Verifying Java programs by integrating ESC/Java2 and PVS

Sun, Ke

Award date:
2006

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Verifying Java Programs by Integrating ESC/Java2 and PVS

by
K.SUN

Supervisors:

dr. Ruurd Kuiper
dr. Cornelis Huizing

Advisor:

dr. Erik Poll

Eindhoven, November 2006

Preface

There will be no silver bullet within ten years.

— *Frederick Brooks*

Four years ago, when I was still a software engineer in China, I read the famous book of Frederick Brooks - *The Mythical Man-Month*. Brooks impressed me with his insightful observations about software development. Software development is a very complicated activity of human beings. Since it is so complex and affected by social, cultural, and psychological factors, I am pessimistic to find silver bullets. But I do believe we could find some useful weapons to deal with the monster of software crisis. Using formal methods is one of the available weapons right now.

Because of this belief, I chose formal verification for Java programs as my graduation project topic. At the moment when the project is done, I am happy to see that the project gave me an interesting view about programming. In the project, I learned how to reason about programs from the mathematical point of view. Rigorous mathematical reasoning can guarantee program correctness, which is very important for safety and security critical applications. Although the tools used in the project still have many problems, I believe the techniques could be able to produce some useful results for industry.

In the project, I received many people's help. First of all, I thank dr. Ruurd Kuiper and dr. Cornelis Huizing. Without their guidance and encouragement, I could not have finished the project. I am grateful to dr. Erik Poll who proposed the topic and helped me a lot in the project. Many ideas in the project were inspired by Erik's suggestions. Without his help, the result of the project would have been totally different. I appreciate prof. Bart Jacobs, dr. Joseph Kiniry, dr. Kees Hemerik, dr. Francien Dechesne, dr. Arjan Mooij, dr. Carl Pulley, Clément Hurlin and Julien Charles who are always kind to answer my questions. They helped me to overcome many difficulties in the project. I thank dr. Erik Luit who carefully read my thesis and pointed out many mistakes. I am also grateful to dr. Judi Romijn, who is kind to join my graduation committee.

Finally, I thank my parents and sisters whom I miss so much. Their supporting and understanding is a source of power in my studying and living.

Contents

Chapter 1

Introduction

Java, as a widely used Object-Oriented programming language, is increasingly used in safety and security critical applications, e.g. Smart Card [2] and Internet Voting System [19]. In order to guarantee the correctness of Java programs, researchers proposed many approaches including formal specification and verification.

Formal specification and verification techniques can greatly improve software quality by rigorous mathematical reasoning, however they are believed to be hard to learn and expensive to use by industry. Furthermore, formal verification suffers from its limitations, e.g. the state space explosion problem of model checking and the hardness of automatic theorem proving.

Since the last two decades, the situation has improved somewhat because of the progress of theoretical research and the development of new verification tools. Also, Java is a well-structured language, more amenable to formal verification than other, older, programming languages. Therefore, we believe that using the latest formal verification techniques in small scale Java programs would give nice results, especially using theorem proving techniques (which do not suffer for the state space explosion problem).

In the thesis project we performed the investigation of formal verification for Java programs with theorem proving techniques. We investigated possibilities to combine the strengths of two state-of-the-art tools, ESC/Java2 (Extended Static Checker for Java, Version 2) and PVS (Prototype Verification System). In order to integrate the two tools, a software component, VC (Verification Condition) translator for PVS, was developed in the project.

1.1 Motivations

ESC/Java2, a static checker originally developed by Compaq [9], takes an annotated Java program as input, and outputs warnings for broken specifications and suspect bugs, e.g. array bound errors, null dereferences, type cast errors, etc. Internally, the tool translates the Java program and its annotations into an intermediate language which is based on Dijkstra's Guarded Command Language. From the intermediate language, a logic formula is derived for each method. The logic formula which is called VC holds iff the method being checked is "correct", i.e. free of certain bugs and no broken specifications. In ESC/Java2, the gen-

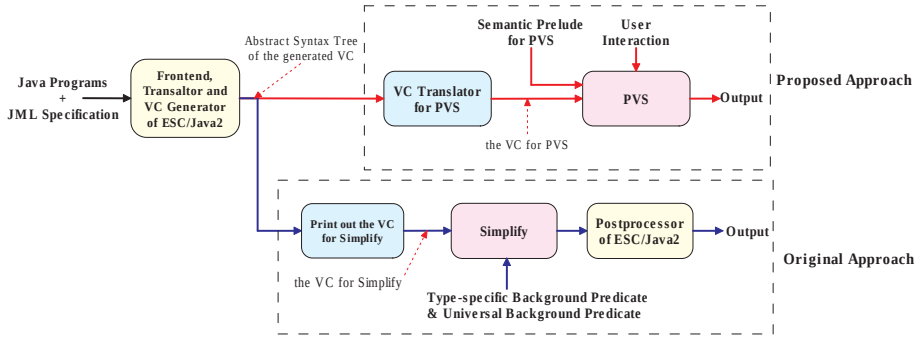

Figure 1.1: The Original and New Proposed Approaches

erated VC is verified by an automatic theorem prover - Simplify. If the VC can not be proved by Simplify, ESC/Java2 will output warnings to indicate the suspect bugs or broken specifications. In the thesis, the approach is called original approach, to distinguish it from our new proposed one.

From the above description, it is clear that the proving power of Simplify affects the result of ESC/Java2. Since Simplify is an automatic theorem prover, sometimes a correct VC can not be proved just because Simplify can not handle it. So, if we can replace Simplify with another powerful, interactive, theorem prover - PVS, we should be able to handle more complicated Java programs although maybe user interaction will be needed in the proving process. Providing an alternative powerful proof engine for ESC/Java2 is the first motivation of our project.

Besides the powerful proof engine, PVS has an expressive specification language and a large library. The replacement of Simplify by PVS gives us an opportunity to utilize the expressivity of PVS. For example, we would like to use the PVS types *set*, *list* in ESC/Java2 since these concepts are hard to express in ESC/Java2 with its annotation language - JML (Java Modeling Language). Providing extra expressivity for ESC/Java2 is the second motivation of the project.

The third motivation of the project is combining the above two points. Utilizing the powerful proof engine and the expressivity of PVS, we should be able to prove Java programs on the abstract level. For example, we can use the PVS *set* to specify a Java program, then reason about the behavior of the Java program with the abstract *set* operations in PVS.

1.2 Proposed Approach

The architecture of ESC/Java2 can be thought of as a pipeline of data processing stages. After the front-end and VC generation procedures, ESC/Java2 internally presents the generated VC by an AST (Abstract Syntax Tree). The syntax of the AST is defined according to the logic of Simplify. The basic idea of our approach is that a newly designed VC translator generates the VC for PVS from the AST, i.e., it translates the VC for Simplify into the VC for PVS.

In addition to the automatically generated VC for PVS, there is also a hand-

written PVS file, the so-called *semantic prelude*. This file defines the basic building blocks for the Java and JML semantics, and defines all the machinery needed, in the form of PVS theories and lemmas, to support the actual work of program verification. From the functional point of view, the semantic prelude is equal to UBP (Universal Background Predicate) in the original approach of ESC/Java2.

The generated VC and the semantic prelude compose the final files. In the thesis, we call the files *proof obligations*. We input every proof obligation into PVS, and prove them via the commands of PVS. If all proof obligations are proved, then we can say the Java program being checked is "correct". Since ESC/Java2 is neither sound nor complete, our approach is also unsound and incomplete. So, the "correct" only means the program has no broken specifications and is free of the bugs that ESC/Java2 checks.

1.3 Project Background

Before we started the project, several people already had done much work on this topic. Clément Hurlin and Carl Pulley developed the initial VC translator for PVS[1] [7]. Julien Charles designed the VC translator for Coq. And Joseph Kiniry wrote the semantic prelude for PVS based on a new sorted logic.

In our initial research proposal, we planned to evaluate the initial VC translator for PVS and investigate the approach to verify Java programs by integrating ESC/Java2 and PVS. However, we soon realized that the initial VC translator for PVS was far from mature. It was not updated to the latest semantic prelude and can only handle one-line methods. Without other options, we decided to update and extend the initial VC translator for PVS first.

The framework of the VC translator was well designed by the pioneers. Actually, the VC translators for different provers are just several classes which implement the abstract superclasses under the framework. It is very convenient to support a new prover. However, in order to extend our VC translator for PVS, it is hard to avoid making some small changes to the framework. After communicating with Joseph Kiniry and Erik Poll, we knew that they plan to reimplement ESC/Java2 entirely in the future. So, our current source code will not be directly included in the future product. Because of this situation, we gave up the compatibility in our design, i.e., we do not guarantee that the VC translators for other provers would work after we tuned the framework for our VC translator for PVS. Because of the experimental nature of our project, we also did not consider efficiency and optimization in our design and implementation.

A problem we met in the project is that the main designers of ESC/Java2 are in other countries and there are no accurate and complete design documents for the tool. This situation forced us to follow a test case driven way in our project, which means: if we want to extend our VC translator with a certain feature, we first design some simple test programs which need the feature, then we develop the VC translator until it passes our experiments. The unsystematic way is convenient for our experiments, but not good for the development of the tool.

[1] They call it VC generator for PVS, but we think VC translator for PVS is a more suitable name since its function is just translating the VC for Simplify into the VC for PVS.

Another difficulty of the project is that we want to translate the original VC based on an unsorted logic into the VC based on a new sorted logic. For historical reasons, ESC/Java2 is built on an unsorted logic which was specially tuned to Simplify. But most modern theorem provers use sorted logic. The successors of ESC/Java2 will be developed on a new sorted logic. So, our VC translator for PVS was required to generate VC based on the new sorted logic although the current front-end and VC generation procedures are still using the old unsorted logic. Because of such inconsistencies, we met many engineering problems in the translation.

1.4 Project Goals

According to the motivations, proposed approach and background information, we defined our goals in the project as follows:

- updating the initial VC translator to the latest semantic prelude and extending it to multi-line methods.
- extending the initial VC translator with some advanced features: quantifier operators, method call, array, and native specifications.
- investigating the advantages of the proposed approach, i.e., answering the question: can we check some programs which can not be handled by the original approach?

1.5 Thesis Overview

In this chapter we have introduced the thesis project motivation, background, proposed approach and goals. An outline of the rest of the thesis is:

Chapter 2: Introduction to the background knowledge. A brief introduction of JML is given since JML is used as the annotation language of ESC/Java2. We also describe the architecture of ESC/Java2 which is important to understand the VC generation procedure. The specification language and some proof commands of PVS are summarized in the final part of the chapter.

Chapter 3[2]: Description of the VC generation procedure. First of all, an intermediate language of ESC/Java2 is introduced. ESC/Java2 translates Java programs into this intermediate language, and derives the VC from the intermediate language; Then, an efficient VC generation algorithm used by ESC/Java2 is discussed. The VC generation algorithm can greatly reduce the size of the generated VC compared to traditional algorithms; Finally, we use a simple example to illustrate the VC generation procedure step by step.

Chapter 4: Introduction to the implementation of the VC translator. In the first part of the chapter, we explain the software architecture of the VC translator from the static point of view. Then in the second part, we describe the

[2]The content of the chapter is mainly summarized from [18], [22], [23], [24], [25] and [26]. The last section is based on our own experiments.

translation procedure from the dynamic point of view. In the remaining sections of the chapter, the way to support a new prover under the current framework is introduced, a brief introduction to some important classes in the implementation is given, and the format of the generated VC is explained. Finally, in order to distinguish our work from other peoples' work (there was an initial VC translator before we started our project[3]), we state our contributions to the VC translator.

Chapter 5: Explanation of how to extend the VC translator with some advanced features. The advanced features include: quantifier operators, method call and array.

Chapter 6: Investigation of supporting native specifications. There are two parts of the chapter. In the first part, we introduce how to support native methods in our proposed approach; in the second part, we investigate the application of native types in JML specifications.

Chapter 7: Investigation of proving Java programs on the abstract level with native specifications. In this chapter, we perform a case study of the so-called celebrity problem. The celebrity program is proved on the abstract level with native specifications.

Chapter 8: Conclusions. We summarize our achievements in the thesis project. Remaining problems and possible solutions are discussed. Finally, future work is pointed out.

[3] [7] describes the initial VC translator.

Chapter 2

Background

2.1 Java Modeling Language: JML

JML is a behavioral interface specification language that is used to specify the behavior and interfaces of Java programs. It combines the design by contract approach of *Eiffel* and the model-based specification approach of *Larch*, with some elements of the refinement calculus.

JML specifications are written in special *annotation comments*, which start with an at-sign (@). The syntax of JML is close to Java. This makes JML easy to learn by Java programmers. JML adds some extensions to Java expressions as shown in Table 2.1. These extensions include a notation to describe the result of a method ($\backslash result$), various kinds of implication (==>, <== and <==>), and a way of referring to the prestate value of an expression ($\backslash old()$).

The semantics of JML is partly described in the JML reference manual [16]. In [5], the denotational semantics of JML is provided in pseudo PVS. However, the complete formal semantics of JML is still lacking.

JML was and is still being developed by Gary T. Leaven and his colleagues at Iowa State University (USA). Researchers all over the world contributed to its development. A range of tools were designed to address the various needs such as reading, writing, and checking JML specifications [27], e.g. the runtime checker, the static checker, the unit testing tool, the documentation generator, etc. In [17], lessons and experience of the JML project are discussed, some initial design considerations are reviewed, and future work is pointed out.

Syntax	Meaning
$\backslash result$	result of method call
a ==> b	"a" implies "b"
a <== b	"a" follows from "b" (i.e. "b" implies "a")
a <==> b	"a" if and only if "b"
$\backslash old(E)$	value of E in prestate

Table 2.1: Some JML's extensions to Java expressions

2.1.1 A First Example

In order to get an impression of JML, consider the following example[1]:

```
1  public abstract class IntHeap {
2
3   //@ public model non_null int[] elements;
4
5   /*@ public normal_behavior
6     @   requires elements.length >= 1;
7     @   assignable \nothing;
8     @   ensures \result == (\max int j;
9     @           0 <= j && j < elements.length;
10    @           elements[j]);
11    @*/
12    public abstract /*@ pure @*/ int largest();
13
14  //@ ensures \result == elements.length;
15  public abstract /*@ pure @*/ int size();
16 };
```

As we already mentioned, all JML specifications are written in special annotation comments, which have the form "/*@ ... @*/" or "//@ ...". Besides the formal specification, an informal description can also be included in JML specifications, e.g. "//@ requires (* x is positive *)" where "x is positive" is an informal description.

Line 3 of the above example defines a model field *elements*. A model field should be thought of as an abstraction of a set of concrete fields used in the implementation. "non_null" means the value of *elements* must not be null.

There are two kinds of specification styles in JML: lightweight and heavyweight specifications. Line 5 to 11 is an example of heavyweight specifications, and line 14 is of the lightweight specification style. In heavyweight specifications, the specification is intended to be complete. It usually includes the keywords: *normal_behavior* or *exceptional_behavior*. The keyword *normal_behavior* tells us that when the precondition of this method is met, then the method must return normally, without throwing an exception. The keyword *exceptional_behavior* specifies that if the precondition is met, the method will return exceptionally with certain properties. In lightweight specifications, users only specify what interests them. For example, nothing else is specified for the method *size* except that line 14 tells us the result of the method should be equal to the length of *elements*.

Line 6 gives the precondition of the method *largest*, and line 8 to 10 give the postcondition. The expression "$\backslash max\ int\ j;\ 0 <= j\ \&\&\ j < elements.length;\ elements[j]$" presents the maximum value of $elements[j]$ for all valid j.

The methods *largest* and *size* are both specified using the JML modifier *pure*. This modifier says that the method has no side effects, and allows the method to be used in specifications if desired. Actually, the use of *pure* gives an implicit frame axiom, i.e. "assignable \nothing".

[1] The example is taken from the JML reference manual [16].

2.1.2 Precondition, Postcondition and Invariant

DBC (Design By Contract) is the key concept in JML. The principal idea behind DBC is that a class and its clients have a "contract" with each other. The client must guarantee certain conditions (precondition) before calling a method defined by the class, and in return the class guarantees certain properties (postcondition) that will hold after the call. In a word, for each method of a class or interface, the contract says what it requires from the the client and what it ensures to the client.

JML uses *requires* clauses to specify preconditions and *ensures* clauses to specify postconditions. Since Java programs may terminate by throwing an exception, JML distinguishes normal and exceptional behaviors. The following example is the JML specification for the method *get* of the class *List*. Line 1 to 5 specify the normal behavior of *get*. Line 2 defines the precondition and line 3 to 5 specify the postconditions for the normal behavior; Line 7 to 9 give the exceptional behavior of *get*. The *signal_only* clause tells us if the parameter *index* is invalid, the only allowed exception *IndexOutOfBoundsException* may be thrown.

```
1  /*@ public normal_behavior
2    @   requires 0 <= index && index < size();
3    @   ensures (\result == null) ||
4    @            \typeof(\result) <: elementType;
5    @   ensures !containsNull ==> \result != null;
6    @ also
7    @ public exceptional_behavior
8    @   requires !(0 <= index && index < size());
9    @   signals_only IndexOutOfBoundsException;
10   @*/
11 /*@ pure @*/ Object get(int index);
```

An object invariant specifies what must hold in all so-called visible states, i.e. the end states of all constructors, and the beginning and end states of all normal methods. The following example defines two invariants for the field *items*. The first one (line 2) says *items* must not be null in all visible states; the second one (line 3) specifies the element type of *items* must be *String*.

```
1  private ArrayList items;
2  //@ invariant items != null;
3  //@ invariant items.elementType == \type(String);
```

2.1.3 Ghost Variable and Model Specifications

Hoare used preconditions and postconditions to describe the semantics of programs in his famous paper [3]. Later he developed the techniques to prove the correctness of data representations in [4]. One of the motivations of Hoare's work is that the reasoning on ADT (Abstract Data Type) level is much more easy than on the concrete level. Another advantage of using ADTs in specifications is that by using ADTs the specification does not have to be changed when the particular data structure used in the implementation is changed.

JML can support ADTs by introducing *ghost variables*, *model fields*, *model methods* and *model classes*. A ghost variable is like a normal field, except that it can only be used in specification. A special *set* command can be used to assign a value to the ghost variable. Model fields are similar to ghost variables,

but should be thought of as the abstract representation of one or more concrete fields. The value of model fields can not be set, and are only determined by their concrete fields.

Besides model fields, the modifier *model* can also be used in the declarations of methods and classes, which means these methods and classes are only used for specification. The following example illustrates a model method which judges the equality of the two objects *o* and *oo*.

```
1  /*@ public normal_behavior
2    @   ensures \result <==>
3    @           (o==oo || (o != null && o.equals(oo)));
4    @ public static model pure boolean nullequals(Object o,
5    @        Object oo);
6    @*/
```

2.1.4 Our Experiences

We had the following experiences with JML.

- JML is a quite expressive specification language. Since its syntax is similar to Java, it is easy to learn by Java programmers.
- The expressivity of JML for exceptional behavior is quite strong. We think using JML to specify and reason about exceptional behavior of Java programs is useful.
- As many programming languages, JML is becoming increasingly large and complicated. We hope JML will not lose its elegance in the evolution.

2.2 Extended Static Checker: ESC/Java2

ESC/Java2 is an extended static checker for Java programs. Here, "static" because the checking is performed without running programs, "extended" because it can catch more errors than conventional static checkers such as type checkers can do. ESC/Java2 is based on the theorem proving technique. It uses an automatic theorem prover - Simplify to reason about the semantics of Java programs. The potential program bugs or the violations of specifications are indicated by warning messages in output.

ESC/Java2 uses JML as its annotation language. The JML specifications can be written together with Java programs, or provided as separate specification files. The second style is very useful when the software is a kind of binary library and its source code is not open. The customers are able to use these separate specification files to check their programs which are built upon the library.

A distinguished feature of ESC/Java2 is *modular checking*: that is, ESC/Java2 checks only one method or constructor at the time. Modular checking is believed to be the essential technique to handle large scale program verification, but it also brings annotation cost, e.g., annotations are needed for the methods which are called by the method being checked.

ESC/Java2 is neither sound nor complete. That means ESC/Java2 may miss some bugs (unsoundness), and may give spurious warnings (incompleteness). In [25], some sources of unsoundness and incompleteness are listed. For example, ESC/Java2 does not check all executions of a loop. By default, it only checks

the first iteration and the condition testing of the second iteration. Suppose the program being checked contains a fragment of the form

```
for (int i = 0; i < 1000; i++) {S}
```

where S never exits abruptly. Then ESC/Java2 will never consider the executions of S from the iteration 2 to 1000 unless it is run with the *-loop* option with an argument not smaller than 1000, which forces ESC/Java2 to check the all iterations. However this would almost certainly result in impractically large verification conditions or impractically slow checking.

2.2.1 The History of ESC/Java2

The ancestor of ESC/Java2, ESC/Java, was developed in Compaq System Research Laboratory in the period from 1996 to 2000. Its main designers include K.Rustan M.Leino, Greg Nelson, and James B.Saxe. Before ESC/Java, they also developed a similar tool, ESC/Modula-3, which performs static checking on Modula-3 programs. After the takeover of Compaq by HP, the development of ESC/Java was abandoned, and the source code was untouched for over two years.

Later, Joseph Kiniry and David Cok took over the project. They updated ESC/Java to ESC/Java2 [10]. ESC/Java2 parses all JML (but only a subset of JML can be used in verification), supports JDK 1.4, and adds processings for model fields and the *represents* clause, etc. Joseph Kiniry maintains a website for the open source project of ESC/Java2 (http://secure.ucd.ie). The latest version of ESC/Java2 is 2.0a9.

Currently, Joseph Kiniry and his collaborators are working on ESC/Java3. They plan to redesign ESC/Java2 entirely, which includes replacing all the source code taken from ESC/Java by their new implementation. They also consider to support multi-provers in ESC/Java3. Our project was proposed as a step towards this goal.

2.2.2 Software Architecture

As we already mentioned, the architecture of ESC/Java2 can be thought of as a pipeline of data processing stages (see Figure 2.1). Its basic components include Front End, Translator, VC Generator, Theorem Prover and Postprocessor.

Front End: The front end of ESC/Java2 is like a Java compiler but it also parses and type checks JML specifications. The front end produces an AST (Abstract Syntax Tree). Some logic formulas called BP (*Type-specific Background Predicates*) are produced as well. These formulas encode type information of the program being checked.

Translator[2]: The component translates the program being checked into an intermediate language based on Dijkstra's GC (Guarded Commands). Actually, the translation is divided into two sub-steps. Firstly, the program being checked is translated into a sugared GC; Secondly, the sugared GC is desugared into a primitive GC.

VC Generator: ESC/Java2 generates a VC (Verification Condition) for each method being checked. A VC is a logic formula which precisely describes

[2]We also call the component developed by ourselves "translator". These two translators are invoked in different phases and have different functions.

