UNIVERSITÄT
KOBLENZ · LANDAU
FB4: Informatik

# Generating Efficient Test Oracles from Specifications

Studienarbeit

im Studiengang Informatik

vorgelegt von

Markus Bender

Betreuer:
Prof. Dr. Bernhard Beckert, Universität Koblenz,
Prof. Dr. Reiner Hähnle, Chalmers Institute of Technology,
Dipl. Inf. Christoph Gladisch,Universität Koblenz,
Dipl. Inf. Philipp Rümmer, Chalmers Institute of Technology

Koblenz, im Juni 2010

## Erklärung

Ich versichere, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet zu haben.

Koblenz, den 13.06.2010 Markus Bender

# Contents

# 1. Introduction

The drawback of using *software testing* to increase a program's quality is the additional need of time which has two causes. Firstly, *test cases* need to be written and evaluated and secondly, running a test case costs time, as well. The first problem has been dealt with by introducing methods to generate test cases automatically. Those methods generate the *oracle* that decides whether a test case succeeded or failed out of the *specification*'s *postcondition* [Engel, 2006, Peters and Parnas, 1994, Cheon et al., 2005].

A possible drawback of this approach might follow from the non uniform view whether specifications have to be executable or not [Hayes and Jones, 1990, Fuchs, 1992a, Gravell and Henderson, 1996], which leads to two problems:

1. A specification can be not executable.
2. A specification can be written in such a way that the performance is slow.

The first case arises from the higher expressiveness of the specification, written as logical formula, compared to programming languages. For example the datatypes of a logical specification can be unlimited, but they are limited for the programming language as there are software, respectively hardware restrictions [Andersen et al., 1992].

As specifications are normally not written to be executed, but to describe the behaviour of a program, the generated program might have a slow runtime [Gravell and Henderson, 1996] which is the second cause for the high time consumption of testing. As a high number of evaluated tests is desirable to increase the quality of the software a short runtime is wanted.

This thesis addresses both of the latter problems. To solve the first problem the given specification is modified to a new one that can be translated by introducing bounds so that the values of occurring datatypes are inside the programming language's limits. The obtained specification is not semantically equal to the original one, but similar in such way that it describes an approximation of the original one for a bounded set of values. The drawback of this method, that is a reduced expressiveness, is accepted as the possibility of execution is a benefit for this thesis.

The idea, presented in this thesis, for reducing the runtime of testcases is to change the structure of the postcondition to achieve a more performant oracle.

This thesis presents the theoretical approach and the implementation into the KeY system (see Section 5), as well as a case study to evaluate the method. Example 1.0.1 illustrates the motivation of this work.

**Example 1.0.1** *Assume formula* $F =$

$$
\begin{array}{l}
\exists i.( \\
\quad i = a * a - 1 \wedge \forall j.( \\
\quad\quad 0 > j \vee j > 100 \vee ( \\
\quad\quad\quad (i < j \Rightarrow p(j)) \wedge (!(i < j) \Rightarrow q(j)) \wedge r(a) \wedge s(b) \\
\quad\quad ) \\
\quad )
\end{array}
$$

)

*is a postcondition and is translated (see Section 2.5) to the oracle program in Listing 1. As one can see in lines 2 and 7 the ranges for both loops are* `MIN_INT` *and* `MAX_INT` *which results in* (`MAX_INT - MIN_INT`)$^2$ *iterations in the worst-case for the inner loop. Lines 16 and 17 show that there are 7 expressions that have to be evaluated in each iteration in the worst-case.*

```
1  boolean oracle(int a, int b){
2    return exists_0(MIN_INT,MAX_INT,a,b);
3  }
4
5  boolean exists_0(int min, int max, int a, int b){
6    int i = min;
7    while(!( i==a*a-1 && forall_1(MIN_INT,MAX_INT,a,b,i)) &&
8           i <= max){
9      i++;
10   }
11 return ! i==max;
12 }
13
14 boolean forall_1(int min, int max, int a, int b,int i){
15   int j = min;
16   while( 0 > j || j > 100 && j <= max && (
17          (i<j?p(j):q(j)) && r(a) && s(b)
18                                  )){
19     j++;
20   }
21 return j==max;
22 }
```

Listing 1: Not optimised generated oracle

*By using the techniques explained in Section 4 the result (see Listing 2) is more efficient due to following points:*

- *Loop execution is omitted if* `r(a)` *or* `s(b)` *are false (line 2).*
- *The nested loop is eliminated.*
- *Iterations of the loop are reduced (line 2).*
- *Statements in loop body are reduced (lines 6,12).*

```
1  boolean oracle(int a, int b){
2    return r(a) && s(b) && forall_0(0,a*a-1) && forall_1(a*a,100); }
3
4  boolean forall_0(int min, int max){
5    int j = min;
6    while(q(j) && j <= max){
7      j++;
8    } return j==max; }
9
10 boolean forall_1(int min, int max){
11   int j = min;
12   while(p(j) && j <= max){
13     j++;
14   } return j==max; }
15 }
```

Listing 2: Optimised generated oracle

# 2. Preliminaries

## 2.1. Specifications

*Specifications* or more particularly *formal specifications* are used to describe the intended behaviour of a program[1] in a precise and definite way by using well defined keywords of a specification language. One possible way of specifying the behaviour is the definition of *preconditions*, respectively *postconditions* for methods, which is supported by many specification languages like VDM-SL[John, 1991], Z[Smith, 1999], OCL[OCL, OMG] or JML[Leavens and Cheon, 2006, JML].

They are logical formulæ with a meaning similar to Hoare formulæ:

*If a method is started in a state in which its preconditions hold, and the method terminates gracefully, then it will terminate in a state in which its postconditions hold.*

There are several different specification languages of which some (like OCL and JML) are widely used and supported by verification and testing tools [Hussmann et al., 2002, Burdy et al., 2005]. In this thesis JML was chosen to be used for examples because the programming language used is Java and as Fuchs [1992b] summarises KowalskiKowalski [1979]:'Kowalski further emphasises that expressing specifications and programs in the same language eases the task of verification.'. Although Kowalski and Fuchs talk about verification, this is beneficial for testing as well. Although JML and Java are not equal, a developer being familiar with Java should be able to read and understand JML specifications.

**JML** The *Java Modelling Language* (JML)[Leavens and Cheon, 2006, JML] was designed as a specification language especially tailored to be used with Java. Therefore

[1] The terms 'program' and 'method' are used synonymously throughout the thesis.

the syntax is similar to Java and one can use the program's methods[2] and variables in the specification according to their visibility. The specification can be added as a so called JML-comment to the method directly inside the Java source file. This comment is treated like an ordinary Java-comment by Java-compilers and therefore the compilation of the program is not affected by the specification.

The concept of defining a program's pre- and postcondition as mentioned in Section 2.1 is realised by using the `requires`-, respectively `ensures`-keyword followed by a boolean expression[3] which represents the pre- or postcondition. Such expressions are allowed to consist of all valid combinations of regular boolean statements in Java and some additional features. The only extension to Java expressions that is considered in this thesis is the possibility to introduce quantified variables in the boolean expressions. This can be achieved by using the `\forall` or `\exists` keyword, followed by the type of the quantified variable, its name, a boolean expression that defines in which ranges the variable should be quantified and finally the formula, containing the quantified variable.

In this work only quantification over integers is regarded, but not quantification over objects. The quantification of objects is harder to handle as it is not uniquely defined how the quantification of objects is interpreted [Leavens et al., 2008]. It is not clear if all objects that are present in the JVM are regarded or even all objects that can possibly be created.

Additionally, it has to be mentioned that the semantics of quantifiers in JML seems not be clearly defined [Leavens et al., 2008, Engel, 2005].

Listing 3 shows a short example to illustrate the principle of pre-, postcondition and quantification in JML. The only precondition that has to hold for the input variable `ar` is that it is not null. The postcondition contains a quantified formula stating that all elements of the created array being returned whose indices are between 0 and the length of the array contain the value 1.

## 2.2. A Common Testing Technique

As mentioned in Section 1 the growing complexity and size of today's software makes it difficult to write software that is error-free and behaves as it is intended in the first attempt. Therefore a developer has to ensure these properties with additional effort. One possible method for this purpose is the use of *Software Testing*.

Software Testing is a collective term for several methods to check for errors by running the code with different input values and comparing the results with the expected values. One particular technique of testing is called *Unit Testing* and is explained in Engel [2005] A single run is called a *test case* and consists of three steps that are illustrated in Figure 1 and explained below:

1. In the *Preamble* the test case is set up. Used variables are instantiated and everything needed for the execution of the test is prepared. There are two different

[2] Only methods that are denoted with the JML modifier `pure` can be used in specifications. This modifier states that the method's execution does not cause any side effects.

[3] Also called formula

```
1  /*@ public normal_behaviour
2    @  requires ar != null;
3    @  ensures \forall int a;
4    @       0 <= a && a < \result.length;
5    @       \result[a] == 1;
6    @ */
7  public static int[] setToOne(int[] ar){
8      int[] res = new int[ar.length];
9      for (int i; i<res.length;i++) {
10         res[i]=1;
11     }
12     return res;
13 }
```

Listing 3: Code and specification for a method that sets all elements of an array to 1


Figure 1: Main steps in testing

ways to check wether the input satisfies the precondition. One is to check right after its generation, before the code is executed, and the other is to check after the execution. In the first case, new values are generated until the input satisfies the precondition and the code is run afterwards. In the second case the current test case is not taken into account and a new test case is executed.

2. In the *Implementation Under Test* (IUT) the code to be tested is executed with attributes and parameters initialised with the values of the preamble. Normally the IUT is encapsulated with safety measures, namely a catch block, to prevent a possibly occurring runtime error in the code from tearing down the whole test case.

3. The *Oracle* checks if the test case failed or succeeded. To do this it compares the results computed by the IUT with the expected results.

The collection of multiple test cases for one program is called *test suite*. Testing increases the chance of finding faults in a program. A higher coverage of the test cases, i.e. more test cases in the test suite, to cover more parts of the code, leads to a higher chance to find existing errors. In theory one can write as many test cases as exists permutations of input values[4] to achieve the maximal probability for finding errors.

[4]This is only possible if the amount of inputs is finite.

A test suite where no test case fails does not state that the program is error-free or like Dijkstra [1979] says: 'Program testing can be used to show the presence of bugs, but never to show their absence!'.

## 2.3. Logic

This thesis uses *first-order logic* (FOL) as formalism which allows expressions like 'For all natural numbers $x$ holds: if $x$ is even then $x + 2$ is even as well.' Its syntax and semantics are introduced in the following similar to Fitting [1996].

### 2.3.1. Syntactical Structure

The set of syntactical elements can be divided into two subsets, where one set is static throughout all the first-order logic formulæ and the elements of the second one vary. The set of elements that are the same in all the first-order logics are introduced first and the altering afterwards.

**Definition 2.3.1** *The following elements are symbols for all FOL languages:*

***Logic Constants*** $\top$ *and* $\bot$.

***Logical Connectives*** $\wedge, \vee, \Rightarrow, \Leftrightarrow$.

***Quantifiers*** $\mathcal{Q}$ *where* $\mathcal{Q} \in \{\forall, \exists\}$ *and*

$\forall$ *is the universal quantifier, read as 'for all'*

*and*

$\exists$ *is the existential quantifier, read as 'there exists'*

***Punctuation*** *')','(' and ':'*

***Variables*** $v_1, v_2, ...$ *for the sake of simplicity written as* $x, y, z...$

After the static elements have been introduced all the elements that can change from FOL language to FOL language are introduced.

**Definition 2.3.2** *A* first-order language $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is determined by specifying:*

1. *A finite or countable set* $\mathbf{P}$ *of* predicate symbols *each of which having a positive integer associated with it, denoting its arity. This set includes the* relational operators*:* $=, \neq, <, \leq, >, \geq$.
2. *A finite or countable set* $\mathbf{F}$ *of* function symbols *each of which having a positive integer associated with it, denoting its arity.*
3. *A finite or countable set* $\mathbf{C}$ *of* constant symbols.

**Definition 2.3.3** *The family of* terms *of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is the smallest set meeting the conditions:*

- *Any variable is a term of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.
- *Any* $x \in \mathbf{C}$ *is a term of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.
- *If* $f \in \mathbf{F}$ *and* $t_1, \ldots, t_n$ *are terms of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$*, then* $f(t_1, \ldots, t_n)$ *is a term of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.

*A term that is free of variables is called* closed.

**Definition 2.3.4** *An* atomic formula *of the language* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is any string of the form* $P(t_1, \ldots, t_n)$ *where* $P \in \mathbf{P}$ *and* $t_1, \ldots, t_n$ *are terms of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$*; also* $\top$ *and* $\bot$ *are taken to be atomic formulæ of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.

*The family of* formulæ *of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is the smallest set meeting the following conditions:*

1. *Any atomic formula of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is a formula of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.
2. *If* $A$ *is a formula of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *so is* $\neg A$.
3. *For a binary connective* $\circ$*, if* $A$ *and* $B$ *are formulæ of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$*, so is* $(A \circ B)$.
4. *If* $A$ *is a formula of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *and* $x$ *is a variable, then* $\mathcal{Q}x.A$ *is a formula of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$.

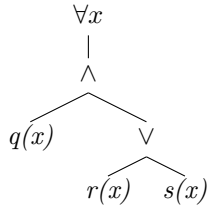
**Definition 2.3.5** Immediate subformulæ *are defined as follows:*

1. *An atomic formula has no subformulæ.*
2. *The only immediate subformula of* $\neg A$ *is* $A$.
3. *For a binary symbol* $\circ$*, the immediate subformulæ of* $(A \circ B)$ *are* $A$ *and* $B$.
4. *For* $\mathcal{Q}x.A$*, the immediate subformula is* $A$*. It is called* matrix (of the quantifier).

**Definition 2.3.6** *Let* $X$ *be a formula. The set of* subformulæ $\mathbf{Sf}_X$ *of* $X$ *is the smallest set* $\mathbf{S}$ *that contains* $X$ *and contains, for each member, the immediate subformulæ of that member.*

**Definition 2.3.7** *For every FOL formula* $F$ *an* abstract syntax tree *(AST)* $T_F$ *can be derived by using the syntax of the FOL. The tree's nodes are logical operators and their subtrees are the operator's subformulæ. The only nodes that have no subtree are the leaves of the tree which are the atomic formulæ of* $F$.

