySort* has arity 0 and is thus a sort constant. The next couple of commands are all instantiations of the function declaration command. They encode the constants and functions of the formula above. The next command is a function definition command that encodes the entire formula as a boolean constant. Finally the *satisfiability* of the formula is determined with the `assert` and `check-sat` commands. If the solver responds with `sat` the formula is satisfied. If the solver responds with `unsat` the formula is not satisfied and we can conclude that the negation of the formula is valid. The solver may also respond with `unknown` if it cannot decide satisfiability for some reason.

Note that an expression such as `(f a)` is correct not only because it is syntactically correct but also because it is well-sorted, i.e. `a` is of the sort that `f` expects as its argument.

In smt scripts a function cannot be overloaded, i.e., we are not allowed to add the function declaration

```
(declare-fun g (Int) MySort)
```

to the script in listing 2.3.

2.3.1 Z3

In this project we use the Z3 solver [DMB08]. We do not consider any of its internals but simply expect it to follow version 2 of the SMT-LIB standard. Z3 is an efficient SMT solver. It won several first and second places at the SMT-COMP'07 competion for SMT solvers. Z3 includes some but not complete support for linear arithmetic.

2.3.2 Notation conventions

Instead of the lisp syntax we will use a notation like the following. Instead of

```
(declare-fun subtypeOf (Class Class) Bool)
```

we write $subtypeOf : Class \times Class \mapsto Bool$. Instead of

```
(declare-fun Object () Class)
```

we write $Object : Class$. Instead of

```
(define-fun spec () Bool (=> pre post))
```

we write $spec : Bool = pre \Rightarrow post$. Instead of

```
(assert (and spec (= i j)))
```

we write $spec \wedge (i \Leftrightarrow j)$. if `i` and `j` have sort `Bool`, otherwise $spec \wedge (i = j)$.

2.4 OpenJML

OpenJDK[2] is an open source version of the Java toolkit that includes the Java compiler javac. Using OpenJDK as a starting point it is possible to alter the javac compiler (although this is complicated, for instance because of the fact that the lexer and parser are hand-written rather than generated, and because of the lack of interfaces for AST nodes and visitor classes [Cok11b]. For these reasons some projects working on extensions to the Java language refrain from modifying javac [Gar10]).

The OpenJML tool set extends OpenJDK by adding support for JML. This includes the ability to parse and type check JML annotated Java code and produce internal ASTs composed by a superset of the nodes defined in OpenJDK.

For instance, in addition to the AST node class JCBinary which are included in OpenJDK, OpenJML also defines class JmlBinary. The JML expression

```
a ==> b
```

is represented by such a JmlBinary node. We can process the tree this node belongs to using the visitor pattern.

In OpenJML (by virtue of its OpenJDK heritage) this is done by extending class `com.sun.tools.javac.tree.JCTree.Visitor`, i.e., all AST node classes have a method with the signature `accept(Visitor)`. Thus a visitor class with the purpose of emitting SMT-LIB commands could be implemented by overriding method `accept(JmlBinary)` like this

```
public void accept(final JmlBinary that)
{
    if (that.op == JmlToken.IMPLIES)
    {
        emit("=> ");
        that.lhs.accept(this);
        emit(" ");
        that.lhs.accept(this);
        emit(")");
    }

    (...more 'if cases' for other operators...)
}
```

Actually we do not emit plain text. Instead we construct a new AST using the jSMTLIB library [Cok11a].

Despite complications the OpenJML project has succeeded in plugging rather

[2]www.openjdk.org

cleanly into OpenJDK modifying relatively few source files and being able to merge with new releases of OpenJDK with little effort. In the past JML tools were based on hand-crafted compilers. This led to an excessive maintenance effort as Java evolved [Cok11b].

Chapter 3

Methodology

As stated in the introduction the goal of this project is to develop a system that is able to compare the JML specifications of two Java methods and decide if they match according to the definitions introduced by Zaremski and Wing in 1997 [ZW97]. We have presented the definitions introduced by Zaremski and Wing in 1997 in section 2.2. For instance, if we consider `m3` (declared in listing 2.2) and `m0` (declared in listing 2.1) we may ask the question:

$$\text{Does } \texttt{m3} \text{ match with method } \texttt{m0} \text{ under plug-in postmatch?} \tag{3.1}$$

This amounts to deciding if the expression in the ensures clause of `m3` implies the expression in the ensures clause of `m0`, that is if (`c & b`) ⇒ (`b & c`). But how can we develop a system that is able to make such a decision?

One solution would be to develop a theorem prover especially for JML expressions. The prover should be able to decide the validity of any formula such as (`c & b`) ⇒ (`b & c`), where (`b & c`) and (`b & c`) are JML expressions. Thus, we would have reached the goal of this project. However, developing such a prover would be a huge undertaking. In fact it would amount to building an entire specialized SMT solver. As mentioned in section 2.3.1 SMT solvers only recently became efficient. Doing the work of developing an SMT solver all over again would be far beyond the scope of a project like this. Instead we choose the compilation alternative and translate JML expressions into one of the many-sorted first-order logics defined by SMT-LIBv2. This will enable us to use any SMT-LIBv2 conforming solver to decide the satisfiability (or, dually, the validity) of a formula such as (`c & b`) ⇒ (`b & c`).

The goal of this project as stated above is thus for a large part reduced to the problem of translating JML expressions into the SMT-LIBv2 command language. Since variables in JML expressions are defined in the surrounding Java context (as described in 2.1) this entails translating large parts of the Java language too.

In other words we need a compiler for JML annotated Java.

One solution would be to develop such a compiler from scratch. This would be a huge undertaking and the result would be subject to the problems mentioned in section 2.4. Instead we plug in to the OpenJML tool set by implementing a Visitor subclass and using jSMTLIB as our target language as outlined in section 2.4. We refer to this as *our compiler*. Even equipped with these tools, however, developing a compiler for all of JML and all of the entailed part of Java would be beyond the scope of a project like this. Therefore we define a limited subset of JML and Java that we will consider.

The subset of JML that we will consider is a subset of JML language Level 0. It includes only the features described in section 2.1. The subset of Java that we will consider consists of class, method and variable declarations and parts of the expression language. In particular we do not consider generics, casting and bitwise operations.

We can divide the encoding of a programming language, such as JML annotated Java, into a logic, such as the many-sorted first order logic defined in SMT-LIB, into three parts: a background theory, an object logic and a type predicate.

