# Specification and verification of sequential Java programs

Master's Thesis Cognitive Artificial Intelligence,
Utrecht University

**Martijn Warnier**

25 January 2002

First supervisor: dr. V. van Oostrom

This thesis is typeset in $\LaTeX\,2_{\varepsilon}$.

## Acknowledgments

Thanks to all the people at the LOOP group (and off course the ITA (Information for Technical Applications) group) at Nijmegen University, especially Bart Jacobs, Cees-Bart Breunesse and Joachim van den Berg (for fixing a number of problems). Also thanks to Vincent van Oostrom and Frank de Boer at the Utrecht University, as well as Jan Bergstra for showing me the way to Nijmegen (and of course his and Marijke Loots's programs which form a central part of the work in this thesis) and my friends at CKI (Bastiaan, Carsten, Janneke, Jeroen, Job, Linda, Thomas and Tinka). Furthermore my "Geological friends": Daan, Juriaan, Kees, Rob, Rosha, Ruby and (off course) Thijs for making 7 years of Utrecht (and many years to come) worth wile. My *roomies*: Dorien, Nïurka, Oele, Pieter and the others, for all the nice years at IBB 179.

And finally my parents, to which I dedicate this Master's Thesis, and my sisters, for giving continuous support whenever needed.

# Contents

# Chapter 1

# Introduction

This thesis is concerned with the specification and verification of Java programs. There are a number of questions one might ask about this topic.

A first question might be:"Why do we need to prove that a program is correct?" Typical (large) programs contain thousands of so-called bugs (as anyone who ever used a computer should know), but millions of peoples use them every day. There are however situations possible in which this type of behavior is not desirable, e.g. when the software of an airplane contains a bug which results in a complete "crash" of the on board computer which in turn results in a plane crash. The latter kind of software is called "Safety Critical Software" (for obvious reasons). So one place where verification of programs is important is in the development of "Safety Critical Systems". Another example are security applications: users do not want to use some kind of security software which turns out to be buggy (with a compromised security as a result).

The observant reader might now ask: "Why are not *all* programs verified before they are distributed?" In an ideal world this would indeed be the case, however as we all know: we do not live in a perfect world. Verification of programs is a very hard task, certainly when "real" (messy) programming languages like C++, COBOL or Java are used. To prove some program correct you need a precise specification. For a lot of programs it would be very hard to write such a specification because most of the time it is not *exactly*[1] clear what a program should do. Moreover, once one has a specification of a program, proving that the specification is correct[2] is a nontrivial task.

This thesis is only concerned with the correctness of specification. Another issue (which is not studied in this thesis) is the completeness of a specification, completeness means that the program can not do any actions which are not in its specification.

We will look at the core part of sequential Java. Some small (but significant) programs from [BL99] will be used. Each program will be specified and proven correct. Possibly, these JML annotated versions of examples from [BL99] will be integrated in a future version of [BL99].

The remainder of this chapter will involve a (first) look at Java. The place of this subject within the field of Cognitive Artificial Intelligence, as well as a look ahead of what is coming in subsequent chapters.

## 1.1 The programming language Java

Since its first release in may 1995 the object oriented programming language Java became popular fast because of its role on the Internet. So called Java *applets* could be used to run programs inside a web document. This feature along with others such as *platform in-dependency* and good *security* made the language ideal for developing programs which were to be used on the Internet.

[1]i.e. people can not say what the program should and should not do in a precise (mathematical) fashion. At the moment there is no suitable mathematics for specifying complex programs, e.g. programs with a lot of threads.

[2]i.e. The program does what the specification says it should do

Nowadays (at the end of 2001) Java applets are widely used on the World Wide Web. There are also other domains where the Java language is used, especially in the area of *embedded systems*[3]. This is partly because Java is relatively easy to use (in comparison with e.g. C++) and partly because Java is to a large extend "open". Open meaning that all (SUN's) API's are open source and low level features and Java Virtual Machines are thoroughly documented. This last feature of Java makes it all the more interesting to look at how the language actually works (i.e. can one give a suitable semantics of Java?).

Java has no official, mathematical semantics (provided by SUN). Since this is obviously an interesting topic for more then only academic reasons it is an important topic in academic research at the moment. See for example the work on LOOP [LOO], Bali [vON99] and others [AF99]. The work in this thesis has been done in Nijmegen at the LOOP group.

## 1.2 Program verification and Artificial Intelligence

There are a lot of possible definitions for the study of artificial intelligence, one of these is the following:

> The study of the computations that make it possible to perceive, reason and act [Win92].

This definition applies to a lot of different research areas, e.g. a psychologist who is studying a computational model of human cognition is (according to the above definition) doing AI, so is a computer scientist who is working on a robot-arm or a linguist who is studying Montague grammar. Since there appear to be a lot of different scientific research areas in the field of AI, the study of AI is sometimes described as a multi-disciplinary study.

A lot of different "established" disciplines are used in AI. To name the most important ones: computer science, logic, psychology, linguistics and biology. So on the one side we see "hard" (hard meaning involving mathematics) studies like computer sciences and logic and on the other hand "softer" studies like linguistics and psychology with between these two extremes biology. This thesis must be placed in the "harder" part of AI.

So what has program verification to do with AI? On first sight it does not seem like an awful lot. Sure, there is the obvious relation with computer sciences and logic, but –according to the above definition– program verification does not seem to be in AI's field. However if we look a bit closer we see that one of the three possible fields within AI is the ability for a computer to *reason*.

The study of reasoning (logic) is probably one of the oldest studies that we know of, it is certain that the ancient Greeks already studied reasoning and it is not unthinkable that earlier civilizations studied this phenomenon. Under the influence of the computer this field took an enormous boost and within AI people studied the possibility to automate reasoning. Since a major part of the reasoning about the programs in this thesis is done solely by the computer (using automated rewriting) it does not seem too far fetched to place this thesis is subject within the scope of AI. Off course this does not mean that this thesis is primarily concerned with automated-reasoning. It is not, but nonetheless the results improve a part of logic and thus improve the understanding of reasoning.

[3]e.g. Java Card™(a Java dialect for implementing Smart Cards), see [WJP] for other Java dialects and API's which are used at the moment

## 1.3 What is still to come

The remainder of this thesis is organized in the following matter:

**Chapter2** takes a closer look at the various languages and tools that were used, namely JavaCck, JML, the LOOP compiler, the Java memory model, the PVS theorem prover and Hoare logic. Each will be explained in (some) detail.

**Chapter 3** describes a number of Java programs from [BL99]. For each of the programs a JML specification is given which will be explained. A detailed description of the program and its behavior is given as well as proofs of the JML annotated programs which are constructed using automatic-rewriting in the semantics in PVS of programs and their specifications.

**Chapter 4** describes some more programs with their JML specification. This time the proofs are developed in an approach using Hoare-logic.

**Chapter 5** concludes with some discussion on the differences between the two methods for obtaining proofs which were described in the previous chapters. Finally some conclusions are drawn and a look-ahead at future work will be given.

# Chapter 2

# Tools & languages

## 2.1 JavaCck

The object oriented programming language Java [GJSB00, AG96] is one of *the* most used programming languages worldwide (certainly in the Academic world). Since Java is a "real" programming language (i.e. not an artificial programming language which is suitable for verification, but not for "real" programming), verification is a non-trivial task.

Rather then using programs written in full-fledged Java we use a subset of Java called the Java Class Construction Kernel used in [BL99]. There were a number of reasons for doing so: (i) JavaCck does not use all Java's features: multi-threading, multiple inheritance[1] and inner-classes are not allowed, furthermore no extending of external classes will be used (thus making the use of the JDK library impossible). All these restrictions make JavaCck a lot less complex then Java (and completely deterministic), making verification easier but still non-trivial (i.e. JavaCck is Turing complete). (ii) By studying a subset of a complex programming language rather then using a artificial programming language (developed solely for the purpose of proving its programs correct) one will still learn a lot about the language itself (and especially its core features). (iii) In [BL99] an extensive study is made for JavaCck, with a lot of programs. These programs can be used for proving, skipping the task of developing a lot of (test) programs.

Another useful feature from [BL99] is the use of class hierarchies. Classes are part of one or more JavaCck Class Families (JCF), a JCF is Java program consisting of at least two classes: a class `s` which consists of the `main` method, `main`'s only action is a call to a method `m()` from a class `c`. All methods that are specified in this thesis will be part of some JCF and will be named `m()`.

## 2.2 JML

The Java Modeling Language [LBR99] is a behavioral interface specification language designed to specify Java programs. Below a (simplified) syntax for a JML specification of an arbitrary method `m()` is given (from [BJ01]).

Each specification starts with `/*@` and ends with `@*/` (the other `@`'s are there for readability). Since the JML annotations are within Java's standard comment notation, the Java program can still be compiled by a normal Java-compiler (and run on one of the standard Virtual Machines). The first line, `behavior`, specifies which kind of behavior is excepted. One kind of behavior is the `normal_behavior` stating that no exceptions are being thrown. Another kind of behavior is the `exceptional_behavior` which is used when exceptions are being thrown. These two can be combined using only `behavior`. The second-line, the so called `requires` clause, states which pre-conditions should be taken into account. An example of this is:

[1] Java does not allow multiple inheritance of classes, but does allow multiple inheritance of interfaces. The latter will not be used in this thesis

`requires !\not_initialized(c)`, stating that the Java class c is not initialized. The next line states which variables are `modifiable`, e.g. `modifiable \static_fields_of(c)` indicates that all static fields of class c are modifiable (and nothing else is). The fourth line of the specification states what kind of results are expected from the method (in the case of `normal_behavior`), e.g. `ensures \result == true;` means that the expected result from the method should be the boolean value true. The `signals` clause states what happens when an exception will be thrown (the signals clause corresponds with the `exceptional_behavior`). An example of this clause is `signals(NullPointerException) b == true;` which states that a NullPointerException will be thrown and the boolean field b will have value true after execution. Note that both the `requires` and `modifiable` clause refer to variables in the pre-condition (before evaluation of the program) and the `ensures` and `signals` clause refer to variables in the post-condition (after evaluation of the program).

The JML annotation language has many more constructs. We will only use a small part of these in the examples shown in the next chapters. More JML keywords will be explained at the moment they are used in specifications.

JML

```
/*@ behavior
  @   requires <pre-condition>;
  @ modifiable <modifiable-clause>;
  @    ensures <post-condition>;      // when terminating normally
  @    signals (E) <post-condition>;  // when terminating abruptly
  @                                   // because of exception E
  @*/
public boolean m()
```

JML

We summarize the meaning of this JML fragment for method `m()`. Assuming the `<pre-condition>` holds; then (1) if it terminates normally the `<post-condition>` of the `ensures` clause will hold and only the variables mentioned in the `<modifiable-clause>` will be altered during execution, or (2) if it terminates abnormally the exception `E` will be thrown and the `<post-condition>` of the `<signals>` clause will hold and again the variables mentioned in the `<modifiable-clause>` will be ale during execution.

