Enhancements to the JML Runtime Assertion Checker Compiler

Kui Dai

A Thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

September 2005

Your file Votre référence
ISBN: 0-494-10284-5
Our file Notre référence
ISBN: 0-494-10284-5

ABSTRACT

Enhancements to the JML Runtime Assertion Checker Compiler

Kui Dai

The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) that can be used to specify the behavior of Java modules. Several tools have been developed for JML, such as the JML (type) checker, JML Runtime Assertion Checker (RAC), ESC/Java2, LOOP and so on. The RAC can be used to translate JML specifications into Java code so that the specifications can be checked at run-time.

In this thesis we describe how two recent enhancements to JML have been formally defined and implemented in the RAC. The RAC semantics (of the newly added functionality) is presented as a large-step operational semantics. An overview of the RAC design is given so as to provide a better understanding of the context in which the enhancements have been made.

The main enhancement to be realized is the added support for arbitrary precision integers in JML specification expressions and model method bodies. We also present prototypical implementation of an alternate semantics for JML assertions that has only very recently been published.

Acknowledgements

I would like to express my gratitude to all those who helped me with my research and the writing of this thesis. I am deeply indebted to my supervisor Dr. Patrice Chalin whose help, stimulating suggestions and encouragement helped me throughout the research and writing of this thesis.

I thank all the members of our Dependable Software Research Group (DSRG), for their help, support and valuable hints. I'm especially obliged to Frederic Rioux for his guidance during my research and comments while reviewing my thesis. I would also like to thank Stuart Thiel for his reading of my thesis.

I thank the JML and MultiJava developers and users who helped me when I modified the projects.

Finally, I thank my parents for their love and support even if they live thousands of miles away.

Contents

List of Figures

List of Tables

1 Introduction

This chapter gives a brief introduction to the Java Modeling Language (JML) and Behavioral Interface Specification Languages (BISLs). It also introduces some of the JML tools. Finally, a summary of contributions is given.

1.1 Java Modeling Language

The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) [6] that can be used to specify the behavior of Java modules. Being a specification language, JML has no relationship with UML, the Unified Modeling Language, although both are named "modeling languages". The term *module* is used to refer to either a Java class or interface. JML is a model-based specification language, like VDM (The Vienna Development Method) [22,23] or Larch [24,25,26], and it has some elements of a refinement calculus [36]. JML specifications are written using a syntax that is based on Java. It supports Design by Contract, quantifiers, specification-only variables, and other enhancements that make it more expressive for specification than Eiffel (which also uses the underlying programming language syntax to capture specifications).

An important part of JML specifications are method contracts. These are expressed in a Hoare-logic style, i.e. using pre and post-conditions. The basic meaning of these conditions is that the preconditions must hold before a method body is executed, and after the method is executed, the post-condition must hold. There are two kinds of post-conditions in JML, one is normal post-condition and the other is exceptional post-condition. The former one is used to indicate behaviors when the method terminates

normally, while the later one is used to indicate behaviors when the method terminates by throwing an exception.

A sample JML specification is given in Figure 1.1. The `Player` class is part of a game. Every player has a life level (field `nLife`), number of times it can jump (field `nJump`), and a rectangle (field `rcImage`) where the picture of player is drawn. Moreover, players can jump, method `jump(nTimes)`, and can move, method `move()`. As well, the image of a player can be drawn using the method `drawImage()`.