In the background theory the specific logic used is defined. This includes a definition of an underlying, possibly many-sorted, first-order logic and could include, if the underlying logic is many-sorted, theories such as a theory on linear arithmetic over integers and reals. In the SMT-LIB standard a logic, such as AUFLIRA or AUFNIRA, constitutes such a background theory. In this project we will use the AUFNIRA logic as our background theory. We choose this logic over the alternatives because it offers the best support for the part of JML and Java we consider in this project. First of all because it does not put restrictions on the array theory which allows for easy encoding of multidimensional arrays of different types as found in Java such as `int[][][]` and `long[][][][][]`. Also, the included theory on non-linear arithmetic over integers and reals provides a foundation for the encoding of linear and non-linear Java expressions such as `2*x + 1` and `(x + 1)*(x + 1)` where `x` may have been declared for instance as `int x;` or `double x;`. Finally, by allowing quantified expressions, the AUFNIRA logic also enables easy encoding of quantified JML expressions. For each SMT-LIBv2 script our compiler produces, this choice of background theory amounts to emitting `(set-logic AUFNIRA)` as the topmost command.

In the object logic the axioms on the programming language are defined in terms of the background theory. For instance this may include an encoding of the programming language's definitions of types and subtyping. (In the next chapter (chapter 4) we define our object logic on JML annotated Java). The concrete result of this definition is a sequence of commands that are always the same and inserted just after the `(set-logic AUFNIRA)` command in every SMT-LIBv2 script our compiler produces.

In the type predicate the axioms specific to the system being reasoned about are defined in terms of the object logic and/or the background theory. In this project the system we reason about are JML specifications of Java methods (with the restrictions mentioned above). Thus a type predicate instance in this project is the encoding of any JML method specification based on the AUFNIRA logic and/or the object logic as introduced above. In concrete terms, for each SMT-LIBv2 script our compiler produces, there is a part which is independent of the specific JML method specification in question. That part is defined by the background theory and the object logic. Secondly there is a part which is dependent on the specific JML method specification. That part is the type predicate.

Sections 4.1 through 4.7 in the next chapter are devoted to the definition of our object logic and the general definition of our type predicate. We do not present these two parts separately. Instead these sections are organized by the different language constructs we consider. E.g. in the section on class declarations we describe both the type predicate a class declaration gives rise to, and the part of the object logic that such a type predicate depends on. In these sections we will also use the term encode. For instance we say that we can encode the conditional expression (`x ? y : z`) using the ite operator of the Core theory like this $ite(x, y, z)$.

Having the desired compiler in place does not completely satisfy the goal of this project. The compiler allows us to express the JML specifications of Java methods in the SMT-LIBv2 command language. But to complete the desired system we need to be able to form match predicates using the type predicate. For instance if we encode `c & b` as $c \wedge b$ and we encode `b & c` as $b \wedge c$ we can encode question 3.1 as

$$c \wedge b \Rightarrow c \wedge b$$

We add this to the SMT-LIBv2 script produced by our compiler. Finally we run the Z3 solver on this script and use the response to decide if the match holds. We explain how we do this in general in section 4.8.

As we describe in chapter 5 our system passes several system tests. However, there are several possibilities for extending our system. For instance we can consider larger subsets of JML and Java. For this reason we sometimes refer to our system as a prototype system.

Chapter 4

Implementation

4.1 Class declarations

The topmost constructs of the Java namespace hierarchy we consider is the class and the interface. This means that references to classes and interfaces in other packages are not allowed in our system. For the purpose of the prototype system we are constructing this limitation is insignificant. One might argue that the 'retrieval for reuse'-application of specification matching would be better illustrated by allowing packages, so that different examples of specification libraries could reside in different packages. However, as we outline in the next section (4.2.1), this would lead to quite long identifiers in the resulting SMT-LIBv2 scripts. While solvers do not care if identifiers are shorter or longer, humans do. During the development of this system we often inspected the SMT-LIBv2 scripts that the compiler produced by hand and we did not want to look at unnecessarily long identifiers. Since at this point the system is still just a prototype system we think that such manual inspection may still be convenient in the further development of the system. Also, at the prototype stage we think it is sufficient to informally regard two classes that formally belong to same (default) package as belonging to different packages, if that makes the potential use in retrieval for reuse clearer.

Concretely we want to be able to express class and interface declarations like

`class C extends D implements I, J, K` †

and

`interface I extends J, K, L` ‡

as type predicates in terms of our object logic. We take the first step towards this by introducing the sort constant

$Class$

in the object logic. The purpose of this sort is to represent the notion of both class and interface reference types in java. As a first approximation in our prototype system we choose to encode class and interface reference types in the same way and introduce only the sort *Class*. An alternative name for this sort could have been *RefType*. However, since, in Java, type variables and array types are reference types along with class and interface types (cf. [Gos00] section 4.3), this would have been imprecise. Another alternative then, could have been *ClassType* or *ClassOrInterfaceType* to distinguish from a sort *Class* that would encode the type `java.lang.Class` which is often used in reflection. However, we have no intention to support reflection so we find it is safe to occupy the name Class for the sort that encodes the notion of class and interface reference types as defined in Java.

As is the case in many other programming languages, types in Java can be related through the subtype relation. We want to encode this relation for class and interface reference types in our object logic. We lay the foundation for this encoding by introducing the uninterpreted function

$\mathit{subtypeOf} : \mathit{Class} \times \mathit{Class} \mapsto \mathit{Bool}$

in the object logic. Since subtyping in Java — as is the case in most programming languages — is defined to be reflexive, transitive and anti-symmetric we qualify the uninterpreted function by adding the assertions

$(\forall c : \mathit{Class})[\mathit{subtypeOf}(c, c)]$

$(\forall c1, c2, c3 : \mathit{Class})[\mathit{subtypeOf}(c1, c2) \wedge \mathit{subtypeOf}(c2, c3) \implies \mathit{subtypeOf}(c1, c3)]$

$(\forall c1, c2 : \mathit{Class})[\mathit{subtypeOf}(c1, c2) \wedge \mathit{subtypeOf}(c2, c1) \implies c1 = c2]$

to the object logic.

In Java subtyping is nominal. A type is only a subtype of *another* type if it is declared to be so. When a class or interface `C` is declared, a new type also named `C` is introduced. This type is a subtype of type `E` ONLY if class `C` is declared a subclass of class `E` as in `class C extends E`; or if class `C` is declared to implement interface `E` as in `class C extends E`; or if interface `C` extends interface `E` as in `interface C extends E`. Thus for every class declaration like † we form the type predicates

$C : \mathit{Class}$

$\mathit{subtypeOf}(C, D) \wedge \mathit{subtypeOf}(C, I) \wedge \mathit{subtypeOf}(C, J) \wedge \mathit{subtypeOf}(C, K)$

where we assume D, I, J and K have been declared previously and are all of sort *Class*.

Likewise for every interface declaration like ‡ we form the type predicates

$I : \mathit{Class}$

$\mathit{subtypeOf}(I, J) \wedge \mathit{subtypeOf}(I, K) \wedge \mathit{subtypeOf}(I, L)$

where we assume J, K and L have been declared previously and are all of sort

$Class$.

In the object logic we also introduce the constant

$Object : Class$

and the assertion

$(\forall c : Class)[subtypeOf(c, Object)]$

to encode the fact that all classes in Java are subclasses of the class `Object` which sits at the top of the class hierarchy.