## 2.3 The LOOP tool

Within the Logic of Object-Oriented Programming [BJ01, LOO] project at Nijmegen University a special purpose compiler, the LOOP tool, has been built. Since its focus is on sequential Java it is ideal for the task at hand. The LOOP tool translates a Java class together with its JML specification into higher order theories for the PVS theorem prover. The JML specification is translated to predicates. We will proof that given the pre-condition mentioned in the `requires` clause the post-condition in the `ensures` and/or `signals` clause will hold for the program. Figure 2.1 shows this process.

The *semantic prelude* in the figure refers to the semantic model for Java and JML. It consists of a great number of theories and lemmas implemented (by hand) in PVS and gives a complete semantics for sequential Java. How this semantic prelude exactly is formalized is not further explained in this thesis. Readers are referred to [JP00] for more information on this subject.

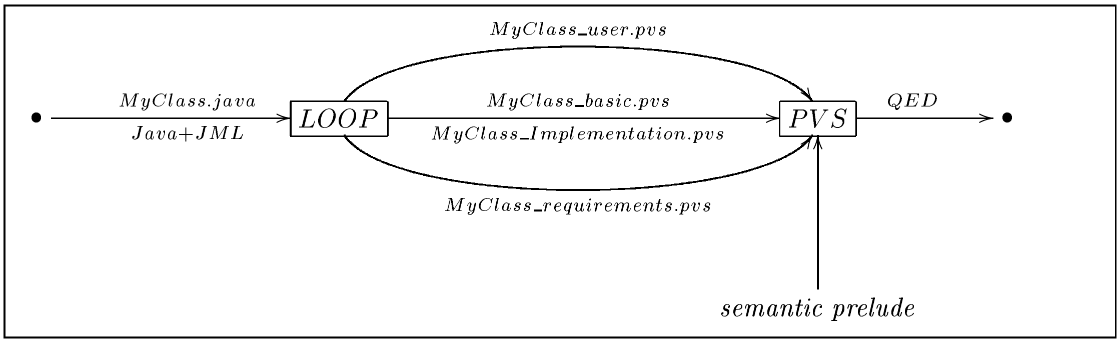


Figure 2.1: the LOOP-compiler

## 2.4 The memory model for Java

We will give a (very) brief introduction to the underlying memory model which will be used.

The most basic notion we find in our memory model is the single memory cell. The memory model consists of an arbitrary number of these cells that can store all the fields from an arbitrary object. The types that the fields of objects can have are limited to the primitive types (byte, short, int, long, char, float and double) and other objects. For each one of these there is a separate part reserved within the single cell. So an object with two integer fields and one boolean field stores its fields in a single cell position: the first integer at place 0 of the integer part of the cell, the second integer at place 1 of the integer part of the cell and the boolean at place 0 of the boolean part of the cell. An empty memory cell is defined with Java's default values.

Object cells represent the main part of the memory. The memory has a heap, stack and static part, for storing the contents of respectively instance variables, local variables and parameters, and static (class) variables. The entry heaptop (resp. stacktop) indicates the next free (unused) memory location on the heap (resp. stack). These change during program execution as, for instance, new objects are created. The LOOP compiler assigns locations in the static memory to classes with static fields. At such locations a boolean `initialised` tells whether static initialization has taken place for this class. One has to keep track of this because static initialization should not be performed more then once [BHJP00].

Accessing a specific value in memory, either for reading or writing, involves three parts: (1) an indication of which memory (heap, stack, static), (2) a memory location of the object cell, and (3) a memory location within the single cell associated with the desired value. One can use `put` and `get` methods to read or alter all these memory parts.

This (simplified) explanation of our memory model should give the reader an idea of the organization of the memory model. In reality the model is implemented using types for the different memory cells and involving $\lambda$-expressions to place and read cell in memory. A complete discussion of the memory model is beyond the scope of this thesis. Interested readers can look in [JP00] for a more detailed introduction to this subject.

## 2.5 The PVS theorem prover

PVS stands for "Prototype Verification System" [SORSC99], it is a prototype environment for specification and verification. The PVS theorem prover (or proof checker) provides a collection of inference procedures that are applied interactively under user guidance within a sequent calculus framework[2]. Using a theorem prover rather then doing proofs by hand has some clear advantages: theorem provers are very precise, they can do lots of, often boring, proof steps in a few seconds, they keep track of the list of proof obligations, and do a lot of bureaucratic administration for the

[2] We used version 2.3 patch 1.2.2.127 - version 2.4 of PVS for all our proves, other versions of PVS might not necessarily produce the same results.

user [BJ01]. One of the most useful features of the PVS theorem prover (in this context) is that it can use so called "auto-rewrite" rules. An auto-rewrite rule is a rule as follows:

$$\tau A \rightarrow \tau B \tag{2.1}$$

where $A$ and $B$ are expressions and $\tau$ an (arbitrary) context. Thus an expression $A$ can be rewritten as a (hopefully simpler) expression $B$.

## 2.6 Hoare Logic

Hoare logic is a logic which was developed to reason about (imperative) program languages on the level of correctness of formulas. It provides rules to derive the correctness of a program from the correctness of its parts. So instead of dealing directly with the semantics of a particular program one can reason on a specification level. Meaning that one can use specifications of other methods in stead of dealing with these methods at a pure semantic level. In Hoare logic a distinction is made between *partial* and *total* correctness:

$$\{P\}\ \mathrm{S}\ \{Q\} \tag{2.2}$$

$$[P]\ \mathrm{S}\ [Q] \tag{2.3}$$

Partial correctness means that *if* the program terminates the program is correct, while total correctness states that the program terminates and it is correct. Formula 2.2 refers to partial correctness and formula 2.3 refers to total correctness. Both rules involve assertions $P$ and $Q$ in some logic (usually predicate logic), and statement S from the programming language one reasons about. $\{P\}$ S $\{Q\}$ expresses that if the assertion $P$ holds in some state $x$ and *if* the statement S, when evaluated in state $x$, terminates normally, resulting in state $x'$, then the assertion $Q$ holds in $x'$. The meaning of formula 2.3 is the same, with the difference that the statement S, when evaluated in state $x$ *must* terminate normally, i.e. one has to *proof* that the programs terminates normally. The proofs we will give in the two next chapters are all *total* correctness proofs. Hoare logic also contains some proof rules, we will present one here: the composition rule.

$$\frac{\{P\}\ \mathrm{S}\ \{Q\} \quad \{Q\}\ \mathrm{T}\ \{R\}}{\{P\}\ \mathrm{S}\ ;\ \mathrm{T}\ \{R\}}\ \text{composition} \tag{2.4}$$

The composition rule from equation 2.4 states that when you have two statements S followed by T (S;T) you have to find a suitable assertion $Q$ in order to proof (in this case partial) correctness for S and T.

In the context that we want to use Hoare logic, some additional rules have to be formulated. Classical Hoare logic does not take things like side-effects, exceptions or abrupt termination of programs into account. These concepts are typically present in object oriented programming languages like Java. Furthermore we use the annotation language JML for our specifications, so we also have to find suitable rules to deal with the modifiable clauses etc. from JML. As an example we show (a simplified version of) the composition rule in PVS notation which will be used in chapter 4:

PVS

```
composition : LEMMA
    (EXISTS(Q: [OM? -> bool]) :
        StatBehavior((# requires := P,
                        statement := S,
                        ensures := Q
        AND
        StatBehavior((# requires := Q,
                      statement := T,
                        ensures := R,))
        IMPLIES
    StatBehavior((# requires := P,
                    statement := S # T,
                    ensures := R,))
```

PVS

At Nijmegen University the LOOP project has developed these rules for the theorem prover PVS. All rules that are used have been proved to be correct (within the used semantics for Java). Some examples will be presented were these rules are used in chapter 4. A more extended look at these rules can be found in [JP00].

# Chapter 3

# Specifications, and their Proofs using automatic rewriting

In [BL99] the programming language Java is studied from an empirical point of view. More precisely, the authors develop an *empirical semantics* for Java.

In this section we will look at some selected programs from [BL99] and for each program we give a specification in JML[1, 2], and subsequent proofs which were developed using the LOOP tool and PVS. All proofs were done on a 1.4 Ghz AMD with 512 MB Ram, noted times for completing a proof may vary depending on the amount of Ghz and RAM's on other machines. The noted time we give here for producing a proof is the time it took to *rerun* a proof. The real proofs took usually much more time to produce.

## 3.1 Initialization of static variables

The first five programs deal with the initialization of static (or class) variables. Static variables are initialized when the class they belong to is initialized. This happens when for the first time a call is made to a method from that class or when a reference to a field or object from that class is made. Initialization starts with the assignment of a default value. This default value is `false` for booleans, `0` for bytes, shorts, ints, longs, reals and doubles and `null` for Objects (including arrays). After initialization one can change the default value by given an appropriate assignment. Thus `static boolean b = true` first leads to the boolean variable `b` getting value `false` when its class is initialized and then the assignment `b = true` is carried out so that in the end `b` has value `true`. Thus something else may happen in between the assignment of the default value `false` and the assignment of the given value `true`. This can lead to some (at first sight) rather unexpected outcomes of methods which we will see in our first program: JCF4.

### JCF4

JCF4 is an adaptation from [Ber01]. The specification is rather simple. It states as a pre-condition that none of the three classes (`c`, `c1` and `c2`) are initialized, furthermore all present (static) fields are modifiable and execution of method `m()` has as result that static boolean fields `result1`, `result3` and `result4` have value `true` and `result2` has value `false`. To the reader who is not (too) familiar with static initialization in Java, this last result might seam a bit puzzling.

To explain why `result2` has value `false` after execution of method `m()` we have to look at what exactly happens during the method invocation. The first thing that happens when `m()` is called is that class `c` is initialized, leading to the initialization of its static fields. Since these are all booleans, their values are set to (the default value) `false`.

[1] JML specifications are derived from existing correctness statements in PVS, developed by Jacobs.
[2] All programs with there specification can be found on the web, see [JCF].

Next the first statement in the body of `m()` is evaluated, this triggers another *chain-reaction*. Class `c1` becomes initialized (because `result1` gets the value from `b1` from class `c1`), again setting the static fields to their default value `false`.

Java with JML specification

```
package jcf4;

class c{

    static boolean result1, result2, result3, result4;

    /*@ normal_behavior
      @   requires !\is_initialized(c)  &&
      @            !\is_initialized(c1) &&
      @            !\is_initialized(c2);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(c1),
      @            \static_fields_of(c2);
      @   ensures  result1  &&
      @           !result2  &&
      @            result3  &&
      @            result4;
      @*/
    static void m(){
        result1 = c1.b1;
        result2 = c2.b2;
        result3 = c1.d1;
        result4 = c2.d2;
    }
}

class c1{
    static boolean b1 = c2.d2;
    static boolean d1 = true;
}

class c2{
    static boolean d2 = true;
    static boolean b2 = c1.d1;
}
```

Java with JML specification

Then the assignment for `b1` is carried out, leading to the initialization of class `c2`, yet again the static fields get their default value `false`.

The process continues with the assignment of `true` to `d2`, and `b2`'s value is set to `false` (the value of `d1`, remember that `d1` is initialized with its default value `false` but the assignment to value `true` is not carried out yet).

After which `b1` gets value `true` and the assignment for `d1` is carried out setting its value to `true`. At last `result1` gets the same value as `d2` which is `true`. Note that static initialization in Java happens in a *pre-order* fashion.