All of the comment lines starting with @ characters are a part of the JML specification for the class. Lines (a) and (b) are two invariant clauses which specify constraints on the two member fields `nJump` and `nLife`, that is, bind the values of these fields to be positive. Lines (c), (d), and (e) are specifications for method `drawImage()`. Line (c) declares the specification visibility and identifies the specification block as expressing the required method behavior when it terminates normally. Line (d) is a requires clause which specifies the pre-conditions that must hold before the execution of the method. Line (e) is a assignable clause which says that no client visible fields or objects can be changed when method `drawImage()` executes. There is no post-condition for this method, so its post-condition can be thought of as implicitly true. Line (f)-(j) are specifications for the method `move()`. Line (h) shows that `rcImage` is permitted to change during the execution of the method. Line (i) and (j) indicate the post-condition that must hold after the method `move()` terminates successfully. In the *ensures* clause, there is an old expression (`\old(nJump)`) which refers to the value of its argument expression as it was *before* the method executed. Line (o) to (s) specify an exceptional behavior which means if the pre condition (line (p)) holds, an exception *InvalidArgument* can be thrown.

```
import java.awt.*;

public class Player {
    protected int nJump;
    protected int nLife;
    protected Rectangle rcImage;

    //@ protected invariant nJump >= 0;                                    //(a)
    //@ protected invariant nLife >= 0;                                    //(b)

    /*@ public normal_behavior                                            //(c)
      @ requires rcImage != null &&
      @          rcImage.height > 0 && rcImage.width > 0;                 //(d)
      @ assignable \nothing;                                              //(e)
      @*/
    public void drawImage() {/*...*/}

    /*@ public normal_behavior                                            //(f)
      @ requires nLife > 0;                                               //(g)
      @ assignable rcImage;                                               //(h)
      @ ensures rcImage.height == \old(rcImage).height &&                 //(i)
      @         rcImage.width == \old(rcImage).width;                     //(j)
      @*/
    public void move() {/*...*/}

    /*@ public normal_behavior                                            //(k)
      @  requires nJump >= nTimes;                                        //(l)
      @  assignable nJump;                                                //(m)
      @  ensures nJump == \old(nJump) - nTimes;                           //(n)
      @ also                                                              //(o)
      @ public exceptional_behavior                                       //(p)
      @  requires nJump < nTime;                                          //(q)
      @  assignable \nothing;                                             //(r)
      @  signals (InvalidArgument e) true;                                //(s)
      @*/
    public void jump(int nTimes) {/*...*/}
}
```

Figure 1.1 Sample JML Specification

1.2 BISLs: Background and Motivation

BISLs are interesting to study because they feature the following advantages [19]:

- Can be integrated into current development practices
- Can be applied to both new and legacy code
- Improve the quality of software design, which is now one of the most important efforts, as application software is becoming much larger and more complex than ever before

Before JML was created, the main BISLs were the Larch family of languages which included LCL (Larch C Language) for C, Larch/C++ for C++, Larch/Smalltalk for

Smalltalk, Larch/Ada for Ada, and LM3 for Modula-3. One distinguishing characteristic of Larch is its two-tiered approach. A *shared* tier involves specifications written in the Larch Shared Language (LSL). These specifications are called traits, and they capture abstractions used in the descriptions contained in the *interface* tier. Each trait defines a multisorted first-order theory. The interface tier contains module specification elements that are specific to a particular programming language, and hence are often based on the programming language in question. Unfortunately, the Larch languages were not popular among software developers. One of the main reasons was the difficulty of using the Larch Shared Language to capture abstractions. So JML was defined according to the main goal of creating a BISL which is both practical and effective [2]. The other goals of JML are [2, Section 1.3]:

- JML must be able to document the interfaces and behavior of existing software, regardless of the analysis and design methods used to create it.
- The notation used in JML should be readily understandable by Java programmers.
- The language must be capable of being given a rigorous, formal semantics, and must also be amenable to tool support.

1.3 JML Tools

Many tools have been developed for JML, such as the JML checker [20] and the Runtime Assertion Checker (RAC) [3], ESC/Java2 [7, 16], LOOP [18], JmlUnit [38], JACK [39], the Daikon invariant detector [40, 41], Houdini [42], JmlDoc [37] and so on. The rest of this section gives a brief description of three of the main tools. The tools chosen can be used to check, with varying levels of effort, whether a Java application satisfies its JML

specifications. We present the tools in order of increasing effort required on the part of the developer.

1.3.1 JML RAC

The JML Runtime Assertion Checker (RAC) can be seen as a JML compiler. It translates JML specifications to Java code appropriately "weaved" in with the Java modules it specifies. The RAC is a part of the Iowa State University (ISU) JML tool suite [15], which is built on top of the MultiJava compiler project [14].

The main purpose of the RAC is to generate extra code that "checks" if a module satisfies its JML specification at run-time. If a check fails, then an exception would be thrown. In such situations, a bug has been uncovered: it might be a bug of the Java code, or the JML specification. Moreover, as it must be trustworthy, the RAC must not produce any false reports, so sometimes, it may ignore possible errors. For example, people can write informational assertions in JML. These assertions are merely English sentences which can only be recognized by a human reader. All these assertions will be treated as "true" in the RAC even if there might be errors in them. Finally, implemented as a compiler, the RAC is compatible with other compilers. That is, the byte code generated by the RAC from a Java program without JML specifications behave the same as the one compiled by a normal Java compiler, though the byte code might be larger due to unnecessary instrumentation.