This concludes the description of our object logic and rules for generating type predicates for class and interface declarations. This part of our object logic is quite minimal consisting only of a sort, $Class$, a constant of this sort, $Object$, and an uninterpreted function, $subtypeOf$, with some assertions. We could have done a more detailed encoding by also taking the field and method descriptions of the declaration bodies into account. This would have involved introducing more functions and sorts in the object logic. To support structural subtyping we would have had to do such a detailed encoding but Java does not support structural subtyping. We want to encode the notion of fields and methods in our object logic regardless of subtyping definitions, so this could be a reason for doing a more detailed encoding. However, a design decision for our object logic, which we will emphasize further in the next section (section 4.2), is to introduce as few new sorts as possible and do the encoding directly in terms of the background logic.

We disregard modifiers all together. On the basis of our current object logic we could include support for the final modifier by letting a class declaration such as

```
final class C
```

give rise to the type predicate

$(\forall D : Class)[\neg subtypeOf(D, C)]$

In our prototype system we have not added support for this or any other class modifiers.

4.2 Field declarations

We consider field declarations of the form

```
t f;
```

where `t` is any Java type and `f` is the name of the field. We disregard all field modifiers.

The type `t` is either an arraytype, another reference type or a primitive type. In this section we consider the latter two. For fields of these types we do not consider initializations, i.e., we treat a field declaration such as `byte b = 127;` simply as `byte b;`. For fields of arraytypes we do consider the field initializers. The next section (section 4.3) is devoted especially to arraytypes.

4.2.1 Names

As mentioned in the previous section (section 4.1) we have not introduced a detailed encoding of classes in the object logic. The type predicates that a class declaration gives rise to do not include anything about the methods or fields of the class. Likewise we do not introduce a function such as $enclosingClass : Field \mapsto Class$ to relate fields to their enclosing classes. We encode a field declaration simply by adding a new constant to the type predicate in the same way as we did for class declarations. However, we base the name of that constant not only on the name of the field but also on the name of the enclosing class. For instance, given a declaration of a field `f` in a class `C` we add a constant with the name _C___f to the type predicate. To accommodate the different ways a field may be accessed in an expression, a field declaration gives rise to several constants of the same sort but with different names: In addition to the constant named _C___f, constants named _C___this_C___f, old_C___f and old_C___this_C___f are added to the type predicate given the declaration of the field `f` in the class `C`. This encodes access to `f` through the current instance of `C` (i.e., `this.f`) and through evaluation of JMLs old-expression (i.e., `old(this.f)` and `old(this.f)`). If the field `f` is static we also encode static access (i.e., `C.f` and `old(C.f)`) by adding constants named _C_C___f and old_C_C___f to the type predicate. If the type of `f` is a reference type `D` we add constants to encode access to the members of class `D` through `f`. For instance, if `D` have field members `a` and `b` those constants are named _C___f_D___a and _C___f_D__b (corresponding to `f.a` and `f.b`). Furthermore, for `a` we add constants _C___this_C___f_D___a, old_C___f_D___a and old_C___this_C___f_D___a, and likewise for `b`. We do not consider possible member access through `a` and `b`.

We also apply the naming scheme described above when we encode variables in expressions. For instance given an expression involving the variable `this.f` we encode this variable as _C___this_C___f. We can refer to _C___this_C___f given our encoding of the declaration of `f` since we declared the constant _C___this_C___f as part our encoding of `f`.

A field `f` is interesting to us only if it is used in a method specification. The purpose of our system is to reason about method specifications, so if a field is not mentioned in any specification, we do not need to add any encodings of its declaration to the type predicate. However, we do all of the encodings mentioned above for every single field declaration. While this is harmless to the soundness of the type predicate it may lead to unnecessarily long SMT-LIBv2 scripts.

The value of `f` may differ from the evaluation of `old(f)`, but the value of `f` does not differ from the value of `this.f` and consequently the value of `old(f)` is identical to the value of `old(this.f)`. Likewise the value of `f.a` may differ from the evaluation of `old(f.a)`, but the value of `f.a` does not differ from the value of `this.f.a` and consequently the value of `old(f.a)` is identical to the value of `old(this.f.a)`. To encode such identities we add assertions like

$\_C\_\_\_f = \_C\_\_\_this\_C\_\_\_f$

to the type predicate.

We designate names of the form $\_C\_\_m$ for methods. Since a field and a method in a class can share the same name, a naming scheme that distinguishes fields and methods is necessary.

For the remainder of this report we will not use these verbose names but simply let f stand for all of the names $\_C\_\_\_f$, $\_C\_\_\_this\_C\_\_\_f$ and $\_C\_C\_\_\_f$; and similarly let old_f stand for all of the names $old\_C\_\_\_f$, $old\_C\_\_\_this\_C\_\_\_f$ and $old\_C\_C\_\_\_f$

4.2.2 Types

Primitive types

We use the sorts *Bool*, *Int* and *Real* from the background theory to represent the primitive types of Java in the object logic. The primitive type `boolean` is encoded using the sort *Bool*. The primitive types `char`, `byte`, `short`, `int` and `long` are encoded using the sort *Int*. The primitive types `float` and `double` are encoded using the sort *Real*. An alternative approach is to add sorts such as *Char*, *Long* and *Double* to the type predicate. However, using this approach we have to encode the appropriate arithmetics of these sorts ourselves in the type predicate. We do not take this approach. Instead we opt for a deep mapping using the sorts available in the theories of the background logic. Below we go through each of the mappings from a primitive Java type to a sort defined in the background theory.

As mentioned Java's primitive type `boolean` is simply encoded using the sort *Bool* from the Core theory which is included in our background theory. Thus for a field declaration

```
bool b;
```

we add the constant

$b : Bool$

to the type predicate.

The primitive types `char`, `byte`, `short`, `int` and `long` are encoded using the sort *Int*. The sort *Int* corresponds to the mathematical integers $\mathbb{Z}$ and thus has an infinite domain. The primitive types of Java on the other hand have a finite domain. We make a first approximation towards taking this semantic difference into account simply by asserting bounds corresponding to the ranges of the primitive types. The range for the type `byte` is -128 to 127. So for instance, given a field declaration

```
byte by;
```

we add the constant

$by : Int$

and the assertions

$-128 \leq by$

$by \leq 127$

to the type predicate.

An alternative approach would be to introduce a function

$isByte : Int \mapsto Bool$

with appropriate assertions

$(\forall i : Int)[isByte(i) \Rightarrow (-128 \leq i \wedge i \leq 127)]$

directly in the object logic, and add

$by : Int$

and

$isByte(by)$

to the type predicate.

However, in neither of these alternatives there is an encoding of the notion of overflow. In Java, if a byte `by` has the value 127, the result of the expression `by++` is -128. In our encoding the result is 128. We accept this as a limitation in our prototype system.

The primitive types `float` and `double` are encoded using the sort *Real* from the background theory. We make no attempts to bridge the semantic gap between these types and the sort *Real*. Given the declaration

```
float f;
```

we simply add the constant

$f : Real$

to the type predicate.