Since all the classes are by now initialized and all assignments in class `c1` and `c2` are carried out, the values of `result2`, `result3` and `result4` should be clear.

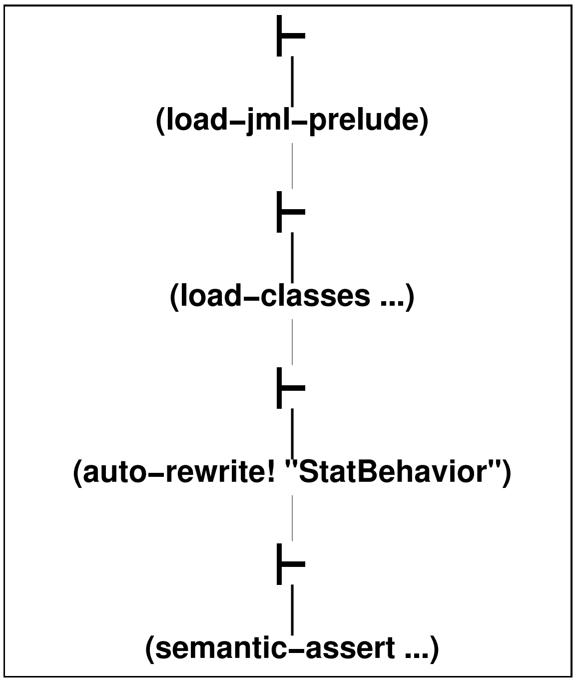


Figure 3.1: Prooftree of method `void m()` from JCF4

Indeed when one changes the class `c1` into the next form (leaving `c` and `c2` the same) the result of invocation of method `m()` will be the assignment of value `true` to all static fields of class `c`.

Java

```
class c1{
    static boolean d1 = true;
    static boolean b1 = c2.d2;
}
```

Java

To prove that the specification is correct, we first use the LOOP compiler to translate the program and its specification into a set of theories for the theorem prover PVS. This is done by execution of the following command: `run jcf4.java -o jcf4JML -class c -class c1 -class c2 -user c`, generating PVS output in four files: `jcf4JML_basic.pvs`, `jcf4JML_requirements.pvs`, `jcf4JML_Implementation` and `jcf4JML_user.pvs`.

The resulting lemma for method `m()` (in the file `jcf4JML_user.pvs`) can then be proved in PVS. The proof is fairly simple, basically what one does is, (i) load the generated (rewrite) theories, and (ii) use automatic rewriting to prove the lemma. In total this takes about 60 seconds, involving 838 rewrite steps. The corresponding proof-tree is displayed in figure 3.1

## JCF15, JCF16 and JCF17

A more complex example of the initialization of static variables can be seen in the following class:

Java with JML specification

```
package jcf15;

class c {

 static boolean result1, result2;

   /*@ normal_behavior
      @   requires !\is_initialized(d1) &&
      @            !\is_initialized(d2);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(d1),
      @            \static_fields_of(d2);
      @    ensures !result1 &&
      @             result2;
      @ also
      @ normal_behavior
      @   requires \is_initialized(d1) &&
      @           !\is_initialized(d2);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(d1),
      @            \static_fields_of(d2);
      @    ensures result1 ==  \old(d1.b1) &&
      @            result2 == !\old(d1.b1);
      @ also
      @ normal_behavior
      @   requires !\is_initialized(d1) &&
      @             \is_initialized(d2);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(d1),
      @            \static_fields_of(d2);
      @    ensures result1 == !\old(d2.b2) &&
      @            result2 ==  \old(d2.b2);
      @ also
      @ normal_behavior
      @   requires \is_initialized(d1) &&
      @            \is_initialized(d2);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(d1),
      @            \static_fields_of(d2);
      @    ensures result1 == d1.b1 &&
      @            result2 == d2.b2;
      @*/
    void m() {
    result1 = d1.b1;
    result2 = d2.b2;
  }
```

```
}

final class d1 {
  static boolean b1 = !d2.b2;
}

final class d2 {
  static boolean b2 = !d1.b1;
}
```

Java with JML specification

The specification of method `m()` of class `c` contains four different specifications which are separated by the JML key-word `also`. The specifications are all different because each has another pre-condition (the `requires` clause is different for each `normal_behavior`). Since we only look at the state of initialization for each of the classes there are 8 theoretically different specifications possible[3]. Since the initialization of class `c` is not relevant in this example we do not say anything about its state of initialization, but instead handle this inside the proof for method `m()`.

When we look at the first part of the specification we see that –as a pre-condition– we demand that both class `d1` and class `d2` are not initialized, all (static) boolean fields are modifiable and the result of execution of method `m()` should be that boolean `result1` equals `false` and `result2` equals `true`. This should not be too hard to see with the previous example in mind: The first assignment in method `m()` triggers the initialization of class `d1`, variable `b1` gets initialized at its default value `false` and then the assignment is carried out, triggering the initialization of class `d2`. The variable `b2` then gets its default value `false` and its assignment is carried out as well, since `b2` is the negation of `b1` it gets value `true` (remember that `b1` had value false after the initialization of class `d1` and that has not changed since). Since `b1` gets the negation of `b2` assigned to it, its value stays `false`.

The next part of the specifications states that class `d1` is initialized and class `d2` is not. Again all static boolean fields are modifiable and since class `d1` is already initialized we state that `result1` equals the value of `d1.b1` before method `m()` is executed and `result2` is the negation of this. We do this using a new key-word from the JML language namely `\old`. This refers to the value from a field or object *before* execution of a method (here `m()`). Note that we have to specify the values of `result1` and `result2` in terms of `d1.b1` because we have no idea what the value of `d1.b1` might be.

The third part of the specification is practically the same as the previous one, with the exception that now class `d1` is not initialized and class `d2` is. The specification should otherwise be obvious.

In the last part of the specification we have even less information since both `d1` and `d2` are initialized. This lack of information makes the ensures clause trivial.

We run the LOOP compiler in the usual way and start the proof by (i) loading the generated theories. After (ii) method expansion we split the four parts of the specification using (iii) the PVS command `(prop)` (propositional simplification). Each of the four resulting sub-trees (corresponding to the four `normal_behaviors`) can now be proved in the same manner: (iv) Make a case whether class `c` is initialized or not and (v) use automatic-rewriting until the proof is complete. This proof is completed in 113 seconds using 4081 rewrite rules. The corresponding prooftree can be seen in figure 3.2.

[3]Each class `c`, `d1`, `d2` is either initialized or not leaving $2^3 = 8$ possible combinations.

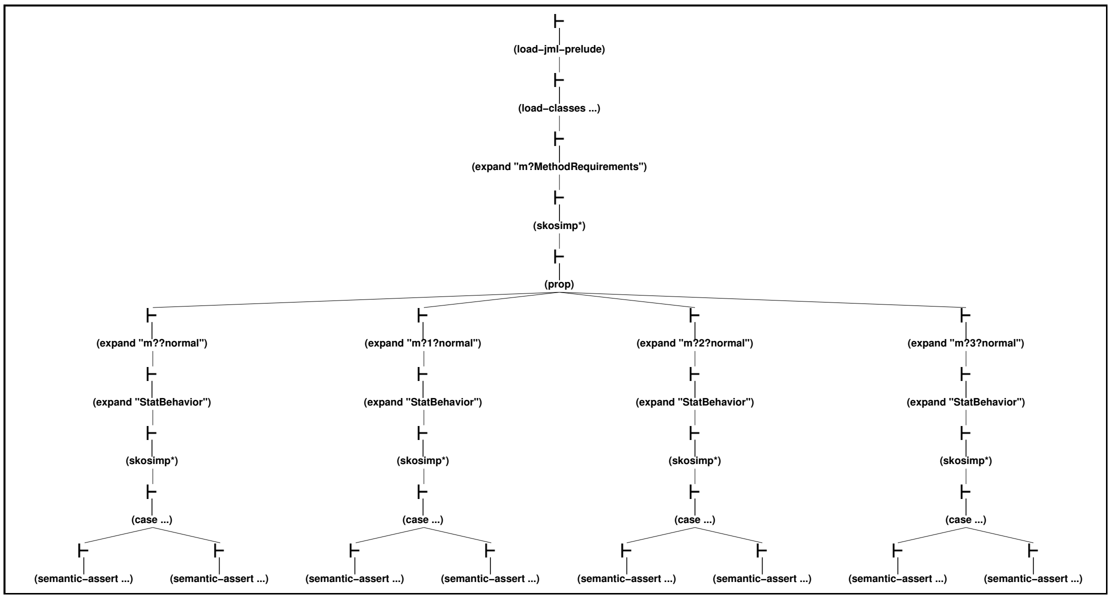


Figure 3.2: Prooftree of method `void m()` from JCF15

Java Class Families jcf16 and jcf17 are minor variations on the same theme as jcf15. The only difference between the three of them is that jcf16 and jcf17 have one extra line in the beginning of method `m()`.

JCF16's and JCF17's method `m()` are shown (in that order) below:

Java

```java
public static void m(){
        boolean b = d2.b2;
        result1 = d1.b1;
        result2 = d2.b2;
    }


public static void m(){
        d2.b2 = d2.b2;
        result1 = d1.b1;
        result2 = d2.b2;
    }
```

Java

The only thing this changes is the order of initialization. The specification and subsequent proofs are easily derived from jcf15 and are not further explained.

## 3.2 Boolean valued methods with side effects

### JCF28 and JCF28b

Consider the program below:

Java with JML specification

```
package jcf28;
class c {

  static boolean result1, result2;

  static boolean b = true;

     /*@ normal_behavior
       @   requires true;
       @ modifiable \static_fields_of(c);
       @   ensures \result == !\old(b);
       @*/
  static boolean f() {
    b = !b;
    return b;
  }

     /*@ normal_behavior
       @   requires b;
       @ modifiable \static_fields_of(c);
       @   ensures !result1 &&
       @            result2;
       @ also
       @ normal_behavior
       @   requires !b;
       @ modifiable \static_fields_of(c);
       @   ensures result1 &&
       @           result2;
       @*/
  static void m() {
    result1 = f() || !f();
    result2 = !f() && f();
  }
}
```

Java with JML specification

The specification of method `f()` states that `f()` has the following `normal_behavior`: there are no pre-conditions (i.e. `requires true;`), all the `static_fields_of` class `c` are modifiable (i.e. `result1, result2 and b`) and the result (here specified as `\result` of calling method `f()` will be the negation of the static boolean field `b`.

The first specification of method `m()` has as a pre-condition that the static boolean field `b` must be true, again all `static_fields_of` class c are modifiable, static boolean field `result1` should be equal to false (after execution of `m()`) and static boolean field `result2` should have the value true. The second `normal_behavior` requires that `b` has value false. The modifiable clause does not change and the fields `result1` and `result2` are both equal to true.