**Example 2.3.1** *If* $F = \forall x.(q(x) \wedge (r(x) \vee s(x)))$ *then* $T_F$ *is*



**Definition 2.3.8** *$Iter_F$ is an iterator that traverses $T_F$ and returns the current subformula. It is a pointer to the current position in the tree. Therefore changes to the current subformula lead to a change of the whole formula $F$.*

*There are two different kinds of iteration directions $Up_F$ and $Down_F$.*

*$Down_F$ starts at the root of the tree and traverses all the nodes in order of a breadth first search. The traversal is finished when the last leaf was visited.*

*$Up_F$ starts at the deepest leaf of the tree. All the nodes are visited in the reverted direction of a breadth first search. The last node visited is the root node.*

**Definition 2.3.9** *The* free-variable occurrences *in a formula are defined as follows:*

1. *The free-variable occurrences in an atomic formula are all the variable occurrences in that formula.*
2. *The free-variable occurrences in $\neg A$ are the free-variable occurrences in $A$.*
3. *The free-variable occurrences in $(A \circ B)$ are the free-variable occurrences in $A$ together with the free-variable occurrences in $B$.*
4. *The free-variable occurrences in $(\mathcal{Q}x)A$ are the free-variable occurrences in $A$, except for occurrences of $x$.*

**Definition 2.3.10** *A variable occurrence is called* bound *if it is not free.*

**Definition 2.3.11** *A* sentence *(also called closed formula) of $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ is a formula of $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ with no free-variable occurrences.*

### 2.3.2. Types

The logic used in this thesis is *typed* which restricts the syntax. I.e. the possible values for constants and variables and the results of functions are restricted to a certain set of values.

The two types that are used in this thesis are *integer* ($\mathbb{Z}$) and *array of integer*.

The array of integer is a logical representation of an array of integer in a programming language.

**Definition 2.3.12** *An* array **arr** *is a list $\{x_0, \ldots, x_n\}$ where each $x_i \in \mathbb{Z}$ for $i \in \{0, \ldots, n\}$. The set $\mathcal{A}$ is the set that contains all elements of type array.*

**Definition 2.3.13** *The function* length $: \mathcal{A} \to \mathcal{N}_0$ *returns the length of the given array's list.*

*For all $arr \in \mathcal{A}$ the notations $length(arr)$ and $arr.length$ are equivalent.*

**Definition 2.3.14** *There exists an operator $[] : \mathcal{A} \times \mathcal{N}_0 \to \mathbb{Z}$ that for all $arr \in \mathcal{A}$ and for all $i \in \mathcal{N}_0$ where $0 \leq i$ and $i < arr.length$ $[](arr, i)$ returns the $i$-th element of arr's list.*

*The notations $[](arr, i)$ and $arr[i]$ are equivalent.*

### 2.3.3. Substitution

A *substitution* defines how variables can be replaced by terms. In the following, $\mathbf{T}$ is the set of all terms of a given language $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ and $\mathbf{V}$ is the set of its variables.

**Definition 2.3.15** *A* substitution *is a mapping $\sigma : \mathbf{V} \to \mathbf{T}$ from the set of variables to the set of terms.*

**Definition 2.3.16** *Let $\sigma$ be a substitution. Then we set:*

1. *$c\sigma = c$ for a constant symbol $c$*
2. *$[f(t_1, \ldots, t_n)]\sigma = f(t_1\sigma, \ldots, t_n\sigma)$ for an $n$-place function symbol $f$.*

**Definition 2.3.17** *The* support *of a substitution $\sigma$ is the set of variables $x$ for which $x\sigma \neq x$. A substitution has* finite support *if its support set is finite.*

**Definition 2.3.18** *Suppose $\sigma$ is a substitution having finite support; say $\{x_1, \ldots, x_n\}$ is the support, and for each $i = 1, \ldots, n$, $x_i\sigma = t_i$. $\{x_1/t_1, \ldots, x_n/t_n\}$ is the notation for $\sigma$, where $\{\}$ is the identity substitution.*

**Definition 2.3.19** *Let $\sigma$ be a substitution and $\sigma_x$ the substitution that is like $\sigma$ but does not change the variable $x$. That is for any variable $y$:*

$$y\sigma_x = \begin{cases} y\sigma & \text{if } y \neq x \\ x & \text{if } y = x \end{cases}$$

**Definition 2.3.20** *A substitution $\sigma$ is extended to formulæ as follows:*

1. *For the atomic cases:*
   - $[A(t_1, \ldots, t_n)]\sigma = A(t_1\sigma, \ldots, t_n\sigma)$
   - $\top\sigma = \top$
   - $\bot\ \sigma = \bot$.
2. $[\neg X]\sigma = \neg[X\sigma]$.
3. $(X \circ Y)\sigma = (X\sigma \circ Y\sigma)$ *for a binary symbol* $\circ$.
4. $[\forall x.X]\sigma = \forall x.[X\sigma_x]$.
5. $[\exists x.X]\sigma = \exists x.[X\sigma_x]$.

### 2.3.4. Semantics

Until now only the syntax of the logic is defined, i.e. at this point one can decide whether or not a given String is a well formed first order logical formula by examining its structure. At this point terms and formulæ have no meaning yet.

In contrast to other logics like propositional logic, where all formulæ are decidable, this is not possible for all FOL-formulæ. This fact is no real drawback because the undecidable formulæ are of such a complex structure that they are normally not of interest in practical issues.

First of all, a *model* is needed which determines of which *domain* constants and variables are and what the source and target set of functions are. Besides this, the model defines the *interpretation* which maps a meaning to the symbols of $\mathbf{C}$, $\mathbf{F}$ and $\mathbf{P}$. A mapping being called an *assignment* can be defined that associates elements of the domain to variables.

Therefore the model is introduced first and the assignment afterwards. The last step in defining the FOL's semantics are the associations of truth values to which we will come at the end of this section.

**Definition 2.3.21** *A* model *for the first-order language* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *is a pair* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *where:*

1. $\mathbf{D}$ *is a nonempty set, called the* domain *of* $\mathbf{M}$ *where* $\mathbf{D} \subseteq \mathbb{Z} \cup \mathcal{A}$.
2. $\mathbf{I}$ *is a mapping, called an* interpretation *that associates:*
   - *to every constant symbol* $c \in \mathbf{C}$, *some member* $c^{\mathbf{I}} \in \mathbf{D}$
   - *to every* $n$*-ary function symbol* $f \in \mathbf{F}$, *some* $n$*-ary function* $f^{\mathbf{I}} : \mathbf{D}^n \to \mathbf{D}$
   - *to every* $n$*-place predicate symbol* $p \in \mathbf{P}$, *some* $n$*-ary predicate* $p^{\mathbf{I}} \subseteq \mathbf{D}^n$

**Definition 2.3.22** *An* assignment *in a model* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *is a mapping* $\mathbf{A}$ *from the set of variables to the set* $\mathbf{D}$. *The image of the variable* $v$ *under an assignment* $\mathbf{A}$ *is denoted by* $v^{\mathbf{A}}$.

**Definition 2.3.23** *Let* $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ *be a model for the language* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$ *and let* $\mathbf{A}$ *be an assignment in this model. To each term* $t$ *of* $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$, *we associate a value* $t^{\mathbf{I},\mathbf{A}}$ *in* $\mathbf{D}$ *as follows:*

1. *For a constant symbol* $c$, $c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}}$
2. *For a variable* $v$,$v^{\mathbf{I},\mathbf{A}} = v^{\mathbf{A}}$
3. *For a function symbol* $f$,$[f(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}})$

As no statement is made how to decide the truth of a given formula, the methods are explained now. It is started by defining the set of truth values in the first step and some operators on it in the second step. Finally it is introduced how the truth value of a formula can be deduced.

**Definition 2.3.24** *The set of* truth values *is* $\mathbf{Tv} = \{true, false\}$.

**Definition 2.3.25**

1. *Negation*

$$\neg(X) : \mathbf{Tv} \to \mathbf{Tv} := \begin{cases} false & for\ X = true \\ true & for\ X = false \end{cases}$$

2. *Conjunction*

$$X \wedge Y : \mathbf{Tv}, \mathbf{Tv} \to \mathbf{Tv} := \begin{cases} true & for\ X = true \ \ and\ Y = true \\ false & else \end{cases}$$

3. *Disjunction*

$$X \vee Y : \mathbf{Tv}, \mathbf{Tv} \to \mathbf{Tv} := \begin{cases} false & for\ X = false \ \ and\ Y = false \\ true & else \end{cases}$$

Two more binary operators, i.e. $\Rightarrow$ and $\Leftrightarrow$, are abbreviations for the combination of known operators:

**Definition 2.3.26**

1. $X \Rightarrow Y : \mathbf{Tv}, \mathbf{Tv} \to \mathbf{Tv} := \neg X \vee Y$
2. $X \Leftrightarrow Y : \mathbf{Tv}, \mathbf{Tv} \to \mathbf{Tv} := (\neg X \wedge \neg Y) \vee (X \wedge Y)$

For later use, the concept of an x-variant has to be introduced.

**Definition 2.3.27** *Let $x$ be a variable. The assignment $\mathbf{B}$ in the model $\mathbf{M}$ is an $x$-*variant *of the assignment $\mathbf{A}$ provided $\mathbf{A}$ and $\mathbf{B}$ assign the same values to every variable except possibly $x$.*

**Definition 2.3.28** *Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for the language $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each formula $\Phi$ of $L(\mathbf{P}, \mathbf{F}, \mathbf{C})$, we associate a truth value $\Phi^{\mathbf{I},\mathbf{A}}$ (true or false) as follows:*

1. *For atomic cases,*
   - $[\mathbf{P}(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = true$ *, iff* $\langle t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}} \rangle \in \mathbf{P}^{\mathbf{I},\mathbf{A}}$,
   - $\top^{\mathbf{I},\mathbf{A}} = true$,
   - $\bot^{\mathbf{I},\mathbf{A}} = false$.
2. $[\neg X]^{\mathbf{I},\mathbf{A}} = \neg[X^{\mathbf{I},\mathbf{A}}]$.
3. $[X \circ Y]^{\mathbf{I},\mathbf{A}} = X^{\mathbf{I},\mathbf{A}} \circ Y^{\mathbf{I},\mathbf{A}}$.

4. $[(\forall x)\phi]^{\mathbf{I},\mathbf{A}} = true \leftrightarrow \phi^{\mathbf{I},\mathbf{B}} = true$ *for every assignment* $\mathbf{B}$ *in* $\mathbf{M}$ *that is an* $x$*-variant of* $\mathbf{A}$.

5. $[(\exists x)\phi]^{\mathbf{I},\mathbf{A}} = true \leftrightarrow \phi^{\mathbf{I},\mathbf{B}} = true$ *for some assignment* $\mathbf{B}$ *in* $\mathbf{M}$ *that is an* $x$*-variant of* $\mathbf{A}$.

**Definition 2.3.29** *A formula* $\phi$ *of* $L(\mathbf{P},\mathbf{F},\mathbf{C})$ *is* true in the model $\mathbf{M} = \langle \mathbf{D},\mathbf{I}\rangle$ *for* $L(\mathbf{P},\mathbf{F},\mathbf{C})$ *provided,* $\phi^{\mathbf{I},\mathbf{A}} = true$ *for all assignments* $\mathbf{A}$. *A formula* $\phi$ *is* valid *iff* $\phi$ *is true in all models of the language. A set* $S$ *of formulæ is* satisfiable *in* $\mathbf{M} = \langle \mathbf{D},\mathbf{I}\rangle$ *provided there is some assignment* $\mathbf{A}$ *(called a* satisfying assignment*) such that* $\phi^{\mathbf{I},\mathbf{A}} = true$ *to all* $\phi \in S$. $S$ *is* satisfiable *if it is satisfiable in some model.*

## 2.4. Notations on Quantifiers

There are several notations related to quantifiers that are of importance for this approach and that are introduced now:

- The *range* of a quantifier or quantified formula describes over which elements it is quantified.
- If the range is not infinite the quantifier has *bounds*, i.e. for all ranges, that are not infinite it is quantified over the set of elements that are between the lower and the upper bound.
- A *range predicate* is a predicate whose operator is '=', '$\leq$', '$\geq$', '$<$' or '$>$' and one of the operators argument is the current quantified variable. Bounds are a special kind of range predicates.
- The range predicates argument being not the quantified variable is called *range value.*
- A *range restriction* is a subformula that is part of the quantifiers matrix. It consits of one bound or a conjunction of two bounds, where one is an upper bound and the other is a lower bound. For a universal quantifier the range restriction is connected with an implication to the matrix ( $\forall x.((a < x \wedge x < b) \Rightarrow F)$ ) and for an existential quantifier it is connected with $\wedge$ ( $\exists x.((a < x \wedge x < b) \wedge F)$ ).

Example 2.4.1 shows some examples for those concepts.

**Example 2.4.1** *Both formulæ* $\forall x.p(x)$ *and* $\exists x.p(x)$ *have no range infos and therefore no bounds, that is their range is infinite. Therefore there is no range restriction.*

*In contrast the formulæ (1) and (2) have three range predicates, namely* $0 < x$, $x < 100$, *and* $x < 50$ *where* $0 < x$ *and* $x < 100$ *are bounds. The range restriction for both formulæ is* $(0 < x \wedge x < 100)$. *That is the quantified formulæ have finite range. Occurring range values in (1) and (2) are* 0, 50 *and* 100

$$\forall x.(0 < x \wedge x < 100) \Rightarrow ((x < 50 \Rightarrow p(x)) \wedge q(x)) \quad (1)$$

$$\exists x.0 < x \wedge x < 100 \wedge ((x < 50 \Rightarrow p(x)) \wedge p(x)) \quad (2)$$

## 2.5. Translation

As explained in Section 1 the translation of a specification to a program has to reflect the structure of the formula to be usable for this approach. An example for such an translation is the one that is used for the examples in this thesis which is introduced now. Other translation methods are usable as long as they are similar to the given one, i.e. they are not changing the structure of the input formula. The Oracle is a Java method with return type `boolean` and all the free variables and constants of the formula as parameter. Its `return` statement for a given formula $F$ is $\mathcal{T}(F)$ with the following rules for $\mathcal{T}(F)$ :

$$\mathcal{T}(F) := \begin{cases} \mathcal{T}_A(F) & \text{if } F \text{ atomic} \\ !\mathcal{T}(A) & \text{if } F = \neg A \\ \mathcal{T}(A)\&\&\mathcal{T}(B) & \text{if } F = A \wedge B \\ \mathcal{T}(A)||\mathcal{T}(B) & \text{if } F = A \vee B \\ \mathcal{T}_\forall(A, x) & \text{if } F = \forall x.A \\ \mathcal{T}_\exists(A, x) & \text{if } F = \exists x.A \end{cases}$$

$\mathcal{T}_A(F)$ translates atomic formulæ into equivalent expressions in Java, where atomic formulæ can be classified into the following two types:

- $=, <, \leq, >, \geq$
- predicates that represent a Java method of type boolean

In the first case the logical relational operators are replaced be their corresponding Java operator (`==`, `<`, `<=`, `>`, `>=`) and in the second case the method is called with the appropriate parameters.