Reference types

We introduce the sort constant

Ref

in the object logic. Values of this sort represent Java variables of class or interface type. The class and interface types are in turn encoded as *Class* constants as described above (section 4.1). To bridge the two sorts we introduce the function

$typeOf : Ref \mapsto Class$

in the object logic. As described in section 4.1 given a class declaration of a class `C` we form the type predicate $C : Class$. On that basis given a field declaration

```
C c;
```

we form the type predicates

$c : Ref$

and

$typeOf(c) = C$

to represent `c` and encode the fact that `c` has reference type `C`.

4.3 Arrays

In this section we consider especially the declaration and initialization of arrays such as

```
byte[] barr = byte[3];
```

To represent the Java array type we use the sort *Array* which is included in our background theory. Since arrays in Java are indexed by integers we always use *Int* as the index sort. As value sort we use either another array sort to encode multidimensional array types, or one of the sorts *Bool*, *Int*, *Real* or *Ref* to encode arrays with other java types as element types as described above (section 4.2.2).

Especially for array declarations we take initializations into account. We do this because we want to add assertions about the length of arrays to the type predicates. The theory of arrays included in our background logic does not include a function *length* that operates on Array sorts. Therefore we define such a function and include it in the object logic. In fact, we define several functions, since we cannot define overloaded functions in our object logic (due to the restrictions mentioned in 2.3). The first few of these uninterpreted functions have the following form

$arrayLength1DSort : (Array\ Int\ Sort) \mapsto Int$

$arrayLength2DSort : (Array\ Int\ (Array\ Int\ Sort)) \mapsto Int$

$arrayLength3DSort : (Array\ Int\ (Array\ Int\ (Array\ Int\ Sort))) \mapsto Int$

where $Sort$ stands for either $Bool$, Int, $Real$ or Ref.

An array declaration like the one above gives rise to the following type predicates

$barr : (ArrayIntInt)$

$arrayLength1DBool(barr) = 3$

$(\forall i : Int)[0 \leq i \wedge i < arrayLength1DBool(barr) \implies -128 \leq select(barr, i) \wedge select(barr, i) \leq 127]$

A declaration of a multidimensional array such as

```
short[][] sharr = new short[3][2];
```

gives rise to following type predicates

$arrayLength2DInt(sharr) = 3$

$arrayLength1DInt(select(sharr, 0)) = 2$

$arrayLength1DInt(select(sharr, 1)) = 2$

$arrayLength1DInt(select(sharr, 2)) = 2$

$(\forall i1 : Int)[i1 < arrayLength2DInt(sharr) \Rightarrow$
$(\forall i2 : Int)[i2 < arrayLength1DInt(select(sharr, i1)) \Rightarrow$
$-32768 \leq select(select(sharr, i1), i2) \wedge select(select(sharr, i1), i2) \leq 32767]]$

Although we had to introduce length operators for arrays in the object logic ourselves, the arrays theory included in our background theory makes encoding of arrays quite simple. It is not entirely trivial to equip the object logic with support for reasoning about arrays on the basis of a background with no notion of this data structure. The authors of the initial version of ESC/Java2 had to include several functions in their object logic to be able to reason about arrays, since their background theory is the unsorted logic of simplify [CK05].

4.4 Method declarations

When encoding a method declaration such as

```
byte m(boolean b, short s) {...}
```

we invoke the rules for forming type predicates that we have introduced in section 4.2. We encode the formal parameters of a method declaration as constants in the type predicate much in same way as we did for field declarations. Given the declaration of the method `m` above we add the constants

$b_m : Bool$

$s_m : Int$

and the assertion

$-32768 <= s_m \wedge s_m <= 32767$

to the type predicate to encode the formal parameters of method `m`. We also add the constant

$m : Int$

and the assertion

$-128 <= m \wedge m <= 127$

to the type predicate.

Thus our encoding of the method declaration above amounts to three constants and some assertions on their ranges. Obviously this is not a complete encoding of `m`. In fact our encoding corresponds to one invocation of `m` with m being the return value and b_m and s_m the actual parameters of that invocation. We will discuss this further in section 4.6.

To allow encoding of methods with no return type we include the constant

$Void$

in the object logic.

4.5 Method specifications

If a method, m, defined as in the previous section (section 4.4), has a specification such as

```
//@ requires b;
//@ requires -11 <= s && s <= 11;
//@ ensures  \result == 11*s;
```

we add the constants

$$\begin{aligned} m_{pre} &: Bool = (b_m \wedge -11 \leq s_m \wedge s_m \leq 11) \\ m_{post} &: Bool = (m = 11 * s_m) \\ m_{spec} &: Bool = (m_{pre} \Rightarrow m_{post}) \end{aligned}$$

to the type predicate to encode the precondition, postcondition and specification, respectively, of `m`. The encoding of the specification, m_{spec}, is expressed in terms of the encodings of the precondition, m_{pre}, and the postcondition, m_{post}, according to the definition presented in section 2.1 and 2.2. m_{post} is an encoding of the Boolean expression in the ensures clause. m_{pre} is a conjunction of the encodings of the Boolean expressions in each of the requires clauses.

In section 4.7 we provide more detail on the encoding of expressions. Here we note that our encoding, s_m, of the parameter `s` is simply the constant that we declared to encode the formal parameter `s` of method `m` as described in the previous section (4.4). Likewise, the encoding, b_m, of the parameter `b` is simply the constant that we declared to encode the formal parameter `b` of method `m`. We achieve this by applying the same naming scheme for the encoding of variable declarations and for the encoding of variables in expressions as mentioned in section 4.2.1. Similarly we encode the keyword `/result` simply as the constant m that we declared as part of the encoding of the declaration of method `m`.

Following the definitions presented in section 2.1, if no requires clause is present in the method specification, we encode the precondition as $m_{pre} : Bool = true$. Likewise if no ensures clause is present we encode the postcondition as $m_{post} : Bool = true$.

4.6 Method invocations

In this section we describe how we encode method invocations in the type predicate. Our approach has some limitations which we discuss.

We proceed by example and consider an expression like the following

```
a < m(true, 11)
```

where `m(true, 11)` is an invocation of the method `m` that we declared in section 4.4 and `a` is either a field or a method parameter. Given this expression we add the following assertions to the type predicate

$b_m = true$

$s_m = 11$

m_{spec}

$a < m$

where we assume m, b_m, s_m and m_{spec} are declared as described in the previous sections (sections 4.4 and 4.5). a is an encoding of a variable `a`. We assume that we have encoded the declaration of `a` elsewhere.

The idea behind this simple approach to encode method invocations is as follows.

Given the three first assertions the solver can substitute $true$ and 11 for b_m and s_m, respectively, in m_{spec} to get

$(true \wedge -11 \leq 11 \wedge 11 \leq 11) \Rightarrow (m = 11 * 11)$

from which the solver can infer

$m = 11 * 11$

and therefore

$a < 11 * 11$

Thus the solver can use our encoding of the specification of method `m` to reason about the value of our encoding of the return value, m, of method `m`.