To see that the above specification is correct the reader has to realize that two different phenomena are present in this example. First there's the notion of *side effect*. Method `f()` has a side effect in that it returns the negation of field `b` and (as a side effect) sets the value of `b` to its negation. Furthermore the logical disjunction (`||`) and conjunction (`&&`) are *conditional* (in logic this is known as *left sequential*) operators meaning that if (in case of a disjunction) the left expression evaluates to `true` the second (and third and fourth etc. ) expression no longer is evaluated and the whole disjunt evaluates to `true`. Or (in case of a conjunction) if the left expression evaluates to false the remainder of the expression is not evaluated making the whole conjunct becomes `false`.

The file `jcf28JML_user` is used to prove that the methods of class `c` are correct according to there specification. For each of the *three*[4] methods a lemma is generated which can (hopefully) be proved.

The proof for method `f()` is rather straightforward: (i) load the generated theories, (ii) expand method definitions, (iii) make a case whether class `c` is initialized or not and (iv) auto-rewrite until Q.E.D. This proof takes about 30 seconds, involving 571 rewrite steps.

Since the specification of method `m()` is somewhat harder, its proof is too. We begin in the usual way:(i) load the generated theories, (ii) expand method definitions then (iii) separate the two specifications using `(prop)` which splits conjunctions and then the proof continues normally with (iv) make a case whether class `c` is initialized or not and (v) auto-rewrite until Q.E.D. Rerunning this proof takes about 4 minutes and uses 2476 rewrite rules.

Java also has *non conditional* logical operators for disjunction (`|`) and conjunction (`&`). If we use these in jcf28 instead of the conditional operators we obtain jcf28b. Since everything else stays the same we only show the specification for method `m()`. Since it is also possible to combine several JML `behavior`'s we will take this opportunity to show this as well:

Java with JML specification

```
/*@ normal_behavior
  @   requires true;
  @ modifiable \static_fields_of(c);
  @   ensures (\old(b) <=> !result1 && result2);
  @*/
 static void m() {
    result1 = f() || !f();
    result2 = !f() && f();
}
```

Java with JML specification

This specification is different because both the expressions of the disjunct are evaluated. We used the bi-implication (`<==>`) to show that the value of both `result1` and `result2` is depending on the value of `b` before execution of method `m()`.

The proof for jcf28b is almost identical to the previous proof, because we combined the two `normal_behavior`'s into one `normal_behavior` we do not have to use `prop` to split the different parts of the specification. All other steps stay the same.

[4]Besides the obvious methods `f()` and `m()`, there is a third method, namely the standard constructor of the class.

## 3.3 Objects with boolean instance fields

In the previous examples we only used the primitive type `boolean`. These were all static (class) variables. The next examples will contain objects with `boolean` instance fields (fields belonging to some specific object). Besides instance fields, objects can also access the (static) fields of a class. The difference between these two is that every object has its own instance fields and static fields are shared for all objects (of the same class).

### JCF31, JCF33 and JCF34

The file `jcf31.java` (shown below) contains objects `x` and `y` (created in the body of method `m()`), since `x` and `y` are instances of class `c1` they have two boolean fields: `b` and `c`.

Java with JML specification

```
package jcf31;

public class c {

  static boolean result1, result2, result3,
                 result4, result5, result6,
                 result7, result8, result9;

   /*@ normal_behavior
     @   requires true;
     @ modifiable \static_fields_of(c),
     @            \static_fields_of(c1);
     @   ensures  !result1 &&  result2 &&
     @            !result3 &&  result4 &&
     @            !result5 && !result6 &&
     @            !result7 &&  result8 &&
     @            !result9;
     @*/
  public static void m() {
    c1 x = new c1();
    c1.d = false;
    c1.e = true;
    x.b = false;
    x.c = true;
    result1 = c1.d;
    result2 = c1.e;
    result3 = x.b;
    result4 = x.c;
    x.b = x.c;
    c1 y = new c1();
    y.b = y.c = x.c = false;
    result5 = y.b;
    result6 = y.c;
    result7 = x.c;
    result8 = x.b;
    result9 = y.d;
  }
}
```

```
class c1 {

  boolean b = true, c = false;

  static boolean d = true, e = false;

}
```

Java with JML specification

The specification of method `m()` should be clear from the previous examples. The only thing we have not seen before is the absence of a pre-condition (`requires true`). Inside the proof we make the case distinctions between classes `c` and `c1` being initialized or not.

The proof is as before except for the second case distinction: (i) load the generated theories, (ii) expand method definitions, (iii) make a case whether class `c` is initialized or not (which results in two branches in the prooftree with an identical continuation of the proof for both branches), (iv) make another case whether class `c1` is initialized or not (again resulting in two branches in the prooftree), and (v) auto-rewrite until Q.E.D.

Rerunning the proof for method `m()` was finished in about 7 minutes and 8701 rewrite steps were used.

Two more examples were done, jcf33 and jcf34. Both are similar to jcf31 and can be found on the web [JCF].

## 3.4 Objects with instance methods

We already saw the use of *static* methods, this sections involves *instance* methods (i.e. methods working on objects). Note that in Java these methods are not "modifiable", this is not true for all object oriented languages.

### JCF41

Java with JML specification

```
package jcf41;

public class c {

    /*@ normal_behavior
      @   requires !\is_initialized(c4) &&
      @            !\is_initialized(R);
      @ modifiable \static_fields_of(R),
      @            \static_fields_of(c4),
      @            R.r[*];
      @   ensures R.r[0]==5 &&
      @           R.r[1]==5 &&
      @           R.r[2]==4 &&
      @           R.r[3]==3 &&
      @           R.r[4]==1 &&
      @           R.r[5]==4 &&
      @           R.r[6]==4 &&
      @           R.r[7]==5 &&
      @           R.r[8]==2 &&
      @           R.r[9]==1 &&
      @           (\forall (int i) 10 <= i && i < R.r.length ==>
      @                           R.r[i]==0);
      @*/
  public static void m() {
    c1.f(c4.x.g2().g2(), c4.x.g1()).y.m();
    c2.f(c4.x.g1().g1(), c4.x.g2()).y.m();
  }
}
class c0 {
  public void m() {R.store(1);}
}
class c1 {
  static c3 f(c3 x, c3 y) {
    R.store(3);
    return x;
  }
}
class c2 {
  static c3 f(c3 x, c3 y) {
    R.store(2);
    return x;
  }
```

```
}
class c3 {
  c0 y = new c0();
  c3 g1() {
    R.store(4);
    return this;
  }
  c3 g2() {
    R.store(5);
    return this;
  }
}
class c4 {
  static c3 x = new c3();
}
class R {
    static int[] r = new int[100];

    static int pos = 0;
    static void store(int result) {
        r[pos++] = result;
    }
}
```

Java with JML specification

The specification of the above program uses some new JML constructs. The requires clause should be familiar. We assume that both class `c4` and `R` are not initialized because we do not want to deal with the difficulties associated with static initialization (as discussed in previous examples).

In the modifiable clause we see our first new JML notation, `R.r[*]` means that all indexes of array `r` from class `R` are modifiable. We have to add this in our specification because when class `R` is initialized all indexes get there default value (in this case `0` because it is an array of integers).

The post-condition states which values the first ten indexes should have after execution of method `m()`. The remainder of the array-indexes do not get modified after initialization so we state that they should have their default value `0`. We use quantification to do this. The arrow's (`==>`) meaning is the same as the standard logical implication.

Note that the `result`'s get first evaluated and after their assignment in the array the `pos` variable is changed so the next index is free for another assignment.

The evaluation of the expressions in the body of `m()` are carried out in *left-most inner-most* order, so lets look at the first expression. The following method are executed in order: `x.g2()`, `x.g2()`, `x.g1()`, `c1.f(x,x)` and `y.m()`. Resulting in the first five indexes of `R.r[]` getting values `5,5,4,3` and `1`. The second expression is evaluated in a similar fashion.

The proof is constructed in the same fashion we have already seen before: (i) load the generated theories , (ii) auto-rewrite until Q.E.D. This proof was completed in 1.5 ours using 5143 rewrite rules.

### JCF43

The JavaCck class jcf43 is a modification of jcf41. Again an array of integers with length 100 is used for storing of the results. There is however a difference in that instead of using one object, as in jcf41, here every method execution results in returning a new object. There are 3 methods `g( )` from which we show one here:

Java

```
c2 g(c1 x) {
       store(1);
       return new c2();
  }
```

Java

The method `m()` we want to prove calls these `g( )` methods in total 18 times resulting in the creation of in total 21 objects (three objects are created inside `m()`'s body, they are needed to call the three `g( )` methods). This makes the proof state a lot bigger, which also has its result on the proof. The proof is exactly the same as the proof from jcf41. However jcf43's proof took almost 87 hours to complete, using 20445 rewrite rules. Naturally if a relative small method like `m()` already takes this long, it should be obvious that we need another approach to show correctness of "real" programs. In the next chapter we will look at such an approach based on Hoare logic, where we also will have a more "in depth" look at JCF43.

## 3.5 Exceptions

Exceptions are a special kind of objects which are returned after an "illegal" action in a Java program occurs. Illegal actions can both be user defined or generated by the language itself[5].

### JCF54

The specification of method `m()` in jcf54 (see below) has no `normal_behavior`. Instead we see an `exceptional_behavior` which indicates that the method throws some kind of exception in stead of terminating normally. The requires clause is as before. However the modifiable clause contains some new items. Besides the usual `\static_fields_of()` we see a new JML notation: `\fields_of()`. This states that the (instance) fields of object `x1` ... `x6` are modifiable. This could be made more precise by explicitly listing which fields of these objects can be modified.

Java with JML specification

```
package jcf54;
public class c {

  static boolean result1, result2, result3, result4,
                 result5, result6, result7;
```

[5] technically speeking these last kind of exceptions are generated by the Java virtual machine. We hold the view that the virtual machine is a part of the language, but there is no general consensus on this topic.

```
  static c1 x1 = new c1();
  static c1 x2 = x1;
  static c1 x3 = x2;
  static c1 x4 = x3;
  static c1 x5 = new c1(x4);
  static c1 x6 = x5.v;
  static boolean b = x6 instanceof c1;

    /*@ exceptional_behavior
      @   requires !\is_initialized(c) &&
      @            !\is_initialized(c1);
      @ modifiable \static_fields_of(c),
      @            \static_fields_of(c1),
      @            \fields_of(x1),
      @            \fields_of(x2),
      @            \fields_of(x3),
      @            \fields_of(x4),
      @            \fields_of(x5),
      @            \fields_of(x6);
      @   signals(NullPointerException)
      @             result1 &&
      @             result2 &&
      @            !result3 &&
      @            !result4 &&
      @             result5 &&
      @             result6 &&
      @            !result7;
      @*/
  public static void m() {
    result1 = x1 == x4;
    result2 = c1.x1.empty;
    result3 = c1.x2.empty;
    result4 = x5.empty;
    result5 = x5.v.empty;
    result6 = b;
    result7 = x6.v.empty;
  }
}
class c1 {

  boolean empty;
  c1 v;
  static c1 x1 = new c1();
  static c1 x2 = new c1(x1.empty);

  c1(){empty = true;}
  c1(boolean b){empty = false;}
  c1(c1 u) {empty = false;v = u;}
}
```

Java with JML specification

Instead of the ensures clause we now get a `signals()` clause. This specifies which exception will be thrown and which post-condition will be obtained in the resulting "exceptional" state. As one can see the exception is a `NullPointerException`, which occurs when a reference to a non existing field or object is made. The post-condition is as usual.