$\mathcal{T}_\forall(\forall x.A, x)$ and $\mathcal{T}_\exists(\exists x.A, x)$ introduce a new boolean method with the signature `forall_ctr`$(min, max, v)$ and `exists_ctr`$(min, max, v)$, where $ctr$ is a consecutively numbered variable, $min$ is the lower bound of the quantification, $max$ is the upper bound and $v$ is a set of program variables representing the constants and free variables of the quantifiers matrix.

Their body is as follows:

```
1 boolean forall_ctr(
2       int min, int max, v){
3   int x = min;
4   while( T(A) && x < max){
5     x++;
6   }
7   return x == max;
8 }
```

Listing 4: Forall

```
1 boolean exists_ctr(
2       int min, int max, v){
3   int x = min;
4   while(! T(A) && x < max){
5     x++;
6   }
7   return !x == max;
8 }
```

Listing 5: Exists

# 3. Problem

To improve the quality of a program, as many errors as possible have to be eliminated. Before this can be achieved the failures in the code have to be revealed first which can be done using software testing. By extending the amount of code that is covered by test cases, which can be achieved by raising the number of test cases run, the chance of finding failures is increased.

The general drawback of testing, i.e. an increased need of time and human interaction, can be reduced by trying to generate test suites or part of them automatically instead of creating them manually. This can be done by using the specification's postcondition as source to derive a part of the test case, namely the oracle, automatically and save manpower. In this work, the oracle is generated out of the program's postcondition, which is represented as a FOL formula.

As mentioned in the introduction (see Section 1) specifications are not written to be executed and therefore the translation might lead to a program with slow execution time, which reduces the amount of test cases that can be run per hour. The main cause that leads to a blowup of runtime is the use of quantifiers because they are translated into loops in the oracle program.

One possible way of handling quantifiers would be to omit the quantification of variables in the specification and therefore substitute the quantified variable with another variable that has to be set up in the preamble of the test case. Although this is possible it wouldn't be wise as one wants to have a certain confidence in the test case which would be reduced by this operation. By simplifying the circumstances of a test case too much one loses this ability and the resulting tests have a lowered probability of finding errors.

A possible way to increase the confidence is the increase of amount of tests run which leads to a higher amount of different starting values for the variables.

But as the newly introduced variable might not influence the runtime behaviour of the IUT this would result in several test cases where the result of the method is exactly the same but only the value of the variable in the specification differs.

To avoid this overhead of redundant IUT executions and keep a high confidence in the test case, quantifiers are not omitted but processed in a logical correct way.

The problem is to find a formula with the same or similar[5] semantics like a given one which can be executed in less time. Let

- $\mathbf{Se}_F$ be the set of all formulæ, that are semantically equivalent to a given formula $F$
- $\mathcal{O}(F)$ be the oracle generated out of formula $F$
- $\mathbf{Rt}(p)$ be the time that is needed to execute the program p

The problem is to find a formula $F' \in \mathbf{Se}_F$ where $\mathbf{Rt}(\mathcal{O}(F')) < \mathbf{Rt}(\mathcal{O}(F))$ if such a formula $F'$ exists.

For example, For $F = true$ there exists no $F'$ with the given properties.

[5] describing an approximation of the original formula for certain values (see Section 1).

# 4. Approach

The idea, presented in this thesis, to reduce the runtime of testcases is to change the structure of the postcondition to achieve a more performant oracle. This approach is only applicable if the program that generates the oracle is using the given postcondition's structure without changes, i.e. the translation is done according to Section 2.5.

The starting point of this approach is a formula representing a program's postcondition where the only datatypes that are permitted in it are boolean and integer types and arrays of type integer. Instead of processing this formula by one huge mechanism the work is split into different independent modules where each is responsible for a single technique.

Before the different parts are explained, Algorithm 1 gives an overview of the whole procedure and the interaction of the singel modules. The algorithm is processed as long as at least one part is applicable to the given formula (lines 1,3). In the case that a preceding module can be applied on the result of a module this is possible as the main modules are executed in a loop.

**Algorithm 1** Main Algorithm for Optimising Runtime

```
 1: F_old := true;
 2: while F_old != F do
 3:     F_old := F.clone();
 4:     F := simplify(F);
 5:     F := negationNormalForm(F);
 6:     F := removeAbbreviation(F);
 7:     F := reducingRanges(F);
 8:     F := reducingScopes(F);
 9: end while
10: F := changeEvaluationOrder(F);
11: F := reorderFomulae(F);
```

## 4.1. Simplification

The goal of this part is the reduction of the formula's complexity by using well known simplification techniques. Not all of them are mentioned here but this mainly covers the application of boolean axioms like

$$\text{simplifyFormula}(F) := \begin{cases} A & \text{if } F = (A \vee (A \wedge B)) \\ A & \text{if } F = (A \wedge (A \vee B)) \\ A & \text{if } F = (A \vee A) \\ A & \text{if } F = (A \wedge A) \\ true & \text{if } F = (A \vee \neg A) \\ false & \text{if } F = (A \wedge \neg A) \\ \dots \end{cases}$$

The Algorithm 2 illustrates the approach, where the method `simplifyFormula()` tries to alter the given formula by applying the procedure mentioned above.

**Algorithm 2** simplify()

```
1: while Up_F.hasNext() do
2:     simplifyFormula(Up_F.next());
3: end while
4: return F
```

## 4.2. Negation Normal Form

In the first step, the formula is transformed into Negation Normal Form (NNF), i.e. the negation symbol ($\neg$) can only occur in front of an atomic formula. This normal form is chosen to simplify the following procedures.

The Example 4.2.1 and the Algorithm 3 illustrate the transformation into NNF. $T_F$ is traversed by $Down_F$ (lines 1,2) whereas it is checked if the current node represents negation and if its subnode is not an atomic formula (line 3). If both conditions are fulfilled, the method `distNeg()` alters the formula like defined in Definition 4.2.1. After termination, $F$ is in NNF and returned.

**Algorithm 3** negationNormalForm($F$)

```
1: while Down_F.hasNext() do
2:     Node tmp = Down_F.next();
3:     if tmp.op() == '¬' && !tmp.sub(0).isAtomic() then
4:         distNeg(tmp);
5:     end if
6: end while
7: return F
```

**Definition 4.2.1**

$$distNeg(F) := \begin{cases} distNeg(\neg A) \wedge distNeg(\neg B) & \text{if } F = \neg(A \vee B) \\ distNeg(\neg A) \vee distNeg(\neg B) & \text{if } F = \neg(A \wedge B) \\ \exists x.distNeg(\neg A) & \text{if } F = \neg\forall x.A \\ \forall x.distNeg(\neg A) & \text{if } F = \neg\exists x.A \\ F & \text{else} \end{cases}$$

**Example 4.2.1** *Let $p(), q(), r()$ be well defined predicates. The transformation of $F$ is done with the following steps, where the used rule is shown after the '$\|$':*

1. $F = \neg\exists x.(p(x) \wedge (q(x) \vee r(x))) \; \| \neg\forall x.A \twoheadrightarrow \exists x.\neg A$
2. $F = \forall x.\neg(p(x) \wedge (q(x) \vee r(x))) \; \| \neg(A \wedge B) \twoheadrightarrow \neg A \vee \neg B$
3. $F = \forall x.(\neg p(x) \vee \neg(q(x) \vee r(x))) \; \| \neg(A \vee B) \twoheadrightarrow (\neg A \wedge \neg B))$
4. $F = \forall x.(\neg p(x) \vee (\neg q(x) \wedge \neg r(x)))$

## 4.3. Remove Abbreviations

One main reason that leads to an increase in runtime is the occurrence of quantifiers in the formula which are represented by loops in the oracle.

A special case where it is possible to get rid of the quantifier without changing the semantics occurs if the quantifier was introduced as an abbreviation for a term, i.e. the formula has the following structure:

$$\exists x.(\cdots \wedge (x = t) \wedge \dots)$$

This use of the existential quantifier as an abbreviation of an complex term simplifies the understanding of the specification as in all places where the complex term $t$ has been before there's only the symbol $x$ now.

After removing this abbreviations, there is one loop less that must be executed which results in a reduced runtime.

The Algorithm 4 illustrates the main steps how abbreviations are detected and removed where Algorithm 5 shows how the abbreviations are determined in detail.

$T_F$ is traversed from the root to the leaves and it is checked if the current node represents an existential quantification (line 3).

If so, the quantified variable is saved (line 4). Afterwards a equation that contains this variable is searched in the node's subtree, i. e. the representation of the quantifiers matrix (Line 5).. This method (see Algorithm 5) checks if the current node is an

**Algorithm 4** removeAbbreviation($F$)

```
 1: while Down_F.hasNext() do
 2:    Node tmp = Down_F.next();
 3:    if tmp.op() == '∃' then
 4:       Variable quanVar = tmp.getQuanVar();
 5:       Equation eq = getEq(tmp.sub(0),quanVar);
 6:       if eq != NULL then
 7:          if eq.sub(0)== quanVar then
 8:             tmp = substitute(eq.sub(0),eq.sub(1),tmp.sub(0));
 9:          else
10:             tmp = substitute(eq.sub(1),eq.sub(0),tmp.sub(0));
11:          end if
12:       end if
13:    end if
14: end while
15: return F
```

equation (line 1) or the conjunctive operator (line 6). In all other cases the method returns `NULL`. If the node is an equation lines 2 and 3 return this equation if it contains the quantified variable. In the case that the node is an '$\wedge$' the method is called recursively on the left subformula (line 7). If this call returns not `NULL` the result is returned in line 11. Lines 8,9 execute the recursive call on the right subformula and return its result.

Back in `removeAbbreviation()` (see Algorithm 4) line 6 checks if the result of `getEq()` is `NULL`. In this case no abbreviation was found in this quantified formula's

matrix and the iteration continues. In the other case line 7 checks if the quantified variable is on the left or the right side of the equation which results in call of the method `substitute(old,new,formula)` with the appropriate parameters in lines 8 or 10, so that all occurrences of the quantified variable are replaced by the term it is equal to.

`substitute(old,new,formula)` replaces all occurrences of `old` in `formula` with `new` and works like the logical substitution (see Section 2.3.3) $[formula]\{old/new\}$.

`tmp.getQuanVar()` returns the variable that is quantified in the node tmp.

**Algorithm 5** getEq(node,var)

```
 1: if node.op() == '=' then
 2:    if node.sub(0) == var || node.sub(1) == var then
 3:       return node;
 4:    end if
 5: else
 6:    if node.op() == '∧' then
 7:       result = getEq(node.sub(0),var);
 8:       if result == NULL then
 9:          return getEq(node.sub(1),var);
10:       else
11:          return result;
12:       end if
13:    end if
14: end if
15: return NULL;
```

Example 4.3.1 shows the benefit of this procedure.

**Example 4.3.1** *In the following example the formula*

$$\exists i.(i = a * a \wedge (a < 0 \Rightarrow result = -i) \wedge (a \geq 0 \Rightarrow result = i))$$

*is given, which defines the postcondition of a method that calculates the square for a given positive number a and the negative square if a is negative. The quantified variable i is introduced to prevent writing $a * a$ multiple times. The oracle resulting from this formula is shown below (see Listing 6) and contains a loop which increases the runtime.*

*The application of Algorithm 4 leads to an elimination of the existential quantifier and the resulting formula is:*

$$a < 0 \Rightarrow result = -(a * a) \wedge a \geq 0 \Rightarrow result = a * a$$

*The translation of this formula into an oracle contains only a case distinction and one equality check which results in a shorter runtime (see Listing 7).*

## 4.4. Reduce Range

As quantified formulæ are the main problem in achieving a short execution time and it is not possible to get rid of them with the method in Section 4.3 in all cases,

```
1  boolean oracle(int a){
2    return exists_0(MIN_INT,MAX_INT,a); }
3
4  boolean exists_0(int min, int max, int a)
5    int x = min;
6    while(! (x == a*a &&
7               (! a < 0 || result == -x) &&
8               (a<0 || result == x)
9            ) && x < max){ x++; }
10 return !x == max; }
```

Listing 6: Not optimised oracle for $a^2 * a/|a|$

```
1  boolean oracle(int a){
2    return ((! a < 0 || result == -a*a) &&  (a<0 || result == a*a);
3  }
```

Listing 7: Optimised oracle for $a^2 * a/|a|$

another optimisation method is introduced now. It aims at reducing the range of the quantification and thus the number of loop iterations by trying to find range predicates in the given quantifier's matrix and use them to create new ranges. A side effect is the shortening of the quantifier's matrix which results in less statements to be executed per loop iteration. Therefore it looks for range predicates in the given matrix and tries to divide the range into disjunctive sections. This approach assumes that if a quantified formula contains at least 2 range predicates, they are bounds and therefore the quantifier has a finite range. If there are less then 2 range predicates, `MIN_INT` and `MAX_INT` are introduced as bounds. Algorithm 6 shows a way to cut down the range which is illustrated in Example 4.4.1.