However, using this strategy where all our encodings of method return values, parameters, and specifications (m, b_m, s_m and m_{spec}, respectively, in this example) are constants, we can only encode one method invocation per method declaration. Using the approach described above, if another invocation of method `m`, say `m(true, 0)`, is encoded in the same type predicate, we would have two assertions on each of the constants m, b_m, s_m and m_{spec}. The two assertions on s_m alone would lead to an unsatisfiable set of assertions, i.e., we would have $s_m = 11$ and $s_m = 0$ which is not satisfiable. Unsatisfiability may well be an expected result when the solver evaluates a type predicate, but we do not want it to stem from an unsound encoding such as the encoding of two or more successive invocations of the same method in the approach described above.

Another approach that allows sound encoding of several invocations of the same method could be as follows. Instead of encoding the return value of a method as a constant we can instead encode it as an uninterpreted function where the number of parameters and their sorts are determined by the variables occurring in the expressions involving the result keyword, and where the result sort is determined by the return type of the encoded method. In our running example this leads to the encoding

$m : Int \mapsto Int$

since the parameter `s` in the expression `result == s` has type `short` which we encode with sort Int and since `m` is declared with return type `int` which we encode with sort Int. Likewise instead of encoding the specification of a method as a constant we can encode it as a function where the number of parameters and their sorts are determined by the variables occurring in the requires and ensures clauses of the method, and with sort $Bool$ as result sort. We give this function a definition that is similar to the one used in the simple approach described above. In our running example this leads to the encoding

$m_{spec} : Bool \times Int \mapsto Int$

$m_{spec}(b, s) = b \wedge -11 \leq s \wedge s \leq 11 \Rightarrow m(s) = 11 * s$

Using this approach we can soundly encode several method invocations of the same method. For instance the expression `a > m(true, 11)` leads to the following assertions in the type predicate

$m_{spec}(true, 11)$

$a < m(11)$

from which the solver can infer that

$a < 11 * 11$

The expression `a > m(true, 0)` on the other hand leads to the assertions

$m_{spec}(true, 0)$

$a > m(0)$

that the solver can use to infer that

$a > 11 * 0$

Now we have encoded two invocations of `m` in a sound way. In particular we have not asserted that the same constant is equal to two different values. In this approach we do not encode method parameters as constants. Instead they only appear in the encoding as the parameters of functions that encode the specification and the return value of the method. This means we must take care that parameters are passed in the right order to these functions. For instance, in the example above we must make sure that 0 and not *true* is passed to m.

We have only implemented the simple approach described in the beginning of this section and not the approach immediately above.

4.7 Other expressions

Boolean expressions in Java can be combined both by standard Boolean operators, viz. `&`, `|`, `^` and `!`, and by the conditional, or "short-circuit", operators `&&` and `||`. The standard Boolean operators have direct counterparts in first-order logic and therefore also in our background theory (which contains the basic elements of Boolean logic since it includes the Core theory as described in section 2.3). The conditional operators can be expressed in terms of the `ite` operator which is also part of our background theory. In table 4.1 the mapping between Boolean Java expressions and type predicates in our object logic is summarized.

Java	SMT-LIB
`a & b`	$and(a, b)$
`a | b`	$or(a, b)$
`a ^ b`	$xor(a, b)$
`!a`	$not(a)$
`a && b`	$ite(a, b, false)$
`a || b`	$ite(a, true, b)$

Table 4.1: In each row the Boolean Java expression in the left column gives rise to the type predicate in the right column.

4.8 Leveraging specification matching

The goal of this project is to develop a system capable of matching JML specifications of Java methods according to the match predicate definitions we presented in section 2.2.

In the previous sections of this chapter we have presented the definition of our object logic and described how we form type predicates to encode JML annotated Java based on this object logic and the background theory. In this section we describe how we use this to achieve the goal of the project.

Say we want to match the specifications of the methods `m1` and `m2`. `m1` is a member of class `A` defined in the source file `A.java`. `m2` is a member of class `B` defined in the source file `B.java`. We run our compiler on both source files to produce two SMT-LIBv2 scripts, `A.smt2` and `B.smt2`. `A.smt2` contains the encoding of the specification of `m1`. `B.smt2` contains the encoding of the specifications of `m2`. The two scripts, `A.smt2` and `B.smt2`, both include a definition of our object logic. We remove the object logic from `B.smt2` and then append `B.smt2` to `A.smt2` to form the script `match.smt2`.

We assume that the signatures of `m1` and `m2` are identical. The names of the formal parameters may vary from `m1` and `m2` though. Even if they do not, in our encoding of the parameters, the names will differ since we prepend the class name to all variable names as described in section 4.2.1. To account for this we append assertions to `match.smt2` to pairwise equate the encodings of the parameters of `m1` and `m2`. For instance, if the parameterlist of `m1` is `(a, b)` and the parameterlist of `m2` is `(x, y)` we append to `match.smt2` the assertions $a_{m1} = x_{m2}$ and $b_{m1} = y_{m2}$. Likewise we equate the encodings of the return value of `m1` and `m2`, i.e. $m1 = m2$.

What remains is to append the actual match predicate. As described in section 4.5 we encode the precondition, the postcondition and the specification of a method as constants of sort *Bool*. We combine these Boolean constants using the connectives $\Leftrightarrow$, $\Rightarrow$ and $\wedge$ to encode the desired match predicate. For instance if we want to decide if `m2` is matched by `m1` under plug-in match we append the constant

$match : Bool = (m2_{pre} \Rightarrow m1_{pre}) \wedge (m1_{pre} \Rightarrow m2_{post})$

to `match.smt2`. Now we do a copy of the script `match.smt2` to a new script `nonmatch.smt2`. We append the assertion *match* to `match.smt2`, and we append the assertion $\neg match$ to `nonmatch.smt2`. We then run the solver on both files. If the solver responds with `unsat` when given `nonmatch.smt2` as input, we conclude that `m1` matches with `m2` under the given match. For testing purposes we also consider the result of running the solver on `match.smt2`. If the solver responds with `unsat` in this case, the match predicate is a contradiction.

Chapter 5

Test and Evaluation

In the previous chapter we described our implementation of a system for specification matching of JML-annotated Java methods. In this chapter we describe the tests of this system that we have performed.

5.1 Test Methodology

For each test case we create two Java source files, `A.java` and `B.java`. In `A.java` we declare a class `A`. In `B.java` we declare a class `B`. We declare a number of methods in each of the classes `A` and `B`. The methods are annotated with JML specifications. A pair of methods consists of a method from class `A` and a method from class `B`. We form all possible pairs of methods. If the number of methods in class `A` is a and the number of methods in class `B` is b, we form $2ab$ pairs. For instance, if class `A` has one method, `a1`, and a class `B` has two methods, `b1` and `b2`, we form the four pairs (`a1`,`b1`), (`b1`,`a1`), (`a1`,`b2`) and (`b2`,`a1`). File `B.java` may coincide with file `A.java`. In that case we form a^2 pairs of methods.