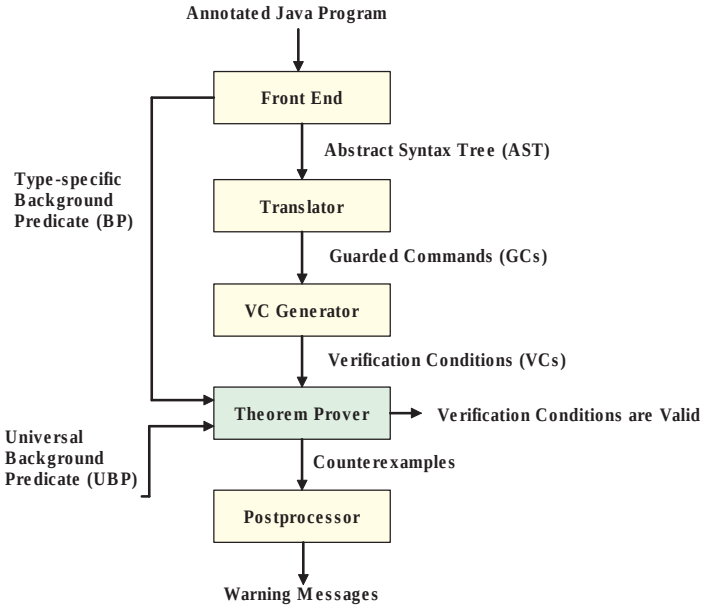

Figure 2.1: The Software Architecture of ESC/Java2

those program states from which no execution can go wrong. The VC generation procedure is based on Dijkstra's Weakest Precondition Calculus, but an efficient algorithm is used instead of the traditional ones (see Chapter 3).

Theorem Prover: The finial proof obligation that combines the generated VC, BP and UBP (*Universal Background Predicate*) is input into an automatic theorem prover - Simplify. The proof obligation looks like:

$$UBP \wedge BP => VC$$

where UBP encodes some general facts about the semantics of Java[3].

Postprocessor: Postprocessor of ESC/Java2 outputs warnings when Simplify can not prove the correctness of a proof obligation. By taking some special labels in the generated VC, ESC/Java2 can produce detailed warning messages including the locations of suspected errors.

2.2.3 Our Experiences

Through the thesis project, we had the following experiences with ESC/Java2.

- Java programmers can learn and use ESC/Java2 easily. It is not difficult to understand most JML specifications. However, some knowledge about program specification and verification is needed if someone wants to master the tool.

[3]In [22], a complete set of UBP for a simple Object-Oriented language is presented.

- ESC/Java2 is good at finding null dereferences, array bounds errors, and type cast errors. However, spurious warnings are produced when the program being checked is complex or the specification is not complete. This increases the cost to run the tool.
- ESC/Java2 is not good at handling recursive specifications. Writing down a recursive specification with JML is easy, but verifying the specification always fails in ESC/Java2. This is because of limitations of Simplify. Actually, this is also one of concrete motivations to provide alternative proof engines for ESC/Java2.
- To reason about the functional requirements of non-trivial Java programs is quite hard. But using ESC/Java2 to check some simple properties, e.g. no unexpected exceptions, seems achievable for small scale applications.

2.3 Prototype Verification System: PVS

PVS was and is being developed at SRI International Computer Science Laboratory at Palo Atlo (USA). It integrates an expressive specification language and a powerful theorem prover to provide an interactive environment for writing and analyzing formal specifications. Since its first release, PVS has been applied successfully to large and difficult applications in both academic and industrial settings.

2.3.1 The Specification Language

The specification language of PVS is quite rich, containing many features. Some specific points are discussed below.

- **Types, Constants and Variables**:
  PVS specifications are strongly typed, meaning that every expression has an associated type. The PVS type system is based on structural equivalence instead of name equivalence. So, types in PVS are closely related to sets, where two types are equal iff they have the same elements.

  There are four kinds of *Type Declarations* in PVS: uninterpreted type declaration, uninterpreted subtype declaration, interpreted type declaration and enumeration type declaration.

  - **uninterpreted type declaration**

    `e.g. Reference : TYPE+`

  - **uninterpreted subtype declaration**

    `e.g. ArrayReference : TYPE+ FROM Reference`

  - **interpreted type declaration**
    predicate subtype:

    ```
    e.g. nonzero : TYPE = {x: nat | x /= zero}
    ```

    function type:

    ```
    e.g. < : [ReferenceType, ReferenceType -> bool]
    ```

tuple type:

```
e.g.  stack : TYPE = [nat, ARRAY[nat -> t]]
```

record type:

```
e.g.  stack : TYPE = [#size: nat, elems: ARRAY[nat -> t]#]
```

- **enumeration type declaration**

```
e.g.  color : TYPE+ = {red, white, blue}
```

A *Variable Declaration* introduces a new variable and associates a type with it.

```
e.g.  elems: VAR ArrayStore.
```

Variable declarations also appear in binding expressions such as FORALL and LAMBDA. Such local declarations "shadow" any earlier declarations.

Constant Declarations introduce new constants. There are both uninterpreted and interpreted constants in PVS. Uninterpreted constants make no assumptions except that they require that the type be nonempty. Some examples of constant declarations are:

```
e.g.  NULL : Reference
      ZERO : int = 0
      refEQ(x, y: Reference) : bool = (x = y)
      inc: [int -> int] = (lambda (x: int): x + 1)
```

- **Parameterized theory**: Polymorphism is not supported in PVS, but it can be approximated by *Parameterized Theory*. To define a polymorphic function, one can put it in a theory which is parameterized with the type variables of the function. However, this approach is not always convenient, because when a function does not use all type parameters of the theory, the unused types should still be instantiated.

```
e.g.
map_theory[Map: TYPE+, Index: TYPE+, Value: TYPE+]: THEORY
  BEGIN
    get: [Map, Index -> Value]
    set: [Map, Index, Value -> Map]
    ...
  END map_theory

...
  IMPORTING
  map_theory[BooleanField, Reference, Boolean],
  map_theory[NumberField, Reference, Number],
...
```

- **Recursive Function**: *Recursive Functions* can be given in PVS. Because all functions in PVS have to be total in their domain, the termination of a recursive function should be shown by giving a *MEASURE* function.

```
e.g.
fibo(i: IntegralNumber): RECURSIVE IntegralNumber =
  IF i = 1 THEN 1
  ELSE (IF i = 2 THEN 1
        ELSE (IF i >= 3 THEN fibo(i-1) + fibo(i-2)
              ELSE 0
              ENDIF)
        ENDIF)
  ENDIF
MEASURE (LAMBDA (i: IntegralNumber): i)
```

The above example is a function to calculate the *Fibonacci* numbers. The MEASURE function shows *fibo* terminates because i decreases with each recursive calling.

- **Overloading**: PVS allows *Overloading*. This means the different declaration can have the same name as long as they have different types. Different functions can have the same name and types if they are in different theories, and the theory name is used as a prefix to distinguish these functions.

```
e.g.
is: [Boolean, JavaType -> bool]
is: [Number, JavaType -> bool]
is: [Reference, JavaTYpe -> bool]

arrayOf(n: IntegralNumber, t: PrimitiveType): ArrayReference
arrayOf(n: IntegralNumber, t: ReferenceType): ArrayReference
```

2.3.2 The Theorem Prover

The PVS theorem prover employs a sequent calculus. The proof is presented as a *proof tree* in the proving procedure. Each node of the proof tree is a *proof goal* which consists of a sequence of formulas called *antecedents* and a sequence of formulas called *consequents*. In PVS, such a proof goal is displayed as

```
antecedents  {-1}  A1
             {-2}  A2
             [-3]  A3
                ...
             |--------
consequents  {1}   B1
             [2]   B2
             {3}   B3
                ...
```

The interpretation of the above proof goal is that the conjunction of the antecedents implies the disjunction of the consequents, i.e., $(A_1 \wedge A_2 \wedge A_3...) \rightarrow (B_1 \vee B_2 \vee B_3...)$. The formula numbers in square brackets (e.g. [-3]) indicate those formulas that are unchanged from its parent proof goal. The numbers in braces (e.g. {2}) highlight those formulas that are either new or different from the parent proof goal.

The PVS theorem prover integrates a typechecker. The typechecker generates some TCCs (Type Checking Conditions) when it checks the specification

being proved. The specification are well-typed only after all TCCs have been proven. Usually, PVS can discharge most TCCs automatically.

Users interact with PVS through proof commands (also called *tactics*). Some proof commands which were used in our project:[4]:

- **Help and Control Commands**
  - (help *command_name*): display the command help.
  - (quit): quit from current proof.
  - (postpone): go to the next proof branch.
  - (undo): undo the last step in the proof.
  - (hide *num*): hide the selected formulas from current proof goal.
  - (lemma "*lemma_name*"): introduce axiom or lemma instance.
- **Basic Proof Commands**
  - (flatten *num*): transform the indicated formula into formulas that contains no disjuncts.
  - (split *num*): split the conjunctive formula into separate formulas.
  - (case *exp*): introduce case splits (e.g. case "x > 0").
  - (skosimp*): repeatedly skolemizes and flattens.
  - (inst *num* "expr"): instantiate existentially quantified variables.
  - (induct *var*): invoke induction.
- **High Automatic Commands**
  - (prop): decision procedure for propositional logic.
  - (assert): decision procedure for equational logic.
  - (grind): a catch-all strategy that is frequently used to automatically complete a proof branch or to apply all the obvious simplifications till they no longer apply.

2.3.3 Our Experiences

We had the following experiences with PVS.

- The specification language of PVS is quite expressive. Its syntax has a mathematical flavor. So, it is not easy to be embraced by ordinary programmers although it is elegant and concise.
- PVS has a powerful interactive theorem prover. The prover can handle recursive specifications better than Simplify.
- The theorem prover of PVS provides a large number of proof commands. To understand the meaning of these commands and know when we should use which commands is not easy. It requires knowledge about logic and quite some proving experience.
- PVS is quite stable on Linux platform and its GUI is user friendly.

[4]The categories are made according to our understanding. They are intuitive rather than sharp and accurate.

Chapter 3

The VC Generation Procedure

Roughly speaking, extended static checking in ESC/Java2 can be divided into two stages. The first stage, *VC Generation*, translates a program fragment and its correctness property into a logical formula, i.e. a VC. A VC has the property that if it is valid then the program fragment satisfies its correctness property. The second stage, *VC Proving*, proves the VC by a theorem prover, e.g. Simplify or PVS. The two stages are tightly coupled in ESC/Java[1] since the logic of ESC/Java is tuned for Simplify.

There are two critical problems in the first stage. First, generating a VC for a practical programming language like Java is not easy. There are many engineering challenges in the VC generation. Second, the traditional VC generation algorithm based on Dijkstra's weakest precondition calculus yields a VC the size of which is exponential in the size of the code fragment being checked in the worst case. Such a large size VC is expensive to create, store and verify. So, how to generate an efficient VC is the other critical problem.

In ESC/Java, the designers provided solutions for both problems. To the first problem, ESC/Java introduces an intermediate language. The language is based on Dijkstra's GC (Guarded Commands). Actually, the Java program being checked is translated into a sugared form GC program first (which we call sugared GC), then transferred to a primitive form GC program (which we call primitive GC). Finally, the VC is derived from the primitive GC program.

For the second problem, the designers of ESC/Java invented an intelligent algorithm for the VC generation. The key idea of the algorithm is that the assignment statements in the program being checked are eliminated. The elimination is done by transforming the assignment statements into *assume* statements. After that, a *dream property* which holds for the transformed program can be used to reduce the size of the VC.

3.1 The GC Language

To derive VCs from Java programs is very complicated. The designers have to make trade-offs involving the frequency of spurious warnings, the possibility of missed errors, the efficiency of the tool, and the annotation overhead, etc. In

[1]Since ESC/Java2 did not change the VC generation procedure of ESC/Java, we talk ESC/Java in the chapter

order to manage complexity and achieve flexibility, the designers of ESC/Java introduced an intermediate language GC (By including the *assume* statement, the language does not need the "guards" which gave Dijkstra's GC its name).

The three-stage procedure (Java → sugared GC → primitive GC → VC) has many benefits. In the translation from Java to sugared GC, many complexities of the Java language are eliminated, such as *switch* statements and expressions with side effects. This part of the process is bulky and tedious, but also relatively stable. Separating this part from other parts which are subject to change in experiments gives great benefits to the tool development. The desugaring from sugared GC to primitive GC is a process where the designers of ESC/Java made many kinds of trade-off explorations. Since sugared GC is a relatively simple language, the explorations are easy to perform.

The basic syntax of primitive GC is as follows[2]:

```
statement ::= assert e
            | assume e
            | x := e
            | A ; B
            | A [] B
            | skip
            | raise
            | A ! B
            | label L: e
where A and B are statements, L is a label,
x is a variable, and e is a predicate expression.
```

The execution of "*assert e*" terminates normally if the predicate e evaluates to $true$ in the current program state, and goes wrong otherwise. The *assume* statement is partial: "*assume e*" terminates normally if e evaluates to $true$, and simply can not be executed from a state where e evaluates to $false$.

The statement "A ; B" denotes the sequential composition of A and B. The execution of the choice statement "A [] B" executes either A or B, but the choice between the two is made arbitrarily. Using the choice statement and the *assume* statement, the if statement of Java language can be expressed as follows:

```
Java statement:     if (x >= 0) A else B;
Primitive GC:       {(assume (x >= 0); A)
                     []
                     (assume boolNot(x >= 0); B)};
```

The *raise* statement raises an exception. ESC/Java uses the exception not only to model the behavior of Java exceptions, but also to model the *break* and *return* statement of Java. In the catch statement "A ! B", the statement B is an exception handler for any exception raised in A. If A terminates normally, then B is not executed.

A labeled expression "label L: e" is semantically equivalent to the expression e, but supplies the label L to Simplify in order to facilitate the production of user-sensible warning messages. There are positive and negative labels. The positive labels with formulas that should be true or the negative labels with formulas that should be false provide counterexample information to ESC/Java.

The desugaring from sugared GC to primitive GC is quite flexible. As an example, let us assume a Java statement "v = o.f" exists in line 27 of a Java program. The sugared GC fragment would be like

[2]Sugared GC includes some additional statements which are not discussed here.

```
check NULL,27,o != null;
v = select(f, o);
```

Then, we have several choices for the desugaring. If we want to check null dereferences, the above statement can be transferred into

```
assert(label NULL@27: o != null);
v = select(f, o);
```

If we do not check null dereferences, the primitive GC after desugaring would be like

```
assume(o != null);
v = select(f, o);
```

If we want to handle null dereferences with an exception, the primitive GC would be

```
{(assume(o != null)
[]
(assume o == null; raise)};
v = select(f, o);
```

3.2 The Efficient VC Generation Algorithm

Before we introduce the efficient VC generation algorithm which is used by ESC/Java, two weakest precondition calculi need to be addressed. The *Weakest Conservative Precondition* of a statement S with respect to a predicate Q on the post-state of S, denoted wp(S,Q), is a predicate characterizing all prestates from which every non-blocking execution of S does not go wrong and terminates in a state satisfying Q. Similarly, the *Weakest Liberal Precondition* of S with respect to Q, denoted wlp(S,Q), characterizes the prestates from which every non-blocking execution of S *either* goes wrong *or* terminates in a state satisfying Q.

Obviously, wp(S,Q) is our ultimate goal. According to the definitions of *wp* and *wlp*, the following equation holds.

$$\forall Q, wp(S,Q) \equiv wp(S,true) \wedge wlp(S,Q)$$

So, in order to get wp(S,Q), we first need to compute wlp(S,Q). However, there is a problem of redundancy. In the computation of wlp(S [] T,Q), which expands to "wlp(S,Q) $\wedge$ wlp(T,Q)", we duplicate Q. This results in a VC which size is exponential in the size of the program being checked in the worst case. We can calculate wlp(S [] T,Q) as "wlp(S,q) $\wedge$ wlp(T,q)" where q = Q. This replacement reduces the size of the VC. However, given a formula like "A $\wedge$ B", the theorem prover of ESC/Java, Simplify, first attempts to prove A and then attempts to prove B. By introducing a name, like q for the common subexpression Q, we do not change the fundamental way in which the theorem prover will attempt to prove the given formula: the theorem prover would still have to consider q as many times as it had to consider Q. The simple replacement does not solve the problem.

Another way to avoid the redundancy is to change the formula into something for which only one independent Q exists. Consider the following *dream*

S	wp(S,Q)	wlp(S,Q)
x := E	Q[x := E]	Q[x := E]
assert E	E ∧ Q	E → Q
assume E	E → Q	E → Q
S ; T	wp(S,wp(T,Q))	wlp(S,wlp(T,Q))
S [] T	wp(S,Q) ∧ wp(T,Q)	wlp(S,Q) ∧ wlp(T,Q)

Table 3.1: Weakest Precondition Semantics

property:

$\forall Q, wlp(S, Q) \equiv wlp(S, false) \vee Q$

If the dream property holds, we can compute wlp(S [] T,Q) as

$(wlp(S, false) \wedge wlp(T, false)) \vee Q$

Then the redundancy problem would be solved since only one independent Q occurs. In [26], the author proves the dream property holds when all statements of the program being checked are *passive commands*, i.e. those statements that terminate without any side effect on the program state.

Since only assignment statements change program states, the key point of the algorithm is to remove all assignment statements. The elimination is done by replacing each assignment statement "x := e" with an assumption "assume $x' =$ e" where x' is a fresh variable. The subsequent references to x will refer to x' in the source statements.

After transferring all source statements to passive commands, we can compute wp(S,Q) efficiently. In [8], it is reported that the approach allows to check large and complex methods which can not be handled by the traditional algorithm due to time and space constraints.

3.3 The Implementation of the VC generation

3.3.1 Implementation Overview

The entry class of ESC/Java2 is *Main* (package: escjava)[3]. It inherits from the class *SrcTool* (package: javafe). The main execution steps of ESC/Java2 are defined in the method *frontEndToolProcessing*() of the class *SrcTool* which invokes the methods *preload*(), *loadAllFiles*(), *postload*(), *preprocess*(), *handleAllCUs*() and *postprocess*(). Some notes for the procedure are in place.

- All execution options are defined in the class *Options* (package: escjava).
- In the method *preload*(), the version of the Java Virtual Machine is checked (ESC/Java2 does not support JDK 1.5).
- The "CU" in *handleCU*() means Compilation Unit.

[3]The following discussion is based on the source code of ESC/Java2 (version 2.09a)

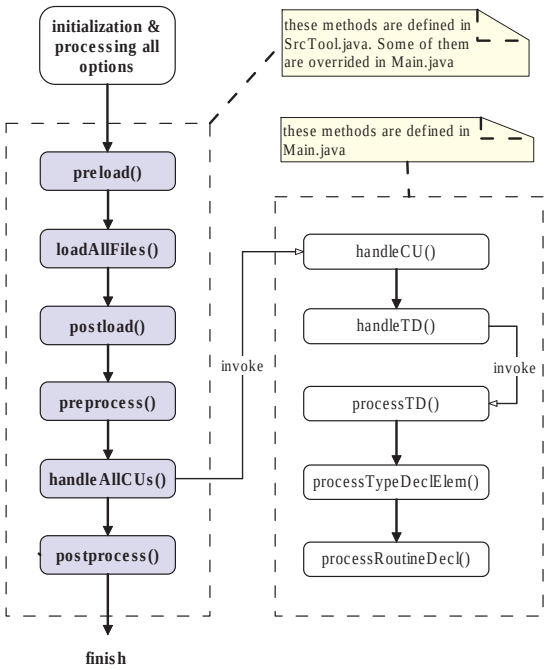

Figure 3.1: The Rough Execution Steps of of ESC/Java2

- The "TD" in *handleTD*() means Type Declaration.

- The theorem prover, Simplify, as a standard-alone component of ESC/Java2, is invoked as a subprocess of ESC/Java2. It communicates with other parts of ESC/Java2 by strings and output files.

- The actual VC generation is mainly performed in the methods *processTD*(), *processRoutineDecl*() and other methods being called by the two methods. Our VC translator is invoked in *processRoutineDecl*().

From the functional point of view, the VC generation of ESC/Java2 can be divided into 5 steps : Type Checking, Generating BP (Type-specific Background Predicate), Translating Methods into GC (Guarded Commands), Converting GC into DSA (Dynamic Single Assignment Form), and finally deriving VC from DSA. The subsequent sections will explain each step based on the following example:

```
1 final class test {
2     //@ ensures \result == a + b;
3     public int add(int a, int b) {
4         return a + b;
5     }
6 }
```

3.3.2 Step 1: Type Checking

The first step is performed in the method *processTD*() of the class *Main*. It does Java type checking. If the type checking fails, ESC/Java2 will exit with errors. Using the option "-pjt", we can output the source codes with type checking information. Some additional specifications are added to the source code automatically. As to our example, the output is:

```
1 final class test
2 {
3    public final int add(int a, int b)
4    /*@ requires (/*boolean*/ (\lblneg Pre (/*boolean*/
                 (/*test*/ this) instanceof test))); *@/
5    /*@ ensures (/*boolean*/ (/*boolean*/
         \old((/*boolean*/ (/*test*/ this) instanceof test)))
          ==> (/*boolean*/ (/*int*/ \result) ==
           (/*int*/ (/*int*/ a:3.23) + (/*int*/ b:3.30)))); *@/
6    /*@ signals_only (java.lang.Exception) (/*boolean*/
         (/*boolean*/ \old((/*boolean*/ (/*test*/ this)
         instanceof test))) ==> (/*boolean*/ false)); *@/
7    /*@ diverges (/*boolean*/ (/*boolean*/ \old((/*boolean*/
         (/*test*/ this) instanceof test)))
          ==> (/*boolean*/ false)); *@/
8    /*@ modifies (/*boolean*/ \old((/*boolean*/
         (/*test*/ this) instanceof test)))
          ==> ((/*UNAVAILABLE*/ \everything)); @*/
9    {
10      return (/*int*/ (/*int*/ a:3.23) + (/*int*/ b:3.30));
11   }
12
13   // <default constructor>
14 }
```

It is interesting to look at line 4. Although we do not provide preconditions for the method *add*(), a default precondition is added. The keyword *\lblneg* indicates *Pre* is a negative label which means a warning will be generated if *Pre* is false. The remainder of the line tells us that *this* is an instance of the class *test*.

The *signals_only* clause specifies the exception *java.lang.Exception* may be thrown if its subsequent predicate is true. Because the precondition protects the predicate to be false in the example, the method *add* will never throw the exception.

The *diverges* clause provides conditions under which the method may diverge and never return to the caller. For the method *add*, it will never diverge since the condition is equivalent to false.

The *modifies* clause specifies what fields can be changed in the method *add*. If we do not give the *modifies* or *assignable* clause in our specification, ESC/Java2 will assume any field can be changed in the methods being checked. If we add a specification "\\@ assignable \nothing" for our example, then line 8 will become:

```
/*@ modifies (/*boolean*/ \old((/*boolean*/
(/*test*/ this) instanceof test))) ==> ((/*void*/ \nothing)); */
```

3.3.3 Step 2: Generating Background Predicate

In Step 2, BP (Background Predicate) will be produced. BP is a set of formulas that encode type information of the class being checked. For example, the generated BP will include a formula like $(\forall t : (t <: test) \rightarrow t == test)$[4] because in our example *test* is a final class. Actually, if we run ESC/Java2 with the option "-guardedVC .", a BP file (named *0.1.12.class.sx*) will be generated.

```
(BG_PUSH (AND
 ...
(FORALL (t) (PATS (<: t T_test)) (IFF (<: t T_test) (EQ t T_test)))
(FORALL (t) (PATS (<: T_test t)) (IFF (<: T_test t)
               (OR (EQ t T_test) (<: |T_java.lang.Object| t)  )))
 ...))
```

The third line says if t is a subtype of *T_test*, then t must be equal to *T_test*. It just encodes the above formula. Furthermore, because the class *test* is a subclass of *Object* (all classes inherit from *Object* in Java), the fourth line describes that if *T_test* is a subtype of t, then t must equal to *T_test* , or t equal to *T_java.lang.Object*, or t is the supertype of *T_java.lang.Object* (this is impossible since *Object* is the top class in Java).