**Algorithm 6** reducingRanges(F)

```
1: while Down_F.hasNext() do
2:    Node tmp = Down_F.next();
3:    if tmp.op() == '∃' || tmp.op() == '∀' then
4:       Variable quanVar = tmp.getQuanVar();
5:       List preds = findRangePreds(tmp.sub(0),quanVar);
6:       if preds.length() < 2 then
7:          preds.add(MIN_INT);
8:          preds.add(MAX_INT);
9:       end if
10:      tmp = rewrite(tmp.sub(0),preds,var,tmp.op());
11:   end if
12: end while
13: return (F)
```

The tree is iterated from the root to the leaves (lines 1,2). If a node is a quantifier (line 3), all range predicates in its subtree are collected by `findRangePreds()` (see Algorithm 7).

**Algorithm 7** findRangePreds(node,var)

```
1: List ranges;
2: if node.op() ∈ {'=','≤','≥','<','>'} then
3:    if node.sub(0) == var || node.sub(1) == var then
4:       ranges.add(node);
5:    end if
6: else if node.op() ∈ {'∧','∨'} then
7:    ranges.add(findRangePreds(node.sub(0),var))
8:    ranges.add(findRangePreds(node.sub(1),var))
9: end if
10: return ranges
```

This methods stores the current node in the list *ranges* if it is a range predicate (lines 2-5) or adds the results of calling itself on both subformulæ (lines 6-9) to *ranges* if the node is a junctor.

*ranges* is then returned to the method `rewrite()` (see Algorithm 8) that creates a new formula .

**Algorithm 8** rewrite(matrix,bounds,var,quan)

```
1: List comps = new List();
2: for i = 0 to bounds.length()-1 do
3:    for j = i to bounds.length() do
4:       comps.add(newFormula('<', bounds.get(i), bounds.get(j));
5:    end for
6: end for
7: Formula pos, neg, not, curr = true;
8: for i = comps.length()-1 to 0 do
9:    pos = newFormula('⇒', comps.get(i), curr);
10:   not = newFormula('¬', comps.get(i));
11:   neg = newFormula('⇒', not, curr);
12:   curr = newFormula('∧', pos, neg);
13: end for
14: Map occ = new Map(bounds.length());
15: for i = 0 to bounds.length() do
16:    occ.put(bounds.get(i),0);
17: end for
18: return createConclusion(curr,matrix,occ,var,quan);
```

To create new disjoint ranges the range values need to be sorted. As they can be program variables that are not instantiated while the formula is created, it is not possible to create the right order directly. Therefore a set of case distinctions has to be established in the formula to introduce a generic way of sorting the range values.

The first step of this approach is to create predicates to compare the range values with each other and store those predicates in a list (lines 1 to 6). This list is iterated from the last to the first element in lines 8 to 13 to create one formula that describes

all the possible relations between the bounds by using conjunctions of implications (lines 9 to 12).

In the tree representation of the formula each path through the tree that doesn't contain the left hand side of an implication defines another ordering of the range values.

This procedure introduces some combinations as well that are not possible but they don't harm this approach as they are going to disappear by simplification later.

In the next step the map *occ* is introduced which maps each range value to an integer (line 14). This is needed for the sorting process later on which is based on counting occurrences of range values. The value is then set to zero for each entity of *occ* (lines 15 to 17). The formula being returned to `reducingRanges()` is then generated by the method `prepareConclusion()` (see Algorithm 9).

**Algorithm 9** prepareConclusion(newForm,oldMatr,occ,var)

```
 1: if newForm == '∧' then
 2:    Formula left = prepareConclusion(newForm.sub(0),oldMatr,occ);
 3:    Formula right = prepareConclusion(newForm.sub(1),oldMatr,occ);
 4:    return left ∧ right;
 5: else if newForm == '⇒' then
 6:    Bound b1 = newForm.sub(0).sub(0);
 7:    Bound b2 = newForm.sub(0).sub(1);
 8:    if sub == '¬' then
 9:       occ.put(b1, occ.get(b1) - 1);
10:       occ.put(b2, occ.get(b2) + 1);
11:    else
12:       occ.put(b1,occ.get(b1) + 1);
13:       occ.put(b2,occ.get(b2) - 1);
14:    end if
15:    return prepareConclusion(newForm.sub(1),oldMatr,occ);
16: else if newForm == 'true' then
17:    return createConclusion(oldMatr,occ,var,quan);
18: end if
```

The main task of `prepareConclusion()` is to determine the ordering of the range values for a certain path through the tree where only paths including right subtrees of an implication are of interest. This methods main idea is to store the occurrences of the different range values in the map *occ* and establish an ordering on those values. The occurrences are counted in the following way:

- If a range value occurs on the left hand side of $<$ its corresponding value in *occ* is increased (line 12).
- If it occurs on the right hand side it is decreased (line 13).
- If the range predicate is negated, both operations are swapped (lines 9,10).

Lines 1 to 4 call the method recursively on both subformulæ of a conjunction, where the current values of *occ* are used as a parameter.

If the formula true is reached, which has been introduced in the beginning by `rewrite()`, the iteration through the tree is finished and a newly created formula

is returned. This formula is created by the method `createConclusion()` (see Algorithm 10) which uses the information on the range values for the current path through the tree which are stored in the parameter *occ*

**Algorithm 10** createConclusion(oldMatr,occ,var,quan)

```
 1: List sortedB = sortBounds(occ);
 2: if quan == '∀' then
 3:     Operator op == '⇒';
 4: else
 5:     Operator op == '∧';
 6: end if
 7: Formula b1 = newFormula(<, sortedB.get(0), var);
 8: Formula b2 = newFormula(<, var, sortedB.get(1)-1);
 9: Formula bounds = newFormula('∧', b1, b2);
10: Formula matr = newFormula(op, bounds, oldMatr);
11: Formula curr = newFormula(quan, var, matr);
12: for i = 1 to occ.length()-1 do
13:     b1 = newFormula(<, sortedB.get(i), var);
14:     b2 = newFormula(<, var, sortedB.get(i+1)-1);
15:     bounds = newFormula('∧', b1, b2);
16:     matr = newFormula(op, bounds, oldMatr);
17:     curr = newFormula('∧', curr, newFormula(quan, var, matr));
18: end for
19: return curr;
```

The method `createConclusion()` uses the ordering of the range values stored in *occ* to create quantified formulæ with disjunctive ranges. Their matrix consists of the matrix of the original quantified formula. In the first step (line 1) the method `sortBounds()` creates a list of the range values that is ordered descending by the integer that is stored in *occ* for each range value. In lines 2 to 6 the operator that connects the range restriction to the remaining matrix is selected which is $\Rightarrow$ if the quantifier is a universal quantifier and $\wedge$ if it's a existential quantifier. In the next step the first two range values are taken out of the list and are used to create the range predicates for the upper and lower bound (lines 7,8) which are used to create the range restriction (line 9). The range restriction and the original matrix of the quantified formula are then conjunctively joined to the new matrix of the quantifier (line 10). In line 11 the quantified formula with the new matrix is created. At this point the new formula consists only of one bounded quantified formula. By using a loop to iterate through all range values and repeat the steps of creating a bounded quantified formula the result is a conjunction of quantified formulæ with disjoint ranges (lines 12 to 18). This formula is then returned to `createConclusion()` which returns it to `rewrite()`.

After the application of `reducingRanges()` the formula consists of conjunctions of implications, where each conjunct describes one ordering of the range values and the corresponding disjunctive quantified formulæ. This whole procedure can lead to a blowup of the formula which can be reduced by using simplification (See Section 4.1).

```
1
2  boolean oracle(int a, int b, int c){
3    return forall_0(MIN_INT,MAX_INT,a,b,c); }
4
5  boolean forall_0(int min, int max, int a, int b, int c )
6    int x = min;
7    while((
8              0 > i || a+1 > i || i < c+1 || i < b+1 ||
9              old[i] = ar[i]
10           ) && x < max ){
11     x++; }
12   return x == max;
13 }
```

Listing 8: Not optimised oracle for a method that checks if all elements of an array stay unchanged

**Example 4.4.1** *In the formula (3)*

$$\forall i.((0 > i) \vee (a + 1 > i) \vee (i < ar.length + 1) \vee (i < b + 1) \vee (old[i] = ar[i])) \quad (3)$$

*there are the four range predicates* $(0 > i), (a + 1 > i), (i < ar.length + 1), (i < b + 1)$ *and the range values* $(0), (a + 1), (ar - length), (b + 1)$*. Without any optimisations the translated formula looks like Listing 8.*

*The loop iterates from* `MIN_INT` *to* `MAX_INT` *which results in a lot of obsolete steps as only some values, i.e. the values where* $i$ *is larger than* 0 *and* $a$ *but smaller than* $b$ *and* $ar.length$*, are of real interest here. In the worst-case there are 6 evaluations that are executed in every step.*

*To optimise the formula Algorithm 6 is applied and some intermediate steps of the application are shown to clarify the procedure:*

*It is started by computing the list* $comps = [(0 < a), (0 < b), (0 < ar.length), (a < b), (a < ar.length), (b < ar.length)]$.

*Then the formula that represents an ordering is created. The recursive steps that build the formula are shown below. To improve the readability of the whole formula the subformulæ of the previous are replaced by the variable* $curr_x$ *where* $x$ *is the iteration.*

$$\begin{aligned}
curr_0 &= ((b < ar.length) \Rightarrow true) \wedge (\neg(b < ar.length) \Rightarrow true) \\
curr_1 &= ((a < ar.length) \Rightarrow curr_0) \wedge (\neg(a < ar.length) \Rightarrow curr_0)) \\
curr_2 &= ((a < b) \Rightarrow curr_1) \wedge (\neg(a < b) \Rightarrow curr_1)) \\
curr_3 &= ((0 < ar.length) \Rightarrow curr_2) \wedge (\neg(0 < ar.length) \Rightarrow curr_2)) \\
curr_4 &= ((0 < b) \Rightarrow curr_3) \wedge (\neg(0 < b) \Rightarrow curr_3)) \\
curr_5 &= ((0 < a) \Rightarrow curr_4) \wedge (\neg(0 < a) \Rightarrow curr_4))
\end{aligned}$$

*After the creation of the formula the map*

$$\begin{aligned} occ = [ & \\ & (`0' \to 0), \\ & (`a+1' \to 0), \\ & (`ar.length' \to 0), \\ & (`b+1' \to 0) \\ & ] \end{aligned}$$

*is initialised and then the right values are calculated for the paths trough the formula. As example for the calculation the steps for the path in Figure 2 are shown in Table 1 where the resulting map is:*

$$\begin{aligned} occ = [ & \\ & (`0' \to 3), \\ & (`a+1' \to 1), \\ & (`ar.length' \to -1), \\ & (`b+1' \to -3) \\ & ] \end{aligned}$$

| Step | $`0'$ | $`a+1'$ | $`ar.length'$ | $`b+1'$ |
|---|---|---|---|---|
| 0 | 1 | $-1$ | 0 | 0 |
| 1 | 2 | $-1$ | 0 | $-1$ |
| 2 | 3 | $-1$ | $-1$ | $-1$ |
| 3 | 3 | 0 | $-1$ | $-2$ |
| 4 | 3 | 1 | $-2$ | $-2$ |
| 5 | 3 | 1 | $-1$ | $-3$ |

Table 1: Single steps for determining the ordering

*The list* $\{`0', `a+1', `ar.length', `b+1'\}$ *is returned by the method* `sortBounds(occ)`. *This list is then iterated to create the new formula:*

$$\begin{aligned} & (\forall i.(0 < i \wedge i < a) \Rightarrow \\ & ((0 > i) \vee (a+1 > i) \vee (i < ar.length + 1) \vee (i < b+1) \vee (old[i] = ar[i]))) \wedge \\ & (\forall i.(a+1 < i \wedge i < ar.length - 1) \Rightarrow \\ & ((0 > i) \vee (a+1 > i) \vee (i < ar.length + 1) \vee (i < b+1) \vee (old[i] = ar[i]))) \wedge \\ & (\forall i.(ar.length < i \wedge i < b) \Rightarrow \\ & ((0 > i) \vee (a+1 > i) \vee (i < ar.length + 1) \vee (i < b+1) \vee (old[i] = ar[i]))) \end{aligned}$$

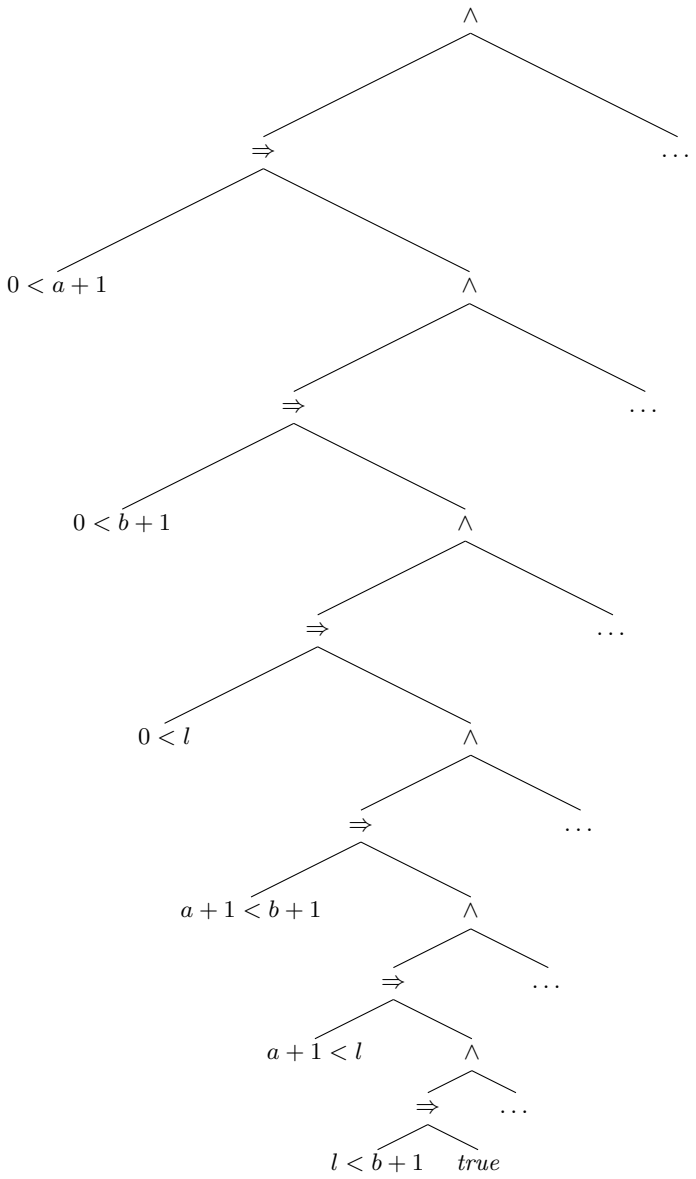



Figure 2: One path through the formula ($l$ is used instead of $ar.length$)