Given a pair of methods, (`m1`, `m2`), we can form a match predicate $match_M(S, Q)$, where S is the specification of `m1` and Q is the specification of `m2`. If $match_M(S, Q)$ holds we say that the pair satisfies M. If $match_M(S, Q)$ is a contradiction we say that the pair contradicts M.

For each of the $2ab$ pairs of methods we let our system decide which matches the pair satisfies or contradicts. We check each of these decisions by hand to see if they are correct. We say that we validate a pair of methods under a match. We may use the relations between the matches as given by the lattice in figure 2.1 to ease the checking by hand. In most test cases we do not consider all matches.

5.2 Test Cases

5.2.1 First order logic

In the first system test we restrict method specifications to consist only of the Boolean literals `true` and `false`. In this test case the two source files coincide. We use the source file `ClassTF.java` given in listing A.1 (The annotation `@MatchMe` can be disregarded for now). We validate all pairs of methods under all matches. For instance we check that our system decides that the pair (`m1`, `m3`) satisfies guarded postmatch, and that the pair (`m1`, `m3`) does not satisfy exact pre/post match.

In the second test case the method specifications consist only of an ensures clause. In this test case the two source files coincide. We use the source file `ClassBB.java` given in listing A.2. All methods have return type `boolean`, and one or two parameters of type `boolean`. The expression in the ensures clause is of the form `result == <expression>`. `<expression>` is a boolean JML expression where the variables are the parameters of the method. The equality operator `==` and the boolean operators `&`, `|` and `!` are allowed in `<expression>`. For instance the method `notnot` that takes a single parameter, `p0`, has the specification `//@ post result == !!p0;`. We validate all pairs of methods under exact pre/post match. The pairs that satisfy exact pre/post match are also validated under all the other matches. For instance we check that our system decides that the pair (`id`, `notnot`) satisfies exact pre/post match, and that the pair (`id`, `not`) contradicts exact pre/post match.

The third test case extends the second by including helper methods to replace the boolean operators, i.e., we write `not(p0)` instead of `!p0`. In this test case the two source files coincide. We use the source file `ClassBBsis.java` given in listing A.3. Helper methods are excluded from the formation of pairs of methods. Only methods annotated with the `MatchMe` annotation are included in the formation of pairs of methods. Thus, any method not annotated with the `MatchMe` annotation is a helper method. The method `not2` is an example of a helper method. This method has the specification `//@ post result == !p0;`. The method `not3` is a helper method with the same specification as `not2`. We include several helper methods with the same specifications, since our system can only handle one invocation of each method, as described in section 4.6. Method `id2` is a helper method with the specification `//@ post result == p0;`. The non-helper method `notnotsis` has the specification `//@ post result == not2(not3(id2(p0)));`. We validate all pairs of methods under exact pre/post match. For instance we check that our system decides that the pair (`id`, `notnotsis`) satisfies exact pre/post match.

In the fourth test case the two source files coincide. We use the source file `Class-DM.java` given in listing A.4. The method specifications has the same form as in the previous two test cases and the only type allowed is `boolean`. We validate all pairs of methods under exact pre/post match. For instance we check that our system decides that the pair (`dmNotAnd`, `dmOrNot`) (cf. listing A.4) satisfies exact pre/post match. We note that this match predicate — viz. $match_{\text{E-pre/post}(S,Q)}$, where S is the specification for `dmNotAnd` and Q is the specification for `dmOrNot` — is an instance of the first of de Morgan's Laws, viz. $\neg(p \wedge q) \Leftrightarrow (\neg p \vee \neg q)$

In the fifth test case we loosen the restrictions compared to the previous four test cases a bit and allow methods with three parameters (still only of type `boolean`) and the boolean operator `==>`. Again the two source files coincide; we use the source file `ClassIM.java` given in listing A.5. We validate all pairs of methods under exact pre/post match. For instance we check that our system decides that the pairs of methods (`pIqIr`, `pAqIr`), (`pIq`, `nqInp`) and (`pIq`, `npOq`) (cf. listing A.5) all satisfy exact pre/post match. We note that these match predicates are instances of Importation/Exportation, Transposition and Material Implication ($[(p \Rightarrow (q \Rightarrow r)) \Leftrightarrow ((p \wedge q) \Rightarrow r)]$, $[(p \Rightarrow q) \Leftrightarrow (\neg q \Rightarrow \neg p)]$ and $[(p \Rightarrow q) \Leftrightarrow (\neg p \vee q)]$) respectively.

In the sixth test case we use two non-coinciding source files. In the first file we declare class `S` as in listing 2.2. In the second file we declare class `Q` as in listing 2.1. We validate all pairs of methods — where `m0` of class `Q` is the second element of the pair — under all matches. This amounts to checking that the decisions our system makes can be summarized as in table 2.3.

5.2.2 Linear arithmetic

In the seventh test case we are concerned with methods with return type `int` and with a single parameter of type `int`. The method specifications are of the form `@ ensures result == <expression>;`. We only allow linear arithmetic. In this test case the two source files coincide. We use the source file `ClassBI.java` given in listing A.6. The helper methods `incr` and `decr` have specifications `//@ post result == i + 1;` and `//@ post result == i - 1;` respectively. The non-helper methods `id` and `idsis` have specifications `//@ post result == i;` and `//@ post result == incr(decr(i));` respectively. We validate all pairs of methods under exact pre/post match. For instance we check that our system decides that the pair of methods (`id`, `idsis`) satisfy exact pre/post match.

In the eighth test case we use the same restrictions as in the seventh test case. Again the two source files coincide; we use the source file `ClassMI.java` given in listing A.7. The helper methods `db` and `halve` have specifications `//@ post result == i * 2;` and `//@ post result * 2 == i;` respectively. The non-helper methods `id` and `idsis` have specifications `//@ post result == i;` and `//@ post result == db(halve(i));` respectively. We validate all pairs of methods under exact pre/post match. For instance we check that our system decides that the pair of methods (`id`, `idsis`) satisfy exact pre/post match.

5.2.3 Non-linear arithmetic

In the ninth test case we exercise our system on non-linear arithmetic. Once again we use a single source file. In this test case we use the file `AI.java` given in listing A.8. The class `ClassAI` in file `AI.java` declare two methods `int lhs(int x)` and `int rhs(int x)` with specifications `//@ post result == (x + 1) * (x + 1);` and `//@ post result == x * x + 2 * x + 1;` respectively. Our system decides that the pair (`lhs`, `rhs`) satisfies exact pre/post match. We expect this, since $(x + 1) \times (x + 1) = x \times x + 2x + 1$. However, when we ask the system to decide if (`lhs`, `rhs`) contradicts exact pre/post match we get no answer, since the solver responds with `unknown`. This is due to the limitations of our solver, Z3. As mentioned section 2.3 Z3 has some but not complete support for non-linear arithmetic. Thus the result `unknown` is expected in this case.