1.3.2 ESC/Java2, an Extended Static Checker for Java

The ESC/Java tool was originally developed at Compaq Research Center; it's current successor, ESC/Java2, is being developed by Cok and Kiniry [7,16]. Compared with the JML RAC, ESC/Java2 is a program checking tool which uses static analysis. A major

improvement of ESC/Java2 as compared to its predecessor is that it supports the entire JML language. Its main function is to determine possible run-time errors in Java programs by static analysis alone. To achieve this, it generates verification conditions (VCs) from the JML annotated Java source and then attempts to prove these VCs using a fully automated theorem prover named Simplify. ESC/Java2 reports back to the user any VCs that Simplify is unable to prove (generally by offering a counter example to the VC).

A feature of ESC/Java2 is that, when confronted with a potential error whose determination may be time/information-intensive, it will make an "educated guess" as to whether there will be an error or not, without fully pursuing it. This leads to a failure to detect some errors, while also reporting some errors which are actually not present (i.e. generating a false positive). The purpose of doing this is to increase the benefit/use effort ratio of the tool. This tactic appropriately meets this purpose, in part because it is not feasible for the ESC/Java to incur the cost of such a full determination, and in part because, as a static analyzing tool, it is impossible for ESC/Java to automatically detect all possible errors.

1.3.3 The LOOP Program Verifier

Both the RAC and ESC/Java2 are tools that can be used to give us some assurance that a Java program satisfies its JML specifications. Sometimes we need to *prove* that the specifications are satisfied. One popular theorem prover is PVS (the Prototype Verification System) [18]. As neither Java code nor JML specifications can be recognized by PVS, a compiler was created to translate them into the language of the PVS in the form of proof obligations. This compiler is called the LOOP (for Logic of Object-Oriented Programs) compiler [18] and is being developed at the University of

Nijmegen. The input of LOOP is a Java program with its JML specifications, and the output is several PVS syntax files which can be imported and proved in PVS.

1.4 Contribution

JML is an open source project being developed by an international team of researchers [28]. Chalin has recently added support for arbitrary precision integers to JML [4,21]. My contribution has been to add support for this language feature to the JML RAC. This involved:

- Studying and understanding the JML checker and RAC code (243 KLOC in 831 files).
- Writing formal translation rules for the RAC semantics of the "arbitrary precision integers" feature.
- Creating test suites and integrating them into the existing JML project testing framework.
- Designing and implementing the features in the JML checker and RAC.
- Studying, understanding and adhering to the software development and configuration management conventions of the JML Sourceforge project.

I also describe experimental changes that were made to the RAC to examine the impact of altering the interpretation of assertions. This was done based on Chalin's latest proposal, given in [29].

1.5 Outline

Structure for the remainder of this thesis is as follows. Chapter 2 gives a brief introduction to the JML language. It is introduced from a lower level to higher one, that is, from JML expressions to method specifications, and then to module specifications. Chapter 3 gives an overview of the JML runtime assertion checker by explaining how it operates, as well as explaining its overall design. This overview was created based on an analysis of the current implementation Chapters 4 to 6 provide the details of the main contributions to the RAC: supporting `\bigint` in JML expressions, supporting `\bigint` in model method bodies, and implementing neutral contexts in the RAC. Chapter 7 presents a summary and discuss possible future work. Chapter 9 and 10 are appendixes used to show the expressions implemented in the RAC and an example of code generated by the RAC.

2 JML: Language Review

Since this thesis covers the semantics and implementation of new JML features, it is necessary to introduce the language in sufficient detail to allow readers, new to JML, to grasp the significance of these additions. That is the purpose of this chapter.

2.1 Annotations

All JML specification elements are expressed via *annotations*. There are two ways to write JML specifications, one is by placing annotations in Java code files, and the other is by using complete files whose extension names are ".jml-refined", ".jml", ".spec", or ".spec-refined". Annotations are always written in the form of comments between the markers /*@ and @*/, also the line after //@, in this way, a Java file with JML specifications can still be compiled by commonly used compilers such as javac.

The JML grammar can be divided into three main phrase class "levels". The first one is JML Expressions and Assertions. The second one is Method Specifications. The last one is Class and Interface Specifications. In the following, I will concisely introduce them, one by one.

2.2 Expressions and Assertions

Expressions in JML use Java syntax, but have two main differences. One is that all the JML constructs must be pure; the other is that JML has its own special expressions.

The concept of purity in the context of JML means having no side-effects. Therefore, JML expressions can't contain assignment operators (=) or compound-assignment operators (e.g. +=, -=), pre(post)-fix operators (++ and --), invocations of non-pure methods, or the *new* operator if said *new* would call a side-effect constructor.