*After simplifying the whole formula, that is the case distinction and the newly created conclusion*

$$
\begin{array}{l}
(\\
\quad (0 < a + 1 \Rightarrow\\
\quad\quad (0 < b + 1 \Rightarrow\\
\quad\quad\quad (0 < ar.length \Rightarrow\\
\quad\quad\quad\quad (a + 1 < b + 1 \Rightarrow\\
\quad\quad\quad\quad\quad (a + 1 < ar.length \Rightarrow\\
\quad\quad\quad\quad\quad\quad (ar.length < b + 1 \Rightarrow\\
\quad\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad\quad \forall i.\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (0 < i \wedge i < a) \Rightarrow\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (old[i] = ar[i])\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad )\wedge\\
\quad\quad\quad\quad\quad\quad\quad\quad \forall i.\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (a + 1 < i \wedge i < ar.length - 1) \Rightarrow\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (old[i] = ar[i])\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad )\wedge\\
\quad\quad\quad\quad\quad\quad\quad\quad \forall i.\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad (\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (ar.length < i \wedge i < b) \Rightarrow\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad (old[i] = ar[i])\\
\quad\quad\quad\quad\quad\quad\quad\quad\quad )\\
\quad\quad\quad\quad\quad\quad\quad )\\
\quad\quad\quad\quad\quad\quad ) \wedge \ldots\\
\quad\quad\quad\quad\quad ) \wedge \ldots\\
\quad\quad\quad\quad ) \wedge \ldots\\
\quad\quad\quad ) \wedge \ldots\\
\quad\quad ) \wedge \ldots\\
\quad )\wedge\\
\quad (\neg 0 < a + 1 \Rightarrow\\
\quad\quad \ldots\\
\quad )\\
)
\end{array}
$$

*As the optimised formula has a larger size than the original one, leading to a larger translation, and the formula allows to see the effects on the runtime of the generated program, the latter is not shown.*

*Although the optimised program contains more lines of code, one can see that the worst case runtime is shorter than the original one, as there are only 2 expressions to be checked per loop iteration. This only holds if the range of the quantified formula contains a large amount of elements.*

## 4.5. Reduce Scope

After trying to get rid of quantified formulæ and reducing the range of the quantified variable the next step is trying to reduce the size of the matrix and getting rid of redundant evaluations. Therefore the matrix of the quantified formula is examined for formulæ that don't need to be in the scope of the quantifier. If such a formula is found we try to pull it out of the scope. The procedure to decide if a formula can be pulled out and how it can be done is described by Algorithm 11 and illustrated in Example 4.5.1.

Like in the two preceding modules, the tree is iterated from the root to the leafs. If a node is a quantifier its subtree is examined by the method `restructure()`. If the subtree consists only of a conjunction of disjunctions (CNF) or disjunctions of conjunctions (DNF) it is checked if one conjunct (disjunctive) contains the bounded variable. If not, the whole subtree is pulled out and prepended conjunctively (disjunctively) before the quantified formula. If the formula is neither in CNF nor DNF a decision can only be made on the top level operator of the matrix. I.e. the tree is descended until the nodes' type differ from the first node's type. At this level all the subtrees are checked for the bounded variable. If it does not occur the subtree is pulled out like before.

**Algorithm 11** reducingScopes()

```
1: while Down_F.hasNext() do
2:    Node tmp = Down_F.next();
3:    if tmp == ∃ || tmp == ∀ then
4:       Variable quanVar = tmp.getQuanVar();
5:       restructure(tmp);
6:    end if
7: end while
8: return (F)
```

**Example 4.5.1** *The following simple formula:*

$$\forall i.(p(a) \vee q(b) \vee r(i))$$

*is translated into the program (see Listing 9)*

```
1 int i = MIN_INT;
2 while((p(a) || q(b) || r(i)) &&
3       i < MAX_INT){
4       i++;
5 }
```

Listing 9: Not optimised oracle for a constructed example

```
1 return p(a) || p(b) || evalR();
2
3 boolean evalR(){
4       int i = MIN_INT;
5       while(r(i) && i < MAX_INT){
6             i++;
7       }
8 return i = MAX_INT;
9 }
```

Listing 10: Optimised oracle for a constructed example

*The two predicates* `p(a)` *and* `q(b)` *are in the scope of the quantifier but they need no quantification. Therefore they can be pulled out what leads to the formula*

$p(a) \vee q(b) \vee \forall i.r(i)$

*respectively the program (see Listing 10)*

*In this optimised form it is possible that the loop is not evaluated. Even if it is executed, there are two evaluations less to be done per loop iteration.*

## 4.6. Change Evaluation Order

After all applications of the above mentioned modules are finished, the algorithm is neither capable of manipulating the range nor the scope of the quantifiers anymore. One remaining possibility that might lead to a shorter runtime is the alteration of the quantifier's evaluation order. This is achieved by trying to find a significant instance, like boundaries, for the quantified variable in the postcondition and instantiate the matrix with it. Depending on the type of quantifier, the instantiated formula is conjunctively (if the quantifier is $\forall$) respectively disjunctively (if the quantifier is $\exists$) added before the original formula. This offers the possibility of a fast success (existential quantification) or fail (universal quantification) of the test.

Due to the fact that quantified formulæ can be nested, this procedure might lead to a blowup in space, i.e. more formulæ in the scope of the quantifier, which can have a negative influence on the runtime of the oracle as well. This fact and the possibility that some instances might be evaluated twice (explicitly before the loop and again during loop iteration) are accepted to gain the possibility of skipping a loop execution.

The Algorithm 12 shows the procedure, which is illustrated in Example 4.6.1. If a quantifier is found during the top-down iteration, its quantified variable is extracted and its matrix is examined by `findSignificantInstances()`. This method searches for predicates containing the quantified variable and alters the relation with sound mathematical operations so that the bounded variable is the only element on one of the operator's sides.

In the next step `createInstantiations()` instantiates the matrix with the values found by `findSignificantInstances()`. This is achieved by replacing all occurrences of the quantified variable in the matrix with the collected instances and storing the resulting formula. If $n$ values have been found, the matrix is instantiated $n$ times and therefore $n$ new formulæ are added.

`createCombinedTree()` connects all the created formulæ and the original quantified formula in such way, that the quantified formula is the most right element, i.e. during evaluation, this formula is evaluated last. The newly generated subtree is used to replace the original formula and the algorithm moves on with the iteration.

**Algorithm 12** changeEvaluationOrder()

```
1: while Down_F.hasNext() do
2:    Node tmp = Down_F.next();
3:    if tmp == ∃ || tmp == ∀ then
4:       Variable quanVar = tmp.getQuanVar();
5:       SetOfIntegers inst = findSignificantInstances(tmp,quanVar);
6:       SetOfNodes newNodes = createInstantiations(inst,tmp);
7:       createCombinedTree(tmp,newNodes);
8:    end if
9: end while
10: return (F)
```

**Example 4.6.1** *The Java program shown in Listing 11 is specified by the formula*

$$\forall\ int\ i.((0 \leq i \wedge i < ar.length) \Rightarrow ar[x] == 1)$$

```
1 public int[] setToOne(int [] ar){
2         for(int i = 0; i < ar.length-1;i++){
3                 ar[i]=1;
4         }
5 }
```

Listing 11: An incorrect implementation of a method that sets all elements of an array to 1

*This formula offers 2 significant instantiations*

- 0
- $ar.length - 1$

```
1 int i = 0;
2 while (ar[i] == 1 && i < ar.length) {
3         i++;
4 }
```

Listing 12: Not optimised oracle for the example

*The translation of this formula is shown in Listing 12. This oracle has to execute the loop ar.length-1 times to get the chance of finding the error in the implementation.*

*If the found significant instantiations are use the following formula can be created.*

$$ar[0] == 1 \wedge ar[ar.length - 1] == 1 \wedge \forall\ int\ i.((0 \leq i \wedge i < ar.length) \Rightarrow ar[x] == 1)$$

*If this formula is translated into an oracle and executed, the possibility of finding the error occurs while executing the second expression. Therefore the execution of the loop can be skipped.*

## 4.7. Reordering of Subformulæ

The last step in the optimisation procedure deals with the order of the formula's subformulæ. By using a given heuristic, costs can be attached to the subformulæ. Those costs are then used the reorder the subformulæ, so that the cheapest formula is evaluated first.

The idea of this approach is to prevent the evaluation of expensive subformulæ, in cases where the evaluation of a cheap subformula already determines the result of the formula.

The heuristic used in this thesis is rather simple[6] and considers only the number of quantified formulæ in the formula and its number of subformulæ. It is shown in Listing 13. For every node in the formula's subtree the costs are increased by one. For each of the subtree's nodes, that represent a quantifier, costs is increased by 100 additionally.

The procedure to reorder the subformulæ is shown in algorithm 14 and illustrated in Example 4.7.1. Two new operators are introduced, namely $\wedge_{mul}$ and $\vee_{mul}$. Their only difference to $\wedge$ and $\vee$ is their arity which is not 2 but arbitrary. I.e. a formula that consists of a conjunction (disjunction) of conjunctions (disjunctions) can be represented with only one operator instead of many. For example $A \wedge B \wedge C \wedge D$ is equivalent with $\wedge_{mul}(A, B, C, D)$. So if a subformular's operator is a binary junctor, it's converted to its n-ary counterpart (lines 1 to 5).

In the next step (line 6) the method `calcCosts()` (see Algorithm 13) is used to calculate the costs for the given subformulæ. The connection between formula and corresponding costs is implemented by using a map (formula -> costs). In line 7 the

[6] This heuristic was chosen because time constraints forbade proper investigations of heuristics that promised a higher benefit (see Section 8).

**Algorithm 13** calcCosts(Formula f)

```
1: int costs=0;
2: while Down_f.hasNext() do
3:    Node tmp = Down_f.next();
4:    costs = costs +1;
5:    if tmp == ∃ || tmp == ∀ then
6:       costs = costs + 100;
7:    end if
8: end while
9: return (costs)
```

subterms are sorted in ascending order according to their costs. The subformulæ are then transformed back to use only binary junctors in such way, that the first element of the list is the leftmost element of the junction (see line 8). I.e. the cheapest formula will be evaluated first.

**Algorithm 14** reorderSubformulae(Formula f)

```
1: Node op = Down_f.next();
2: while Down_f.hasNext() do
3:    Node tmp = Down_f.next();
4:    if op == ∧ || op == ∨  then
5:       List subs = toMulti(tmp);
6:       Map costs = getCosts(subs);
7:       sort(subs,costs);
8:       toBinary(subs);
9:    end if
10: end while
```

**Example 4.7.1** *The evaluation of the formula*

$$\forall x.(\exists y.(p(x,y)) \wedge r(x)) \wedge (a < b \Rightarrow p(a,b)) \wedge (s(a))$$

*starts with the leftmost subformula. That is a nested quantified formula has to be evaluated which is expensive. If the predicate $s(a)$ evaluates to false this previous evaluation was a waste of time.*

*After* `reorderSubformulae` *is applied the formula has the following form:*

$$(s(a)) \wedge (a < b \Rightarrow p(a,b)) \wedge \forall x.(r(x) \wedge \exists y.(p(x,y)))$$

*If the predicate $s(a)$ evaluates to false now, no expensive evaluation are done and therefore no time is being wasted.*

# 5. Implementation

## 5.1. KeY

The implementation of the developed ideas, which is called *Optimiser*, is integrated into the KeY system [KeY, Beckert et al., 2007]. KeY is an automated theorem prover developed at the University of Karlsruhe, University of Koblenz-Landau and Chalmers Institute of Technology and is written in Java. It was created as a tool for verification of JavaCard programs and works by applying a sequent calculus on formulæ in Java Dynamic Logic [Beckert, 2000]. Although this thesis is not dealing with verification, KeY has the following features that make it attractive to be used as a base for the implementation of the described ideas:

- An import mechanism, that allows to use JML, OCL and logical formulæ as input for specifications.
- An export mechanism, for exporting specifications in JML, OCL or as logical formulæ and for generating oracles in Java.
- An adjustable mechanism for rewriting formulæ (*Rules Applicator*).
- A predefined set of rules for the rewriting mechanism to realise logical simplification.
- The ability to generate test cases, respectively test suites which can be used to evaluate this approach [Engel, 2006].
- KeY is licenced under GPL which guarantees access to the sources and allows their modification.

### 5.1.1. Overview

At the current state, the implementation of the mentioned ideas is not finished. Therefore not all the implementation issues are discussed in the following sections.

The integration is done with mainly three interfaces between KeY and the Optimiser which can be seen in Figure 3.

In the first step the specification is given from KeY to the Optimiser that processes the formula. Therefore KeY's Rules Applicator is used with specially defined rule sets for the different optimisation techniques(see Section 5.1.3) and some additional operations are executed. Latter are the limitation of the quantifier's bounds (see Section 5.1.4) and the reordering of subterms (see Section 4.7). After all possible algorithms have been applied the formula is given back to KeY and can be exported into the wanted output format.

Due to the fact that multiple output formats are supported, this tool can be used in combination with other tools in a flexible way.

As the export part is not finished yet, the resulting formula can only be printed to the screen.

For the sake of simplicity figures and diagrams show only parts of the system. Explanations of modifications of the KeY source code are simplified and rough as well.