To see why the exception is thrown we have to look at last assignment in the body of `m()` which states : `result7 = x6.v.empty;`. We see that `c6` is the same as `x5.v` which is the same as `c4`. And since `c4` has an empty `v` field there is also no `v.empty` so a `NullPointerException` occurs.

The proof was constructed as seen before: (i) load definitions and (ii) rewrite. It took some 2 minutes using 2365 rewrite rules.

## 3.6 Inheritance

*Inheritance* is a mechanism which is typical for object oriented languages (such as Java). It means that classes can use the fields and methods from their super-class. Another phenomenon related to inheritance is *hiding* (also called shadowing). When a class inherits a field from (one of its) super class(es) it can hide this field by defining a field with the same name. This same mechanism also applies to methods and is then called *overriding*. Hiding is present in the next example: jcf63.

### JCF63 and JCF65

Java with JML specification

```
package jcf63;
public class c extends c1 { }
class c1 extends c2 {

  static boolean result1, result2, result3, result4, result5, result6;

    /*@ normal_behavior
      @   requires !\is_initialized(c1) &&
      @            !\is_initialized(c2) &&
      @            !\is_initialized(c3);
      @ modifiable \static_fields_of(c1),
      @            \static_fields_of(c2),
      @            \static_fields_of(c3);
      @   ensures  !result1 &&
      @             result2 &&
      @            !result3 &&
      @            !result4 &&
      @             result5 &&
      @            !result6;
      @*/
  static void m() {
    result1 = a;
    result2 = b;
    result3 = c;
    result4 = d;
    result5 = e;
    result6 = !f;
  }
```

```
}

class c2 extends c3 {

  static boolean b = true;
  static boolean c = false;
  static boolean d = !c3.d;
}

class c3 {

  static boolean a = false, b = false, c = false,
                 d = true,  e = true,  f = true;
}
```

Java with JML specification

The specification of method `m()` is as usual. The first assignment in `m()`'s body is a clear example of inheritance. Class `c1` inherits a field `a` from class `c2` who again inherits it from class `c3`. So `result1` gets the same value assignment to it as `c3.a` namely, `false`. The next assignment also shows us hiding of a field. Again the field `b` is inherited from class `c2`. Class `c3` also has a field `b`, but this field is hidden. Meaning that `result2` has value `false` assigned to it. The other assignments are obtained in a similar fashion.

The proof for jcf63 is again a (i) load generated theories, and (ii) auto-rewrite sequence. It took one minute using 1363 rewriting steps to prove that the specification for `m()` was correct.

The Java file jcf65.java is similar to jcf63.java, the only difference being that overriding for methods is showed instead of hiding of fields. The corresponding proof for jcf65 the same as before. It took 1.5 minutes and used 1267 rewriting steps.

## JCF80

Java with JML specification

```
package jcf80;

public class c {

    static c3 x = new c3();

    /*@ normal_behavior
      @   requires  !\is_initialized(c)  &&
      @             !\is_initialized(c1) &&
      @             !\is_initialized(c2) &&
      @             !\is_initialized(c3);
      @ modifiable   \static_fields_of(c),
      @              \fields_of(x);
      @    ensures   !\result;
      @*/
    public static boolean m(){ return x.m();}
}

class c1 {
    private boolean q = true;
}

class c2 extends c1 {
    boolean q;
}

class c3 extends c2 {
    public boolean m(){ return q;}
}
```

Java with JML specification

The program above (jcf80) is another example which uses hiding and inheritance. Again we assume that all classes are not initialized and all fields are modifiable. Object `x` inherits a boolean field `q` from class `c2` (the super class from `class c3`). The same field hides the boolean field from `c3`. This field gets no value assigned to it so it has default value `false`.

The proof was as usual and took 5 and a half minute. 1348 rewriting steps were used.

# Chapter 4

# Specifications, and their Proofs using Hoare logic

As we saw in the last chapter (especially at page 27) there are some drawbacks when using only automatic rewrite rules to prove (JML) specification correct for Java programs. These proofs can take quite a while to complete, moreover for more complex examples these kind of proofs do not work at all (e.g. with while loops[1]). In this section we will use another approach to prove the correctness of Java programs: so called Hoare-logic (see also section 2.6), especially adapted to JML.

Typically in Hoare-logic we use composition-rules to separate statements. This involves the introduction of an intermediate state in the program. Once an individual statement is isolated we can (hopefully) use the *semantic* proof method seen in the previous chapter to finish the proof. But sometimes the structure of individual statements has to be decomposed further with additional Hoare logic rules.

The composition rule mentioned above is not the only Hoare rule that we can use. Basically there are separate rules to handle every possible construct within (sequential) Java, for instance, `if..then..else`, `while` , `for`, `new` and `&&` all have there own Hoare rule. A Hoare rule is a lemma in higher order logic within PVS that is proven to be correct within the Java semantics. Typically we do not use the lemma directly, but instead use a strategy which applies the lemma and (tries) to prove generated TCC's (proof obligations) automatically. All Hoare style proofs start with the PVS strategy `(jml-start)`. This strategy applies a number of rewrite rules and basically generates a proof state on which the user can start using Hoare rules. We will discuss some of these in more detail as we encounter them in our examples.

Typically it is not hard to determine *which* Hoare rule we have to use. The Hoare rules are syntax driven. In other words they follow the syntactic form of the program. The prooftree of a (proven) specification therefore has the same structure as the the program (we will see this in the next sections). What *is* hard is that we have to introduce intermediate predicates for many Hoare rules. Such a predicate typically specifies what values are assigned to certain fields and which fields are modifiable. We have to prove that the intermediate predicate holds in the intermediate state that is created by using the Hoare rule.

The remainder of this chapter describes Hoare style proofs for three different programs. We will start with non static variants of JCF28 and JCF43 and will end this chapter with a proof of JCF4 (which we opened the previous chapter with). We have chosen to begin with non static variants, because there are at the moment no suitable Hoare rules which deal with static initialization. Hence we use an ad hoc solution.

This chapter describes the details of applying the Hoare logic for JML for three case studies. It will necessarily be full of technical details, which may be hard to follow by those who are not

[1] Recall that JavaCck is Turing complete.

intimately familiar with Hoare logic, Java, JML and PVS. But hopefully the reader will grasp the essence of the proofs described here.

## 4.1 JCF28ns

We already saw a proof for the static-variant of JCF28 (in the previous Chapter), here we will give a detailed proof of a non static variant of JCF28.

We start with running the LOOP compiler as we did before, however we use an additional argument so that the generated files are suitable for using Hoare logic. This additional argument is (not surprisingly) `-hoare`. After type-checking we can now start with the correctness proof for method `boolean f()`.

### Method `boolean f()`

The specification for method `f()` is slightly different from the one we used at page 21:

Java with JML specification

```
 /*@ normal_behavior
   @   requires true;
   @ modifiable b;
   @   ensures b == !\old(b);
   @*/
boolean f(){
   b = !b;
   return b;
}
```

Java with JML specification

Of course since field `b` is no longer static, the modifiable clause changes slightly, but (more surprisingly) our `ensures` clause also changes. We state that `b == !\old(b)` holds in the post state instead of `\result == !\old(b)`. We do this because we want to use the specification of `boolean f()` in our proof for `void m()`. If we only state that `\result == !\old(b)` should hold in the post state we have no way of knowing what the value of `b` is in the post state. We could of course add that `\result=b`, but this would make the ensures clause more complicated then necessary. The (Hoare) proof for this method is trivial, we will not further explain it here. Rerunning this proof took 1 minute.

### Method `void m()`

Proving the specification for method `m()` involves some subtleties as we shall see shortly. We want to keep the specification as general as possible so we combine the two different `normal_behavior`'s into one (like with JCF28b). This gives the following specification:

Java with JML specification

```
 /*@ normal_behavior
   @   requires true;
   @ modifiable b,result1,result2;
   @   ensures  (\old(b)  ==> !result1 && result2) &&
   @            (!\old(b) ==>  result1 && result2);
   @*/

void m(){
   result1 = f() || !f();
   result2 = !f() && f();
}
```

Java with JML specification

After the `(jml-start)` strategy. we use a `(composition-replace..)` so we can separate the two expression in `m()`'s body. This helps us to get rid of the (default) assumption that nothing is modifiable (which no longer holds in the post state). We also assert that after evaluation of the first expression `b` has value `true`, `\old(b)`$\rightarrow$ `result1=false`, $\neg$`\old(b)`$\rightarrow$ `result1=true` and `result2` is not modified.

To see why this is so we continue with our proof. We have to use a Hoare rule for the conditional or (`||`). Here we have a difficulty, because when `\old(b)` is true we only evaluate the first expression (before `||`), but if `\old(b)` is false we also have to evaluate the second expression. One possible solution might be to make a case distinction whether `\old(b)` is true or not, but there is a more elegant solution. The `(orelse)` rule we use here takes as an argument a state (like the `(composition)` rule) and an additional argument from type `bool` which states if only the first or both parts of the conditional or have to be evaluated (i.e. if the first part is true or not).

What we do is that we let the value from the `bool` depend on the value from `\old(b)`. So in case `\old(b)` is true the first expression returns `false` triggering the second expression to be evaluated (again resulting in false) and `b` getting value `true` ($\neg\neg$`b = b`). While if `\old(b)` has value false the first expression returns `true` , the second expression is not evaluated anymore and `b` also has value `true`. The remainder of this branch is handled by using the specification of method `f()`. The strategies `(inst-spec)` and `(use-spec)` handle this for us. Basically what these do is load the specification from a method and instantiate it with the current state and then produce two branches, (i) the right branch in which must be proved that the specification can be used here (i.e. that the `requires` clause of the used specification holds) and (ii) the left branch in which the "real" proving is done (i.e. the argument to the `(orelse)` strategy).

The second expression in `f()`'s body is evaluated in a similar fashion, only now we use the `(andelse)` rule. This rule also gets as an argument a state an a `bool`, the `bool` is again used to determine if both or only the first statement must be evaluated. Since we know that the value of `b` is `true` we can simply say that the `bool` in the rule is also true (i.e. both sides of the conditional and (`&&`) must be evaluated). Figure 4.1 shows the resulting prooftree which took a bit more then 2 minutes to (re)produce.