Besides the above two lines, other type information is also encoded into the BP file. For example, the following formula specifies the class *Throwable* is the subclass of *Object*. The general type information is included in every generated BP file.

```
(<: |T_java.lang.Throwable| |T_java.lang.Object|)
```

3.3.4 Step 3: Translating Methods into Guarded Commands

Step 3 translates the Java code fragments, i.e. methods, to the intermediate language GC. The translation is mainly performed in the method *computeBody*() of the class *Main*. The class *Translate* (package: escjava.translate) implements actual translation handlers. The translated GC for the method *add*() is:

```
1  ASSUME is(this, \type(test));
2  ASSUME isAllocated(this, alloc);
3  ASSUME refNE(this, null);
4  ASSUME is(a:3.23, \type(int));
5  ASSUME is(b:3.30, \type(int));
6  ASSUME (\lblneg Pre boolAnd(is(this, \type(test)),
           refNE(this, null)));
7  ASSUME (\forall anytype brokenObj;
           refEQ(java.lang.Throwable#_stackTrace(state, brokenObj),
           getStackTrace##state(state, brokenObj)));
8  ASSUME (\forall anytype brokenObj<1>;
           refEQ(java.lang.Throwable#_stackTrace(state, brokenObj<1>),
           getStackTrace##state(state, brokenObj<1>)));
9  VAR int a@pre:3.23; int b@pre:3.30 IN
10    a@pre:3.23 = a:3.23;
11    b@pre:3.30 = b:3.30;
```

[4] In the logic of ESC/Java2, the predicate "p <: q" means p is a subtype of q or p and q are the same type.

```
12    {  RES = integralAdd(a:3.23, b:3.30);
13       ASSUME (\lblpos trace.Return^0,4.8 true);
14       EC = ecReturn;
15       RAISE
16    !  SKIP
17    };
18    RESTORE a:3.23 FROM a@pre:3.23;
19    RESTORE b:3.30 FROM b@pre:3.30
20 END;
21 ASSUME (\forall anytype brokenObj;
           refEQ(java.lang.Throwable#_stackTrace(state, brokenObj),
           getStackTrace##state(state, brokenObj)));
22 ASSUME (\forall anytype brokenObj<1>;
           refEQ(java.lang.Throwable#_stackTrace(state, brokenObj<1>),
           getStackTrace##state(state, brokenObj<1>)));
23 ASSERT (\lblneg Exception@5.4 anyEQ(EC, ecReturn));
24 ASSERT (\lblneg Post:2.8@5.4 boolImplies(
           boolAnd(anyEQ(EC, ecReturn),
           is(this, \type(test)), refNE(this, null)),
           integralEQ(RES, integralAdd(a:3.23, b:3.30))));
25 ASSERT (\lblneg Post:2.4@5.4 boolImplies(
           boolAnd(anyEQ(EC, ecThrow),
           typeLE(\typeof(XRES), \type(java.lang.Exception))),
           boolNot(boolAnd(is(this, \type(test)),
           refNE(this, null)))))
```

The *assume* clauses from line 1 to 8 are the preconditions of *add*(). The *assert* clauses from line 23 to 25 define the postconditions. Line 7, 8, 21 and 22 concern the *aliasing* checking.

The predefined variable *alloc* represents the current allocation time. The predicate in line 2 just says *this* was allocated before current allocation time.

The predicate *refNE* defines the unequal relation between two reference objects. The predicate *refEQ* returns true if the two reference objects are equal.

The special variable *EC* in line 14 is used to model method return. By convention, the GC programs generated by the translation always set *EC* (and possibly *RES* or *XRES*) before performing a raise. More specifically, before a raise that corresponds to a Java return, the guarded command sets *EC* to the special literal *ecReturn* and sets *RES* to the return value, if there is one. Before a raise that corresponds to a Java throw, the guarded command sets *EC* to the special literal *ecThrow* and sets *XRES* to the exception thrown.

3.3.5 Step 4: Converting Guarded Commands into Dynamic Single Assignment Form

As introduced in the section 3.2, ESC/Java2 will convert the ordinary statements to a kind of passive form statements in the efficient VC generation algorithm. The basic idea of the conversion is to replace each assignment statement "$x := e$" by an assumption "$assume\ x' = e$", where x' is a fresh variable. Since the example *test* has no assignment statements, let us consider another example *test1* as follows:

```
1  class test1 {
```

```
2      //@ ensures \result >= 0;
3      public int abs(int x) {
4          if (x < 0) x = -x;
5          return x;
6      }
7  }
```

Before the conversion, the GC program for *test1* will be:

```
   ...
1  VAR int x@pre:3.23 IN
2     x@pre:3.23 = x:3.23;
3     { {  ASSUME integralLT(x:3.23, 0);
4          ASSUME (\lblpos trace.Then^0,4.19 true);
5          x:3.23 = integralNeg(x:3.23)
6       []
7          ASSUME boolNot(integralLT(x:3.23, 0));
8          ASSUME (\lblpos trace.Else^1,4.8 true)
9       };
10      RES = x:3.23;
11      ASSUME (\lblpos trace.Return^2,5.8 true);
12      EC = ecReturn;
13      RAISE
14   !  SKIP
15   };
16   RESTORE x:3.23 FROM x@pre:3.23
17 END;
   ...
24 ASSERT (\lblneg Post:2.8@6.4 boolImplies(
     boolAnd(anyEQ(EC, ecReturn), is(this, \type(test1)),
     refNE(this, null)), integralGE(RES, 0)));
   ...
```

In the GC program, the statement "$x = -x$" is represented as "$x : 3.23 = integralNeg(x : 3.23)$". After the conversion, part of the DSA for *test1* is:

```
   ...
1  {  { ASSUME integralLT(x:3.23, 0);
2       ASSUME (\lblpos trace.Then^0,4.19 true);
3       ASSUME anyEQ(x:4.19, integralNeg(x:3.23));
4       ASSUME anyEQ(x:3.23<1>, x:4.19)
5     []
6       ASSUME boolNot(integralLT(x:3.23, 0));
7       ASSUME (\lblpos trace.Else^1,4.8 true);
8       ASSUME anyEQ(x:3.23<1>, x:3.23)
9     };
10    ASSUME (\lblpos trace.Return^2,5.8 true);
11    RAISE
12 !  SKIP
13 };
   ...
18 ASSERT (\lblneg Post:2.8@6.4 boolImplies(
     boolAnd(anyEQ(ecReturn, ecReturn), is(this, \type(test1)),
     refNE(this, null)), integralGE(x:3.23<1>, 0)));
   ...
```

Comparing the GC and DSA, we could notice that the assignment statement of line 5 in the GC is replaced by the *assume* clauses in line 4 and 8 in the DSA. A fresh variable "x:3.23<1>" is introduced in the translation. Furthermore, in the postcondition (line 24 in the GC and line 18 in the DSA), the fresh variable "x:3.23<1>" is used instead of *RES* (= "x:3.23").

3.3.6 Step 5: Deriving VC from Dynamic Single Assignment Form

In Step 5, the VC is derived from the DSA. Using the option "-pvc", the output of the VC for the method *add*() can be obtained[5]:

```
1  (EXPLIES
2    (LBLNEG |vc.test.add.2.4|
3      (IMPLIES
4        (AND
5          (EQ |elems@pre| elems)
6          (EQ elems (asElems elems))
7          (< (eClosedTime elems) alloc)
8          (EQ LS (asLockSet LS))
9          (EQ |alloc@pre| alloc)
10         (EQ |state@pre| state)
11       )
12       (NOT
13         (AND
14           (EQ |@true| (is this T_test))
15           (EQ |@true| (isAllocated this alloc))
16           (NEQ this null)
17           (EQ |@true| (is |a:3.23| T_int))
18           (EQ |@true| (is |b:3.30| T_int))
               ...
31           (EQ RES (+ |a:3.23| |b:3.30|))
               ...
65         )
66       )
67     )
68   )
69   (AND (DISTINCT |ecReturn| |ecThrow|))
70 )
```

Line 5 to 10 is the premise of the VC. It concerns some predefined variables. For example, *elems* is used to model the state of all arrays, and *LS* is used in multi-threaded Java program verification (neither is needed in the example).

Line 69 indicates that the *ecReturn* and *ecThrow* are different. This is true because the *ecReturn* represents the normal execution path and the *ecThrow* represents the exceptional execution path.

The generated VC is stored in an AST (Abstract Syntax Tree). The class *VcToString* (package: escjava.translate) prints out the VC for Simplify from the AST. Actually, our VC translator for PVS is also invoked in this step while taking the AST as an input. The functions of *VcToString* and our VC translator for PVS are the same, except that *VcToString* is for Simplify, and our VC translator is for PVS.

[5]In order to make the VC easier to understand, we have changed its layout in the presentation.

Chapter 4

The VC Translator for PVS

The VC translator for PVS is invoked after the VC is generated in ESC/Java2. It translates the VC for Simplify into the VC for PVS. More specifically, the VC translator parses the Abstract Syntax Tree of the generated VC (we call it the old AST in the following discussion); then it builds a new AST from the old AST; finally, the VC for PVS is produced from the new AST.

The old and new AST are the central data structures of the VC translator. The old AST is based on an unsorted logic which means type information is encoded into separate logic formulas. The new AST is built on a sorted logic which requires that type information is bound together with variables and constants. For example, assume we have a Java statement: "$b = a + 1$" where a and b are both *integral* numbers. If we translate the statement into unsorted logic, the result would be like:

```
(typeof(a) = int) /\ (typeof(b) = int) /\ (b = a + 1)
```

If we consider sorted logic, the statement would be translated as:

```
(declare a and b as int variables): b = a + 1
```

where a and b are declared as *integral* numbers instead of using separate logic formulas to describe the type of a and b.

One of the key tasks of our VC translator is to build a typed new AST from the untyped old AST, i.e. give the right type to every variable and constant in the old AST. In order to accomplish the type assignment, at the beginning the VC translator retrieves all type information from the old AST. Whenever a variable or constant is encountered in building the new AST, the VC translator looks up the retrieved types and gives the variable or constant the right type. In the future, if the successor of ESC/Java2 is built on the new sorted logic from the beginning, the transformation from the old AST to the new AST will not be needed.

With the options "-vc2dot" and "-pToDot" of ESC/Java2, the VC translator can output the graphic representations of the old and new AST separately. The figure 4.1 shows parts of an old AST and a new AST. The old AST part represents the formula:

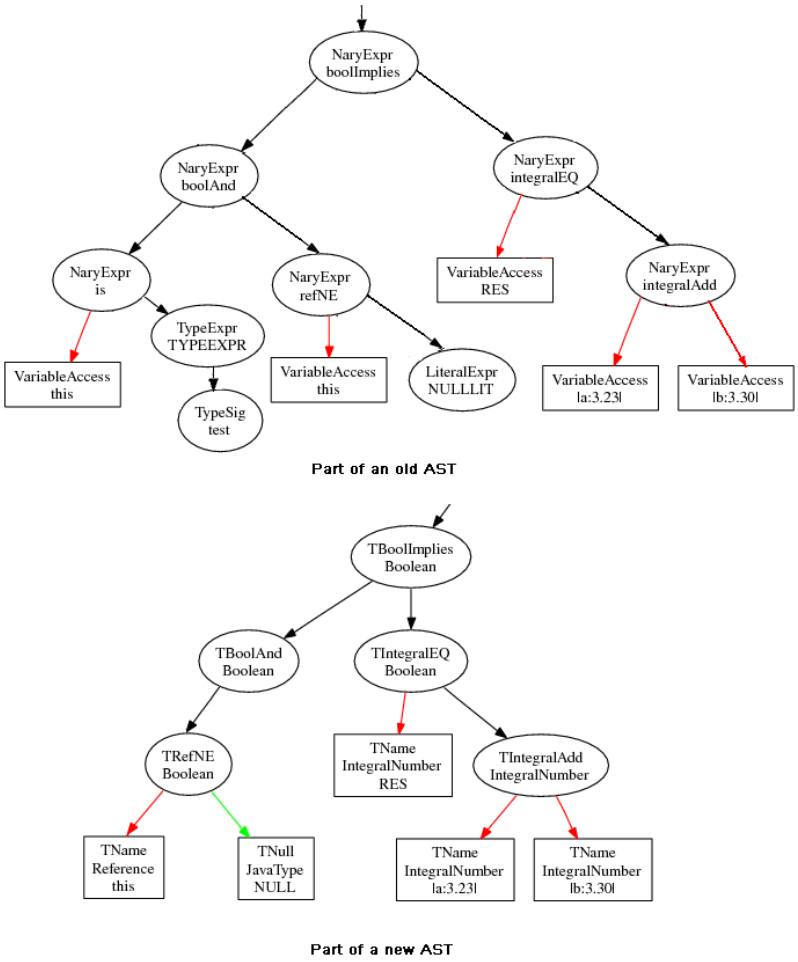

Figure 4.1: Abstract Syntax Tree

$$(is^1(this, test) \wedge this \neq NULL) \rightarrow (RES = |a : 3.23|^2 + |b : 3.30|)$$

And the new AST part represents the formula:

$$(this \neq NULL) \rightarrow (RES = |a : 3.23| + |b : 3.30|)$$

where the subformula "$is(this, test)$" does not occur since the type of $this$ is already included in its declaration.

[1] "is" is a predefined predicate in the unsorted logic of ESC/Java2. It says the type of "this" is "test".

[2] $|a : 3.23|$ is a variable name that "a" comes from the name of the original Java variable, "3" and "23" indicate the line number and column number of the Java variable declaration. This is similar for $|b : 3.30|$.

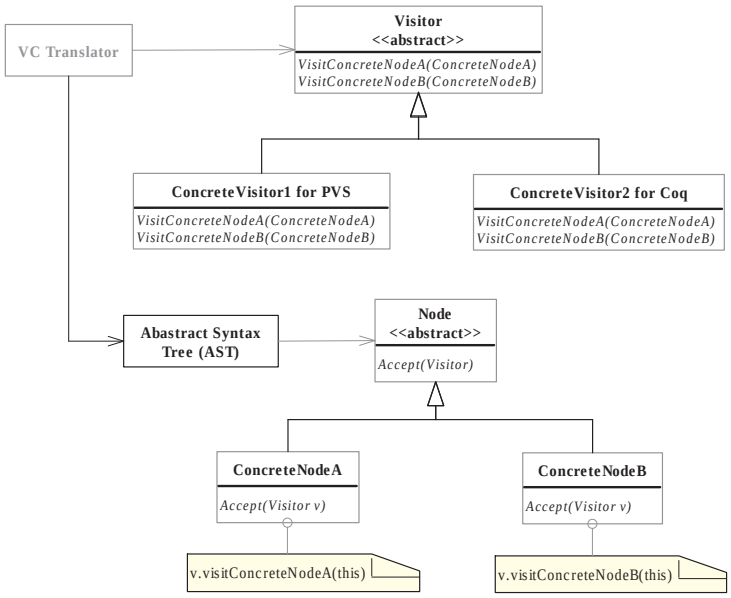

Figure 4.2: The Visitor Pattern

4.1 Software Architecture

Besides our VC translator for PVS, VC translators for other provers, e.g. the VC translator for Coq, have been developed by others. The different VC translators all share the same framework, i.e., they inherit from the same superclasses and share common top-level procedures, e.g. the construction of the new AST. Since supporting new provers is expected in the future, the framework of the VC translators should be flexible to support this extension.

Because of this consideration, the architecture of our VC translator is based on the design pattern *Visitor* (two good references for Design Patterns are [14] and [29]). Figure 4.2 describes the class diagram of the *Visitor* pattern as applied to our VC translator. The basic ideas of the *Visitor* pattern are the following:

- **Motivation**: There are some operations on nodes. New operations could be added in the future. If we define these operations directly in the class *Node*, when a new operation is added we have to recompile all the node classes. The software would be inflexible and hard to maintain. We need a solution where the node classes will not be affected when a new operation is added.

- **Solution**: We abstract operations into a class *Visitor*. Every operation will be defined as a concrete *Visitor*. The nodes invoke these operations via a method *Accept*(*Visitor v*). At the beginning, every concrete *Visitor* is instantiated, and is passed as a parameter to the method *Accept*.

- **Consequence**: The *Visitor* pattern allows us to add new operations easily. We can define a new operation simply by adding a subclass of *Visitor*.

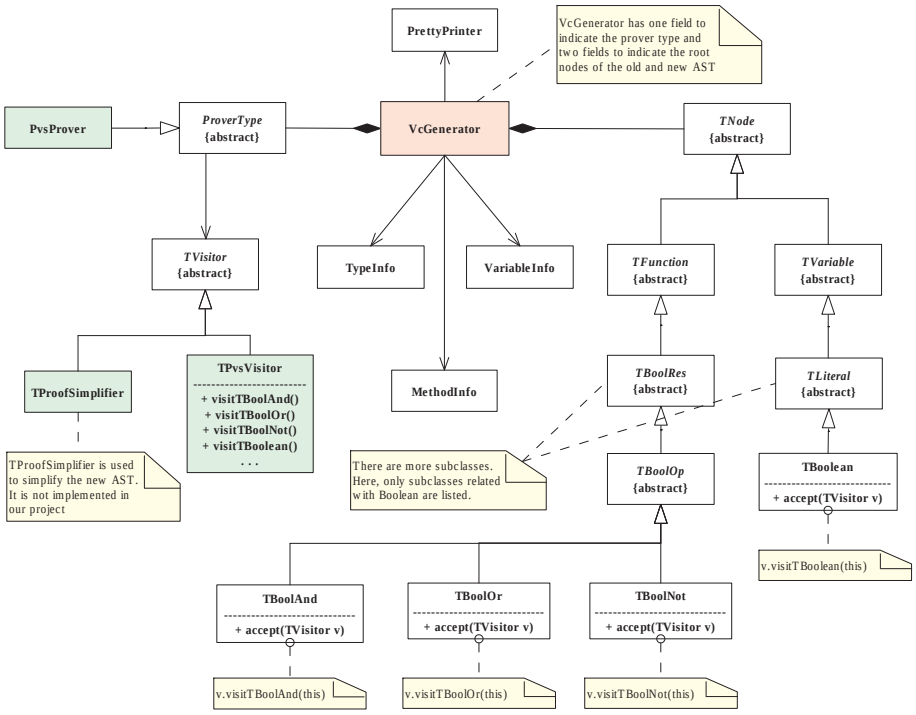

Figure 4.3: Software Architecture of the VC Translator

> In the class, the operation for each node is implemented as a *visit* method. The node classes need not be recompiled when a new operation is added. The *Visitor* pattern also has its drawbacks. It is inconvenient to add a new node since each new node gives rise to a new abstract method in the class *Visitor* and a corresponding implementation in every subclass of *Visitor*.

The figure 4.3 depicts the software architecture of our VC translator for PVS. *VcGenerator* is the entry class of the VC translator. *TNode* and *TVisitor* correspond to the class *Node* and *Visitor* in the *Visitor* pattern. Since the node classes are relatively stable after the release of the VC translator, the drawback of the *Visitor* pattern is not a problem in our application context. Under this architecture, it is easy to support a new prover - just by implementing a concrete *Visitor* and a prover interface for the target prover[3].

Some notes on the architecture:

- The class *PrettyPrinter* is used to format the output of generated VC. The class *TypeInfo* stores the type information, and *VariableInfo* and *MethodInfo* save variable and method information separately. These classes are explained in more detail in Section 4.4.

[3]This is described in detail in section 4.3.

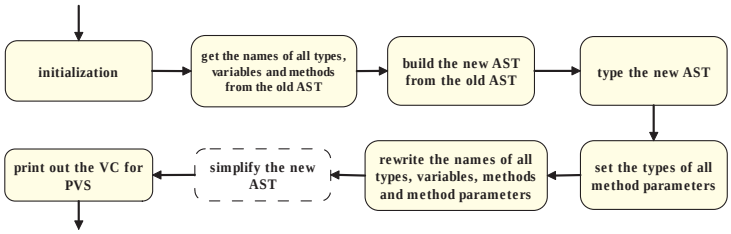

Figure 4.4: The Translation Procedure

- The actual VC translation procedure is managed by *PvsProver*. It invokes *TPvsVisitor* by calling the method *accept*(*TVisitor v*) of the root node of the new AST. Every node of the new AST is accessed by recursively calling the method *accept*.
- The class *TProofSimplifier* simplifies the new AST by eliminating some duplicate nodes and unnecessary formulas, e.g., after the introduction of the sorted logic, some formulas which encode type information would be useless. In practice, the size of the generated VC increases very fast as the program being checked becomes larger. However, we did not implement *TproofSimplifier* since we do not consider optimization in the project.
- The actual node classes' hierarchy is much more complicated than is shown here. There are about 90 node classes in our implementation.

4.2 The Translation Procedure

The translation procedure (from the VC for Simplify to the VC for PVS) is showed in figure 4.4. The initialization step is performed in the constructor of the class *VcGenerator*. The remaining steps are executed in the method *getProof* of *VcGenerator*. Each step is explained below.

- **Initialization**: In this step, we set the prover type and save the root node of the old AST. The predefined types, variables and methods are initialized and saved in the *HashMap* fields of the classes *TypeInfo*, *VariableInfo* and *MethodInfo*.
- **Get all types, variables and method names from the old AST**: In order to serve the subsequent process, the names of all types are retrieved from the old AST. The retrieved type names are stored in a *HashMap* field of the class *TypeInfo*. Similar processing is done for the variable names and method names.
- **Build the new AST from the old AST**: A new AST is built from the old AST in the method *generateNewAST* of the class *VcGenerator*. Each node in the old AST is translated to a typed node in the new AST.
- **Type the new AST**: In this step, we give a type to every node of the new AST. The type assignment is performed according to the context

of the node. For example, if the parent of a node is *refEQ* which is a predicate on *references*, then the node type must be *reference*. Since all variables, constants and methods are represented as certain nodes in the new AST, they are all typed after we type the whole new AST. Here, we have to type the new AST twice. This is because we access the new AST in a certain order. Sometimes, the type of a node which represents a variable or method can not be determined in the current place, but can be determined in other places later. In this situation, we should give the node the right type according to information coming from other places. We execute the typing process twice to overcome this problem.

- **Set method parameters**: Although we retrieved method names in the previous step, the types of method parameters can only be determined after we typed the new AST. The types of the method parameters are stored in the *HashMap* field of the class *MethodInfo* together with the method name.

- **Rewrite all types, variables and method names**: Not all names coming from the Simplify logic are legal in PVS setting. Identifers in PVS are composed of letters, digits, and the characters "_" or "?", and they must begin with a letter. In this step, we rewrite all types, variables and method names according to the PVS syntax.