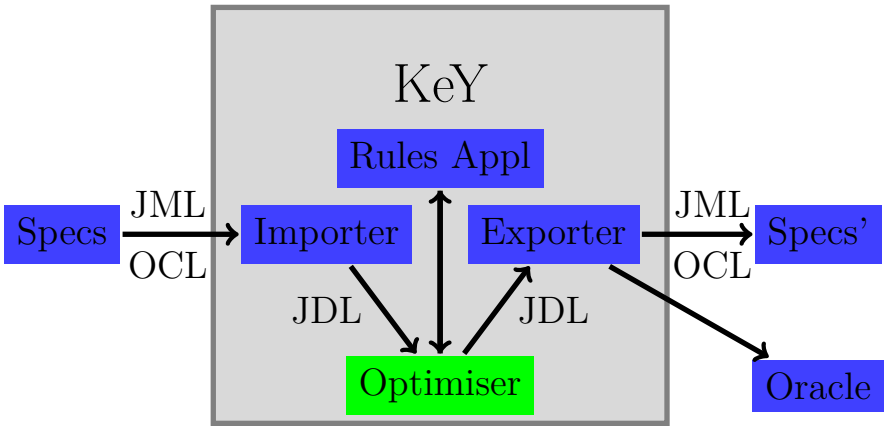



Figure 3: Layout of the approach

### 5.1.2. Import

KeY has several possibilities to import a formula. The key to turn 'some data' into a proof that can be worked on by KeY is the `ProofOblInput` interface. There are several classes that implement this interface with different nuances. A class that implements `ProofOblInput` is called proof obligation.

KeY has a several possibilities to import a formula. The key to turn 'some data' into a proof that can be worked on by KeY is the `ProofOblInput` interface. There are several classes that implement this interface with different nuances. A class that implements `ProofOblInput` is called proof obligation.

The class `KeYUserProblemFile`, for instance, transforms the formulæ from a loaded .key-file into the appropriate proof. The class `EnsuresPostPO` on the other hand combines the source code and the JML specifications that have been extracted from a .java-file into a proof to verify that the code s correct with respect to the specification. Figure 4 gives a rough overview over the classes that are involved in opening a .java-file to proof the correctness of a Java method with respect to the specification.

The `ProblemLoader` is initiated with the Java-file, that contains the source and the JML specification. Then the method `doWork` in `ProblemLoader` is called. Its first step is to initialise a new `ProblemInitialiser`. Then both objects do some operations to prepare the following steps. The `POBrowser` is called afterwards. It then creates a new `EnsuresPostPO` object. This object then uses the given information to create a new proof. This proof is then added to a newly created `ProofAggregate` object which is returned.

Although there are several different proof obligations, there was none to start a new proof with a formula which can be selected from a proof that is already loaded. Therefore the class `TermProofObl` was written. It implements `ProofOblInput` and creates a new proof to show that $F \Rightarrow false$ holds for a given formula $F$. This proof

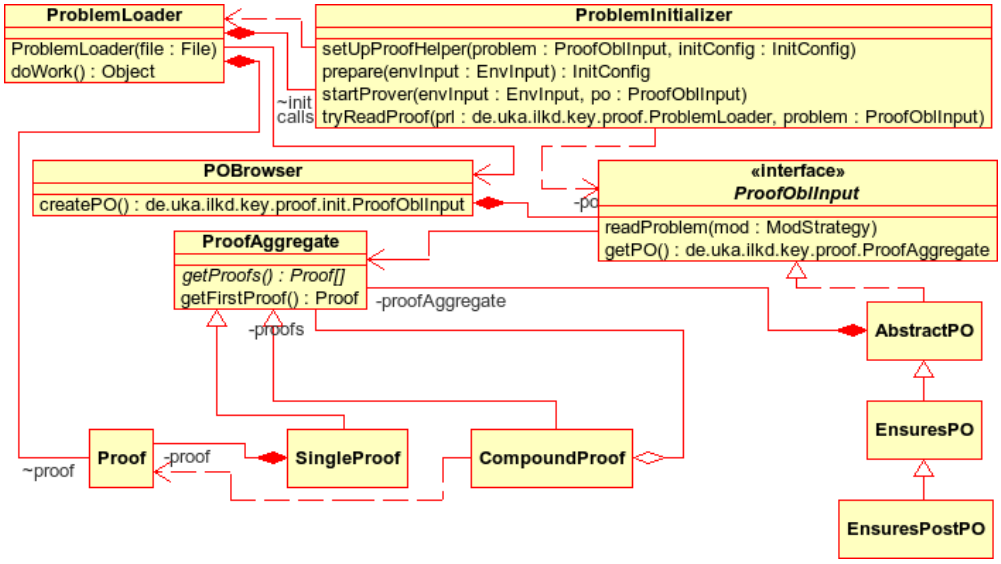



Figure 4: A simplified view on the cooperation of classes being involved in importing a Java-file with JML and creating a proof

obligation is not intentend to prove something but to offer a possibility to use KeY's Rules Applicator to transform arbitrary formulæ.

### 5.1.3. Rule Application and Strategies

To show a formula's validity, KeY tries to rewrite it to the symbol *true*. This procedure is called a proof. A formula is rewritten by applying one of about 1500 rewrite *rules* that are defined in KeY. The rewrite steps are repeated while the formula is not syntactically equal to *true* and there are rules that are applicable[7].

A rule defines how and under which circumstances a formula $F$ is transformed to $F'$. Rules with the same purpose are arranged in *rule sets* where a rule can belong to several rule sets. For example, the rule that rewrites $F \wedge \mathit{true}$ to $F$ and the rule that rewrites $\mathit{true} \wedge F$ to $F$ have the same purpose so they would share at least one rule set. There are about 140 rule sets in KeY where rule sets can intersect or be sub-/supersets of each other.

If multiple rules can be applied on the same formula there needs to be a determinate way of choosing which rule to apply. KeY achieves this by assigning costs to rule sets and always selects the cheapest applicable rule. Those costs aren't statically bound to the rules but can be calculated and changed. Figure 5 gives an abstract overview on the classes that are involved in calculating those costs.

[7] Additionally KeY allows to set a maximal number of steps or a time period after which rewriting is stopped.

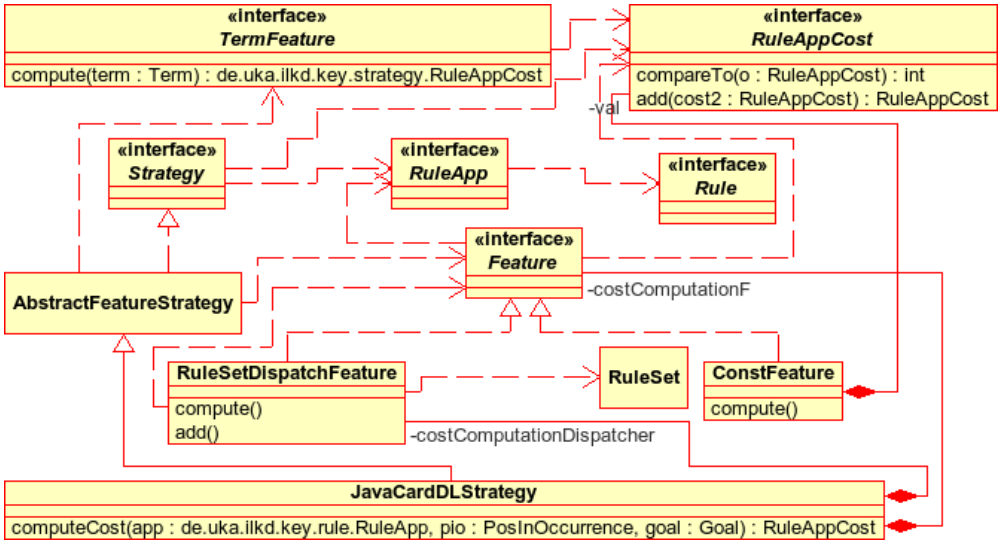



Figure 5: Simplified view on the cooperation of classes being involved in calculating the costs for a Rule

KeY uses certain classes to decide how to bind costs to a rule set. Those classes represent *strategies*, which are all derived from `Strategy`. This leads to a high flexibility as one can influence the behaviour of the rewriting system by changing the costs, which can be achieved by introducing a new strategy. In this diagram the class `JavaCardDLStrategy` is chosen as example.

It's the strategy used for verifying Java programs. It implements the method `computeCost` from `Strategy`. This method returns the costs to apply a rule to a part of a formula. Costs are modeled with the class `RuleAppCost` and the application of a rule is modeled by `RuleApp`.

As the diagram shows, `JavaCardDLStrategy` does not implement `Strategy` directly but extends the `AbstractFeatureStrategy`. The latter class introduces so called features.

Classes implementing `Feature` introduce a language to calculate the costs for rules dynamically. For example one is able to define something like: *if feature A is true for the given formula and the given rule then set the costs to -500 else check if feature B is true; if so, the costs are infinite, if not, the costs are -1000.* Infinite costs prevent the rule from being applied at all.

The class `ConstFeature` is a kind of atomic element of the language, as it does not represent logical or arithmetical operations but only an integer value.

The interface `TermFeature` is not derived from `Feature` but it's used to calculate costs as well. In contrast to `Feature` it does not combine features to compute the costs but it's able to examine the formula the rule should be applied on. Therefore

one can define costs like *If the given term's operator is a junctor and neither the left nor the right subformula contain a quantifier, the costs are -5000 and infinite else*

The class `RuleDispatchFeature` is a feature that manages the costs for all registered rule sets. If more than one feature is assigned to a rule set the `RuleDispatchFeature` creates a new combined feature and stores it. It also takes care of adding the features if a rule belongs to multiple rule sets. The `computeCost` of the `JavaCardDLStrategy` delegates to `computeCost` of `RuleDispatchFeature`

As the feature language is highly complex and it's hardly needed for this thesis, further explanations are omitted.

Figure 6 shows the new strategies that have been implemented for adjusting the costs to achieve the Rules Applicator's wanted behaviour for this thesis.

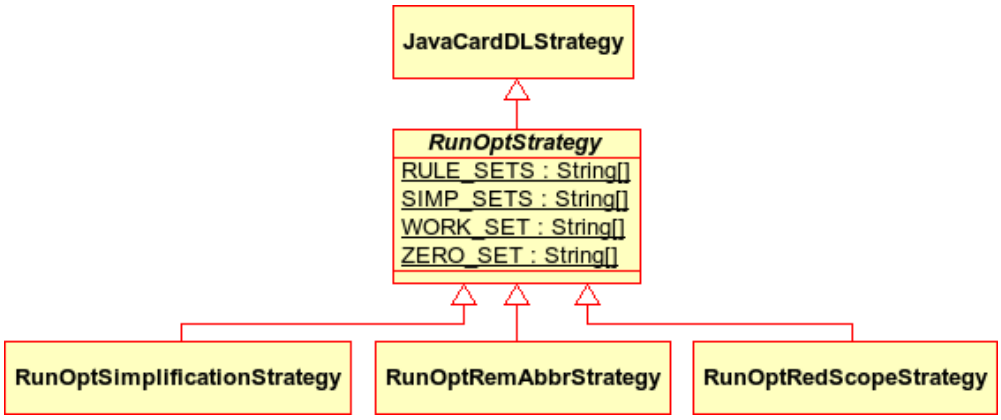



Figure 6: Overview on newly introduced strategy classes

The abstract class `RunOptStrategy` creates a general base for the other strategies. In the first step it erases the costs for all known rule sets which might have been set in a superclass. Therefore it iterates over all elements in the array `RULE_SETS` which contains the names of all rule sets, and deletes them from the `RuleDispatchFeature`. In the next step the costs of all rule sets in `SIMP_SETS` are set to -9000. This array contains the rules that lead to a simplification or reduction of the formula. Those rule sets, that are wanted in this work are stored in the array `WORK_SET` and their costs are set to -7000. The array `ZERO_SETS` contains those rule sets that intersect with or are super-/subsets of rule sets which are in `SIMP_SETS` or `WORK_SET`. As the costs are added if a rule is a member of multiple rule sets the costs of all of `ZERO_SETS`'s members is set to 0 to avoid the modification of the already processed rule sets.

In addition the cost of rule set that allows splitting of proofs is set to -6000 if splitting is allowed by an option in the Optimiser. If the option is not set the costs are infinite and therefore splitting is not used.

For the needed functionality three subclasses with matched behaviour were created. The first one is `RunOptSimplificationStrategy`. This class extends the abstract

class `RunOptStrategy` without any modification and is used to simplify formulæ and establish the negation normal form(see Sections 4.1 and 4.2 ). The second class is `RunOptRemAbbrStrategy`. It sets the costs of the rule set that is used to eliminate abbreviations (see Section 4.3) to -6000. `RunOptRedScopeStrategy` introduces a new `TermFeature` to check if the subformulæ contain the quantified variable of the current quantifier. If both subterms do, the costs for the the rule set that allows distribution of quantifiers is set to infinite, else to -9000. In combination with a ordering (see Section 5.1.5) this strategy is used to implement the reduction of the quantifier scope (see Section 4.5).

The mechanism that KeY uses to apply the rules on a proof is roughly depicted in Figure 7.

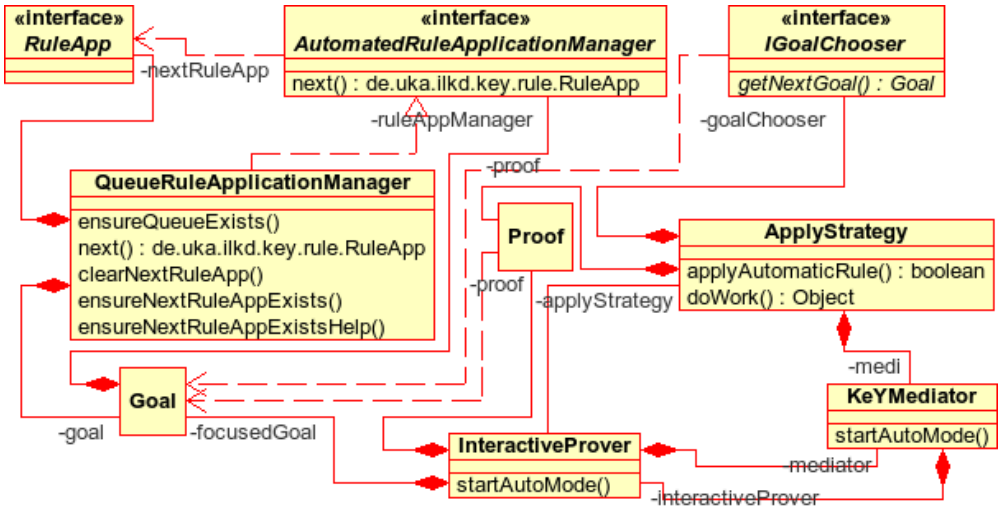



Figure 7: Simplified view on the cooperation of classes being involved in applying rules

The `KeYMediator` does some preparations and checks the existence of an open proof in KeY that the rules can be applied on. Then it calls `startAutoMode` in the `InteractiveProver` to work on that proof. This class does some checks and preparations as well and then calls the method `start` in `ApplyStrategy`. This method does some more preparations and creates a new thread that runs the method `doWork`. This method is the actual core of the rules application. It executes `applyAutomaticRule` while rules can be applied and some other conditions are true. In `applyAutomaticRule` the `IGoalChooser` is used to return a goal from the proof to work on. Then the method `next` of the goal's `AutomatedRuleApplicationManager` is called. This method ensures, that there is a queue containing all the rules that are applicable to the current formula. `next` then return the cheapest rule that is applicable which is then applied in `applyAutomaticRule`.