Chapter 6

Related Work

In this chapter we mention some of the work related to the work we present in this thesis. Firstly we mention the work of Zaremski and Wing [ZW97]. In addition to introducing definitions of specification matching, these authors also implement a system for specification matching of software components based on their definitions on specification matching. The software components considered by these authors are written in the programming language ML and specified in the Larch/ML specification language. Zaremski and Wing define specification matching not only for functions but also for modules (sets of functions). The authors show that module matching can be used to determine subtyping for object types for different definitions of subtyping for objects. Zaremski and Wing defines component matching as a conjunction between signature matching and specification matching. The authors note that we can think of signature matching as a filter used to prune the search space before trying the possibly more expensive specification matching. Klein and König-Ries propose a matching algorithm that does not split signature and specification matching [KKR04]. Instead the algorithm is able to alter the signatures of components if their specifications match.

Other related work concerns the challenge of representing programming languages like Java in first-order logic. The initial version of the verification system ESC/-Java2 uses Simplify as its solver and is based on an object logic tuned for the unsorted logic of this solver [CK05]. Given their starting point in this unsorted logic the authors of the initial version of ESC/Java2 have included several axioms in their object logic to reason about arrays. A future task within the OpenJML project is to complete the SMT-LIBv2 interface for static checking [Cok11b]. The KeY project develops a system for verification of the Java Card language [Häh07]. An instance of dynamic logic called Java Card DL is used as the logical basis of this verification system. Either OCL or JML can be used as specification language in this system.

Chapter 7

Summary and Future Work

In this thesis we present a system for specification matching of JML-annotated Java methods. The system is able to compare the JML specifications of two Java methods and decide if they match according to any of the definitions of method specification matching given by Zaremski and Wing in 1997 [ZW97]. To our knowledge the system presented in this thesis is the first to realize specification matching as defined by Zaremski and Wing atop a mainstream programming language such as Java. However, our system is only defined for a subset of JML and Java.

A main ingredient in our system is the translation of JML-annotated Java into a many-sorted first-order logic defined by version 2 of the SMT-LIB standard. As our background theory we use the logic AUFNIRA from the SMT-LIB. This logic includes a theory of arrays and a combined theory on integers and reals. This is a convenient basis for the definition of our object logic of JML-annotated Java. Our object logic includes axioms on subtyping and provides the length operator missing in the background theory. On the basis of the background theory and our object logic we form type predicates of class, field, array and method declarations and of JML specifications in the form of pre- and postconditions. Thus, we can form type predicates corresponding to any of the specification matching definitions introduced by Zaremski and Wing in 1997 [ZW97]. We give the resulting SMT-LIBv2 script as input to the conforming solver Z3.

Our system passes system tests with specifications involving Boolean expressions and linear arithmetic, respectively. The system also passes a test case on non-linear arithmetic. Here one of the expected results is that the solver Z3 cannot decide satisfiability due to the limited support for non-linear arithmetic in that solver.

An inherent limitation of our system is implied by the limitations of the underlying solver. It is also a limitation that our system is only defined for a subset JML and Java. Other limitations stem from the approximations in our object logic and our rules for forming type predicates. One such limitation is our representation of primitive Java types such as `char`, `int` and `long`. These types have a finite domain. We represent these types with the sort Int which have an infinite domain. We do not properly bridge this semantic gap, neither in our object logic nor by means of rules for forming type predicates. Another limitation of our system is the encoding of method invocations as we discuss in section 4.6. Further work on the system we present in this thesis include work on the limitations mentioned above.

Bibliography

[BCC$^+$03] L. Burdy, Y. Cheon, D. Cok, M.D. Ernst, J. Kiniry, G.T. Leavens, K. Rustan, M. Leino, and E. Poll, *An overview of JML tools and applications1*, Electronic Notes in Theoretical Computer Science **80** (2003), 75–91.

[Bro87] F.P. Brooks, *No silver bullet: Essence and accidents of software engineering*, IEEE computer **20** (1987), no. 4, 10–19.

[BST10] C. Barrett, A. Stump, and C. Tinelli, *The smt-lib standard: Version 2.0*, 2010.

[CK05] D.R. Cok and J.R. Kiniry, *Esc/java2: Uniting esc/java and jml*, Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (2005), 108–128.

[Cok11a] D. Cok, *jSMTLIB: Tutorial, Validation and Adapter Tools for SMT-LIBv2*, NASA Formal Methods (2011), 480–486.

[Cok11b] ______, *OpenJML: JML for Java 7 by Extending OpenJDK*, NASA Formal Methods (2011), 472–479.

[DMB08] L. De Moura and N. Bjørner, *Z3: An efficient SMT solver*, Tools and Algorithms for the Construction and Analysis of Systems (2008), 337–340.

[FS97] B. Fischer and J. Schumann, *NORA/HAMMR: Making deduction-based software component retrieval practical*, Proc. CADE-14 Workshop on Automated Theorem Proving in Software Engineering, vol. 96, Citeseer, 1997, p. 97.

[Gar10] M. Garcia, *Compiler plugins can handle nested languages: AST-level expansion of LINQ queries for Java*, Object Databases (2010), 41–58.

[Gos00] J. Gosling, *The Java language specification*, Prentice Hall, 2000.

[Häh07] Reiner Hähnle, *A new look at formal methods for software construction*, Verification of Object-Oriented Software. The KeY Approach (Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, eds.), Lecture Notes in Computer Science, vol. 4334, Springer Berlin / Heidelberg, 2007, 10.1007/978-3-540-69061-$0_1, pp. 1--18$.

[Hem05] D. Hemer, *A formal approach to component adaptation and composition*, Proceedings of the Twenty-eighth Australasian conference on Computer Science-Volume 38, Australian Computer Society, Inc., 2005, pp. 259–266.

[JC95] Jun-Jang Jeng and Betty H. C. Cheng, *Specification matching for software reuse: a foundation*, SIGSOFT Softw. Eng. Notes **20** (1995), no. SI, 97–105.

[KKR04] M. Klein and B. König-Ries, *Coupled signature and specification matching for automatic service binding*, Web Services (2004), 183–197.

[LBR99] G. Leavens, A. Baker, and C. Ruby, *JML: A notation for detailed design*, KLUWER INTERNATIONAL SERIES IN ENGINEERING AND COMPUTER SCIENCE (1999), 175–188.

[LC05] G.T. Leavens and Y. Cheon, *Design by Contract with JML*, Draft, available from jmlspecs. org **1** (2005), 4.

[LPC$^+$06] G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. M
"uller, J. Kiniry, P. Chalin, and D.M. Zimmerman, *JML reference manual*, Department of Computer Science, Iowa State University (2006).

[MBNR69] M.D. McIlroy, JM Buxton, P. Naur, and B. Randell, *Mass produced software components*, Software Engineering Concepts and Techniques (1969), 88–98.