JML also has its own special operators and/or language constructs for expressions which are not a part of Java. Some of the most commonly used are shown next (also see [2, Section 3.1]):

- (* ... *) Informal assertion where "..." can be any text
- ==> and <== logical and reverse implication
- <==> and <=!=> logical equivalence and inequivalence
- <: subtype relation
- \duration(e) time for execution

  eg. `\duration(f())` : the maximum number of virtual machine cycles needed to execute the method f().
- \elemtype(e) returns the static type of e

  ```
  eg. \elemtype(\type(int[])) == \type(int)
  ```
- \fields_of(e) fields and array elements in objects
- \forall and \exists universal and existential quantifiers

  ```
  eg. (\forall int i; 0 <= i && i < 100;)
  ```
- \fresh(e) asserts that objects are freshly allocated
- \max, \min, \product, and \sum

  generalized quantifiers

  ```
  eg. (\sum int i; 1 <= i && i < 3; i) == 1 + 2 + 3
  ```
- \old(e) values in the pre-state
- \result return value of a method
- \typeof returns the dynamic type
- A[*] all items in array A
- A[E1 .. E2] array ranges
- new Set comprehensions

  eg. `new JMLObjectSet {Person p | PersonInClass.has() && p.age() == 20}` : a JMLObjectSet consists of all Person objects found in the set PersonInClass and whose age is 20.

2.3 Method Specifications

JML use the Hoare-logic style pre and post-conditions for method specifications. Furthermore, it provides many extensions to them, including two forms of specification (heavyweight and lightweight), visibility of specifications, normal and exceptional post-conditions, frame conditions and redundant specification elements [3, Section 2.3]. These features will be explained briefly in the following:

2.3.1 Specification Clauses

The following lists some of the most important clauses that can be used for method specifications:

- **requires** specifies a precondition which must be satisfied before the method is invoked. Actually a *requires* clause specifies a conjunction of the method's precondition (since more than one clause can be used in a method contract).

  ```
  eg.  //@ requires i > 0;
       //@ requires i < 10;
  ```

  This specification means that before the method is called, the variable i must be larger than 0 but less than 10.

- **ensures** specifies a normal post-condition; if the method terminates normally, it must satisfy the given boolean expression.

  ```
  e.g. //@ ensures nSize == i + 1;
  ```

  This statement means that if the method terminates normally, the variable `nSize` must be equal to `i + 1`.

- **signals** gives an exceptional post-condition; if the method terminates abruptly by throwing an exception, it must satisfy the given boolean expression.

  ```
  eg: //@ signals (NullPointerException) x == null;
  ```

  This specification indicates if the method throws a NullPointerException exception, the variable `x` must be equal to null.

- **assignable** defines a frame condition [8] used to restrict the possible side-effects of a method, i.e. only the named locations can be modified during the execution of a method.

  ```
  eg: //@ assignable theStack;
  ```

  This statement means that only the variable (or field) `theStack` can be modified in the following method.

- **diverges** specifies under which conditions a method fails to terminate (i.e. diverges).

- **accessible** gives locations that can be directly accessed in the body of the method.

- **callable** specifies methods that a method will be able to call,

- **duration** specifies the maximum time needed to call a method or constructor.

- **working_space** provides a guarantee of the maximum amount of heap space used by a call.

2.3.2 Heavyweight and Lightweight Specifications

A *heavyweight specification* is a specification that is "completely detailed" and uses one of the behavior keywords (`behavior`, `normal_behavior`, and `exceptional_behavior`). On

the other hand, a *lightweight specification* is a specification which is "less detailed" and does not use a behavior keyword. As a lightweight specification has no visibility modifier (i.e. public, protected, private) JML treats it as having the same level of visibility as the method it specifies. In a lightweight specification, the default for an omitted method specification clause is `\not_specified`, which tools are free to interpret in their own way [3, Section 2.3.2]. The meanings of the behavior keywords are:

- **normal_behavior** if the precondition holds, the execution of a method is "normal", that is it must end normally without throwing an exception.
- **exceptional_behavior** if the precondition holds, the execution of a method is "exceptional", that is it must end abruptly by throwing an exception.
- **behavior** if the precondition holds, the execution of a method can be either "normal" or "exceptional" (as specified in the given ensures and/or signals clauses).

2.3.3 Other features

- **visibility of specifications** The visibilities of JML specification constructs are the same as those in Java: public, protected, package-visible, and private.
- **also** The keyword *also* is used to separate two or more specifications blocks.
- **"_redundantly" keyword suffix** Almost every JML keyword can be suffixed with "_redundantly" (e.g. requires_redundantly) which means "both that the stated property is specified to hold and that this property is

believed to follow from the other properties of the specification" [2, Section 2.2.2.2].

2.4 Module Specifications

Java declarations, such as member fields, can be used in JML module specifications. Moreover, JML also includes some extensions such as invariants, history constraints, specification-only member declarations.