- **Simplify the new AST**: As we explained in section 4.1, this step was not performed in our project.

- **Generate the VC for PVS**: After the new AST is well typed and all required information is retrieved from the old AST, we can generate the VC for PVS. This step is performed in the method *getProof* of descendants of the abstract class *ProverType*. As to our VC translator for PVS, the class *PvsProver* implements *ProverType* and overrides *getProof*.

4.3 How to Support a New Prover

Basically, only two classes are needed to support a new prover under the current framework: the implementations of the abstract classes *ProverType* and *TVisitor*.

ProverType provides an interface when adding a new prover. The following methods are defined in *ProverType*[4]:

- *abstract public void init()*
  Performs some initialization work, e.g. initializing predefined types, variables and methods.

- *abstract public void getProof(Write out, String name, TNode root)*
  Generates VC for the target prover. The parameter *name* is the given VC name; *root* indicates the root node of the new AST; *out* is the output stream.

[4]We ignore the exception handling in the discussion

- *abstract public TNode rewrite(TNode n)*
  Simplifies the new AST. It is not implemented in our project.

- *abstract public String getTypeInfo(TypeInfo ti)*
  Returns a valid type name of *ti* for the target prover.

- *abstract public String getVariableInfo(VariableInfo vi)*
  Returns a valid variable name of *vi* for the target prover.

- *abstract public String getMethodInfo(MethodInfo mi)*
  Returns a valid method name of *mi* for the target prover.

- *abstract public Expr addTypeInfo(InitialState s, Expr e)*[5]
  Adds some background predicates to the generated VC, e.g. "$|state@pre| = state \wedge eClosedTime(elems) < alloc$".

- *abstract protected TVisitor visitor(Writer out)*
  Returns a concrete *Visitor* for the target prover.

- *abstract protected void generateDeclarations(Writer out, HashMap vars)*
  final public void generateDeclarations(Writer out, TNode n)
  Generates type, variable and method declarations. They should be called in the method *getProof*.

- *final public void generateTerm(Writer out, TNode n)*
  Generates the terms of VC by calling "$n.accept(visitor(out))$". It should be called in *getProof* as well as *generateDeclarations*.

- *abstract public String labelRename(String label)*
  Rewrites the given VC name.

There is not too much to say about the class *TVisitor*. Its function and application are explained in the *Visitor* pattern. *TVisitor* uses the class *PrettyPrinter* to format its output - the VC for the target prover. Every node of the new AST has an abstract *visit* method in *TVisitor*. For example, the *visit* method for the node *TBoolean* is:

```
abstract public void visitTBoolean(TBoolean n)
```

Its implementation in the VC translator for PVS is:

```
public void visitTBoolean(TBoolean n) {
   if (n.value)
      lib.appendN("TRUE");
   else
      lib.appendN("FALSE");
}
```

where *lib* is a *PrettyPrinter* field in *TVisitor*.

[5]The method comes from the initial VC translator. We think the method name does not explain its intention well.

4.4 Some Important Classes

TypeInfo
Type information is saved in the class *TypeInfo*. There are two fields in *TypeInfo*: *old* and *def*. *old* records the original name of the type in the VC for Simplify. It should be unique. *def* saves the new name for the target prover. *TypeInfo* has a static field of type *HashMap*: *typeMap*. The field is used to contain all types occurring in the old AST. The method *getTypeAST* of the class *VcGenerator* retrieves type information from the old AST, and saves this in *typeMap*.

VariableInfo
The function of *VariableInfo* is similar to *TypeInfo*, but it saves variable information instead of type information. There are also two fields, *old* and *def*, to save the original name and new name of a variable. A field, *type*, stores the type for the variable. A static field of type *HashMap*, *variableMap*, contains all variables occurring in the old AST.

MethodInfo
MethodInfo is similar to *TypeInfo* and *VariableInfo*. It saves method information. The *Vector* field, *argsType*, is used to save the parameter types of a method. The field *returnType* stores the return type of the method.

TNode
TNode is the superclass of all nodes in the new AST. It has a field, *type*, to indicate the type of the node. Another field, *parent*, records the parent node of current node. There are two direct subclasses of *TNode*: *TFunction* and *TVariable*. Only nodes of type *TFunction* can have children in the new AST. One important method[6] of *TNode* is *typeTree*. It types the current node and the child nodes of the current node. Every node class should provide an implementation for *typeTree*.

PrettyPrinter
The class *PrettyPrinter* is used to format the output of the generated VC. In the constructor of *PrettyPrinter*, it initializes the symbols which represent tab, left bracket, right bracket and "\n". A field of type *StringBuffer* records indentation in the output. In order to get an impression, let us look at one method of *PrettyPrinter*.

```
public Writer append(String s) {
    out.write(indentation.toString());
    out.write(s);
}
```

This method first outputs indentation, then outputs the String *s*. Another method *appendIwNl* performs the following processing on the output: increase indentation by a tab space, output an indentation space, output a left bracket and the String *s*. The counter-method of *appendIwNl* is *reduceIwNl* which reduces indentation by a tab space.

```
public Writer appendIwNl(String s) {
```

[6] Another important method is *accept* which was already introduced in section 4.1.

```
    indentation.append(TAB);
    out.write(indentation.toString());
    out.write(LBR + s);
}

public Writer reduceIwNl() {
    out.write(RBR);
    indentation = indentation.delete(0, TAB.length());
    return out;
}
```

4.5 The Format of the VC for PVS

The format of the generated VC for PVS is as follows:

```
theory_name: THEORY
BEGIN

IMPORTING escjava2_logic
%% import the semantic prelude

class_name: TYPE+ FROM JavaType
%% introduce a type for the class being checked

model_method_declarations
%% declare the model methods

theorem_name: THEOREM
Forall(
variable_declarations):
the_body_of_the_VC

END theory_name
```

where *theory_name* is composed of the class name, the name of the method being checked, the line and column number of the method. Consider the program in section 3.3:

```
1 final class test {
2     //@ ensures \result == a + b;
3     public int add(int a, int b) {
4         return a + b;
5     }
6 }
```

The generated VC for the method *add* is about 120 lines. An impression can be obtained from the following[7]:

```
vc_test_add_2_4: THEORY
BEGIN

IMPORTING escjava2_logic
```

[7]The complete VC is included in appendix B.

```
test: TYPE+ FROM JavaType

vc_test_add_2_4: THEOREM
Forall(
b_3_30: IntegralNumber,
brokenObj: Reference,
RES: IntegralNumber,
a_3_23: IntegralNumber,
brokenObj_1_: Reference):

(
  (
    (elems_pre = elems)
    AND
    ((eClosedTime(elems)) < alloc)
    AND
    ...
  )
  IMPLIES
  (NOT
    (TRUE
      AND
      ...
      AND
      (
        (NOT(ecReturn = ecReturn))
        OR
        (
          (ecReturn = ecReturn)
           AND
          (
            (NOT
              (
                ((ecReturn = ecReturn) AND (this /= NULL))
                IMPLIES
                (RES = (a_3_23 + b_3_30))
              )
            )
            OR
            ...
          )
        )
      )
    )
  )
)

END vc_test_add_2_4
```

4.6 Our Work on the VC Translator

As introduced in the Chapter 1, before we started the project, Clément Hurlin and Carl Pulley already had developed an initial VC translator (see [7]). Our

work and the content of this chapter are greatly affected by their work. Here we explain what we did concerning their initial work.

- We reimplemented the whole program except for the node classes and the class *PrettyPrinter*. From our point of view, although the framework of the VC translator was well designed in the initial work, its implementation was not perfect. We found at least five bugs in the implementation. The VC translation procedure was not very clearly designed in the initial VC translator. For example, in the initial design, step 2 (get types, variables and method names) was not separated from step 3 (build the new AST) in the translation procedure. It builds the new AST as well as retrieves type and variable information from the old AST. The approach is efficient, but not very clear from the functional point view and not very convenient for our experiments.

- The initial design follows the Object-Orient design style. It gives much flexibility to the software. But we think it abused inheritance and overriding somewhat sometimes. For example, in the abstract interface class *ProverType*, the methods *generateDeclarations* and *generateTerm* are not interface methods at all since they are just invoked by another method *getProof* of *ProverType*. It is not a good design to define *generateDeclarations* and *generateTerm* in the interface class. So, in our implementation, we try to avoid the effect of these imperfect designs.

- In order to extend our VC translator, we had to change the design of the initial VC translator somewhat. For example, we added the node class *TSum* to support quantifier operators, and we also added the class *MethodInfo* to support method call, etc. Details about the extension are discussed in the next chapter.

Chapter 5

Extending the VC Translator

As introduced in Chapter 1, we followed a test case driven approach in the project. More specifically, if we want to extend our VC translator to a certain feature, we follow the approach:

- design a test program.
- investigate the GC for the test program.
- investigate the old AST for the test program.
- develop the VC translator according to the investigations.
- prove the test program by the developed VC translator.

In this chapter, we follow the same road to explain how to extend our VC translator to some advanced features: quantifier operators, method call, and array.

5.1 JML Quantifier Expression

In JML, we can use expressions with the quantifiers *\forall* and *\exists*. For example, the expression

```
(\forall int i, j; 0 <= i && i < j && j < 10; a[i] <= a[j])
```

says the array a is sorted at indexes between 0 and 9. Although quantifier expressions are very useful in practice, the initial VC translator does not support them. So, we decided to extend our VC translator to handle quantifier expressions in the project.

◇ The Test Program

```
1 class quan {
2   //@ ensures (\forall int i; 0 <= i && i < 6; \result >= i);
3   public int m() {
4     return 5;
5   }
6 }
```

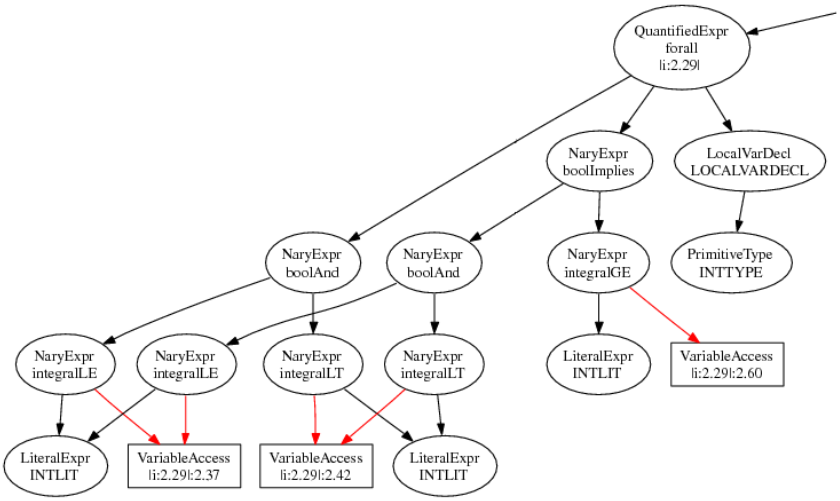

Figure 5.1: The Quantifier Expression in the Old AST

◇ **The GC for The Quantifier Expression**
The generated GC for the quantifier expression in line 2 is as follows:

```
(\forall int i:2.29; boolImplies(
    boolAnd(integralLE(0, i:2.29), integralLT(i:2.29, 6)),
    integralGE(RES, i:2.29)))
```

where the variable "$i : 2.29$" corresponds to the specification variable i and the RES corresponds to the JML expression "\result".

◇ **The Quantifier Expression in the Old AST**
The figure 5.1 shows the quantifier expression in the old AST. The expression is represented by three branches. The first branch is the guard of the quantifier expression, i.e. "$0 <= i \;\&\&\; i < 6$"; the second branch represents the body of the quantifier expression, i.e. "$0 <= i \;\&\&\; i < 6 \rightarrow \backslash result >= i$"; the third branch encodes the type information about the bound variable i.

◇ **Develop the VC Translator**
In order to support quantifier expression, we add a node class *TQuantify* and its subclasses *TForall* and *TExist* into our VC translator. An array field *pars* with type *VariableInfo*[ ] is defined in *TQuantify*. This array is used to contain bound variables of quantifier expressions.

Before building the new AST, all variables including bound variables of quantifier expressions are retrieved from the old AST by the method *getVariableAST*. After that, building the new AST from the old AST for quantifier expressions

is much easier to implement. The source code which parses the old AST and builds the new AST for the node *TForall* is as follows:

```
1 private TNode generateNewAST(ASTNode n) {
2   TNode currentNode = null;
3
4   switch (n.getTag()) {
5     ...
6     case TagConstants.FORALL: {
7          currentNode = new TForAll();
8          QuantifiedExpr m = (QuantifiedExpr) n;
9          int num = m.vars.size();
10         ((TForAll) currentNode).pars = new VariableInfo[num];
11         for (int i = 0; i < num; i++) {
12            GenericVarDecl v = m.vars.elementAt(i);
13            String name =
14               Atom.printableVersion(UniqName.variable(v));
15            ((TForAll) currentNode).pars[i] =
16               VariableInfo.getVariable(name);
17         }
18         break;
19    ...
20  }
21  ...
22 }
```

where line 9 to 17 are about saving bound variables in *pars*. Here, the method *getVariable* in line 16 is used because before building the new AST all bound variables are already parsed and saved in *VariableInfo* (by the method *getVariableAST* of the class *VcGenerator*).

We also need to define the *visit* methods for the node classes *TForall* and *TExist*. Since the processes for the two nodes are almost the same, a method *quanOp* is defined in the class *TPvsVisitor*, and the *visit* methods of *TForall* and *TExist* invoke *quanOp* with different parameters.

◇ Prove the Test Program
For our test program, the generated VC for the quantifier expression is as follows:

```
(FORALL (i_2_29: IntegralNumber):
  (
    ((0 <= i_2_29) AND (i_2_29 < 6))
    IMPLIES
    (5 >= i_2_29)
  )
)
```

The VC of the test program was proved in PVS with the tactics *grind*. As expected, if we change the number "5" in line 2 to "6", the proof can not be established.

◇ Discussion
Supporting quantifier expressions in our VC translator is quite easy and straight forward since PVS has the keywords "FORALL" and "EXISTS" as well. The only tricky part is how to handle bound variables and their types. Actually, we

noticed that the VC translator for Coq did not correctly process bound variables when we read their source code during the project.

5.2 Method Call in Specification

It is very useful to use method call in specification as well as in implementation. Firstly, methods are units of reuse in specification. Secondly, methods provide a means of abstraction. In JML, we call the specification methods model methods. Model methods should be side effect free which means they should not interfere with program execution. Methods without side effect are called *pure* methods. If pure methods allocate and initialize new objects, they are *weak pure*. Otherwise they are considered to be *strong pure*. In the project, we focused on the strong pure model methods.

◇ **The Test Program**

```
1 class methodcall {
2   //@ ensures \result == i + 1;
3   //@ model pure static int inc(int i);
4
5   //@ ensures \result == inc(c);
6   public int next(int c) {
7     return ++c;
8   }
9 }
```

◇ **The GC for The Model Method**
ESC/Java2 does not generate VC for model methods. So, for the test program, only two VCs are produced: one for the method *next*; the other one for the default constructor. The specifications about the model method *inc* are translated into the *assume* and *assert* clauses in the preconditions and postconditions of the method *next*. The following statements show the GC concerning the model method *inc*.

```
...
ASSUME (\forall int i:3.38;
        integralEQ(method.inc.3.26(i:3.38),
                   integralAdd(i:3.38, 1)));
ASSUME (\forall int i:3.38;
        is(method.inc.3.26(i:3.38), \type(int)));
...
ASSERT (\lblneg Post:5.8@8.4
        boolImplies(boolAnd(anyEQ(EC, ecReturn),
                    is(this, \type(methodcall)), refNE(this, null)),
                    integralEQ(RES, methodcall.inc.3.26(c:6.24))));
...
```

The first *assume* clause corresponds to the specification of line 2. The second *assume* clause specifies that the return type of *inc* is *int*. The *assert* clause encodes the postcondition in line 5.

◇ **The Model Method in the Old AST**
The figure 5.2 represents the first *assume* clause in the old AST. We could notice

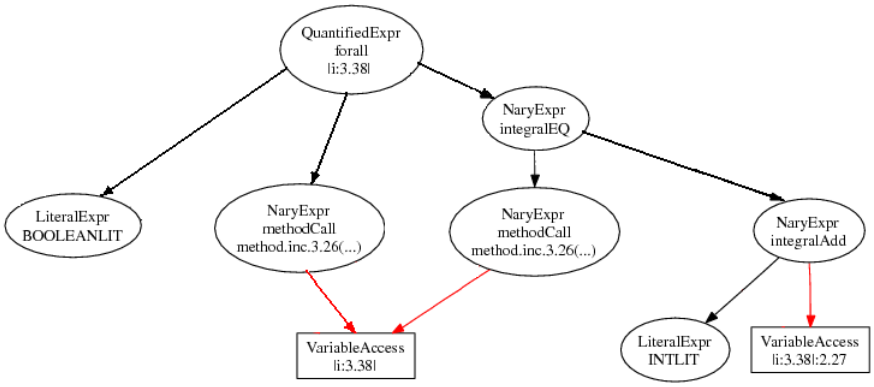

Figure 5.2: The Method Call in the Old AST

that the parameter "$|i : 3.38|$" is encoded into a child node of its method node.

◇ **Develop the VC translator**
In order to support model methods, we define a class *MethodInfo* to store method information. A node class *TMethodCall* is introduced into our VC translator as well. Two methods *getMethodAST* and *setMethodArgsType* are defined in the class *VcGenerator*: *getMethodAST* retrieves the names of model methods from the old AST; *setMethodArgsType* sets the types of return value and parameters of model methods.

In the VC translator, the names of model methods are parsed and saved in *MethodInfo* before building the new AST. Then after typing the new AST, we set the types of return value and parameters for all model methods. When generating the VC for PVS, the declarations of model methods are produced before outputting the main VC body.

◇ **Prove the Test Program**
Part of the generated VC concerning the model method *inc* is shown below:

```
...
methodcall_inc_3_26(p0: IntegralNumber): IntegralNumber

vc_methodcall_next_5_4: THEOREM
Forall(
...
i_3_38: IntegralNumber,
c_6_24: IntegralNumber,
c_7_17: IntegralNumber):
(
  ...
  AND
  (FORALL (i_3_38: IntegralNumber):
     (methodcall_inc_3_26(i_3_38) = (i_3_38 + 1))
  )
  ...
```

```
  AND
  ( ...
    (
      (ecReturn = ecReturn)
      AND TRUE
      AND (this /= NULL)
    )
    IMPLIES
    (c_7_17 = methodcall_inc_3_26(c_6_24))
  )
)
```

We proved the VC in PVS with the tactics *grind*.

◇ **Discussion**
Supporting strong pure model methods is easy in our VC translator. The only troubling thing is determining the types of model methods. In the future, if ESC/Java2 is built on a sorted logic from the frontend, the work concerning types would be much easier.

An important area that was not covered in our project is weak pure model methods. In [1], the authors proposed a sound approach to handle weak pure model methods. It is interesting to investigate the approach and consider how to implement the approach in our VC translator in the future.

5.3 Array in Implementation

Although almost all non-trivial Java programs need the type array, the initial VC translator did not support it. So, we investigated ways to support arrays in the project. From a theoretical point of view, there are no difficulties to handle arrays in our proposed approach. But there are some problems in the implementation.

5.3.1 The Issues about Type Assignment

◇ **Problem 1**:

Consider the following program:

```
1  class arr1 {
2    private int[] t;
3
4    //@ requires a != null;
5    public void init(int[] a) {
6      t = a;
7      for (int i = 0; i < a.length; i++) {
8        t[i] = 0;
9      }
10   }
11 }
```

The generated VC includes the following sub-formula ϕ:

```
elems_7_8_0 = (set(elems, tmp0_t_7_8_0_8_12,
                   (set(get(elems, tmp0_t_7_8_0_8_12)), 0, 0)))
```

where *elems* is a predefined variable with the type *ArrayStore* which models the state of all arrays. The methods *get* and *set* are defined in the theory *map_theory* in the semantic prelude. They model memory operations - read and write separately. *tmp0_t_7_8_0_8_12* is an intermediate variable generated by ESC/Java2. Its type must be *ArrayReference* since another expression

```
0 < (arrayLength(tmp0_t_7_8_0_8_12))
```

exists in the generated VC.

However, the only instantiation of *map_theory* for arrays in the semantic prelude is

```
map_theory[ArrayStore, ArrayName, ArrayReference]
```

which introduces

```
get: [ArrayStore, ArrayName -> ArrayReference]
set: [ArrayStore, ArrayName, ArrayReference -> ArrayStore]
```

This causes a problem: there is no proper *get* and *set* matching the types of the formula ϕ since ϕ requires:

```
get : [ArrayStore, ArrayReference -> ArrayReference]
set : [ArrayStore, ArrayReference, ArrayReference -> ArrayStore]
```

In order to solve this problem, we propose that the following instantiation of *map_theory* should be added to the semantic prelude:

```
map_theory[ArrayStore, Reference, ArrayReference]
```

which introduces the proper *get* and *set* methods

◇ **Problem 2**:

However, the solution to problem 1 does not solve all the problems with type assignment. Consider another program:

```
1 class arr2 {
2   //@ requires a != null;
3   //@ requires a.length >= 2;
4   public void init(int[] a) {
5     a[0] = 0;
6     a[1] = 1;
7     //@ assert a[0] != a[1];
8   }
9 }
```

The generated VC for line 7 is:

```
(get(get(elems_2_, a_4_23)),0) /= (get(get(elems_2_, a_4_23)),1)
```

where the type of "elems_2_" is *ArrayStore* and the type of "a_4_23" is *ArrayReference*. This subformula can not pass the type checking of PVS since more than one instantiation of *map_theory* matches it. For example, the suitable instantiations include:

```
1. map_theory[ArrayReference, IntegralNumber, Boolean]
   which produces get:[ArrayReference, IntegralNumber -> Boolean]
2. map_theory[ArrayReference, IntegralNumber, Number]
   which produces get:[ArrayReference, IntegralNumber -> Number]
3. map_theory[ArrayReference, IntegralNumber, Reference]
   which produces get:[ArrayReference, IntegralNumber -> Reference]
```

In order to avoid ambiguous interpretations, we could provide extra information to assist PVS in choosing an unique instantiation of *map_theory*. For example, we can rewrite the VC as:

```
(get[ArrayReference, IntegralNumber, Number]
    (get(elems_2_, a_4_23)),0) /= (get(get(elems_2_, a_4_23)),1)
```

However, this modification is hard to implement in our VC translator since the VC translator does not know what the semantic prelude defines and when it should generate the extra information.

5.3.2 The Issues about Modeling New Arrays

◇ The predicate *arrayFresh* in the original unsorted logic

- One-dimensional array

In the original unsorted logic of ESC/Java, there is a predefined predicate *arrayFresh* to model the allocation of new arrays. For example, consider a Java statement:

```
int[] x = new int[3];
```

ESC/Java2 generates a GC for the statement which looks like:

```
ASSUME arrayFresh(x, alloc, alloc', elems, arrayShapeOne(3),
                  \type(int[]), 0)
```

where x is the newly allocated array; *alloc* and *alloc'* are the allocation times just before and after the allocation of x; *elems* is a global variable modeling the state of all arrays; *arrayShapeOne* constructs an array shape. Intuitively, a shape is a nonempty list of integers, representing the dimensions of a rectangular array. For example, *arrayShapeOne*(3) represents the shape of an one-dimensional array of length 3; *int*[ ] is the type of the array; 0 is the default initial value for the elements of the array.