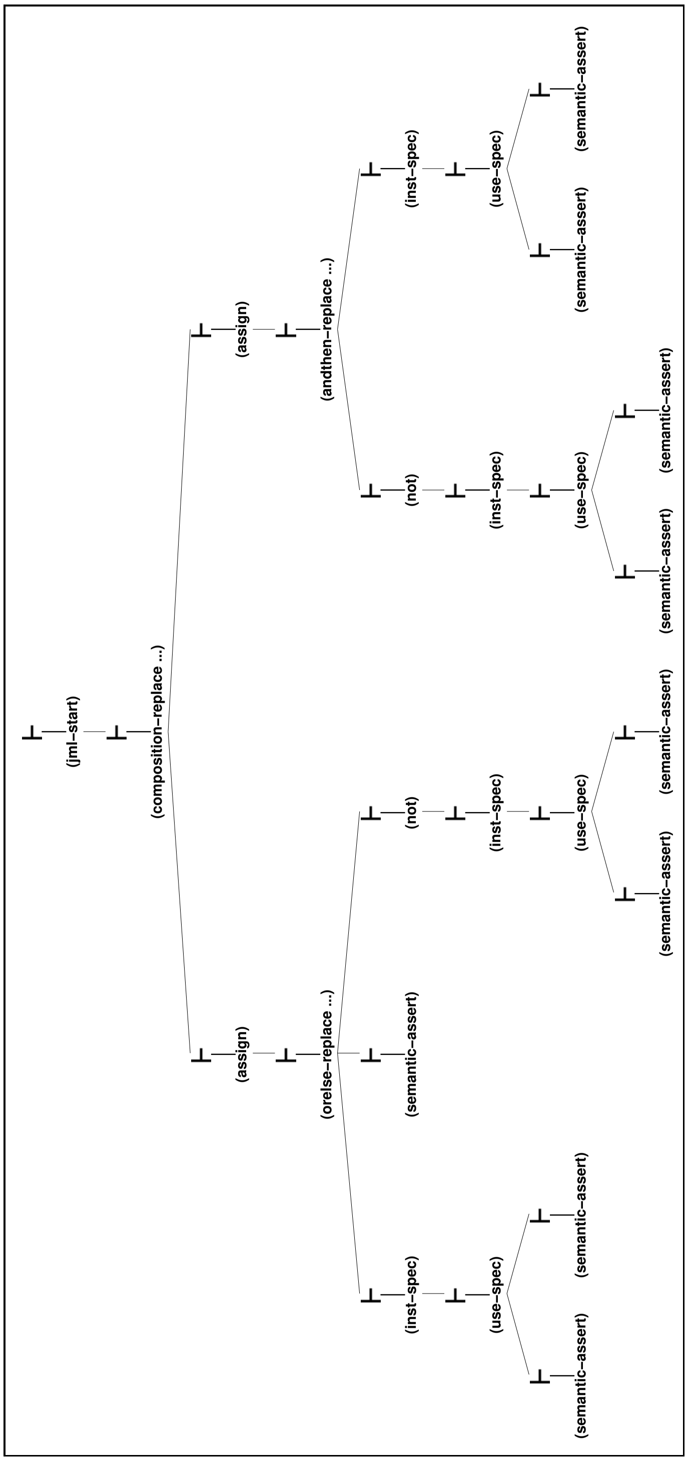


Figure 4.1: Prooftree of method `void m()` from JCF28ns

## 4.2 JCF43ns

JCF43ns contains 4 classes from which 3 classes are empty (class `c1`, `c2` and `c3`). The empty classes are only used to create instances (i.e. objects) of that class. The main class (class `c`) contains 6 methods: the default constructor (which will not be used), method `void store(int result)`, method `c2 g(c1 x)`, method `c3 g(c2 x)`, method `c1 g(c3 x)` and the main method `void m()`. We will give Hoare-style proofs for 3 of these[2].

### Method `void store(int result)`

We start with the method `void store(int result)`, this method is used in the three similar `g()` methods to store integers in a array. The method and it's JML specification can be seen here:

Java with JML specification

```
int[] r = new int[100];
int pos = 0;

/*@ normal_behavior
  @   requires  r        != null     &&
  @             pos      >= 0        &&
  @             pos      <  r.length;
  @ modifiable  r[pos],
  @             pos;
  @   ensures   r[\old(pos)] == result &&
  @             pos == \old(pos) + 1   &&
  @             pos <= r.length;
  @*/
void store(int result) {
   r[pos++] = result;
}
```

Java with JML specification

There are two *instance fields* present, an array from integers called `r` and an integer `pos` which is used as index for the array. All the method does is that it puts an integer `result` at the position `pos` in array `r` and then increases the value of `pos` with 1. Because we want to keep our specification as general as possible we have to make sure that the `requires` clause is not too strong. So we state that r is not null (otherwise a NullPointerException will be thrown) and we state that `pos` must be within the bounds of the array (otherwise an ArrayOutOfBoundsException will be thrown). Note that in case the length of array `r` equals zero we know that evaluation of `void store(int result)` results in throwing an ArrayOutOfBoundsException. This is however no problem in our proof because in that case our requires clause is `false`: $\texttt{pos} \geq 0 \wedge \texttt{pos} < 0 \rightarrow \texttt{false}$. With a false assumption we can prove anything.

It should be clear that `pos` and the index of array `r` at position `pos` are modifiable[3]. The `ensures` clause states that the index of array `r` at position `pos` *before* method execution becomes equal to the integer `result`, the integer `pos` will be incremented with 1 and after method execution `pos` is smaller *or equal* to the length of array `r`[4].

[2] the three `c? g(c? x)` methods are all similar so only one proof of these 3 methods will be presented here.

[3] Recall that in the modifiable clause statements and expressions are evaluated in the *pre-condition*, i.e. the state before execution of a method. So we do not have to use the JML key word `\old`.

[4] The integer `pos` can be equal to the length of `r` if `pos` in the pre-condition is equal to the last index of `r`.

Since this method is obviously very simple the proof for this method is quite trivial, in fact we could easily use the semantic proof-method from the last chapter to show its correctness. However since this chapter involves Hoare logic we will use this to construct the proof. The proof can be completed with a number of standard Hoare rules all of which do not take any arguments. So the user of PVS only has to select the appropriate rule and PVS does the rest. It suffices to say that the proof took about 100 seconds to reproduce. The next method we look at is more interesting since we use the specification from `void store(int result)` in its proof.

### Method `c2 g(c1 x)`

Methods `c2 g(c1 x)`, `c3 g(c2 x)` and `c1 g(c3 x)` are all similar, the only difference is that each returns another object, each accepts another object and each stores another integer in array `r` ( 1, 2 and 3 in that order). We will here only look at the first method. Method and specification look as follows:

Java with JML specification

```
/*@ normal_behavior
  @   requires  r        != null     &&
  @             pos      >= 0        &&
  @             pos      <  r.length;
  @ modifiable  r[pos],
  @             pos;
  @   ensures   r[\old(pos)] == 1       &&
  @             pos == \old(pos) + 1    &&
  @             pos <= r.length         &&
  @             \invariant_for(\result) &&
  @             \result != null;
  @*/
c2 g(c1 x) {
    store(1);
    return new c2();
}
```

Java with JML specification

The specification is almost equal to that of the previous method (which is not surprising considering that the first statement in the body is a method invocation of `void store(int result)`). We now know which value the integer `result` has so we specify that `r[\old(pos)]` has value `1` after method execution. Furthermore because a new object will be created we demand in our specification that this object should satisfy its invariant and that it is not `null` (this last information will be needed in the proof for method `m()` where we use this specification).

We start the proof in the usual way with the PVS strategy `(jml-start)` (the prooftree for this method can be found in figure 4.2). After this step is completed we use the `composition` lemma (see section 2.6) to split the body of `c2 g(c1 x)` in two parts, the two parts being the two statements `store(1)` and `return new c2()`. We know that after execution of `store(1)` the first three statements in our ensures clause will be true. So when `Z` is the pre-condition and `P` is the post-condition we introduce a new state `ZP` which is the state of the method after evaluation of `store(1)` and before evaluation of `return new c2()`. There is one more subtlety involved here. Often, we assume that the `requirements` clause stays intact during the whole evaluation of a method body. In PVS this looks as follows:

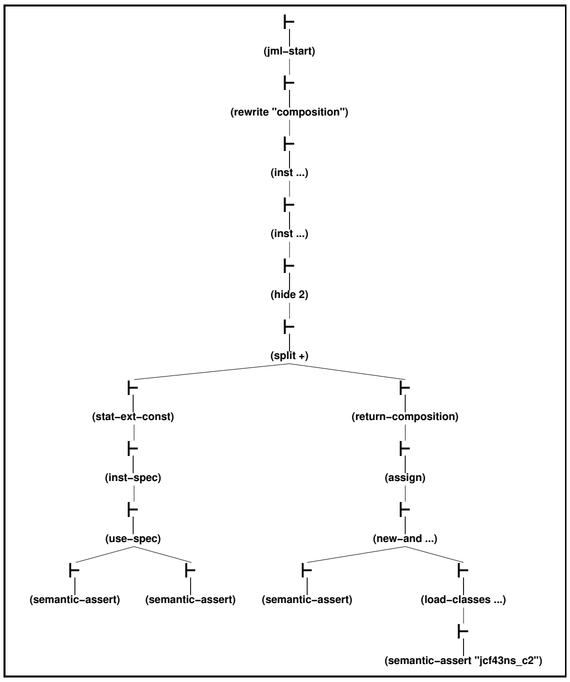


Figure 4.2: Prooftree of method `c2 g(c1 x)` from JCF43ns

PVS

```
requires:=
     LAMBDA (x: OM?):
       ...
       (g?c1?normal_requires(abstract(c?!1)(p?!1))(p?!1)(x))
       AND
       ...
```

PVS

where `OM?` stands for object memory, `x` is a post state and `p?!1` is "this" i.e. the object on which the method is invoked. We also see that we are dealing with the requires clause for method `c2 g(c1 x)` because of the identifier `g?c1?` [5] and that "this" (`p?!1`) is an instance of class `c` (`abstract(c?!1)(p?!1)`).

The problem in this case is that one condition in our `requirements` clause does not hold after evaluation of `store(1)`: the requirement that `pos < r.length`. Just assume that `pos = r.length -1` before execution. This does not violate the `requirements` clause so the method can be executed, but after evaluation of `store(1)` `pos = r.length` so our original requirement does not hold anymore. Thus state `ZP` should be same as state `Z` with the assumption `pos < r.length` in state ZP removed and `r[\old(pos)] == 1`, `pos == \old(pos) + 1` and `pos <= r.length` added. In this particular case we do this by rewriting the composition lemma and instantiate the lemma with the values we want (see figure 4.2).

[5] In PVS we use the question mark "?" in our names because these are illegal in Java names, making it easy to see what is PVS and what is not.

We now have two branches (because we used composition) . We continue with the first (left) branch (where `store(1)` is evaluated) were we use the strategy `(stat-ext-const)` after which we can use the specification from the method `void store(int result)`. We do this in two steps, first we instantiate the method in our current state (using `(inst-spec)`) and then we use the specification with the strategy `(use-spec)`. This leads again to two branches (as seen in the example for `void m()` from JCF28ns), both of which are trivial so they are proved with the strategy `(semantic-assert)`.

The second branch we obtain after the use of the `composition` lemma is evaluated in state `ZP`. Here a new object from class `c2` is created and returned. We first use the strategy `returncomposition` which removes the `return` in a suitable way. A `return` statement leads to a from of abnormal termination. What the `(return-composition)` strategy does is remove all statements after the return statement (because these will not be evaluated).