As this mechanism is rather complex and does more things than are actually needed for this work, the class `RunOptProofStarter` has been written. It is a simple approach to combine the tasks of `KeYMediator`, `InteractiveProver` and `ApplyStrategy`. It is initialised with a formula that is then used to create a proof. In the initialisation the environment is set up as well. The method `applyAutomaticRule` is then called and mainly behaves like a simpler version of the `ApplyStrategy`'s method with the same name.

### 5.1.4. BoundsWorker

The `BoundsWorker` is the implementation of the ideas shown in Section 4.4. As the class mainly works on its own, besides the use of KeY's datastructure for formulæ and terms and the use of the `RunOptStarter` for simplification, no diagram is given.

With exception to one point, it's the Java adaptation of the shown pseudocode. Therefore only the difference to the pseudocode is explained.

This difference is the treatment of the range values. In the Algorithm 7 all found range values are put in one list. In the implementation however, there are three lists, where the first one contains only lower bounds, the second one only upper bounds, and the third one contains all the range values that are not bounds. This modification lessens the blowup of the generated formulæ as the generation of the case distinctions is handled separately for each list and therefore the number of comparisons is reduced. This has a huge impact on the generated formula's size as latter growths exponentially with the number of comparisons.

Additionally, the mechanism to introduce `MIN_INT` and `MAX_INT` as lower/upper bound if needed that is shown in Algorithm 6 only works under the assumption that if there are at least two range values, there exists at least one upper and one lower bound. Due to the separate lists for lower/upper bound and range values the processing of lower and upper bound now works under any circumstances and `MIN_INT`/`MAX_INT` are only introduced if they are really needed.

A slightly changed version of this method for dealing with bounds has been adapted for the master branch of KeY to augment KeY's testing capabilities.

As KeY was designed with different goals some of its internal characteristics make the implementation of the needed simplification difficult. Therefore it was not completely implemented and human interaction is needed to finish this process.

### 5.1.5. Reorder

The reordering of subformulæ explained below is mainly the implementation of the ideas from Section 4.7 but it's needed for the reduction of quantifier scopes (see Section 4.5) as well.

The main idea was to use the Java classes `TreeSet` and `Comparator` to do the sorting. Figure 8 shows the classes involved.

The class `TermSorter` offers the two public methods `presort` and `postsort` that both work identically but use different comparators.

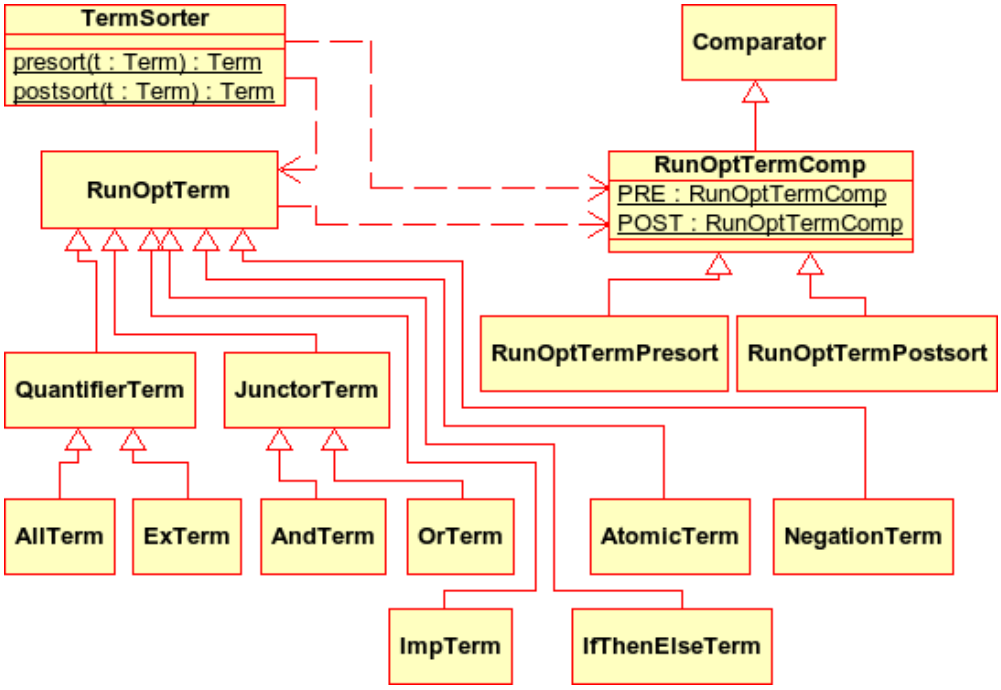



Figure 8: Simplified view on the cooperation of classes being involved in sorting formulæ

Both used comparators are derived from the class `RunOptComparator`. This class implements `Comparator` and contains a static instance of `RunOptTermPresort` and `RunOptTermPostsort`. Those two instances are the only possibility to get a instance of the two latter classes.

`RunOptTermPresort` is used as preprocessing step in the reduction of the quantifier's scope. As KeY rules assume that the subformula that can be removed from the scope is the rightmost element this comparator is used in the following ordering based on the quantified variables that are in scope in a formula:

1. If one formula contains no quantified variable and the other does, the one with none is moved to the right.

2. If none contains quantified variables they are ordered by their hash code. This ordering has no real effect but a ordering was needed to conform Java guidelines on `compare` and `equal`.

3. If both contain quantified variables the ordering is achieved by comparing the occurring quantified variables with the occurrence of quantifiers. I.e. if there are

nested quantifiers the formula that contains the quantified variable of the most nested quantifier goes to the left.

`RunOptTermPostsort` implements the sorting behaviour that was explained in Section 4.7, i.e. counting the number of subformulæ and occurring quantified formulæ.

If `presort` or `postsort` is called, the formula, being given as parameter, is then used to create a `RunOptTerm` to represent the formula. The class itself is abstract, therefore no instance can be created of it, but there are several classes derived from it. Each class represents another kind of formula depending on the formula's top level operator. The most important one is `JunctorTerm` with its derived classes `AndTerm` and `OrTerm` as they are responsible for the reordering. The latter classes use a `TreeSet` to hold the subformulæ that, in combination with the type of the class itself, is used to implement $\wedge_{mul}$, respectively $\vee_{mul}$.

The `AtomicTerm` is used to wrap the predicates of the original formula.

After the `RunOptTerm` is created, `presort` (`postsort`) calls its method `sort` with `PRE` (`POST`) as argument. This method initialises the `TreeSet` with the given comparator and the according subformulæ of the current `JunctorTerm` which leads to the reordering.

Finally `presort` or `postsort` calls the method `getTerm` of `RunOptTerm` which leads to the creation of KeY's datastructure for formulæ out of the `RunOptTerm`. This formula is reordered in terms of this section.

## 5.2. Jet

JET is an automated unit testing tool for Java classes annotated with JML specifications[Jet]. It is written in Java by Yoonsik Cheon and Perla M. Escarcega at the University of Texas at El Paso. It was chosen as it is able to work on Java-files that have been enriched with JML specifications. Another reason is the license used for publishing as it is the GPL which allows modification of the code which is needed for this work.

The version of JET being used is 0.78a with mainly the two following minor adjustments:

1. The original code was tailored to be highly multithreaded which lead to a wide spread in the runtimes. To optimise the program's behaviour to be used for benchmarking, multithreading was stripped from the important methods.

2. A stopwatch to meassure and display the time needed to run a test case method was incorporated in code.

# 6. Evaluation

## 6.1. Examples

In this section only the classical first order logic notation, i.e. not JML, of the examples' postcondition are shown. The full code of the examples can be found in the Appendix A. The following examples have been constructed for the means of evaluation.

**Reduce Range.** The first method that is to be evaluated is the mechanism to reduce the ranges of a quantifier. The following formula is used as example.

$$
\begin{aligned}
&\forall x. \\
&\quad 0 \leq x \wedge x < 2 * a.length \Rightarrow \\
&\qquad 0 \leq x \wedge x < a.length \Rightarrow \\
&\qquad\quad result[x] = 13 \\
&\qquad \wedge \\
&\qquad a.length \leq x \wedge x < 2 * a.length \Rightarrow \\
&\qquad\quad result[x] = 17
\end{aligned}
$$

This formula consists of one quantified formula in whose matrix case distinctions are made. Those case distinctions depend on the quantified variable are used and splits the quantifier's bounds in to ranges. If the quantified variable is in the first half of the quantifier's bounds ($0 \leq x \wedge x < a.length$) it's checked if the result is equal to 13. If the quantified variable is in the last half of the quantifiers bounds ($a.length \leq x \wedge x < 2 * a.length$) it's checked if the result equals to 17.

The formula being created by this optimisation is shown below. Its quantified formula has been broken up into two quantified formulæ that are conjunctively connected. The case distinctions of the source formula are used as bounds for the new ones.

$$
\begin{aligned}
&\forall x. \\
&\quad 0 \leq x \wedge x < a.length \Rightarrow \\
&\qquad result[x] = 13 \\
&\wedge \\
&\forall x. \\
&\quad a.length \leq x \wedge x < 2 * a.length \Rightarrow \\
&\qquad result[x] = 17
\end{aligned}
$$

**Remove Abbreviation.** The example to evaluate the removal of an abbreviation is shown below. The quantified variable $y$ is introduced as an abbreviation for the term $t$. In this specific case an abbreviation does not improve the readability of the formula, but the introduction of a more complex term was discarded to keep the examples simple.

$$
\begin{array}{l}
\forall x. \\
\quad 0 \leq x \wedge x < a.length \Rightarrow \\
\quad\quad \exists y. \\
\quad\quad\quad\quad y = t \wedge result[x] = y
\end{array}
$$

As JET was not able to execute this example due to the fact, that the existential quantified formula can not be executed, this example could not be used for evaluation. Therefore the example shown below is used which is similar to the original one and executable by JET.

$$
\begin{array}{l}
\forall x. \\
\quad 0 \leq x \wedge x < a.length \Rightarrow \\
\quad\quad \forall y. \\
\quad\quad\quad\quad -2147483648 \leq y \wedge y \leq 2147483647 \Rightarrow \\
\quad\quad\quad\quad\quad y = t \Rightarrow result[x] = y
\end{array}
$$

The optimisation removes the inner quantifier and its quantified variable is replaced by its value.

$$
\begin{array}{l}
\forall x. \\
\quad 0 \leq x \wedge x < a.length \Rightarrow \\
\quad\quad result[i] == t
\end{array}
$$

**Reduce Scope** The example to evaluate the reduction of the quantifier's scope is shown below. The expression $\neg wrong = true$ is evaluated in the quantifier's scope. This is not needed as the expression does not depend on the quantified variable.

$$
\begin{array}{l}
\forall x. \\
\quad 0 \leq x \wedge x < a.length \Rightarrow \\
\quad\quad result[x] = 13 \wedge \neg wrong = true
\end{array}
$$

By moving the expression out of the quantifiers scope there is one operation less to be executed in the quantifiers scope.

$$
\begin{array}{l}
\forall x. \\
\quad 0 \leq x \wedge x < a.length \Rightarrow \\
\quad\quad result[x] = 13 \\
\wedge \\
\neg wrong = true
\end{array}
$$

## 6.2. Evaluation Approach

To evaluate this approach's benefit, the 3 main methods, i.e. the reduction of the quantifier's scope (see Section 4.5), the reduction of its bounds (see Section 4.4) and the removal of abbreviations (see Section 4.3), are each applied on an example. Then the time to test the unmodified and the modified version is measured. For running the tests KeY's own testing method and JET are used.

**JET.** JET's execution time for a single test case of the examples lies in the magnitude of $10^{-4}$s. This duration is so short, that it is prone to influences, e.g. the Java garbage collector, or errors of measurement and therefore hardly usable. JET offers the possibility to execute several test attempts in one test run. The execution of 1000 attempts has been tried and its runtime lies in the magnitude of $10^{-2}$s which is considered less inexact. Therefore the time measured in the evaluation is not the time for a single attempt but for 1000.

First experiments showed that the execution times for an example vary in huge margins. As this behaviour makes a reliable judgement of the results hardly possible, a loop has been added to execute the testing mechanism 1000 times and calculate the average time for the 1000 gathered test runs.

**KeY.** In KeY the execution time of a single test lies between 1 and 2 seconds. Therefore the time needed to run one attempt is considered usable.

As with JET, the test times vary and therefore the test mechanism is executed 1000 times and the mean is calculated and shown in the results.

In both cases the standard deviation $\sigma$ is given as well to show how strong the single results scatter. It was calculated by using the formula

$$\sigma = \sqrt{\frac{1}{n} * \sum_{i=0}^{n} (x_i - x_{avg})^2}$$

where $n$ is the number of executions used and $x_{avg}$ is the mean.

All tests were run on the same hardware (an Intel Pentium M (Banias) with 1.4Ghz and 2GB of DDR2 RAM).

## 6.3. Results

Running the optimised and not optimised examples with KeY's test generator leads to the results shown in Table 2. The most significant result is that running the example for the remove abbreviation method in its unchanged form takes more than 90 minutes. After 90 Minutes, not even one single test case was finished and the execution was cancelled. The test for the optimised formula though takes roughly the same time as the test of the other examples.

The results of the two remaining test examples (reduce ranges, reduce scope) shows not significant benefit between the original and the optimised version.