The meaning of *arrayFresh* for an one-dimensional array is precisely defined by the following axiom:

```
(ALL a, t1, t2, e, n, T, v::
   arrayFresh(a, t1, t2, e, arrayShapeOne(n), T, v) ==
     t1 <= vAllocTime(a) && vAllocTime(a) < t2 && a != null &&
     typeof(a) == T && arrayLength(a) == n &&
     (ALL i:: e[a][i] == v)
```

- Multi-dimensional array

Similarly, consider a multi-dimensional array

```
   int[][] y = new int[5][6];
```

ESC/Java2 generates the following GC:

```
ASSUME arrayFresh(y, alloc, alloc', elems,
                  arrayShapeMore(5, (arrayShapeOne(6)),
                  \type(int[][]), 0)
```

where *arrayShapeMore* is similar to *arrayShapeOne*, but it constructs a multi-dimensional array shape.

The meaning of *arrayFresh* for a multi-dimensional array is defined by the following axiom:

```
(ALL a, t1, t2, e, n, T, v::
   arrayFresh(a, t1, t2, e, arrayShapeMore(n, s), T, v) ==
     t1 <= vAllocTime(a) && vAllocTime(a) < t2 && a != null &&
     typeof(a) == T && arrayLength(a) == n &&
       (ALL i::
          arrayFresh(e[a][i], t1, t2, e, s, elemType(T), v) &&
          arrayParent(e[a][i]) == a &&
          arrayPosition(e[a][i] == i)
```

The predicates *arrayParent* and *arrayPosition* ensures that the arrays allocated as part of of the multi-dimensional array are distinct (see [23]).

◇ How does ESC/Java2 reason about array operations with *arrayFresh*

There is no document that completely explains how ESC/Java2 uses *arrayFresh* to reason about array operations in Java programs (the only document we know is [23]). But we do know that array bound checking is one of key characteristics of ESC/Java. Consider the following program:

```
1  class arr3 {
2    public void m() {
3      int[] x = new int[3];
4      x[4] = 0;
5
6      int[][] y = new int[5][6];
7      y[7][0] = 0;
8      y[2][8] = 0;
9      //@ assert y[0] != y[1];
10   }
11 }
```

Obviously, line 4, 7 and 8 cause array bound errors. ESC/Java2 generates the GC for the program as follows:

```
   ...
1  ASSUME arrayFresh(tmp0!new!int[]:3.18, alloc, alloc<1>, elems,
                     arrayShapeOne(3), \type(int[]), 0);
   ...
2  x:3.14 = tmp0!new!int[]:3.18;
3  tmp1!x:4.8 = x:3.14;
4  ASSERT (\lblneg Null@4.9 refNE(tmp1!x:4.8, null));
5  ASSERT (\lblneg IndexNegative@4.9 integralLE(0, 4));
6  ASSERT (\lblneg IndexTooBig@4.9
```

```
            integralLT(4, arrayLength(tmp1!x:4.8)));
   ...
7  ASSUME arrayFresh(tmp2!new!int[][]:6.20, alloc, alloc<2>, elems,
                     arrayShapeMore(5, arrayShapeOne(6)),
                     \type(int[][]), 0);
   ...
8  y:6.16 = tmp2!new!int[][]:6.20;
9  ASSERT (\lblneg Null@7.9 refNE(y:6.16, null));
10 ASSERT (\lblneg IndexNegative@7.9 integralLE(0, 7));
11 ASSERT (\lblneg IndexTooBig@7.9
            integralLT(7, arrayLength(y:6.16)));
12 tmp3:7.9 = select(select(elems, y:6.16), 7);
13 ASSERT (\lblneg Null@7.12 refNE(tmp3:7.9, null));
14 ASSERT (\lblneg IndexNegative@7.12 integralLE(0, 0));
15 ASSERT (\lblneg IndexTooBig@7.12
            integralLT(0, arrayLength(tmp3:7.9)));
   ...
16 ASSERT (\lblneg Null@8.9 refNE(y:6.16, null));
17 ASSERT (\lblneg IndexNegative@8.9 integralLE(0, 2));
18 ASSERT (\lblneg IndexTooBig@8.9
            integralLT(2, arrayLength(y:6.16)));
19 tmp4:8.9 = select(select(elems, y:6.16), 2);
20 ASSERT (\lblneg Null@8.12 refNE(tmp4:8.9, null));
21 ASSERT (\lblneg IndexNegative@8.12 integralLE(0, 8));
22 ASSERT (\lblneg IndexTooBig@8.12
            integralLT(8, arrayLength(tmp4:8.9)));
   ...
```

Line 1 of the GC corresponds to line 3 of the Java program. According to the axiom of *arrayFresh*, line 1 of the GC provides the information

```
arrayLength(tmp0!new!int[]:3.18)==3
```

This causes the assertion in line 6 to fail since "$integralLT(4, 3)$" is false. Then ESC/Java2 can detect the error. Similarly, *arrayFresh* in line 7 causes the assertions in line 11 and 22 to fail.

Besides the length information, the definition of *arrayFresh* also helps Simplify to reason about the assertion in line 9 of the Java program. The *arrayFresh* in line 7 of the GC introduces the following sub-formulas:

```
arrayPosition(select(select(elems, y), 0))==0 &&
arrayPosition(select(select(elems, y), 1))==1
```

If the assertion is right, i.e. "select(select(elems, y), 0)==select(select(elems, y), 1)", then Simplify can deduce "0==1". This is obviously wrong.

◇ Interpret *arrayFresh* in the new sorted logic

Because our VC translator transforms the VC for Simplify into the VC for PVS, it needs to interpret the predicate *arrayFresh* in the translation procedure. The most safe and reasonable way to deal with *arrayFresh* is to interpret it line by line in the sorted logic according to its definition in the unsorted logic. However, Joseph Kiniry changed the semantics for array in the sorted logic (we are not clear about his intention of the change). For example, the predicates *arrayFresh*, *arrayShapeOne*, *arrayShapeMore*, *arrayParent* and *arrayPostion* are

not provided in the current semantic prelude. This makes the interpretation not straight forward.

While deleting some old predicates, the current semantic prelude also provides some new predicates:

```
typeOf(r: Reference): ReferenceType
elemtype: [ArrayType -> JavaType]
array_constructor: [JavaType -> ArrayType]
arrayOf(n: IntegralNumber, t: PrimitiveType): ArrayReference
arrayOf(n: IntegralNumber, t: ReferenceType): ArrayReference
arrayLength(r: ArrayReference): IntegralNumber
```

Consider the two Java statements:

```
int[] a = new int[n];
int[][] b = new int[n][m];
```

As to the array *a*, the VC for Simplify which models its allocation looks like:

```
arrayFresh(a, t1, t2, elems, arrayShapeOne(n), array(T_int), 0)
```

We can interpret it in the new sorted logic as follows:

```
newArray(a, t1, t2, elems, n, array_constructor(T_int))
```

where *newArray* is a new predicate we defined in the semantic prelude. Its definition is:

```
newArray: [ArrayReference, Time, Time, ArrayStore,
           IntegralNumber, ArrayType -> bool]

newArray_axiom: AXIOM
   FORALL (arr: ArrayReference, t1: Time, t2: Time,
           e: ArrayStore, n: IntegralNumber,
           t: ArrayType):
   newArray(arr, t1, t2, e, n, t) IFF
   t1 <= vAllocTime(arr) AND vAllocTime(arr) < t2 AND
   arr != null && typeof(arr) == t AND
   arrayLength(arr) == n
```

As to the array *b*, the VC for Simplify which models its allocation looks like:

```
arrayFresh(b, t1, t2, elems, arrayShapeMore(n, arrayShapeOne(m)),
           array(array(T_int)), 0)
```

We can interpret it in the new sorted logic as follows:

```
newArray(b, t1, t2, elems, n,
         array_constructor(array_constructor(T_int))) AND
FORALL (i: IntegralNumber):
   0 <= i AND i < n AND
   newArray(get(get(elems, b), i), t1, t2, elems, m,
            array_constructor(T_int))
```

There are two obviously missing points in our interpretation. Firstly, since without the predicate *arryPosition* and *arrayParent*, our interpretation does not distinguish the different parts of a multi-dimension array. This would cause problems in proving the assertions like "b[0] != b[1]" if *b* is a multi-dimension

array; Secondly, we do not provide the specification to default initial values of the elements of array. Actually, we are not clear about the application of initial values in the static checking of ESC/Java2.

We tested our interpretation on some small Java programs with arrays. But the interpretation is done by hand, i.e., we modified the generated VC by hand according to the above description. We did not implement the interpretation approach in our VC translator because of the complexity of the interpretation and the time limitation of the project. According to our experiments, the interpretation is able to detect array bound errors. In the future, more investigations are needed and the interpretation should be implemented in the VC translator.

Chapter 6

Supporting Native Specifications

When doing program verification with JML, one has to translate Java implementations and JML specifications into the specification language of the target prover. In this thesis, we call prover oriented specifications *native specifications*. Since native specification languages would provide additional expressibility over JML, it is very useful to integrate native specifications into JML specifications directly. For example, consider a set Q:

Q = {i: integer | ∃ k (0 ≤ k < m) & A[k] = i}

where $1 \leq m \leq A.length$. Obviously, Q represents the set of values of the first m elements of A. Although this is a common mathematical concept, it is hard to specify in ESC/Java2 since JML has no built-in set concept[1]. But we can specify Q in PVS like:

```
Q: set = {i: int | EXISTS(k: nat): 0 <= k AND k < m AND A(k) = i}
```

In [21], a new keyword *native* is proposed to indicate native specifications in JML. Using the keyword, the verification tool (Jack in [21]) can perform some processing on native specifications since the tool can distinguish native specifications from ordinary JML specifications. Although adding the new keyword can help us express our intention more clearly and give verification tools more freedom, we can use model classes and model methods to achieve a similar result as the one in [21]. In our project, we chose the second approach (use model classes and model methods to define native specifications) since it allows us to avoid modifying the frontend of ESC/Java2.

As to native specifications, we consider two kinds of applications. Firstly, we want to introduce *native methods* into JML specifications. These methods are defined in semantic preludes or predefined libraries of provers. Secondly, we also want to use *native types* in our specifications. These types are build-in types of provers or user-defined types in semantic preludes. After introducing native types, we can use them to declare ghost variables or model fields in JML specifications.

[1] We can specify it using the JML runtime library class *JMLObjectSet*, but the runtime library is not supported by ESC/Java2. Actually, since JML is a large specification language, most of the existing JML tools only support certain subsets of JML.

6.1 Adding Native Methods to ESC/Java2

We restrict native methods considered in our project to be pure methods that do not allocate and initialize new objects. Furthermore, we also assume that native methods always terminate and do not throw any exceptions.

6.1.1 The Approach to Support Native Methods

The approach to support native methods requires three steps in our experiments:

1. define a function with the PVS specification language in the semantic prelude;
2. declare a native method as a model method in the JML specification;
3. map the native method to the function in the generated VC.

STEP 1[2]
There are several ways to define a function in PVS. Here, we give some ways used in our project.

- declare a function type, then use axioms to specify its properties.
  example:

```
fClosedTime: [Field -> Time]     % declare a function type
fClosedTime_definition: AXIOM    % specify its properties
   FORALL (r: Reference, f: Field, t: Time):
      (fClosedTime(f) < t AND isAllocated(r, t))
      IMPLIES isAllocated(get(f, r), t)
```

- declare an interpreted constant which defines the desired function.
  example:

```
g(int x): int = x + 1
```

- give the recursive definition for the desired function.
  example:

```
factorial(x: nat): RECURSIVE nat =
   IF x = 0 THEN 1
   ELSE x * factorial(x - 1)
   ENDIF
MEASURE (LAMBDA (x: nat): x)
```

STEP 2
Since we did not introduce the new keyword *native* into ESC/Java2, native methods are just declared as model methods in JML specifications.
example:

```
//@ public model pure static int pvs_sum(int i);
```

[2]If we map a native method to a predefined function of provers, step 1 is not needed.

Here, the keyword *model* means the method is a model method; the keyword *pure* says the model method is side effect free. After the declaration, we can use the method in our specification. ESC/Java2 does not know the model method actually is a native method. It interprets the method in the generated VC just as other ordinary model methods.

STEP 3
Mapping the declared native method to the function defined in the semantic prelude or the predefined function of provers is currently done by hand. However, this step could be done automatically by ESC/Java2 if we introduce the new keyword *native* into ESC/Java2.

As to the example of the step 2, the mapping is done by providing the following definition at the beginning of the generated VC.

```
test_pvs_sum_9_3(p0: IntegralNumber): IntegralNumber = sum(p0)
```

where *sum* is a function defined in the semantic prelude.

6.1.2 Test Cases and Result Analysis

TEST CASE 1

In the test case, we use a native method *sum* to specify a Java program which calculates the sum from 1 to *n*. Then we use PVS to prove the correctness of the program, i.e., the return value equals the required sum.

step 1: define a function *sum* in the semantic prelude.

```
sum(i, j: IntegralNumber): IntegralNumber =
  IF i > j THEN 0
  ELSE ((i+j) * (j-i+1) / 2)
  ENDIF
```

step 2: declare a native method *pvs_sum* in the Java program.

```
class test1 {
    //@ public model pure static int pvs_sum(int i, int j);

    /*@ requires n >= 0;
      @ ensures \result == pvs_sum(0, n);
      @*/
    public int sum(int n) {
        int i = 0;
        int s = 0;

        //@ loop_invariant s == pvs_sum(0, i);
        //@ loop_invariant 0 <= i && i <= n;
        while (i < n) {
            s = s + i + 1;
            i++;
        }

        return s;
    }
}
```

step 3: map *pvs_sum* to *sum* in the generated VC[3].

```
test1_pvs_sum_2_33(p0: IntegralNumber, p1: IntegralNumber):
   IntegralNumber = sum(p0, p1)
```

Result: we proved the VC with the tactics *grind* in PVS.

TEST CASE 2

In the test case, we use a native method to specify and verify a Java program which calculates Fibonacci numbers.

step 1: define a function *fibo* in the semantic prelude.

```
fibo(i: IntegralNumber): RECURSIVE IntegralNumber =
  IF i = 1 THEN 1
  ELSE (IF i = 2 THEN 1
        ELSE (IF i >= 3 THEN fibo(i-1) + fibo(i-2)
              ELSE 0
              ENDIF)
        ENDIF)
  ENDIF
MEASURE (LAMBDA (i: IntegralNumber): i)
```

step 2: declare a native method *pvs_fibo* in the Java program.

```
class test2 {
    //@ public model pure static int pvs_fibo(int i);

    /*@ requires n >= 1;
      @ ensures \result == pvs_fibo(n);
      @*/
    public int fibo(int n) {
        int a = 0;
        int b = 1;
        int swap = 0;
        int i = 1;

        //@ loop_invariant a == pvs_fibo(i-1);
        //@ loop_invariant b == pvs_fibo(i);
        //@ loop_invariant 1 <= i && i <= n;
        //@ decreases n - i - 1;
        while (i < n) {
            swap = b;
            b = a + b;
            a = swap;
            i++;
        }

        return b;
    }
}
```

step 3: map *pvs_fibo* to *fibo* in the generated VC.

[3]ESC/Java2 adds the class name, the column and line number to the name of the method.

```
test2_pvs_fibo_2_33(p0: IntegralNumber):
   IntegralNumber = fibo(p0)
```

Result: We proved the VC with the tactics *grind* in PVS.

In the past, we tested the same program by giving the recursive specification of *fibo* with JML. ESC/Java2 can not prove the program. It shows that we can handle recursive specification better under the current approach (by integrating ESC/Java2 and PVS) compared to the old approach (using ESC/Java2 only).

6.2 Adding Native Types to ESC/Java2

There are two kinds of native types. In the first category the types are provided by provers. For example, PVS and its predefined libraries provide many types including *set*, *list*, *graph*, etc; the second category consists of user-defined types. We can define our own types in the semantic prelude. Both kinds of types can be introduced into JML specifications by model classes in our proposed approach.

6.2.1 The Approach to Support Native Types

The approach to support native types requires three steps in our experiments:

1. define a type with the PVS specification language in the semantic prelude;
2. declare a native type as a model class in the JML specification;
3. map the native type to the type in the semantic prelude in the generated VC.

STEP 1[4]
PVS provides quite flexible ways to define types. For example, we can define subtypes, function types, tuple types, and record types. There is also a quite powerful way to define abstract data types.

STEP 2
Native types are declared as empty model classes in JML specifications.
example:

```
//@ public model pure class pvs_set {};
```

After introducing native types, we can use them to declare ghost variables or model fields.
example:

```
//@ private ghost pvs_set s;
//@ public model pvs_set m;
```

STEP 3
Since native types are declared as model classes, our VC translator generates type declarations for these native types at the beginning of the generated VC. However, we should modify these declarations by hand because the default type

[4]If the type is a type provided by provers, step 1 is not needed.

for model classes is *JavaType*. For example, for the native type *pvs_set*, we should change the VC as follows:

pvs_set: TYPE+ FROM JavaType $\Longrightarrow$ pvs_set: TYPE+ = set

6.2.2 Test Cases and Result Analysis

TEST CASE 1

With an abstract Java class *myset*, we investigate the approach using native types to declare ghost variables. We introduce a native type *pvs_set* which corresponds to the PVS type *set*. Some native methods related to the native type are introduced into *myset* in order to perform our investigation. These methods include *create*, *member*, *add* and *delete*. With the native type and native methods, we can specify the behaviors of *myset*.

step 1: define a ghost variable with the native type *pvs_set* in the Java program.

```
//@ public model pure class pvs_set {}

abstract class myset {

  //@ ghost pvs_set s = create();

  //@ public model pure static pvs_set create();
  //@ public model pure static boolean member(pvs_set q, Object r);
  //@ public model pure static pvs_set add(pvs_set q, Object r);
  //@ public model pure static pvs_set delete(pvs_set q, Object r);

  //@ ensures member(s, o) == true;
  public void insert(Object o) {
    //@ set s = add(\old(s), o);
  }

  //@ ensures member(s, o) == false;
  public void remove(Object o) {
    //@ set s = delete(\old(s), o);
  }
}
```

step 2: map *pvs_set* to *set* in the generated VC.
As already introduced, the mapping is done by setting the type of *pvs_set* to be *set* in the generated VC.

```
IMPORTING sets[Reference]
...
pvs_set: TYPE+ = set
```

Result: We proved the VC by the tactics *grind* in PVS.

There are two problems which forced us to modify the generated VC by hand.

The first problem concerns the type of the ghost variable s[5]. ESC/Java2 treats ghost variables as object fields in the generated VC. So, our VC translator sets the type of s to be *field*. This is necessary since a predicate "fClosedTime(s) < alloc" exists in the VC and the type of s must be *field* according to the defintion of *fClosedTime*. However, the type of s should also be *pvs_set* since it is used as a parameter of the native methods *member*, *add* and *delete*.

We solved this problem by changing the type of s to *pvs_set* and deleting the predicate "fClosedTime(s) < alloc" in the generated VC. This modification should be OK since ghost variables have no effect on program states.

However, this modification gives us another problem. Since ESC/Java2 treats all ghost variables as fields, every read and write of s is modeled by the *get* and *set* predicates. The *get* and *set* predicates are defined in the parameterized theory *map_theory*. There is no proper instantiation for s since its type is *pvs_set*. Although we can add an instantiation of *map_theory* for s according to its type, the approach is very inconvenient since we have to define a proper instantiation in the semantic prelude every time we introduce a new native type.

We solved the problem by simply changing the VC to read and write s directly without using *get* and *set*. This modification also should be OK since s is not a real field and its operations have no effect on the program state.

If we introduce the new keyword *native* into JML, the two problems can be solved by ESC/Java2 automatically. With the new keyword, ESC/Java2 can distinguish native specifications from ordinary JML specifications. If ESC/Java2 detects that the type of a ghost variable is native, it generates direct reads and writes in the generated VC without using *get* and *set*.

TEST CASE 2

In the second test case, we investigate the application of a model field with a native type. We use a model field s with the native type *pvs_set* to specify the behaviors of *arrset*. *arrset* is a set class which is implemented by an array $A[]$. A native method *toSet* is defined as the representation function between s and the array $A[]$ (see [4]).

step 1: define a model field with the native type *pvs_set* in the Java program.

```
//@ public model pure class pvs_set {}

class arrset {
  //@ public model pvs_set s;

  //@ public model pure static pvs_set toSet(int[] a, int n);
  //@ public model pure static boolean member(pvs_set q, int r);

  private int[] A;
  //@ in s;
  //@ maps A[*] \into s;
  //@ represents s <- toSet(A, A.length);
  //@ invariant A != null;
```

[5]In the generated VC, s corresponds to *s_5_22* where "5" and "22" indicate the line and column number of s in the Java program.

```
  //@ invariant A.owner == this;

  private int m;
  //@ invariant 0 <= m && m <= A.length;

  public arrset() {
    A = new int [100];
    m = 0;
    //@ set A.owner = this;
  }

  //@ ensures member(s, i) == true;
  public void insert(int i) {
    if (m <= A.length - 1) {
      A[m] = i;
      m++;
    }
  }
}
```

step 2: define the representation function *toSet* in the semantic prelude.

```
toSet(A: ArrayReference, n: IntegralNumber): RECURSIVE set =
  IF n <= 0 THEN empty?
  ELSE add(toSet(A, n-1), get(A, n-1))
  ENDIF
MEASURE (LAMBDA (n: IntegralNumber): n)
```

Result: We failed to prove the VC in PVS.

Although we did not find a way to prove the program, the generated VC passed the type checking of PVS. This shows that using native types to declare model fields is feasible.

However, the failure uncovers a shortcoming of our proposed approach. In the original approach, if Simplify can not prove a VC, ESC/Java2 can indicate broken specifications or suspect bugs. It is very useful to help users track and analysis the reason why it failed. In our proposed approach, if we fail to prove a VC, it is quite difficult to know the reason: it is unclear whether the failure is caused by the VC itself or by an improper proof; if it is the problem of the VC, which lines of implementations or specifications cause the failure?

According to our understanding, the reason why ESC/Java2 can indicate broken specifications or suspect bugs is because ESC/Java2 introduces special labeled expressions in the VC for Simplify. Simplify can utilize these labeled expressions to indicate broken specifications or suspect bugs. In our VC translator, these labeled expressions are ignored since they are useless from the semantic point of view. How to deal with these labeled expressions in our proposed approach should be investigated in the future.

6.3 Conclusions

On the basis of our experiments, we conclude that our proposed approach can support native specifications quite well. We believe that supporting native spec-

ifications is a big advantage of our proposed approach. This advantage is demonstrated in the fibonacci program which can not be handled by ESC/Java2 but can be proved in our proposed approach.

However, in order to make the mapping from native specifications to the PVS methods and types being done automatically, i.e., without the need to change the generated VC by hand, we should introduce the new keyword *native* into ESC/Java2 as [21] did for Jack.