Next an (implicit) assignment is carried out (using the strategy `(assign)`). We can now use the `(new-and)` strategy to create the new object. Here we have to give the strategy an argument:`(new-and "Lambda(x:OM?): heap?top(Z?!1) <= heap?top(x) -1")`. This states that the heap will be incremented with 1 (i.e. the new object). So we strengthen our original assumption `"Lambda(x:OM?): heap?top(Z?!1) <= heap?top(x)"` (which is the default assumption) to the one we use as an argument in the `(new-and)` strategy. Where `x` is the post-condition `P`. This again results in two branches, in the left the `heap?top` will be incremented. This branch is once again proved using the strategy `(semantic-assert)`. In the other branch the constructor from class `c2` is called. Since we do not have a specification for this (default) constructor we continue in this branch with a proof on Implementation level. We load the implementation from the classes we need (`c2` and `java.lang.Object`) and finish the proof with a `(semantic-assert)`. Rerunning this proof took about 3 minutes.

## Method `void m()`

Java with JML specification

```
/*@ normal_behavior
  @   requires r        != null          &&
  @            pos      >= 0             &&
  @            pos      <  r.length - 15 &&
  @            r.length >= 15;
  @ modifiable r[pos .. pos+14], pos;
  @ ensures  r[\old(pos)]   ==1 &&  r[\old(pos)+1] ==2 &&  r[\old(pos)+2] ==3 &&
  @          r[\old(pos)+3] ==1 &&  r[\old(pos)+4] ==2 &&  r[\old(pos)+5] ==2 &&
  @          r[\old(pos)+6] ==3 &&  r[\old(pos)+7] ==3 &&  r[\old(pos)+8] ==1 &&
  @          r[\old(pos)+9] ==1 &&  r[\old(pos)+10]==2 &&  r[\old(pos)+11]==3 &&
  @          r[\old(pos)+12]==2 &&  r[\old(pos)+13]==3 &&  r[\old(pos)+14]==1;
  @*/
void m() {
    c1 x1 = new c1();
    c2 x2 = new c2();
    c3 x3 = new c3();
    g(x1); g(x2); g(x3);
    g(g(x1)); g(g(x2)); g(g(x3));
    g(g(g(x1))); g(g(g(x2)));
}
```

Java with JML specification

Method `void m()` (displayed above) is the main method of JCF43ns[6] it concerns inside-out evaluation. Again we want the specification to be as general as possible so we state in the `requires` clause the assumptions we have seen before that array `r` should not be `null` and `pos` should be bigger then or equal to 0. In addition we want to say that `pos < r.length` we strengthen this assumption by demanding that `pos < r.length - 15`. We do this because we know that 14 successive positions into `r` will get a value between 1 and 3 assigned to it. Therefore we also state that the length of `r` should be equal to or bigger then 14 (again because otherwise an ArrayOutOfBoundsException will be thrown). Knowing this it is easy to see why the `modifiable` clause holds. The `ensures` clause of the specification uses the JML keyword `\old()` a lot, because we can only say something about the different values relative to the value of `pos` in the pre state. The remainder of this clause should by now be clear to the reader.

We start the proof from this method with the (by now familiar) strategy `(jml-start)`. Then we can use the `composition` rule to isolate the first statement. We do with the following arguments:

[6] We have taken the liberty of removing one statement out of `m`'s body, there was also a call to `g(g(g(x3)))`. However PVS could not handle this last statement as its proof-buffer became too large and it crashed. PVS can not handle proof-buffers larger then 500 MB

PVS

```
(composition-and "Lambda (x:OM?) : heap?top(Z?!1) <= heap?top(x) - 1
                       AND
                       reference??(get?ref(stack?(st!1, 0))(x))
                       AND
                       LET ref1 = refpos?(get?ref(stack?(st!1, 0))(x))
                       IN
                          ref1 = heap?top(x) - 1
                          AND
                          SubClass?(get?type(ref1)(x), \"c1\")
                          AND
                          get?dimlen(ref1)(x) = (0,0)")
```

PVS

We use `composition-and` because we already have the assumption `LAMBDA (x:OM? ): modifiable_nothing(st!1)(Z?!1)(x)`[7] which we want to keep using for now. The arguments we introduce here state that in the post state a new object has been created: `x1` (corresponding with `ref1` above). Because we use this object in later statements we have to take this information with us for the remainder of the proof. The branch which results from this is similar to the one we saw before in the previous section (involving `c2 g(c1 x)`) after the `(return-composition)` rule in the proof. The only difference being that we now load the implementation for class `c1` and `java.lang.Object`. The next two compositions are similar, again we have to take the information concerning the new object with us (and the previously created ones).

We now are at the point that for the first time another method is called. Before we use another composition rule we again weaken our specification. We do this because `pos < r.length - 15` does not have to hold in the state after which `pos` has been increased by 1.

We continue with the next PVS command (again `composition`) which states that after the call to method `c2 g(c1 x)` the value of `pos` will be incremented by 1 and `r[\old(pos)]` will have value 1 assigned to it. We use the replace command because we want to weaken our assumptions. The assumption `LAMBDA (x:OM? ): modifiable_nothing(st!1)(Z?!1)(x)` will not hold anymore because both `pos` and `r[pos]` will be modified after the evaluation of `g(x1)`. All other values of array `r` will not be modified by this method call so we add

$$\forall\ \texttt{(i:nat)}:\ \texttt{i} \geq \texttt{pos} \wedge \texttt{i} < \texttt{r.length} \rightarrow$$
$$\texttt{get?int(heap?(refpos?(r), i))(post?)} = \texttt{get?int(heap?(refpos?(r), i))(pre?)}$$

to meet this requirement. In our semantics in PVS this looks as follows:

[7] besides the usual assumptions which will not be altered during the proof, like the `requirements` and remainder of the `modifiable` clause.

PVS

```
(composition-replace "LAMBDA (x: OM?):
....
AND
LET
   pos =
   get?int(FIELD?POSITION((\"c\" :: string))((\"pos\" :: string))(p?!1))(x),
   oldpos =
   get?int(FIELD?POSITION((\"c\" :: string))((\"pos\" :: string))(p?!1))(Z?!1),
   r =
   get?ref(FIELD?POSITION((\"c\" :: string))((\"r\" :: string))(p?!1))(x)
IN
   pos = oldpos + 1
   AND
   reference??(r)
   AND
   oldpos >= 0
   AND
   oldpos < proj_2(get?dimlen(refpos?(r))(x))
   AND
   get?int(heap?(refpos?(r), oldpos))(x) = 1
   AND
   (Forall(i:nat) : i >= (pos) AND i < proj_2(get?dimlen(refpos?(r))(x))
         IMPLIES
    get?int(heap?(refpos?(r), i))(x) = get?int(heap?(refpos?(r), i))(Z?!1))
   AND
   ...
```

PVS

where we have not shown the information about the objects `x1`, `x2` and `x3` (at the dots). The resulting branch in our proof tree is solved by using the method-specification for `c2 g(c1 x)` like in the previous section this involves using the `(int-spec)` and `(use-spec)` strategies.

After two more `(composition-replace)` commands which handle `g(x2)` and `g(x3)` we have to separate `g(g(x1))` from the remainder of the method body. Once again we use `(composition-replace)` with as arguments that after evaluation `pos` will have value `\old(pos) + 5` , the assigned values for `r[\old(pos)..\old(pos) + 4]` and the usual information about object `x1`, `x2` and `x3` and the (non) modifiablity of the remainder of the array.

The resulting branch can then be proved by splitting the two method calls by using the `(expr-ext-replace ..)` (were `expr-ext` stands for ExpressionExtension) strategy. Here we have to make sure that the object which is created by the call to `g(x1)` is passed on to the next call from `g()` (recall that Java uses innermost evaluation). Since we have no direct information about the object created by `g(x1)` (we use its specification and not the Hoare rule for `new`) we do not know what its position on the `heaptop` will be. However we do know that the object is a reference, that it will be under the current `heaptop` and that it is an instance of class `c2`. We give this information (as well as all information about `pos`, `r`, etc. we know) as an argument with the `(expr-ext-replace ..)` rule. Now it is also clear why we specified that `c2 g(c1 x)`'s `\result` should satisfy its invariant and can not be `null`. We need this information, because otherwise we do not have enough information to complete this branch. We conclude the proof of this branch by using the specifications from `c1 g(c1 x)` and `c3 g(c2 x)`.

The remainder of the proof continues in a similar fashion were we have to use the `(expr-ext-replace .. )` two times in the branches with three calls to `g()`. For an impression of the overview of the proof the reader can look in figure 4.3 where the resulting prooftree is displayed. Notice that the prooftree resembles the (syntactic) structure of `m`'s body. Rerunning this proof took 7 hours (significantly shorter then the proof seen at page 27). But of course the construction of the proof took much more time (about two weeks).

## 4.3 JCF4

In this section we give a Hoare proof for a program from which we already showed the correctness of the specification (using automatic rewriting) in the previous chapter. The program and its specification are exactly like the one we saw at page 15. The proof itself however is completely different.

Normally we would now proceed with the PVS strategy `(jml-start)`, there are however some difficulties. At the moment no suitable Hoare rules for dealing with static initialization have been developed. Since all of the programs we have seen in the last chapter have static fields and/or methods this poses a problem. We have decided to deal with this problem by giving a Hoare-style proof for JCF4 (using Hoare-rules whenever possible) and using the proof technique from the previous chapter when Hoare rules can not be used. So keep in mind that the proof which we are about to present is not a typical Hoare-style proof.

### Method `void m()`

We can not use the `(jml-start)` strategy, since it crashes at some point. Instead we use `(jml-start$)`. Now PVS applies the rules from the strategy until a command crashes. So we can use at least part of it. Another reason why we want to do this is that the `(jml-start)` rule sets some global variables which are used in later strategies to speed up rewriting. After handling some semantic-details (which the strategy normally applies for us) we end up with two branches: One branch for the case that class `c` is initialized and another branch for the case that it is not. The branch were class `c` is initialized is trivial since we have the assumption that it is not. Since this leaves us with `false` as an assumption we can conclude anything.

Continuing in the other branch we have to weaken our assumptions. In the normal case we assume that our assumptions (which come out of the JML `requires` clause) hold for the entire execution of a method. This is what we normally want, for instance, when we use an array and state in the JML `requires` clause that it should not be `null`, we want this to be so for the entire execution of the method. However since we now have as a requirement that class `c`, `c1` and `c2` are not initialized, we do not want this to hold after static initialization.

We use the `requires` lemma to do this. The lemma lets the user weaken his assumptions (our ad assumptions). Now we can start with the static initialization of class `c`. First we separate the static constructor (which handles class initialization) from the body of method `m()`. The `(composition)` rule lets us do this, as an argument we give the state after the static initialization. We know that after initialization all static fields of `c` have value `false` (the default value for booleans) and class `c` is now initialized. Furthermore classes `c1` and `c2` are not initialized and their fields have not changed value either. The branch with the static constructor is further handled semantically using the strategy `(semantic-assert)`. This strategy is a combination of loading the JML classes and then applying automatic rewriting (like in the previous chapter).