[8] Execution stopped

| Name of Method | $t_{avg}$ [ms] | $\sigma$ [ms] |
|---|---|---|
| Reduce Ranges | 1590.953 | 22.338 |
| Reduce Ranges (opt) | 1549.151 | 30.266 |
| Remove Abbreviation | —[8] | —[8] |
| Remove Abbreviation (opt) | 1549.040 | 21.680 |
| Reduce Scope | 1566.759 | 33.523 |
| Reduce Scope (opt) | 1549.178 | 25.674 |

Table 2: Results for testing the examples with KeY.

The results with JET are shown in Table 3. As in KeY there is no significant benefit visible in the runtimes.

| Name of Method | $t_{avg}$ [ms] | $\sigma$ [ms] |
|---|---|---|
| Reduce Ranges | 48.955 | 10.199 |
| Reduce Ranges (opt) | 47.173 | 6.502 |
| Remove Abbreviation | 47.546 | 6.876 |
| Remove Abbreviation (opt) | 47.315 | 3.453 |
| Reduce Scope | 49.073 | 11.011 |
| Reduce Scope (opt) | 48.085 | 6.606 |

Table 3: Results for testing the examples with JET.

# 7. Related Work

While looking for some information for this thesis a lot of projects working on executable specifications were found. They are using several languages in their projects for the specification as well as for the target programming language being executed. Although the main goal, i.e. execution of specification, is equal in those projects, it is possible to distinguish them by taking a look at the details. A significant criterion for classification is the field of application of the generated program. This leads to two different groups.

In the approach of Fuchs [1992b] he tries to generate the actual program out of a specification. I.e. one writes the specification of the wanted program as a Hoare tuple and the program is generated out of it. Fuchs describes two different methods to generate the program where the first one uses if-halves of the specification in a readable and easy understandable way. As the program generated with this method has a slow execution time, he introduces another method that leads to shorter runtimes for the generated programs and is based on structural induction.

The second group of projects uses the generated program to check the correctness of their already existing programs. In Peters and Parnas [1994] a way to create a program out of formal documentations that are given by using Limited Domain Relations and

tabular expressions is shown. The target language as well as the language of the code to be tested is C where it is mentioned that the oracle might be implemented in C++ as well. The only statement that is made about an optimised runtime is the fact that the translation is building the program in such way, that it is tried to get a true expression in the case of an existential quantification and a false expression in the case of an universal quantification as fast as possible.

Tim Wahls is involved in two different approaches. In Wahls et al. [2000] and Wahls and Leavens [2001] he, Leavens and Baker describe and prove methods for translating Specs-C++ specification into the concurrent constraint programming language AKL. It is mentioned that the occurrence of an existential quantification leads to a blowup of the set of constraints, which leads to an increase of the runtime. There is no optimisation method mentioned to prevent such cases. In contrast to above mentioned approaches the result is not fixed to creating an oracle but to be used as prototype of the specified program as well.

In Krause and Wahls [2006] Krause and Wahls are working on a method of translating the JML ensures-clause to a constraint program that can be executed with the Java Constraint Kit(JCK). A technique that is mentioned in the paper, that might lead to a reduced runtime for the generated program is the use of a simplifier on the set of constraints that have been gathered. This approach focuses on the generation of prototypes while developing the formal specification.

The paper Zee et al. [2007] describes a method to translate specification written in Higher Order Logic to a subset of Java. To optimise the runtime of the generated program the authors describe a technique that is similar to the method explained in Section 4.3 to remove quantified variables being introduced as abbreviation for a term. The goal of this work is to support the prover Jahob with runtime checking to prevent errors in the specification or the code.

In Fröhlich [1997] it is assumed that the postcondition of a program is defined by using abstract functions that are defined by their behaviour instead of their implementation. Those functions, written in VDM-SL are given to a framework that allows to execute them. No statements about optimisation techniques or the runtime are made in the paper.

# 8. Conclusion and Outlook

This minor thesis shows a way to optimise a generated oracle to achieve shorter runtimes. Shorter runtimes of test cases allows the execution of more test cases in the same time. The execution of more test cases leads to a higher confidence in the software's quality.

Oracles can be derived from specifications. However specifications are used for different purposes and therefore are not necessarily executable. Even if the are executable it might be with only a high runtime.

Those two facts come mostly from the use of quantifiers in the logic. If the quantifier's range is not bounded, respectively if the bounds are outside the target language's datatype's limits, the specification is too expressive to be exported into a program.

Even if the bounds inside the used datatype's limits, the quantification is represented as a loop which leads to a runtime' blowup, especially if quantifiers are nested.

This work explains four different possibilities to reduce the execution time of the oracle by manipulating the quantified formulæ whereas this approach is only applicable if the quantified variables are of type Integer.

The first part shows a way to get rid of a special kind of quantification, i.e. an abbreviation. An abbreviation is an existential quantifier that contains a conjunctively added equality that defines the value of the quantified variable.

The second method tries to find information about the quantified variable and use it to cut down the ranges of the quantifier. It tries to remove case distinction that depend on the quantified variable by splitting the quantifier in multiple quantified formulæ with disjunctive ranges.

The third method offer a way to try to shrink the scope of a quantifier by moving subformulæ that do not depend on the quantified variable out of its scope.

The fourth method is not actually working on the quantified formula, but trying to achieve its fail respectively its success by introducing instantiations that might be significant. If they are significant for a certain test case the execution of the loop is omitted which reduces the runtime. This four core methods are supported by common logical simplification techniques to ensure that the formula is free of meaningless subformulæ.

Additionally a method for reordering subformulæ was introduced. The idea is to achieve an quick fail or success of a formula by using short circuit evaluation and therefore avoid not needed evaluations.

The benchmarks presented in this thesis show no significant benefits in the time needed for testing, when using optimisation techniques. The only visible benefit so far is the reduction of the runtime for the Remove Abbreviation example when KeY is used as testing tool.

However the method of changing the evaluation order has not been evaluated and the idea of reordering formulæ hast not been completely developed.

There are some tasks, that are still open and might be of interest.

Besides the completion of the implementation there is one more important step that might be done to improve the whole approach. The idea of ordering formulæ by costs could be refined by using other means to calculate the costs. The next step would be to use their evaluation cost, i.e. define how much it costs to evaluate a formula and take the cheapest first. Thereby some formulæ are cheap to execute but true respectively false in most cases so their evaluation does not lead to a real advantage in the test case. At this point the idea of developing a probability for a formula to be true respectively false was introduced. The combination of both aspects, called estimated costs, should be used to order the subformula in such a way that formulæ that are cheap to evaluate and that have a high value of being false (if it is conjunctively connected) respectively true (if it is disjunctively connected) are evaluated first.

Another open point is the possibility to use KeY's Rules Applicator in combination with a `TermFeature` (see Section 5.1.3) for the reordering of formulæ. This might be more elegant then the current approach.

# A. Appendix

```
1  /*@ public normal_behavior
2    @ requires a!=null;
3    @ ensures (\forall int i; 0<=i && i < 2*a.length;(
4    @      (0<=i&&i< a.length ==>\result[i]==13 &&
5    @      (a.length<=i&&i<2*a.length ==>\result[i]==17))
6    @ ));
7    @ assignable \nothing;
8    @ */
9  public static int[] redBounds1(int[] a){
10         int l = a.length;
11         int ll = 2*a.length;
12         int[] res = new int[ll];
13         for (int i=0; i < l; i++){
14                 res[i] = 13;
15         }
16         for (int i= l; i < ll; i++){
17                 res[i] = 17;
18         }
19         return res;
20 }
21
22 /*@ public normal_behavior
23   @ requires a!=null;
24   @ ensures (
25   @      (\forall int i; 0<=i && i < a.length;\result[i]==13) &&
26   @      (\forall int i;a.length<=i&&i<2*a.length;\result[i]==17)
27   @ );
28   @ assignable \nothing;
29   @ */
30 public static int[] redBounds2(int[] a){
31         int l = a.length;
32         int ll = 2*a.length;
33         int[] res = new int[ll];
34         for (int i=0; i < l; i++){
35                 res[i] = 13;
36         }
37         for (int i= l; i < ll; i++){
38                 res[i] = 17;
39         }
40         return res;
41 }
```

Listing 13: The code used for evaluating the reduction of ranges

```
1  public static int t = 13;
2
3  /*@ public normal_behavior
4    @ requires a!=null;
5    @ ensures (\forall int i; 0<=i && i < a.length;
6    @      (\forall int x ; -2147483648 <= x && x <= 2147483647 ;
7    @              x==t ==> \result[i]==x
8    @      )
9    @ );
10   @ assignable \nothing;
11   @ */
12 public static int[] remAbb1(int[] a){
13         int[] res = new int[a.length];
14         for (int i=0; i < a.length; i++){
15                 res[i] = t;
16         }
17         return res;
18 }
19
20 /*@ public normal_behavior
21   @ requires a!=null;
22   @ ensures (\forall int i; 0<=i && i < a.length;
23   @      \result[i]==t
24   @ );
25   @ assignable \nothing;
26   @ */
27 public static int[] remAbb2(int[] a){
28         int[] res = new int[a.length];
29         for (int i=0; i < a.length; i++){
30                 res[i] = t;
31         }
32         return res;
33 }
```

Listing 14: The code used for evaluating the removal of Abbreviations

```
1  public static boolean wrong=false;
2
3  /*@ public normal_behavior
4    @ requires a!=null;
5    @ ensures (\forall int i; 0<=i && i < a.length;
6    @      (\result[i]==13 &&
7    @      wrong!=true)
8    @ );
9    @ assignable \nothing;
10   @ */
11 public static int[] redScope1(int[] a){
12         int[] res = new int[a.length];
13         for (int i=0; i < a.length; i++){
14                 res[i] = 13;
15         }
16         return res;
17 }
18
19 /*@ public normal_behavior
20   @ requires a!=null;
21   @ ensures (\forall int i; 0<=i && i < a.length;
22   @      (\result[i]==13) &&
23   @      wrong!=true
24   @ );
25   @ assignable \nothing;
26   @ */
27 public static int[] redScope2(int[] a){
28         int[] res = new int[a.length];
29         for (int i=0; i < a.length; i++){
30                 res[i] = 13;
31         }
32         return res;
33 }
```

Listing 15: The code used for evaluating the reduction of scopes

# References


M. Andersen, R. Elmstrom, P. Lassen, and P. Larsen. Making specifications executable – using iptes meta-iv, 1992.

B. Beckert. A dynamic logic for java card. In *In Proceedings, 2nd ECOOP Workshop on Formal Techniques for Java Programs*, pages 111–119. Springer, 2000.

B. Beckert, R. Hähnle, and P. H. Schmitt. *Verification of Object-Oriented Software. The KeY Approach: Foreword by K. Rustan M. Leino (Lecture Notes in Computer Science)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007. ISBN 354068977X.

L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, 2005.

Y. Cheon, M. Kim, and A. Perumendla. A complete automation of unit testing for Java programs. In H. R. Arabnia and H. Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume I, Las Vegas, Nevada, June 27-29, 2005*, pages 290–295. CSREA Press, 2005.

E. Dijkstra. Structured programming. pages 41–48, 1979.

C. Engel. A translation from JML to javadl, 2005.

C. Engel. Verification based test case generation. Master's thesis, University Karslruhe, August 2006.

M. Fitting. *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 0-387-94593-8.

B. Fröhlich. Program generation based on postconditions. In M. Hmaza, editor, *Software Engineering (SE'97)*, pages 62–66. IASTED, IASTED/ACTA Press, November 1997.

N. E. Fuchs. Specifications are (preferably) executable. *IEE/BCS Software Engineering Journal*, 7(5):323–334, 1992a.

N. E. Fuchs. Hoare logic, executable specifications and logic programs. *Structured Programming*, 13(3):129–135, 1992b.

A. M. Gravell and P. Henderson. Executing formal specifications need not be harmful. *IEE/BCS Software Engineering Journal*, 11(2):104–110, 1996.

I. J. Hayes and C. B. Jones. Specifications are not (necessarily) executable. Technical Report 148, St. Lucia, Queensland, Australia 4072, 1990.

H. Hussmann, B. Demuth, and F. Finger. Modular architecture for a toolset supporting ocl. *Sci. Comput. Program.*, 44(1):51–69, 2002. ISSN 0167-6423.

Jet. Jet. Website at `http://cs.utep.edu/cheon/download/jet/index.php`, 2008.

JML. The java modeling language. Website at `http://www.eecs.ucf.edu/~leavens/JML/`.

D. John. *The VDM-SL Reference Guide*. Routledge, UK, 1991.

KeY. KeY Project. Website at `www.key-project.org`, 1999.

R. A. Kowalski. *Logic for Problem Solving*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979. ISBN 0444003657.

B. Krause and T. Wahls. jmle: A tool for executing jml specifications via constraint programming. In L. Brim, B. R. Haverkort, M. Leucker, and J. van de Pol, editors, *FMICS/PDMC*, volume 4346 of *Lecture Notes in Computer Science*, pages 293–296. Springer, 2006. ISBN 978-3-540-70951-0.

G. T. Leavens and Y. Cheon. Design by contract with JML, 2006.

G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. JML reference manual, 2008.

OCL. The object constraint language. Handbook, 2006. URL `http://www.omg.org/cgi-bin/doc?formal/2006-05-01`.

OMG. The object modeling group. Website at `http://www.omg.org/`.

D. Peters and D. L. Parnas. Generating a test oracle from program documentation: work in progress. In *ISSTA '94: Proceedings of the 1994 ACM SIGSOFT international symposium on Software testing and analysis*, pages 58–65, New York, NY, USA, 1994. ACM. ISBN 0-89791-683-2.

G. Smith. *The Object-Z Specification Language*, volume 1 of *Advances in Formal Methods*. Springer-Verlag, 1999.

T. Wahls and G. T. Leavens. Formal semantics of an algorithm for translating model-based specifications to concurrent constraint programs. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 567–575, New York, NY, USA, 2001. ACM. ISBN 1-58113-287-5.

T. Wahls, G. T. Leavens, and A. L. Baker. Executing formal specifications with concurrent constraint programming. *Automated Software Engg.*, 7(4):315–343, 2000. ISSN 0928-8910.

K. Zee, V. Kuncak, and M. Rinard. Runtime Checking for Program Verification Systems. In *Workshop on Workshop on Runtime Verification (collocated with AOSD)*, 2007.