[NRB69] P. Naur, B. Randell, and F.L. Bauer, *Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968*, Scientific Affairs Division, NATO, 1969.

[PP02] D. Perry and S. Popovich, *Inquire: Predicate-based use and reuse*, Knowledge-Based Software Engineering Conference, 1993. Proceedings., Eighth, IEEE, 2002, pp. 144–151.

[ZW97] AM Zaremski and JM Wing, *Specification matching of software components*, ACM Transactions on Software (1997).

Appendix A

Test Files

Listing A.1: Source file `ClassTF.java` declaring class `ClassTF`

```
class ClassTF
{
    //@ requires true;
    //@ ensures  true;
    @MatchMe void m1() {}

    //@ requires false;
    //@ ensures  true;
    @MatchMe void m2() {}

    //@ requires true;
    //@ ensures  false;
    @MatchMe void m3() {}

    //@ requires false;
    //@ ensures  false;
    @MatchMe void m4() {}
}
```

Listing A.2: Source file `ClassBB.java` declaring class `ClassBB`

```java
class ClassBB
{
    //@ post \result == p0;
    @MatchMe
    boolean id(boolean p0) {return p0;}

    //@ post \result == !p0;
    @MatchMe
    boolean not(boolean p0) {return !p0;}

    //@ post \result == (p0 & p1);
    @MatchMe
    boolean and(boolean p0, boolean p1) {return p0 & p1;}

    //@ post \result == (p0 | p1);
    @MatchMe
    boolean or(boolean p0, boolean p1) {return p0 | p1;}

    //@ post \result == (!p0 & p1);
    @MatchMe
    boolean imp(boolean p0, boolean p1) {return !p0 & p1;}

    //@ post \result == (p0 == p1);
    @MatchMe
    boolean eq(boolean p0, boolean p1) {return p0 == p1;}

    //@ post \result == !!p0;
    @MatchMe
    boolean notnot(boolean p0) {return !!p0;}

    //@ post \result == (p0 & p0);
    @MatchMe
    boolean anduna(boolean p0) {return p0 & p0;}

    //@ post \result == (p0 | p0);
    @MatchMe
    boolean oruna(boolean p0) {return p0 | p0;}

    //@ post \result == (p0 & true);
    @MatchMe
    boolean andrew(boolean p0) {return p0 & true;}

    //@ post \result == (p0 | false);
    @MatchMe
    boolean orpheus(boolean p0) {return p0 | false;}
}
```

Listing A.3: Source file `ClassBBsis.java` declaring class `ClassBBsis`

```
class ClassBB
{
    //@ post \result == p0;
    @MatchMe
    boolean id(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id2(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id3(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id4(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id5(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id6(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id7(boolean p0) {return p0;}

    //@ post \result == p0;
    boolean id8(boolean p0) {return p0;}


    //@ post \result == !p0;
    @MatchMe
    boolean not(boolean p0) {return !p0;}

    //@ post \result == !p0;
    boolean not2(boolean p0) {return !p0;}

    //@ post \result == !p0;
    boolean not3(boolean p0) {return !p0;}

    //@ post \result == !p0;
    boolean not4(boolean p0) {return !p0;}

    //@ post \result == !p0;
    boolean not5(boolean p0) {return !p0;}


    //@ post \result == (p0 & p1);
    @MatchMe
    boolean and(boolean p0, boolean p1) {return p0 && p1;}
```

Listing A.4: Source file `ClassDM.java` declaring class `ClassDM`

```
class ClassDM
{
    /*
     *  [1]        [2]
     *  !(p â^§ q) â†" !p â^¨ !q             //de Morgan I
     *
     *  [3]        [4]
     *  !(p â^¨ q) â†" !p â^§ !q
//de Morgan II
     *
     */

    //[1]
    //@ post \result == !(p & q);
    @MatchMe
    boolean dmNotAnd(boolean p, boolean q) {return !(p & q);}

    //[2]
    //@ post \result == (!p | !q);
    @MatchMe
    boolean dmOrNot(boolean p, boolean q) {return (!p | !q);}

    //[3]
    //@ post \result == !(p | q);
    @MatchMe
    boolean dmNotOr(boolean p, boolean q) {return !(p | q);}

    //[4]
    //@ post \result == (!p & !q);
    @MatchMe
    boolean dmAndNot(boolean p, boolean q) {return (!p & !q);}
}
```

Listing A.5: Source file `ClassIM.java` declaring class `ClassIM`

```
class ClassIM
{
    /*
     *  [1]              [2]
     *  (p â†’ [q â†’ r]) â†" ([p â^§ q] â†’ r)
//(Im/Ex)portation
     *
     *  [3]        [4]
     *  (p â†’ q) â†" (!q â†’ !p)
//Transposition
     *
     *  [3]        [5]
     *  (p â†’ q) â†" (!p â^¨ q)
//Material Implication
     */

    //[1]
    //@ post \result == (p ==> (q ==> r));
    @MatchMe
    boolean pIqIr (boolean p, boolean q, boolean r) {return true;}

    //[2]
    //@ post \result == ((p & q) ==> r);
    @MatchMe
    boolean pAqIr (boolean p, boolean q, boolean r) {return true;}

    //[3]
    //@ post \result == (p ==> q);
    @MatchMe
    boolean pIq   (boolean p, boolean q) {return true;}

    //[4]
    //@ post \result == (!q ==> !p);
    @MatchMe
    boolean nqInp (boolean p, boolean q) {return true;}

    //[5]
    //@ post \result == (!p | q);
    @MatchMe
    boolean npOq  (boolean p, boolean q) {return true;}
}
```

Listing A.6: Source file `ClassBI.java` declaring class `ClassBI`

```
class ClassBI
{
    //@ post \result == i;
    @MatchMe
    int id(int i) {return i;}

    //@ post \result == i + 1;
    int incr(int i) {return i + 1;}

    //@ post \result == i - 1;
    int decr(int i) {return i - 1;}

    //@ post \result == incr(decr(i));
    @MatchMe
    int idsis(int i) {return i;}
}
```

Listing A.7: Source file `ClassMI.java` declaring class `ClassMI`

```
class ClassMI
{
    //@ post \result == i;
    @MatchMe
    int id(int i) {return i;}

    //@ post \result == i * 2;
    int db(int i) {return i * 2;}

    //@ post \result * 2 == i;
    int halve(int i) {return i / 2;}

    //@ post \result == db(halve(i));
    @MatchMe
    int idsis(int i) {return db(halve(i));}
}
```

Listing A.8: Source file `ClassAI.java` declaring class `ClassAI`

```
class ClassAI
{
    //@ post \result == (x + 1) * (x + 1);
    @MatchMe
    int lhs(int x) {return (x + 1) * (x + 1);}

    //@ post \result == x * x + 2 * x + 1;
    @MatchMe
    int rhs(int x) {return x * x + 2 * x + 1;}
}
```