Chapter 7

Verifying the Celebrity Programs

In the final period of the thesis project, we decided to verify programs for the so-called celebrity problem using the techniques we developed so far. Since we already realized that our VC translator is still far from mature, the main goal of the experiments is to investigate the ability of our approach, especially the advantages of using native specifications in JML.

In order to achieve the goal, we carefully chose a problem which is quite simple but still interesting. In the experiments, we propose a new approach to verify the program for the celebrity problem on the abstract level. The approach was inspired by the work of Hoare [4]. The novel point of our work is that our approach is machine-checkable comparing to Hoare's pen and paper proving.

7.1 The Celebrity Problem

Among a group with n persons, a celebrity is someone who is known by everyone but does not know anyone. If we are allowed to ask questions of the form "does person x know person y", our task is to identify all celebrities in the group.

According to the properties of celebrities, it is easy to find out that there is no celebrity or just one celebrity in the group. Assume a and b are both celebrities in the group, then b must know a (everyone knows celebrities) and a must not know b (celebrities do not know anyone). Then b is not a celebrity since a does not know him/her. This gives a conflict to our assumption.

In order to make the problem clearer and simpler, we assume that the group has more than one person and there is a celebrity in the group. Then the problem can be described as follows:

```
There is a finite set G which has more than one element. The
binary relation set B on G (B = G -> G) represents the "knows"
relations (->) between elements of G. If there is a celebrity c
in G, request to identify c which has the following properties:

for all x in G, x != c,
(1) x -> c (everyone knows the celebrity);
(2) not(c -> x) (the celebrity does not know anyone);
```

7.2 The First Attempt to the Celebrity Problem

At the beginning, we provide a straight forward solution to the celebrity problem. We use the natural number 0 to n-1 to represent people in the group, and a two-dimensional array *knows* to represent the relations between people.

```
1  class celebrity1 {
2     /*@ requires n >= 2;
3       @*/
4     /*@ requires knows != null && knows.length == n;
5       @*/
6     /*@ requires (\forall int i; 0 <= i && i < n;
7       @            knows[i] != null && knows[i].length == n);
8       @*/
9     /*@ requires (\exists int c; 0 <= c && c < n;
10      @          (\forall int j; 0 <= j && j < n && j != c;
11      @           knows[j][c] == true && knows[c][j] == false));
12      @*/
13    /*@ ensures (\forall int i; 0 <= i && i < n && i != \result;
14      @           knows[i][\result] == true);
15      @*/
16    /*@ ensures (\forall int i; 0 <= i && i < n && i != \result;
17      @           knows[\result][i] == false);
18      @*/
19    public int findCeleb(int n, boolean[][] knows) {
20        int a, b;
21        a = 0;
22        b = n - 1;
23
24        /*@ loop_invariant (\exists int c; a <= c && c <= b;
25          @      (\forall int j; 0 <= j && j < n && j != c;
26          @       knows[j][c] == true && knows[c][j] == false));
27          @*/
28        while (a != b) {
29            if (knows[a][b])
30                a++;
31            else
32                b--;
33        }
34        return a;
35    }
36 }
```

ESC/Java2 can not prove the above program. It gives a warning that the loop invariant (line 24 to 27) possibly does not hold. Although we are not clear about the reason for the failure, the result is reasonable considering the complexity of the specification. We guess there are two difficulties when Simplify tries to prove the program. Firstly, Simplify has difficulties handling specifications with two embedded quantifier operators. Secondly, the result of *knows*[a][b] includes an implication: if *knows*[a][b] is true, then a can not be the celebrity; if *knows*[a][b] is false, then b can not be the celebrity. We are not sure Simplify is smart enough to derive and utilize the implication automatically.

We also used our VC translator to generate a VC for PVS. PVS can not prove the VC with the tactics *grind* automatically. We did not find a way to

prove the VC in our experiments, although we almost sure the proof exists. The reason for the failure is mainly because the generated VC is too large (about 1000 lines). It is hard for us to fully understand the VC and to use low level tactics to prove the VC interactively.

7.3 The Second Attempt to the Celebrity Problem

Since we guess the failure of the first attempt is because of the specifications with two embedded quantifier operators and the unexposed implication of *knows*[a][b], a natural consideration would be rewriting the program and specification to overcome the two problems. In order to do this, we introduce a model method *isCelebrity*. We assume that *isCelebrity* returns true if its parameter indicates the celebrity, otherwise it returns false.

We add two assumptions in the method *knows*. These assumptions guarantee the postconditions of *knows* hold. We can not prove the postconditions without these assumptions since *isCelebrity* is not defined explicitly.

```
1  class celebrity2 {
2     int N;
3     //@ invariant N >= 2;
4
5     boolean[][] B;
6     //@ invariant B != null;
7     //@ invariant B.owner == this;
8     //@ invariant B.length == N;
9     //@ invariant (\forall int i; 0 <= i && i < B.length;
                     B[i] != null);
10    //@ invariant (\forall int i; 0 <= i && i < B.length;
                     B[i].length == N);
11
12    int celeb;
13    // the celebrity in the group
14
15    //@ model pure static boolean isCelebrity(int i);
16
17    //@ requires k >= 2;
18    public celebrity2(int k) {
19        N = k;
20        B = new boolean[N][N];
21        //@ set B.owner = this;
22    }
23
24    //@ requires 0 <= p && p < N;
25    //@ requires 0 <= q && q < N;
26    //@ requires p != q;
27    //@ ensures \result == true ==> isCelebrity(p) == false;
28    //@ ensures \result == false ==> isCelebrity(q) == false;
29    public boolean knows(int p, int q) {
30        //@ assume B[p][q] == true ==> isCelebrity(p) == false;
31        //@ assume B[p][q] == false ==> isCelebrity(q) == false;
32        return B[p][q];
33    }
```

```
34
35    //@ requires (\exists int c; 0 <= c && c <= N - 1;
                    isCelebrity(c));
36    //@ ensures isCelebrity(celeb);
37    public void findCeleb() {
38        int a = 0;
39        int b = N - 1;
40
41        //@ loop_invariant (\exists int i; a <= i && i <= b;
                               isCelebrity(i));
42        while (a != b) {
43            if (knows(a, b))
44                a++;
45            else
46                b--;
47        }
48
49        celeb = a;
50
51        // @ assert false;
52    }
53 }
```

The program passed the checking of ESC/Java2. However, PVS can not prove the generated VC for the method *findCeleb* with the tactics *grind* automatically. This gives us an impression that Simplify is more powerful in automatic reasoning than PVS. We guess it is because the logic of ESC/Java2 is highly tuned for Simplify. So, Simplify can fully utilize the character of the logic in its proving, but PVS has no such kind of privilege.

As a workaround, we found a way to prove the method *findCeleb*. The basic idea is that we replace line 43 to 46 by an assume clause as follows:

```
//@ assume (\exists int i; a <= i && i <= b; isCelebrity(i))
```

Then we can prove the generated VC with the tactics *grind*. We can also prove the loop invariant holds for line 43 to 46 with the tactics *grind* after we put them in a separate method and transform the loop invariant as the precondition and postcondition of the method.

7.4 The Third Attempt to the Celebrity Problem

7.4.1 A New Approach to Verify the Celebrity Program

Inspired by the work of Hoare [4], we found a new approach to prove the celebrity program. The main ideas of the approach are as follows:

- Define a specification variable with the abstract data type *set*.
- Implement some basic operation units and use the abstract *set* operations to specify them.
- Prove the correctness of data representations, i.e., that the concrete implementations of the basic operation units indeed represent the abstract *set* operations.

- Prove the correctness of *findCeleb* on the abstract level, i.e., reason about the behavior of *findCeleb* with the abstract *set* operations.

Using abstract data to specify and verify programs has two advantages: firstly, it should be easier to reason about the behavior of programs on the abstract level than on the concrete implementation level; secondly, we do not need to rewrite the specification if the concrete implementation is changed later on.

Since ESC/Java2 does not support the data type *set*, we have to introduce a data type *pvs_set* with native specifications. The *pvs_set* is mapped into the PVS type *set* in the generated VC. We also need to change the previous celebrity program to facilitate our experiments. The modified solution for the celebrity problem is described as follows:

```
create a set Q which initially includes all persons;

choose and remove two different elements a and b from Q;

while (Q is not empty) {
  if (a knows b) {
     choose and remove another element c from Q;
     a = c;
  }
  else {
     choose and remove another element c from Q;
     b = c;
  }
}

if (a knows b)
   b is the celebrity;
else
   a is the celebrity;
```

7.4.2 Specifying the Celebrity Program with Set Operations

Following the ideas in the last section, we rewrote the celebrity program and specified the program with the abstract set operations as follows:

```
1 /*@ public model pure class pvs_set {
2   @   public model pure static boolean member(pvs_set s, int e);
3   @   public model pure static pvs_set remove(pvs_set s, int e);
4   @   public model pure static boolean empty(pvs_set s);
5   @ }
6   @*/
7
8 class celebrity3 {
9     int N;
10     //@ invariant N >= 2;
11
12     boolean[][] B;
13
14     int celeb;
15
16     //@ private ghost int c;
```

```
17
18      int lowerBound;
19      int upperBound;
20
21      //@ private model pvs_set Q;
22      //@ private model pure static pvs_set toSet(int n, int m);
23      //@ represents Q <- toSet(lowerBound+1, upperBound-1);
24
25      //@ requires k >= 2;
26      public celebrity3(int k) {
27          N = k;
28
29          B = new boolean[N][N];
30
31          lowerBound = -1;
32          upperBound = N;
33      }
34
35      //@ ensures (\result == true)  ==> (c != p);
36      //@ ensures (\result == false) ==> (c != q);
37      public boolean knows(int p, int q) {
38          return B[p][q];
39      }
40
41      //@ ensures Q == pvs_set.remove(\old(Q), lowerBound);
42      public void chooseLower() {
43          lowerBound++;
44      }
45
46      //@ ensures Q == pvs_set.remove(\old(Q), upperBound);
47      public void chooseUpper() {
48          upperBound--;
49      }
50
51      //@ ensures (\result == true)
                    ==> (pvs_set.empty(Q) == true);
52      //@ ensures (\result == false)
                    ==> (pvs_set.empty(Q) == false);
53      public boolean empty() {
54          return ((upperBound - lowerBound) < 2);
55      }
56
57      //@ requires pvs_set.member(Q, c);
58      //@ ensures celeb == c;
59      public void findCeleb() {
60          chooseLower();
61          chooseUpper();
62          //@ assert c == lowerBound || c == upperBound ||
                          pvs_set.member(Q, c);
63
64          //@ loop_invariant c == lowerBound ||
                c == upperBound || pvs_set.member(Q, c);
65          while (!empty()) {
66              if (knows(lowerBound, upperBound))
```

```
67                  chooseLower();
68              else
69                  chooseUpper();
70          }
71
72          //@ assert c == lowerBound || c == upperBound;
73
74          if (knows(lowerBound, upperBound)) {
75              celeb = upperBound;
76          }
77          else {
78              celeb = lowerBound;
79          }
80      }
81  }
```

The ghost variable c is used to indicate the celebrity in the group. The method *findCeleb* finds the celebrity and stores it in the variable *celeb*. From line 1 to 6, we introduce a native type *pvs_set* and some related native methods: *member*, *remove* and *empty*. In the generated VC, we map *pvs_set* and the native methods into the PVS type *set* and corresponding set operations as follows:

```
pvs_set: TYPE+ = set

pvs_set_member_2_6(p0: JavaType, p1: pvs_set, p2: IntegralNumber):
    Boolean = member(p2, p1)
pvs_set_remove_3_6(p0: JavaType, p1: pvs_set, p2: IntegralNumber):
    pvs_set = remove(p2, p1)
pvs_set_empty_4_6(p0: JavaType, p1: pvs_set):
    Boolean = (p1 = emptyset)
```

where the parameter $p0$ is generated by ESC/Java2 to indicate the state of the program. Since our native methods are pure, $p0$ is not useful anymore.

Line 21 introduces an abstract set Q into the specification of the celebrity program. Line 22 and 23 defines the representation function *toSet* to Q. The function *toSet* is defined in the semantic prelude as follows:

```
toSet(n: IntegralNumber, m: IntegeralNumber):
      set = {k: IntegralNumber | n <= k AND k <= m}
```

7.4.3 Checking the Method knows and Array Operations

Unlike in the first and second attempts, we separate our concerns in the third attempt. Firstly, we check the correctness of the method *knows* and also absence of array errors in the implementation, e.g. array null dereferences, array bounds errors, etc. These things are directly related with the concrete implementation, and have no direct relations with the abstract *set* operations. So, we just used ESC/Java2 to check them.

For the method *knows*, its postconditions are defined as follows:

```
//@ ensures (\result == true)  ==> (c != p);
//@ ensures (\result == false) ==> (c != q);
```

In order to serve the checking, we delete the specifications with *set* which are not supported by ESC/Java2, and add necessary preconditions for *knows*. These preconditions specify the valid parameters of *knows* and the properties of the celebrity *c*.

```
1 class celebrity31 {
2     int N;
3     //@ invariant N >= 2;
4
5     boolean[][] B;
6     //@ invariant B != null;
7     //@ invariant B.owner == this;
8     //@ invariant B.length == N;
9     //@ invariant (\forall int i; 0 <= i && i < B.length;
                     B[i] != null);
10    //@ invariant (\forall int i; 0 <= i && i < B.length;
                     B[i].length == N);
11
12     int celeb;
13
14     //@ private ghost int c;
15
16     int lowerBound;
17     int upperBound;
18
19     //@ requires k >= 2;
20     public celebrity31(int k) {
21         N = k;
22
23         B = new boolean[N][N];
24         //@ set B.owner = this;
25
26         lowerBound = -1;
27         upperBound = N;
28     }
29
30     //@ requires 0 <= p && p < N;
31     //@ requires 0 <= q && q < N;
32     //@ requires p != q;
33     //@ requires 0 <= c && c < N;
34     //@ requires (\forall int i; 0 <= i && i < N && i != c;
                     B[i][c] == true);
35     //@ requires (\forall int i; 0 <= i && i < N && i != c;
                     B[c][i] == false);
36     //@ ensures (\result == true)  ==> (c != p);
37     //@ ensures (\result == false) ==> (c != q);
38     public boolean knows(int p, int q) {
39         return B[p][q];
40     }
41
42     //@ ensures lowerBound == \old(lowerBound) + 1;
43     public void chooseLower() {
44         lowerBound++;
45     }
```

```
46
47      //@ ensures upperBound == \old(upperBound) - 1;
48      public void chooseUpper() {
49          upperBound--;
50      }
51
52      public boolean empty() {
53          return ((upperBound - lowerBound) < 2);
54      }
55
56      //@ requires 0 <= c && c < N;
57      //@ requires (\forall int i; 0 <= i && i < N && i != c;
                      B[i][c] == true);
58      //@ requires (\forall int i; 0 <= i && i < N && i != c;
                      B[c][i] == false);
59      //@ requires lowerBound == -1;
60      //@ requires upperBound == N;
61      public void findCeleb() {
62          chooseLower();
63          chooseUpper();
64
65          while (!empty()) {
66              if (knows(lowerBound, upperBound))
67                  chooseLower();
68              else
69                  chooseUpper();
70          }
71
72          if (knows(lowerBound, upperBound)) {
73              celeb = upperBound;
74          }
75          else {
76              celeb = lowerBound;
77          }
78      }
79  }
```

The above program passed the checking of ESC/Java2. This means that the postconditions of *knows* hold and there are no array errors in the scope of ESC/Java2's checking. After this checking, we can focus on the correctness of data representations and the method *findCelb* which are the main goals in the experiment.

7.4.4 Some Axioms Used in the Proving Procedure

We defined some axioms in the semantic prelude in order to facilitate our proof. Defining axioms is dangerous in PVS since it is easy to introduce inconsistency. In order to avoid inconsistency, we tried to prove these axioms before using them (by changing axioms to lemmas).

toSet_axiom_1

```
toSet_axiom_1: AXIOM
  FORALL(n: IntegralNumber, m: IntegralNumber):
```

```
    toSet(n+1, m) = remove(n, toSet(n, m))
```

PVS can not prove the axiom *toSet_axiom_1* with the tactics *grind*. The essence of the axiom is about the equality of *set*. Although we think there should be a way to prove the axiom directly, we did not find the way in our experiments. As a workaround, we proved the following two lemmas with *grind*:

```
toSet_axiom_1_prove_1: THEOREM
  FORALL(i: IntegralNumber n: IntegralNumber, m: IntegralNumber):
    member(i, toSet(n+1, m)) IMPLIES
      member(i, remove(n, toSet(n, m)))

toSet_axiom_1_prove_2: THEOREM
  FORALL(i: IntegralNumber n: IntegralNumber, m: IntegralNumber):
    member(i, remove(n, toSet(n, m))) IMPLIES
      member(i, toSet(n+1, m))
```

The correctness of the two lemmas should be sufficient to guarantee the correctness of *toSet_axiom_1*.

toSet_axiom_2

Similar to *toSet_axiom_1*, we also defined and proved the axiom *toSet_axiom_2* as follows:

```
toSet_axiom_2: AXIOM
  FORALL(n: IntegralNumber, m: IntegralNumber):
    toSet(n, m-1) = remove(m, toSet(n, m))
```

toSet_axiom_3

```
toSet_axiom_3: AXIOM
  FORALL(n: IntegralNumber, m: IntegralNumber):
    (n > m) <=> (toSet(n, m) = emptyset)
```

According to the definition of *emptyset* and *empty?*, the above axiom is equivalent to the following axiom:

```
toSet_axiom_3_prove: AXIOM
  FORALL(n: IntegralNumber, m: IntegralNumber):
    (n > m) <=> (empty?(toSet(n, m)))
```

We proved *toSet_axiom_3_prove* with the tactics *grind*.

get_and_set_axiom

```
get_and_set_axiom: AXIOM
  FORALL(f: Field, r: Reference, v: Number):
    get(set(f, r, v), r) = v
```

Actually, the axiom *get_and_set_axiom* is an instantiation of the existing axiom *get_and_set_definition* in the semantic prelude. We provided it in order to avoid tedious typing work in the proving procedure.

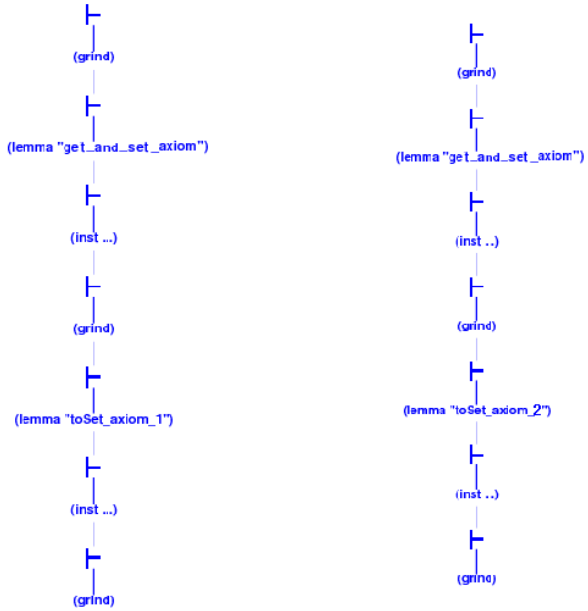

Figure 7.1: Proof trees for the methods *chooseLower* and *chooseUpper*

7.4.5 Proving the Correctness of Data Representations

Before proving the correctness of the method *findCeleb*, we need to prove its basic composition units: *chooseLower*, *chooseUpper*, *empty*. Since we use *set* to specify the behavior of these methods, the correctness of these methods means they indeed implement the abstract *set* operations.

Prove *chooseLower*
For the method *chooseLower*, we need to prove its implementation represents the following abstract operation:

```
//@ ensures Q == pvs_set.remove(\old(Q), lowerBound);
```

The generated VC for *chooseLower* is about 260 lines. With the axiom *toSet_axiom_1*, we proved the VC for *chooseLower*. The left part of figure 7.1 shows the proof tree for *chooseLower*.

Prove *chooseUpper*
For the method *chooseUpper*, we need to prove its implementation represents the following abstract operation:

```
//@ ensures Q == pvs_set.remove(\old(Q), lowerUpper);
```

The generated VC for *chooseLower* is about 260 lines. With the axiom *toSet_axiom_2*, we proved the VC for *chooseUpper*. The right part of figure 7.1 shows the proof tree for *chooseUpper*.

Prove *empty*
The postconditions for *empty* are defined as follows:

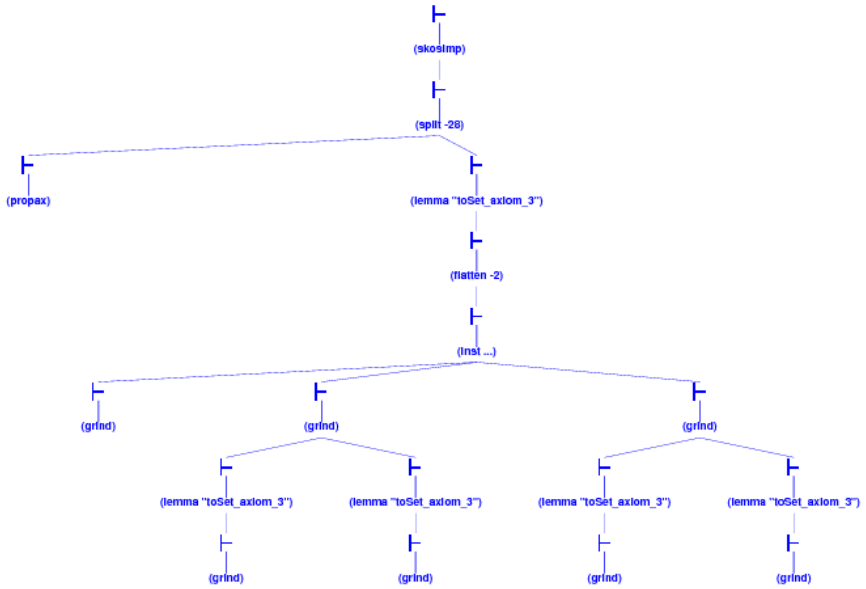

Figure 7.2: Proof tree for the method *empty*

```
//@ ensures (\result == true)  ==> (pvs_set.empty(Q) == true);
//@ ensures (\result == false) ==> (pvs_set.empty(Q) == false);
```

Figure 7.2 shows the proof tree for *empty*. The axiom *toSet_axiom_3* is used in the proof.

7.4.6 Proving the Correctness of the Method findCeleb

After the proof of *knows*, *chooseLower*, *chooseUpper* and *empty*, the correctness of *findCeleb* can be proved on the abstract level since ESC/Java2 performs modular checking, i.e., only preconditions and postconditions of the methods *chooseLower*, *chooseUpper*, *empty* and *knows* will be considered when checking *findCeleb*.