We continue with `m()`'s body. Again we use the `(composition)` rule to separate the first statement from the others. In PVS this looks as follows:

PVS

```
(COMPOSITION-REPLACE "Lambda (x:OM?) :
    (get?boolean(STATIC?FIELD?POSITION((\"c\"))((\"result1\"))))(x) = true
AND (get?boolean(STATIC?FIELD?POSITION((\"c\"))((\"result2\"))))(x) = false
AND (get?boolean(STATIC?FIELD?POSITION((\"c\"))((\"result3\"))))(x) = false
AND (get?boolean(STATIC?FIELD?POSITION((\"c\"))((\"result4\"))))(x) = false
AND IS?INITIALISED?((\"c\"))(x)
AND IS?INITIALISED?((\"c1\"))(x)
AND IS?INITIALISED?((\"c2\"))(x)
AND (get?boolean(STATIC?FIELD?POSITION((\"c1\"))((\"d1\"))))(x) = true
AND (get?boolean(STATIC?FIELD?POSITION((\"c1\"))((\"b1\"))))(x) = true
AND (get?boolean(STATIC?FIELD?POSITION((\"c2\"))((\"d2\"))))(x) = true
AND (get?boolean(STATIC?FIELD?POSITION((\"c2\"))((\"b2\"))))(x) = false ")
```

PVS

With some puzzling this should be clear:
`get?boolean(STATIC?FIELD?POSITION((\"c\"))((\"result1\")))(x)` means get the boolean at the memory place reserved for the `result1` from class `c` (note that its in the static part of memory). Remember that the assignment triggered the static initialization from both class `c1` and `c2`.

We now would like to continue using Hoare logic to handle the initialization from both class `c1` and `c2`, however at the moment this is not possible since we do not have any Hoare rules we can apply here. So instead we deal with this branch as we did in the previous chapter by loading the implementation of all classes and auto-rewrite till the branch is proved.

Two more `composition`-rules let us finish the remainder of `m`'s body. All branches are handled using `(semantic-assert)`. The proof took about 2 minutes to (re)-run.

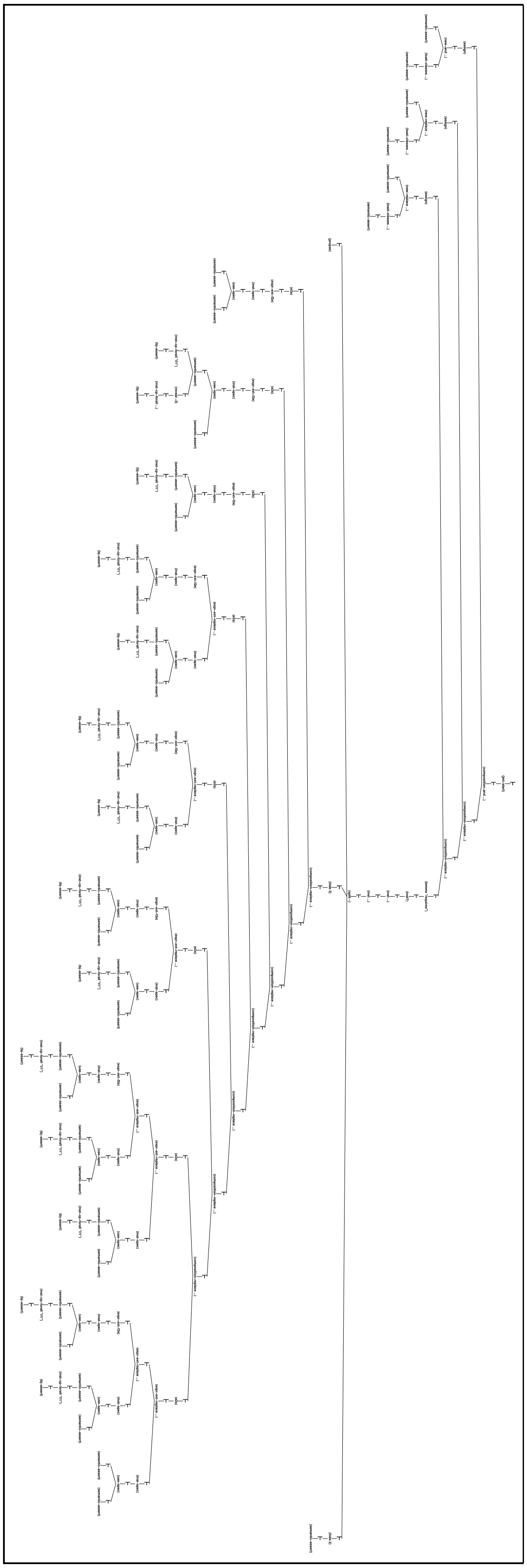

Figure 4.3: Prooftree of method `void m()` from JCF43ns

# Chapter 5

# Discussion & Conclusions

This chapter concludes with a general overview and some points of discussion about the work done in this thesis. We will end with some conclusions and a look at future work.

## 5.1 Problems encountered during Specification & Verification

The specifications and proofs described in this thesis have been constructed in an iterative manner. This means that first a specification was written which we tried to prove correct. If this did not succeed we had to look were the problem originated. This could be either of the following: (i) we had made a mistake in the proof, (ii) we had an incorrect or not strong enough specification, (iii) there was an error in the translation from Java + JML to the higher order logic of PVS, (iv) there was a bug in PVS or in a PVS strategy and (v) there was an error in the semantics for Java and JML. All five kinds of problems have been encountered in some form.

Most problems (luckily) were due to either (i) or (ii). These can be seen as real "faults" from the user. Fixing these was often a matter of looking again at the specification and the proof and determining were a possible error could have been made.

However some problems were not so easy to fix, all the tools and languages used in this thesis have at least one thing in common. They are not "finished" projects:

- The most stable language we used is Java (at the moment of writing there is a beta release of version 1.4 of the Java 2 Platform edition available, version 1.3.1 being the latest stable release). We do not expect that the results presented in this thesis will change in any drastic way for the beta release or indeed any future version for Java (unless SUN decides to change the (core features of the) language in a drastic way).

- JML is still undergoing (minor) changes. This is especially so because JML is more and more used these days. This leads to feedback from users which again leads to minor refinements in the language. We also used a JML construct which is not (yet) officially part of the language: the `\static_fields_of()` keyword.

- The LOOP compiler is still under development, it is currently only used within the LOOP group at Nijmegen University. New versions of PVS have also led to number of modifications in the compiler. For most cases however it works perfectly.

- The semantic-prelude (with the Java and JML semantics) is "finished" in the sense that it has been used on a number of projects (including this thesis) and no (major) modifications have been necessary. There are however still a number of JML constructs which have so far not been included in the semantic model.

- The PVS theorem prover is still heavily under development. In the period that this thesis was done (august - december 2001) a number of new versions from PVS have been released, the most recent being version 2.4. This last version has been used in the proofs in the previous chapter, while the proofs in chapter 3 were mostly done in version 2.3 patch 1.2.2.127. As noted, new versions of PVS did not maintain backward compatibility. Which has led to a number of modifications of both the LOOP compiler and the strategies which are used in the proofs (e.g. `(jml-start)`).

- The Hoare-rules are also not finished yet. As noted on page 44 there are no suitable rules for static initialization.

All of the reasons mentioned above have led to a number of difficulties.

## 5.2 Automatic rewriting vs. Hoare logic

The two main approaches we used in this thesis are different in a number of ways. In chapter 3 we constructed proofs at implementation level using only automatic rewrite rules. This has some clear advantages since no detailed knowledge about the Java program (at least to run the proof, the specification is an entirely different matter) is needed. However there are also some dis-advantages, (i) proofs will in general take a long time to run since everything is evaluated at the implementation level, (ii) not all programs will be provable with this method (see page 27), and (iii) it is not an compositional procedure in the sense that no specifications from other methods can be used (even if these other methods have been proofed to be correct).

The method we applied in chapter 4 also has some clear advantages. Here we can use specifications from other methods in our proofs and (at least in principal) all methods can be proved. Furthermore, because we do not reason at the implementation level but at the (more abstract) specification level proofs tend to be faster (or at least, rerunning the proof takes less processor time). As for the dis-advantages, the user has to have a clear understanding of the Java program as well as of the semantics for Java and JML and of course the Hoare rules as well.

In the long run the second approach seams more promising. Especially since the goal of all this is to get correct specifications for (non trivial) Java programs and a detailed knowledge about the program at hand will be necessary in any case if one wants to write a (correct) specification.

One last question one might ask is: "Can this procedure be used for *real* programs (instead of the somewhat far fetched examples presented here)?" The answer is yes, to some extend. Currently at the LOOP group a case study has been done which looks at a (part of a) JavaCard Applet. This has the advantage that the programs can not be to big (because of the limited resources available on the card). The reader is referred to [BJB01] for more information on this topic.

## 5.3 Conclusions

We have drawn a number of conclusions:

- It is possible to specify (sequential) Java programs in JML and then give a correctness proof of the specification using the LOOP tool and the theorem prover PVS.

- Specifying and proving of (sequential) Java programs is still hard, especially because one has to use a lot of different tools and languages which (almost) all have to some extend incorrect behavior.

- From the two approaches described here the one involving Hoare logic seems to work better, especially for large and/or complex programs.

## 5.4 Future work

There is obviously a lot of room for improvement at almost all levels encountered in this thesis. Some suggestions for future work include:

- New and/or other theorem provers (as well as newer version of PVS) could possibly improve the work done here. Because of the problems with PVS described earlier.
- An extension of the semantics so that one can use (and thus proof programs with) threads seems a logical next step.
- A (further) extension of the Hoare rules so that one can reason about static initialization with these.

# Bibliography


[AF99] Jim Alves-Foss, editor. *Formal Syntax and Semantics of Java.* Springer, 1999.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[Ber01] Jan Bergstra. Application to Java: static field initialization, August 2001.

[BHJP00] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A Type-Theoretic Memory Model for Verification of Sequential Java Programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, number 1827 in Logic of Object-Oriented Programming, pages 1–21. Springer, Berlin, 2000.

[BJ01] J.A.G.M. van den Berg and B.P.F. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Software (TACAS)*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.

[BJB01] Cees-Bart Breunesse, Bart Jacobs, and Joachim van den Berg. Specifying and Verifying an example: a decimal representation in Java for smart cards, December 2001.

[BL99] Jan Bergstra and Marijke Loots. Emperical Semantics for Object Oriented Programs. Artificial Intelligence Preprint Series 007, Onderwijsinstituut CKI, Department of Philosophy, Utrecht University, July 1999.

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition.* Addison-Wesley, 2000.

[JCF] Website with all used programs and their specifications, `http://www.phil.uu.nl/~warnier/thesis.html`.

[JP00] B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2000.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A Notation for Detailed Design. In Haim Kilov, Bernhard Rumpe, and William Harvey, editors, *Behavioral Specification for Businesses and Systems*, chapter 12, pages 175–188. Kluwer Academic Publishers, 1999.

[LOO] The LOOP website, `http://www.cs.kun.nl/~bart/LOOP/index.html`.

[SORSC99] N. Shanker, S. Owre, J.M. Rushby, and D. Stringer-Calvert. *PVS prover guide*, 1999. Version 2.3.

[vON99] David von Oheimb and Tobias Nipkow. Machine-Checking the Java Specification: Proving Type-Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, 1999.

[Win92] Partick Henry Winston. *Artificial Intelligence.* Addison-Wesley, 1992.

[WJP] Java products, `http://java.sun.com/products/`.

# List of Figures