However, the generated VC for *findCeleb* is too large (over 4600 lines). It is hard to understand and prove such large VC in PVS. As a workaround, we divided *findCeleb* into four parts, and proved the four parts separately. Although we can not guarantee that the correctness of *findCeleb* is equivalent to the correctness of the four parts, the proof demonstrates the feasibility of proving a program on the abstract level which is the main goal of the experiment.

```
1  //@ requires pvs_set.member(Q, c);
2  //@ ensures celeb == c;
3  public void findCeleb() {
4     chooseLower();
5     chooseUpper();
```

```
6     //@ loop_invariant c == lowerBound ||
                          c == upperBound || pvs_set.member(Q, c);
7     while (!empty()) {
8        if (knows(lowerBound, upperBound))
9            chooseLower();
10       else
11           chooseUpper();
12    }
13
14    if (knows(lowerBound, upperBound)) {
15        celeb = upperBound;
16    }
17    else {
18        celeb = lowerBound;
19    }
20 }
```

The first part includes line 4 and 5; The second part includes the loop from line 7 to 12. In the second part, we replace the loop body by an *assume* clause; The third part is the loop body from line 8 to 11; Line 14 to 19 are in the fourth part.

Prove the first part of *findCeleb*
The first part of *findCeleb* is as follows:

```
1 //@ requires pvs_set.member(Q, c);
2 //@ ensures c == lowerBound || c == upperBound ||
                  pvs_set.member(Q, c);
3 public void findCeleb1() {
4     chooseLower();
5     chooseUpper();
6 }
```

We proved the generated VC with the tactics *grind* in PVS.

Prove the second part of *findCeleb*
The second part of *fidCeleb* is as follows:

```
1 //@ requires c == lowerBound || c == upperBound ||
                   pvs_set.member(Q, c);
2 //@ ensures c == lowerBound || c == upperBound
3 public void findCeleb2() {
4     //@ loop_invariant c == lowerBound || c == upperBound ||
                              pvs_set.member(Q, c);
5     while (!empty()) {
6         //@ assume c == lowerBound || c == upperBound ||
                         pvs_set.member(Q, c);
7     }
8 }
```

findCeleb2 comes from the loop statement of *findCeleb* with the replacement of the loop body by an assume clause. The actual loop body is proved in the third part. We do this in order to generate a smaller VC and simplify the proof procedure.

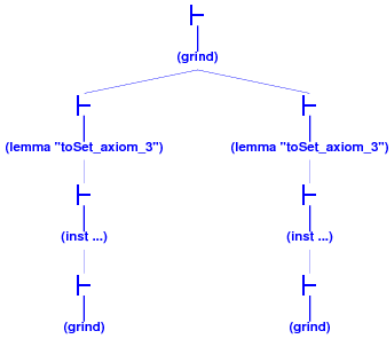

Figure 7.3: Proof Tree for the method *findceleb2*

Figure 7.4: Proof Tree for the method *findceleb3*

The figure 7.3 shows the proof tree for *findCeleb2*.

Prove the third part of *findCeleb*

The third part of *findCeleb* contains the actual loop body.

```
1  //@ requires c == lowerBound || c == upperBound ||
                    pvs_set.member(Q, c);
2  //@ ensures c == lowerBound || c == upperBound ||
                   pvs_set.member(Q, c);
3  public void findCeleb3() {
4     if (knows(lowerBound, upperBound))
5         chooseLower();
6     else
7         chooseUpper();
8  }
```

Figure 7.4 shows the proof tree for *findCeleb3*.

Prove the fourth part of *findCeleb*

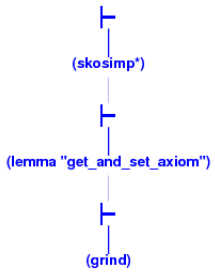

Figure 7.5: Proof Tree for the method *findceleb4*

The fourth part of *findCeleb* is as follows:

```
1  //@ requires c == lowerBound || c == upperBound;
2  //@ ensures celeb == c;
3  public void findCeleb4() {
4      if (knows(lowerBound, upperBound)) {
5          celeb = upperBound;
6      }
7      else {
8          celeb = lowerBound;
9     }
10 }
```

Figure 7.5 shows the proof tree for *findCeleb4*.

7.5 Problems and Solutions

From the theoretical point of view, proving programs on the abstract level with native specification works. However, there are many problems in the engineering aspect. The most serious problem is how to scale the approach to handle non-trivial programs. As mentioned, the generated VC is over 4600 lines for the method *findCeleb* in our experiments. It is hard to prove such a large VC in PVS, even for a PVS expert.

Another serious problem concerns the type assignment in our VC translator. For example, in the experiments we found the following GC:

```
1 {   ASSUME RES;
2     tmp0!old!a:10.13 = a:7.12;
3     a:7.12 = integralAdd(tmp0!old!a:10.13, 1)
4   []
5     ASSUME booleNot(RES);
6     tmp0!old!a:12.13 = a:7.12;
7     a:7.12 = integralAdd(tmp0!old!a:12.13, 1)
8 };
9 RES = a:7.12;
```

The *RES* in line 1 and 5 is different from the *RES* in line 9. Actually, they have different types: the *RES* in line 1 and 5 has the type *Boolean*, and the *RES* in line 9 must have the type *IntegralNumber*. This GC does not give any problem for ESC/Java2 since ESC/Java2 is based on an unsorted logic where the type information is encoded into separate sub-formulas. However, it is a problem for our VC translator since our VC translator can not give a proper type to *RES* since our VC translator has no way to distinguish the two *RES*s.

The third problem is about the semantic prelude. In the experiments, we noticed that it is necessary to add a type *IntegralField* in the semantic prelude. The motivation to add *IntegralField* is that we found the following formula exists in a generated VC:

```
(0 < get(f, this)) AND (get(f, this) < 2)
```

where f is a field of the class being checked. If the type of f is *IntegralField*, PVS can judge that the value of f must be 1. If the type of f is *NumberField*, PVS has no way to get the result since f can be a real number. Because of this reason, we added the following definition in the semantic prelude:

```
IntegralField : TYPE+ FROM NumberField
```

The fourth problem is caused by the type system in the semantic prelude. In the proof procedure, we found many unproved TCCs (Type Checking Conditions) in PVS. All unproved TCCs are of two forms:

```
FORALL (f: IntegralField, brokenObj: Reference):
       IntegralNumber_pred(get(f, brokenObj) + 1);

FORALL (s: pvs_set):
       Boolean_pred(s = emptyset[IntegralNumber]);
```

For the first unproved TCC, the type of "get(f, brokenObj)" is *IntegralNumber*. PVS can not prove that the type of "get(f, brokenObj)+1" is still *IntegralNumber* since *IntegralNumber* is defined as a subtype of *int* in the semantic prelude.

For the second unproved TCC, the type of "s = emptyset[IntegralNumber]" is *bool*. PVS can not prove the TCC since *Boolean* is defined as a subtype of *bool* in the semantic prelude.

If we modify the definition of *IntegralNumber* and *Boolean* as follows, all unproved TCCs disappear after the modification.

```
old:
IntegralNumber : TYPE+ FROM int
Boolean:         TYPE+ FROM bool

new:
IntegralNumber : TYPE+ = int
Boolean:         TYPE+ = bool
```

7.6 Conclusions

The experiments with the celebrity problem demonstrate the feasibility of proving Java programs on the abstract level with native specifications. The approach has potential advantages since the specification on the abstract level is usually clearer and simpler than the specification on the concrete implementation level.

Chapter 8

Conclusions

The main purpose of the thesis project is to develop an approach to verify Java programs by integrating ESC/Java2 and PVS. As we state in Chapter 1, there are three specific goals in the project:

- updating the initial VC translator to the latest semantic prelude and extending it to multi-line methods.
- extending the initial VC translator to some advanced features: quantifier operators, method call, array, and native specifications.
- investigating the advantages of the proposed approach, i.e., answering the question: can we check some programs which can not be handled by the original approach?

We would like to say we achieved the goals in the thesis project. The VC translator was updated and extended in the project, and the proposed approach was demonstrated to prove some programs which can not be handled by ESC/Java2, e.g. the fibonacci program and the celebrity program.

8.1 Achievements

We summarize our achievements from the educational and research point of views separately.

Achievements from the educational point of view:

- We improved our understanding about ESC/Java2. For example, we are more clear about how the VC is generated and processed in ESC/Java2.
- We gained experience in using PVS. Since PVS is a quite complicated tool, we are still beginners. However the experience in the project already gave us some ideas about how to use and master PVS. For example, we are clear about the usage of some tactics like *skosimp*, *induct*, etc. We also noticed that the prelude of PVS provides good material to study its specification language.

- We got better understanding of program specification and verification techniques, e.g. Weakest Precondition Calculus, Guarded Command Language, Hoare logic, JML, etc. This understanding is valuable since it is achieved by studying theory and investigating real software.

Achievements from the research point of view:

- We updated and extended the initial VC translator for PVS to quantifier operators, strong pure method call and array operations. Although our VC translator is still far from dealing with real industry applications, it is already able to handle some small programs.
- We developed a methodology to support native specifications in JML. We believe supporting native specifications is one of big advantages of the proposed approach.
- We developed a methodology to reason about Java programs on the abstract level with native specifications. The experiments for the celebrity problem demonstrate the feasibility and advantages of the methodology.

8.2 Problems and Solutions

Besides our achievements, there are still many problems left in both theoretical and practical aspects.

- The unsorted logic of ESC/Java2 caused many difficulties and problems in the VC translator.
  Description: The task of the VC translator is translating the original VC for Simplify into the VC for PVS. The original VC is based on an unsorted logic while the VC for PVS is based on the sorted logic of PVS. The discrepancy between the unsorted and sorted logic caused many difficulties and problems in the development of the tool. For example, in order to give variables with right types, we have to access the abstract syntax tree to get all type information in advance. The procedure is not very robust. Furthermore, in the VC translator we have to build a new sorted abstract syntax tree since the original abstract syntax tree is unsorted. This increases the complexity of the VC translator.

  Solution: The successor of ESC/Java2 should be developed on a sorted logic. If ESC/Java2 is built on a sorted logic from its front-end, the work of the VC translator would be much easier. Actually, Joseph Kiniry and his colleagues in Dublin are working on this topic right now.

- The generated VC of ESC/Java2 is usually too large to be understood by a human being.
  Description: We noticed that the generated VC of ESC/Java2 is quite large even for a simple program like the celebrity program. If we want to prove a VC in PVS, we usually need to understand the structure of the VC. However, it is hard to understand a formula of over a thousand lines like we got in the experiments with the celebrity problem.

Solution: There are two possible ways to deal with the problem. Firstly, maybe we can introduce a postprocessor to analyze the generated VC and compress it. Secondly, ESC/Java2 generates one VC for each method being checked. The VC integrates many small proof obligations which are used to check different properties like aliasing errors, invariants, loop invariants, etc. Actually, as we know, similar tools to ESC/Java2 like Jack generate several proof obligations to each method being checked. Since each proof obligation focuses on only one property, such proof obligations would be much easier to understand than the VC of ESC/Java2. So, we wonder whether we can provide an option to ESC/Java2 to split a big VC into many small proof obligations according to the semantics of programs.

- Using model classes and model methods to support native specifications is not perfect.
  Description: As we state in Chapter 6, using model classes and model methods to support native specifications is a temporary trick in our project. Since ESC/Java2 can not distinguish native methods from ordinary model methods, we have no way to map these native methods into the methods of PVS automatically. There is a similar problem for native types.

  Solution: We suggest to introduce a new keyword *native* to indicate native specifications in ESC/Java2 as [21] did in Jack. With the new keyword, ESC/Java2 can distinguish native methods from other model methods, and the VC translator can map these native methods into the methods of PVS automatically.

- The soundness of the semantic prelude is not investigated.
  Description: Since ESC/Java2 is not sound, the proposed approach is unsound as well. Being an important part of the proposed approach, the soundness of the semantic prelude is very important since a small problem in the semantic prelude would cause all of our proofs in PVS to be invalid.

  Solution: We should seriously investigate the soundness and completeness of the semantic prelude.

- How to identify the broken specifications or suspect bugs in source code.
  Description: If ESC/Java2 fails to check a program, Simplify can provide some information to identify the accurate positions of broken specifications or suspect bugs in source code. This is very useful for users. But in PVS, we did not find a way to support such feature. If a VC can not be proved in PVS, we only know maybe there is an error in the program. We are not clear about what and where the error is.

  Solution: We are still not very clear about how to solve the problem. The reason why Simplify can identify the broken specifications or suspect bugs is because Simplify utilizes some special labeled expressions in the VC for Simplify. In our current VC translator, we ignored these special labels when generating the VC for PVS. Further investigations are needed to solve the problem.

8.3 Future Work

There is plenty of work to do in the future. First of all, we need to consider how to solve the problems given in the previous section. However, we think it is better to tackle these problems after the successor of ESC/Java2 is developed on the sorted logic from its front-end.

Secondly, we need to develop the VC translator to handle allocating new arrays in programs. In the thesis project, we only proposed a solution. Further investigation and implementation is needed.

Finally, it is interesting to support weak pure method calls in ESC/Java2. In the thesis project, we only considered strong pure method calls which require that the model methods are side effect free and no new objects are allocated in the methods. In [1], the authors proposed an approach to support weak pure method calls which allow allocating and initializing new objects in the model methods. We think the approach could possibly be introduced in ESC/Java2.

Appendix A

An Example of the VC for Simplify

The following VC is generated by ESC/Java2 for the method *add* of the program *test* in the section 3.3.

```
(EXPLIES
  (LBLNEG |vc.test.add.2.4|
    (IMPLIES
      (AND
        (EQ |elems@pre| elems)
        (EQ elems
          (asElems elems)
          )
        (<
          (eClosedTime elems)
          alloc)
        (EQ LS
          (asLockSet LS)
          )
        (EQ |alloc@pre| alloc)
        (EQ |state@pre| state)
       )
      (NOT
        (AND
          (EQ |@true|
            (is this T_test)
            )
          (EQ |@true|
            (isAllocated this alloc)
            )
          (NEQ this null)
          (EQ |@true|
            (is |a:3.23| T_int)
            )
          (EQ |@true|
            (is |b:3.30| T_int)
            )
          (LBLNEG |Pre|
            (AND
              (EQ |@true|
                (is this T_test)
                )
              (NEQ this null)
              )
            )
          (FORALL
            (brokenObj)
            (EQ
              (|java.lang.Throwable#_stackTrace| state brokenObj)
              (|getStackTrace##state| state brokenObj)
              )
            )
          (FORALL
```

```
  (|brokenObj<1>|)
  (EQ
    (|java.lang.Throwable#_stackTrace| state |brokenObj<1>|)
    (|getStackTrace##state| state |brokenObj<1>|)
    )
  )
(EQ RES
  (+ |a:3.23| |b:3.30|)
  )
(LBLPOS |trace.Return^0,4.8|
  (EQ |@true| |@true|)
  )
(FORALL
  (brokenObj)
  (EQ
    (|java.lang.Throwable#_stackTrace| state brokenObj)
    (|getStackTrace##state| state brokenObj)
    )
  )
(FORALL
  (|brokenObj<1>|)
  (EQ
    (|java.lang.Throwable#_stackTrace| state |brokenObj<1>|)
    (|getStackTrace##state| state |brokenObj<1>|)
    )
  )
(OR
  (NOT
    (LBLNEG |Exception@5.4|
      (EQ |ecReturn| |ecReturn|)
      )
    )
  (AND
    (LBLNEG |Exception@5.4|
      (EQ |ecReturn| |ecReturn|)
      )
    (OR
      (NOT
        (LBLNEG |Post:2.8@5.4|
          (IMPLIES
            (AND
              (EQ |ecReturn| |ecReturn|)
              (EQ |@true|
                (is this T_test)
                )
              (NEQ this null)
              )
            (EQ RES
              (+ |a:3.23| |b:3.30|)
              )
            )
          )
        )
      (NOT
        (LBLNEG |Post:2.4@5.4|
          (IMPLIES
            (AND
              (EQ |ecReturn| |ecThrow|)
              (<:
                (typeof XRES)
                |T_java.lang.Exception|)
              )
            (NOT
              (AND
                (EQ |@true|
                  (is this T_test)
                  )
                (NEQ this null)
                )
              )
            )
          )
        )
      )
```

```
            )
          )
        )
      )
    )
  )
(AND
  (DISTINCT |ecReturn| |ecThrow|)
  )
)
```

Appendix B

An Example of the VC for PVS

The following VC is generated by the VC translator for the method *add* of the program *test* in the section 3.3.

```
vc_test_add_2_4: THEORY
BEGIN

IMPORTING escjava2_logic
test: TYPE+ FROM JavaType

vc_test_add_2_4: THEOREM
Forall(
b_3_30: IntegralNumber,
brokenObj: Reference,
RES: IntegralNumber,
a_3_23: IntegralNumber,
brokenObj_1_: Reference):
(
  (
    (elems_pre = elems)
    AND True
    AND
    (
      (eClosedTime(elems))
       < alloc)
    AND True
    AND
    (alloc_pre = alloc)
    AND
    (state_pre = state)
  )
   IMPLIES
  (NOT
    (TRUE
      AND
      (isAllocated(this, alloc))
      AND
      (this /= NULL)
      AND TRUE
      AND TRUE
      AND
      (TRUE
         AND
        (this /= NULL)
      )
      AND
      (FORALL (brokenObj: Reference):
        (java_lang_Throwable_stackTrace(state, brokenObj)
           = getStackTrace_state(state, brokenObj)
        )
      )
      AND
```

```
      (FORALL (brokenObj_1_: Reference):
        (java_lang_Throwable_stackTrace(state, brokenObj_1_)
           = getStackTrace_state(state, brokenObj_1_)
        )
      )
      AND
      (RES =
        (a_3_23 + b_3_30)
      )
      AND TRUE
      AND
      (FORALL (brokenObj: Reference):
        (java_lang_Throwable_stackTrace(state, brokenObj)
           = getStackTrace_state(state, brokenObj)
        )
      )
      AND
      (FORALL (brokenObj_1_: Reference):
        (java_lang_Throwable_stackTrace(state, brokenObj_1_)
           = getStackTrace_state(state, brokenObj_1_)
        )
      )
      AND
      (
        (NOT
          (ecReturn = ecReturn)
        )
         OR
        (
          (ecReturn = ecReturn)
           AND
          (
            (NOT
              (
                (
                  (ecReturn = ecReturn)
                  AND TRUE
                  AND
                  (this /= NULL)
                )
                 IMPLIES
                (RES =
                  (a_3_23 + b_3_30)
                )
              )
            )
             OR
            (NOT
              (
                (
                  (ecReturn = ecThrow)
                   AND
                  (
                    (typeOf(XRES))
                     <= T_java_lang_Exception
                  )
                )
                 IMPLIES
                (NOT
                  (TRUE
                     AND
                    (this /= NULL)
                  )
                )
              )
            )
          )
        )
      )
    )
  )
)

END vc_test_add_2_4
```

Appendix C

The Semantic Prelude

The following file (*escjava2.pvs*) is the semantic prelude used in our project. *escjava2_predefined*, *escjava2_test_sum*, *escjava2_test_fibo*, *escjava2_test_set_array* and *escjava2_test_celebrity* are theories which are added for our experiments in the thesis project.

```
% The ESC/Java2 sorted logic.
%
% This is the sorted logic used by ESC/Java2.
%
% $Id: escjava2.pvs,v 1.1 2005/10/15 11:21:54 jkiniry Exp $
%
% It was written by Joe Kiniry, Cesare Tinelli, Patrice Chalin, and
% Clement Hurlin in 2005.
%
% This is the canonical sorted logic which is to be translated
% automatically into SMT-LIB by either an external script or by a new
% extension to PVS (to be written by folks at UCD).

escjava2_types : THEORY
  BEGIN
    %S : TYPE+

    % Java base types.
    Boolean : TYPE+ FROM bool
    % Eventually we'll refine IntegralNumber to bounded, modular ints.
    % All of these numeric types inherit from the PVS type "number".
    IntegralNumber : TYPE+ FROM int
    % The same thing holds true for this bad-boy.
    FloatingPointNumber : TYPE+ FROM real
    BigIntNumber : TYPE+ FROM int
    RealNumber : TYPE+ FROM real
    % We would love to define these as supertypes of these component
    % types but we cannot because we are relying upon PVS prelude
    % types for their semantics.  Number is the supertype of all
    % Java integral types and JML numeric types.
    Number : TYPE+ = number
    BasicValue : TYPE+ = [Boolean + Number]
    Reference : TYPE+
    ArrayReference : TYPE+ FROM Reference
    JMLNumber : TYPE+ = [Number + BigIntNumber + RealNumber]

    % Sorts representing the various kinds of object fields in Java.
    Field : TYPE+
    BooleanField, NumberField, ReferenceField : TYPE+ FROM Field

    % These two PVS types represent the actual Java types.
    JavaType : TYPE+
    PrimitiveType, ReferenceType : TYPE+ FROM JavaType
    ArrayType : TYPE+ FROM ReferenceType
    JavaTypes_are_disjoint : AXIOM
      FORALL(p : PrimitiveType, r : ReferenceType): p /= r
```

```
    Time : TYPE FROM int
    Lock : TYPE+
    Path : TYPE+

    % @review JoeK, CesareT: It is unclear if we need Object at
    % all; reconsider later.
    Object : TYPE+

  END escjava2_types

map_theory[Map : TYPE+, Index : TYPE+, Value : TYPE+] : THEORY
  BEGIN

    get : [ Map, Index -> Value ]
    set : [ Map, Index, Value -> Map ]

    get_and_set_definition : AXIOM
      FORALL(m : Map, i : Index, v : Value ) :
             get(set(m, i, v), i) = v

    set_only_changes_one_index : AXIOM
      FORALL(m : Map, i,j : Index, v : Value) :
             i /= j IMPLIES get(set(m, i, v), j) = get(m, j)

  END map_theory

escjava2_java_typesystem : THEORY
  BEGIN
    IMPORTING escjava2_types,
              orders[JavaType]
    t, u : ReferenceType

    % <: is <=
    % < is <
    <,<= : [ ReferenceType, ReferenceType -> bool ]

    % === ESCJ 8: Section 1.1

    % We will use '<=' in PVS for '<:' in the Simplify logic.  Thus,
    % <= is reflexive, transitive, and antisymmetric.

    comparison_is_a_strict_order : POSTULATE strict_order?(<)
    % Add an axiom to relate these two orders and define subtyping (<=) as a
    % partial order.
    subtype_definitions: AXIOM
      FORALL(t : ReferenceType, u : ReferenceType) :
             t <= u IFF t < u OR t = u

    % The base type in Java (java.lang.Object).

    T_java_lang_Object : ReferenceType

    % Primitive types are final.

    T_boolean, T_char, T_byte, T_short, T_int, T_long, T_float, T_double : PrimitiveType

    % primitive? is no longer necessarily because the existence of
    % PrimitiveType.

    % extends? is Java's "extends" and "implements" (direct subtype)
    extends? : [ ReferenceType, ReferenceType -> bool ]
    extends_is_irreflexive : POSTULATE irreflexive?(extends?)

    subtype_includes_extends : AXIOM
      FORALL(t : ReferenceType, u : ReferenceType) :
        extends?(t, u) IMPLIES t <= u

    subtype_is_a_relation_that_contains_extends : AXIOM
      FORALL(t : ReferenceType, u : ReferenceType) :
        t <= u AND t /= u IMPLIES
        EXISTS(v : ReferenceType) : extends?(t, v) AND v <= u

    % Note that this is a higher-order axiom that cannot be translated
```

```
    % to SMT-LIB.
    subtype_is_the_smallest_relation_that_contains_extends : AXIOM
      FORALL(r : [ ReferenceType, ReferenceType -> bool ],
             t : ReferenceType, u : ReferenceType) :
        t <= u AND t /= u IMPLIES r(t, u)

    primative_types_are_final : AXIOM
      FORALL (t : JavaType, p : PrimitiveType): t <= p IMPLIES t = p
    primative_types_have_no_proper_supertypes : AXIOM
      FORALL (p : PrimitiveType, t : JavaType): p <= t IMPLIES p = t

    java_lang_Object_is_Top : AXIOM
      FORALL (t : ReferenceType): t <= T_java_lang_Object

    % === ESCJ 8: Section 1.2

    typeOf(r : Reference) : ReferenceType
    NULL : Reference

    isa?(r : Reference, t : ReferenceType) : bool =
      ((r = NULL) OR typeOf(r) = t)

    % === ESCJ 8: Section 1.3

    T_java_lang_Cloneable : ReferenceType

    elemtype : [ArrayType -> JavaType]
    array_constructor : [JavaType -> ArrayType]

    arrays_are_cloneable : AXIOM
      FORALL (t : JavaType): array_constructor(t) <= T_java_lang_Cloneable
    elemtype_definition : AXIOM
      FORALL (t : JavaType): elemtype(array_constructor(t)) = t
    array_subtyping : AXIOM
      FORALL (t0 : ArrayType, t1 : JavaType): t0 <= array_constructor(t1) IFF
                                              elemtype(t0) <= t1

    % === ESCJ 8: Section 2.1

    % The static type predicate.

    % Again, we would prefer to wrap this up in a single is(), but
    % because of the PVS type hierarchy (discussed above) we have to
    % overload instead.
    is : [ Boolean, JavaType -> bool ]
    is : [ Number, JavaType -> bool ]
    is : [ Reference, JavaType -> bool ]

    % cast is dealt with the same way.
    cast : [ Boolean, PrimitiveType -> Boolean ]
    cast : [ Number, PrimitiveType -> Number ]
    cast : [ Reference, ReferenceType -> Reference ]

    redundant_cast_removal_boolean : AXIOM
      FORALL (x : Boolean, t : JavaType): is(x, t) IMPLIES cast(x, t) = x
    redundant_cast_removal_number : AXIOM
      FORALL (x : Number, t : JavaType): is(x, t) IMPLIES cast(x, t) = x
    redundant_cast_removal_reference : AXIOM
      FORALL (x : Reference, t : JavaType): is(x, t) IMPLIES cast(x, t) = x

    % === ESCJ 8: Section 2.2

    % Not in ESCJ8, but should be

    refEQ(x, y : Reference): bool = x = y

    refNE(x, y : Reference): bool = x /= y

  END escjava2_java_typesystem

escjava2_java_boolean_ops : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem
```

```
    % === ESCJ 8: Section 5.2

    boolAnd (a, b : Boolean): bool = a AND b
    boolEq (a, b : Boolean): bool = a IFF b
    boolImplies (a, b : Boolean): bool = a IMPLIES b
    boolNE (a, b : Boolean): bool = a /= b
    boolNot (a : Boolean): bool = NOT a
    boolOr (a, b : Boolean): bool = a OR b

    % === ESCJ 8: Section 5.3

    % Java's ternary 'conditional' operator (? :)

    termConditional (b : Boolean, x, y : Boolean): Boolean =
      IF b THEN x ELSE y ENDIF
    termConditional (b : Boolean, x, y : Number): Number =
      IF b THEN x ELSE y ENDIF
    termConditional (b : Boolean, x, y : Reference): Reference =
      IF b THEN x ELSE y ENDIF

  END escjava2_java_boolean_ops

escjava2_java_integral_types : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem

    % === ESCJ 8: Section 2.2.1

    % Axioms to express the size of the basic types.
    range_of_char : AXIOM
      FORALL (x : Number): is(x, T_char) IFF 0 <= x AND x <= 65535
    range_of_byte : AXIOM
      FORALL (x : Number): is(x, T_byte) IFF -128 <= x AND x <= 127
    range_of_short : AXIOM
      FORALL (x : Number): is(x, T_short) IFF -32768 <= x AND x <= 32767
    range_of_int : AXIOM
      FORALL (x : Number): is(x, T_int) IFF -2^31 <= x AND x <= 2^31-1
    range_of_long : AXIOM
      FORALL (x : Number): is(x, T_long) IFF -2^63 <= x AND x <= 2^63-1
    range_of_float : AXIOM
      FORALL (x : Number): is(x, T_float) IFF -(2-(2^-23))*(2^127) <= x
                           AND x <= (2-(2^-23))*(2^127)
    range_of_double : AXIOM
      FORALL (x : Number): is(x, T_double) IFF -(2-(2^-52))*(2^1023) <= x
                           AND x <= (2-(2^-52))*(2^1023)

  END escjava2_java_integral_types

escjava2_java_integral_ops : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem

    % === ESCJ 8: Section 5.1

    % Define using modulo_arithmetic theory.
    integralMod(x, y : IntegralNumber): {v: mod(y) | EXISTS (r: mod(y)): v = x * y + r}
    integralDiv(x, y : IntegralNumber): {r: mod(y) | EXISTS (v: mod(y)): v = x * y + r}

    integralMod_def : LEMMA
      FORALL (a, b : IntegralNumber): b /= 0 IMPLIES a = (a / b) * b + (integralMod(a, b))

    integralEQ(x,y : IntegralNumber) : IntegralNumber = (x = y)
    integralGE(x,y : IntegralNumber) : IntegralNumber = (x >= y)
    integralGT(x,y : IntegralNumber) : IntegralNumber = (x > y)
    integralLE(x,y : IntegralNumber) : IntegralNumber = (x <= y)
    integralLT(x,y : IntegralNumber) : IntegralNumber = (x < y)
    integralNE(x,y : IntegralNumber) : IntegralNumber = (x /= y)

    % === Axioms about properties of integral &, |, and /

    integralAnd, integralOr, integralXor, intShiftL, longShiftL :
      [ IntegralNumber, IntegralNumber -> IntegralNumber ]

    integralAnd_definition1 : LEMMA
```

```
      FORALL (x, y : IntegralNumber): (0 <= x OR 0 <= y) IMPLIES
                        0 <= integralAnd(x, y)
    integralAnd_definition2 : LEMMA
      FORALL (x, y : IntegralNumber): 0 <= x IMPLIES
                        integralAnd(x, y) <= x
    integralAnd_definition3 : LEMMA
      FORALL (x, y : IntegralNumber): 0 <= y IMPLIES
                        integralAnd(x, y) <= y
    integralOr_definition : LEMMA
      FORALL (x, y : IntegralNumber): (0 <= x AND 0 <= y) IMPLIES
                        x <= integralOr(x, y) AND y <= integralOr(x, y)
    integralDiv_definition : LEMMA
      FORALL (x, y : IntegralNumber): (0 <= x AND 0 <= y) IMPLIES
                        0 <= integralDiv(x, y) AND integralDiv(x, y) <= x
    integralXor_definition : LEMMA
      FORALL (x, y : IntegralNumber): (0 <= x AND 0 <= y) IMPLIES
                        0 <= integralXor(x, y)
    intShiftL_definition : LEMMA
      FORALL (n : IntegralNumber): (0 <= n AND n < 31) IMPLIES
                     1 <= intShiftL(1, n)
    longShiftL_definition : LEMMA
      FORALL (n : IntegralNumber): (0 <= n AND n < 63) IMPLIES
                     1 <= longShiftL(1, n)

  END escjava2_java_integral_ops

escjava2_java_floating_point : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem

    % === A few floating point axioms - DRCok

    floatingEQ(x, y : FloatingPointNumber) : FloatingPointNumber = (x = y)
    floatingGE(x, y : FloatingPointNumber) : FloatingPointNumber = (x >= y)
    floatingGT(x, y : FloatingPointNumber) : FloatingPointNumber = (x > y)
    floatingLE(x, y : FloatingPointNumber) : FloatingPointNumber = (x <= y)
    floatingLT(x, y : FloatingPointNumber) : FloatingPointNumber = (x < y)
    floatingNE(x, y : FloatingPointNumber) : FloatingPointNumber = (x /= y)

   floatingADD(x, y : FloatingPointNumber) : FloatingPointNumber = (x + y)
   floatingMUL(x, y : FloatingPointNumber) : FloatingPointNumber = (x * y)
   floatingNEQ(x : FloatingPointNumber) : FloatingPointNumber = (- x)
   floatingMod(x, y : FloatingPointNumber): {v: mod(y) | EXISTS (r: mod(y)): v = x * y + r}

  END escjava2_java_floating_point

escjava2_array_store : THEORY
  BEGIN

    % Mimics the 'elems' of SRC ESC/Java.
    ArrayStore : TYPE+
    % This is the syntactic, fully-resolved name of the Java array
    % being indexed.  In SRC ESC/Java, it is encoded in Translate as a
    % standard VariableAccess.
    ArrayName : TYPE+

  END escjava2_array_store

escjava2_java_field_representation : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem,
              escjava2_jml_semantics,
              escjava2_array_store,
              map_theory[BooleanField, Reference, Boolean],
              map_theory[NumberField, Reference, Number],
              map_theory[ReferenceField, Reference, Reference],
              map_theory[ArrayReference, IntegralNumber, Boolean],
              map_theory[ArrayReference, IntegralNumber, Number],
              map_theory[ArrayReference, IntegralNumber, Reference],
              map_theory[ArrayStore, ArrayName, ArrayReference]

    elems : var ArrayStore

    % === ESCJ 8: Section 3.0
```

```
    vAllocTime : [ Reference -> Time ]
    isAllocated (r : Reference, t : Time): bool = vAllocTime(r) < t

    % === ESCJ 8: Section 3.1

    fClosedTime : [ Field -> Time ]
    fClosedTime_definition : AXIOM
      FORALL (r : Reference, f : Field, t : Time):
        (fClosedTime(f) < t AND isAllocated(r, t)) IMPLIES
        isAllocated(get(f, r), t)

    % === ESCJ 8: Section 3.2
    eClosedTime : [ ArrayStore -> Time ]

    eClosedTime_definition : AXIOM
      FORALL (a : ArrayName, i : IntegralNumber, t : Time):
        (eClosedTime(elems) < t AND isAllocated(get(elems, a), t)) IMPLIES
        isAllocated(get(get(elems, a), i), t)

  END escjava2_java_field_representation

escjava2_java_strings : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem,
              escjava2_jml_semantics

    T_java_lang_String : ReferenceType

    stringCat (x, y : Reference) : Reference

    stringCat_definition1 : AXIOM
      FORALL (x, y : Reference): stringCat(x, y) /= NULL AND
                        typeOf(stringCat(x, y)) <= T_java_lang_String

  END escjava2_java_strings

escjava2_java_semantics : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem,
              escjava2_java_boolean_ops,
              escjava2_java_integral_types,
              escjava2_java_integral_ops,
              escjava2_java_floating_point,
              escjava2_java_field_representation,
              escjava2_java_strings

  END escjava2_java_semantics

escjava2_lock_semantics : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem,
              escjava2_jml_semantics,
              escjava2_java_field_representation,
              map_theory[Lock, Reference, bool]

%     % === ESCJ 8: Section 4

    LS : Lock
    maxLockset : Reference

    lockLE (l : Lock, x : Reference, y : Reference): bool
    lockLT (l : Lock, x : Reference, y : Reference): bool

    max(l : Lock) : Reference =
      maxLockset

    % null is in lockset (not in ESCJ 8)
    null_is_in_lockset : AXIOM
      FORALL (l : Lock, r : Reference): get(l, NULL) = true

    % all locks in lockset are below max(lockset) (not in ESCJ 8)
    all_locks_in_lockset_are_below_max_lockset : AXIOM
      FORALL (l : Lock, r : Reference): get( l, r) = true IMPLIES
```

```
                          lockLE(l, r, max(l))

    % null precedes all objects in locking order (not in ESCJ 8)
    null_precedes_all_objects : AXIOM
      FORALL (l : Lock, x : Reference): typeOf(x) <= T_java_lang_Object IMPLIES
                     lockLE(l, NULL, x)

  END escjava2_lock_semantics

escjava2_arrays : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem,
              escjava2_java_field_representation,
              escjava2_java_integral_types

    arrayOf( n : IntegralNumber, t : PrimitiveType) : ArrayReference
    arrayOf( n : IntegralNumber, t : ReferenceType) : ArrayReference

    % arrayLength(r : ArrayReference) : IntegralNumber
    arrayLength(r : Reference) : IntegralNumber

    arrayLengthDef1 : AXIOM
      FORALL(n : IntegralNumber, t : PrimitiveType, r : Reference) :
        arrayLength(r) = n IFF r = arrayOf(n, t)

    arrayLengthDef2 : AXIOM
      FORALL(n : IntegralNumber, t : ReferenceType, r : Reference) :
        arrayLength(r) = n IFF r = arrayOf(n, t)

  END escjava2_arrays

escjava2_jml_semantics : THEORY
  BEGIN
    IMPORTING escjava2_java_typesystem

    % === Define typeof for primitive types - DRCok
    typeof : [ Boolean -> PrimitiveType ]
    typeof : [ IntegralNumber -> PrimitiveType ]
    typeof : [ Reference -> ReferenceType ]
    % WRONG - not equivalent; need to review
%     typeof_definition : AXIOM
%       FORALL (x, y : S): primitive?(y) AND is(x, y) IFF typeof(x) = y

    typeof_char : AXIOM
      FORALL (x : IntegralNumber): is(x, T_char) IFF typeof(x) = T_char
    typeof_byte : AXIOM
      FORALL (x : IntegralNumber): is(x, T_byte) IFF typeof(x) = T_byte
    typeof_short : AXIOM
      FORALL (x : IntegralNumber): is(x, T_short) IFF typeof(x) = T_short
    typeof_int : AXIOM
      FORALL (x : IntegralNumber): is(x, T_int) IFF typeof(x) = T_int
    typeof_long : AXIOM
      FORALL (x : IntegralNumber): is(x, T_long) IFF typeof(x) = T_long
    typeof_float : AXIOM
      FORALL (x : IntegralNumber): is(x, T_float) IFF typeof(x) = T_float
    typeof_double : AXIOM
      FORALL (x : IntegralNumber): is(x, T_double) IFF typeof(x) = T_double

    % === ESCJ 8: Section 2.3

    typeof_reference_definition : AXIOM
      FORALL (r : Reference, t : ReferenceType): t <= T_java_lang_Object IMPLIES
                                      is(r, t) IFF (r = NULL OR typeof(r) <= t)
  END escjava2_jml_semantics

escjava2_predefined: THEORY
  BEGIN
    IMPORTING escjava2_types, escjava2_arrays

  % predefined types
  T_java_lang_Exception: TYPE+ FROM ReferenceType
  T_java_lang_Object: TYPE+ FROM ReferenceType

  % predefined variables
```

```
  ecReturn: Path
  ecThrow: Path
  alloc: Time
  alloc_pre: Time
  elems: ArrayStore
  elems_pre: ArrayStore
  state: JavaType
  state_pre: JavaType
  this: Reference
  XRES: Reference
  T_java_lang_Exception: ReferenceType
  T_java_lang_Object: ReferenceType

  IntegralField : TYPE+ FROM NumberField

  % predefined methods
  getStackTrace_state(s: JavaType, o: Reference): Reference
  java_lang_Throwable_stackTrace(s: JavaType, o: Reference): Reference

  ecReturn_ecThrow_axiom: AXIOM
    ecReturn /= ecThrow

  this_not_null_axiom: AXIOM
    this /= NULL

  get_and_set_axiom: AXIOM
    FORALL(f: Field, r: Reference, v: Number):
      get(set(f, r, v), r) = v

  END escjava2_predefined

escjava2_test_sum: THEORY
  BEGIN
    IMPORTING escjava2_types, escjava2_arrays

  sum(i: IntegralNumber, j: IntegralNumber): IntegralNumber =
    IF i > j THEN 0
    ELSE ((i + j) * (j - i + 1) / 2)
    ENDIF

  END escjava2_test_sum

escjava2_test_fibo: THEORY
  BEGIN
    IMPORTING escjava2_types, escjava2_arrays

  fibo(i: IntegralNumber): RECURSIVE IntegralNumber =
    IF i = 1 THEN 1
    ELSE (IF i = 2 THEN 1
          ELSE (IF i >= 3 THEN fibo(i-1) + fibo(i-2)
                ELSE 0
                ENDIF)
          ENDIF)
    ENDIF
    MEASURE (LAMBDA (i: IntegralNumber): i)

  END escjava2_test_fibo

escjava2_test_set_array: THEORY
  BEGIN
    IMPORTING escjava2_types, escjava2_arrays
    IMPORTING map_theory[ArrayReference, IntegralNumber, IntegralNumber]
    IMPORTING map_theory[ArrayStore, ArrayReference, ArrayReference]
    IMPORTING sets[IntegralNumber]

  toSet(A: ArrayReference, n: IntegralNumber): RECURSIVE set =
    IF n <= 0 THEN empty?
    ELSE add(toSet(A, n-1), get(A, n-1))
    ENDIF
  MEASURE (LAMBDA (n: IntegralNumber): n)

  END escjava2_test_set_array

escjava2_test_celebrity: THEORY
```

```
  BEGIN
    IMPORTING escjava2_types, escjava2_arrays
    IMPORTING map_theory[ArrayReference, IntegralNumber, IntegralNumber]
    IMPORTING map_theory[ArrayStore, ArrayReference, ArrayReference]
    IMPORTING map_theory[IntegralField, Reference, IntegralNumber]
    IMPORTING sets[IntegralNumber]

  toSet(n: IntegralNumber, m: IntegralNumber):
        set = {k: IntegralNumber | n <= k AND k <= m}

  toSet_axiom_1: AXIOM
    FORALL(n: IntegralNumber, m: IntegralNumber):
      toSet(n+1, m) = remove(n, toSet(n, m))

  toSet_axiom_2: AXIOM
    FORALL(n: IntegralNumber, m: IntegralNumber):
      toSet(n, m-1) = remove(m, toSet(n, m))

  toSet_axiom_3: AXIOM
    FORALL(n: IntegralNumber, m: IntegralNumber):
      (n > m) <=> (toSet(n, m) = emptyset)

  END escjava2_test_celebrity

escjava2_logic : THEORY
  BEGIN
    IMPORTING escjava2_java_semantics,
              escjava2_jml_semantics,
              escjava2_lock_semantics,
              escjava2_arrays,
              escjava2_predefined

    %IMPORTING escjava2_test_sum
    %IMPORTING escjava2_test_fibo
    %IMPORTING escjava2_test_set_array
    %IMPORTING escjava2_test_celebrity

  END escjava2_logic
```

Bibliography

[1] Á. Darvas and P. Müller. *Reasoning About Method Calls in Interface Specifications*, in Journal of Object Technology, Vol.5, No.5, Jun. 2006.

[2] B. Jacobs, H. Meijer, and E. Poll. *VerifiCard: A European Project for Smart Card Verification*, in Newsletter 5 of the Dutch Association for Theoretical Computer Science, 2001.

[3] C. A. R. Hoare. *An Axiomatic Basis for Computer Programming*, in Communications of the ACM, Vol.12, No.10, Pages 576-583, Oct. 1969.

[4] C. A. R. Hoare. *Proof of Correctness of Data Representations*, in Acta Informatica, Vol.1, No.4, Pages 271-281, 1972.

[5] C. Breunesse. *On JML: topics in tool-assisted verification of JML programs*, PhD thesis, Faculty of Science, Mathematics and Computer Science, Radboud University Nijmegen, Aug. 2006.

[6] C. Breunesse, E. Poll and M. Warnier. *Introduction to Theorem Proving using PVS*, IPA Course on Formal Method, 2006. http://www.cs.ru.nl/E.Poll

[7] C. Hurlin. *Verifcation Condition Generator for ESC/Java2*, Internal Document of ESC/Java2, Oct. 2005.

[8] C. Flanagan, J. B. Saxe. *Avoiding Exponential Explosion: Generating Compact Verification Conditions*, in POPL'01, London, UK, Jan. 2001.

[9] C. Flanagan, K. R. M. Leino, M. Lilibridge, et al. *Extended Static Checking for Java*, in PLDI'02, Berlin, Gemany, Jun. 2002.

[10] D. Cok and J. Kiniry. *ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2*, RUN NIII Technical Report NIII-R0413, Faculty of Science, Mathematics and Computing Science, Radboud University Nijmegen, 2004.

[11] D. Cok, J. Kiniry, and E. Poll. *Design by Contract and Automatic Verification for Java with JML and ESC/Java2*, FM'05 Tutorial Slides. http://secure.ucd.ie

[12] D. Cok. *Reasoning with Specifications Containing Method Calls and Model Fields*, in Journal of Object Technology, Vol. 4, No. 8, Pages 77-103, Oct. 2005.

[13] E. W. Dijkstra. *A Discipline of Porgramming*, Prentice-Hall, Inc., 1976.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*, Addison-Wesley, 1995.

[15] G. T. Leavens and Y. Cheon. *Design by Contract with JML*, Internal Document of JML. http://www.cs.iastate.edu/∼leavens/JML

[16] G. T. Leavens, E. Poll, C. Clifton, et al. *JML Reference Manual*, Internal Document of JML. http://www.cs.iastate.edu/∼leavens/JML

[17] G. T. Leavens and C. Clifton. *Lessons from the JML Project*, ISU Technical Report 05-12a, Computer Science, Iowa State University, Jul. 2005.

[18] G. Nelson. *A Generalization of Dijkstra's Calculus*, in ACM Transactions on Programming Languages and Systems, Pages 517-561, 1989.

[19] J. Kiniry, A. Morkan, F. Fairmichael, et al. *The KOA Remote Voting System: A Summary of Work To-Date*, submitted to EVT'06, 2006.

[20] J. Crow, S. Owre and J. Rushby, et al. *A Tutorial Introduction to PVS*, in WIFT'95, Florida, USA, 1995.

[21] J. Charles. *Add Native Specifications to JML*, in FTfJP06, Nantes, France, 2006.

[22] K. R. M. Leino. *Ecstatic: An Object-Oriented Programming Language with an Axiomatic Semantics*, in FOOL4, Paris, Jan. 1997.

[23] K. R. M. Leino and J. B. Saxe. *The Logic of ESC/Java*, Internal Document of ESC/Java, 1998.

[24] K. R. M. Leino, J. B. Saxe, and R. Stata *Checking Java Programs via Guarded Commands* Compaq SRC Technical Note 1999-002, May 1999.

[25] K. R. M. Leino, G. Nelson and J. B. Saxe. *ESC/Java User's Manual*, Compaq SRC Technical Note 2000-002, Oct. 2000.

[26] K. R. M. Leino. *Efficient Weakest Preconditions*, Microsoft Research Technical Report 2004-34, Apr. 2004.

[27] L. Burdy, Y. Cheon, D. Cok, et al. *An Overview of JML Tools and Applications*, in FMICS'03, Volume 90 of Electronic Nots in Theoretical Computer Science, Pages 73-89, Jun. 2003.

[28] N. Shankar, S. Owre and J. M. Rushby, et al. *PVS Prover Guide*, SRI International, 2001.

[29] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*, Prentice-Hall, Inc., 2003.

[30] S. Owre, N. Shankar and J. M. Rushby, et al. *PVS Language Reference*, SRI International, 2001.