2.4.1 Invariants

A *class invariant* is indicated by the keyword `invariant` followed by an assertion. For example:

```
//@ private invariant 0 <= nSize;
```

An invariant is a predicate that remains true between method calls. It doesn't have to hold during the execution of a method, but it must hold before and after. "Helper" methods are an exception to this rule—a class invariant is not required to hold before or after a helper method is executed [11, Section 7.1.1.4]. A typical way of using this keyword is to specify constraints on a class field as in the above example. The default visibility level of an invariant is package-visible.

Invariants can be divided into two different kinds, one is *static invariant*, indicated by the modifier `static`, and the other one is *instance invariant* (which requires no special modifier in class specification, but requires the `instance` modifier in Java interface specifications). Without an explicit modifier, an invariant is an *instance* invariant if it is a member of a class, or a *static* invariant if it belongs to an interface. The major difference between *static* invariants and *instance* invariants is that a *static* invariant can only refer to

static type members, whereas an *instance* invariant can refer to all kinds of type members.

2.4.2 History Constraints

The *history constraints* (*constraints* for short) have the syntax: `constraint` *predicate [* `for` *constrained-list];* and the optional part is used to list method(s) that the constraint applies to [11, Section 8.3]. For example:

```
/*@ public constraint vRectClient == \old(vRectClient)
  @                   for OnDraw(), OnPaint();
  @*/
```

A constraint is used to define "relationships that should hold for the combination of each visible state and any visible state that occurs later in the program's execution" [11, Section 8.3]. A typical way of using this keyword is to specify how a value changes before and after a method. Therefore, constraints usually use an `old` expression which refers to the value before a method is executed as shown in the example above.

Like invariants, constraints are also separated into static and instance; The implicit visibility of constraints is also package-visible. Constraints can still only work for non-helper methods. One difference between them is that invariants can't contain `\old` terms.

2.4.3 Specification-only Attributes

There are four kinds of specification-only attributes: model fields, ghost fields, model methods, and model modules (so far, the last kind is not supported by JML compilers). The main feature of these attributes is that they can only be used in specifications, and are invisible to Java code. Moreover, they don't have to be implemented and there is no storage directly combined with them [2, Section 2.1.1].

A model field is declared by the keyword `model`, and a `represents` clause is normally followed to define how its value is related to the module state. The following is an example:

```
long nLength, nWidth;
//@ public model long lArea;
//@ public represents lArea <- nLength * nWidth;
```

These two statements declare a model field `lArea` whose value is equivalent to the expression `nLenght * nWidth`.

A ghost field declaration starts with the modifier `ghost`, and is similar to a model field. A ghost field is a specification-only field whose value is explicitly set using `set` statements (rather than using a represents clause). An example of a ghost field is shown in the following:

```
//@ public static ghost int nCounter = 0;

void f() {
   //@ set nCounter = nCounter + 1;
}
```

The ghost field `nCounter` is used to record the number of times `f()` is invoked.

JML also supports specification-only methods, called *model methods*. Such method declarations start with the `model` keyword and are entirely contained within a JML annotation (i.e. comment). It can be treated as a Java method which can only be seen by specifications. An example of model method is shown in the following:

```
//@ public model long lArea;
//@ public represents lArea <- getArea();

//@ private model pure long getArea() {
//@   return nLength * nWidth;
//@ }
```

This example has the same function as the example for model field mentioned above.

2.4.4 Inheritance of Specifications

In JML, specification inheritance stipulates that a sub-type inherits the specifications of its super-type(s); this is called *behavioral subtyping* [12]. Hence the Java keyword `extends` is bound by extra semantics to mean that a subclass inherits specifications from its super-classes, or a sub-interface inherits specifications from its super-interface. The keyword `implements` indicates a class inherits specifications from its interface(s).

A new means of specification inheritance supported by JML is refinement, which is indicated by the keyword *refine*. All specifications, regardless of their visibility, are inherited by the refining type from its refined type. For example, the following refine declaration states that the class A in the current file (e.g., `A.java`) "refines" the same class in the file `A.jml`. The class A in the current file is called a *refining type* and the one in `A.jml` is called a *refined type*.

```
//@ refine "A.jml";
public class A { /* ... */ }
```

JML provides a particular kind of behavioral subtyping which is called *weak behavioral subtyping*. In this subtyping form, the subtype's additional non-overriding methods can ignore the history constraints declared in its supertypes. When the opposite is true, it is called a *strong behavioral subtyping*. A weak behavioral subtyping is indicated by using a keyword *weakly*. A strong behavioral subtyping is the default kind of inheritance, if no modifier is explicitly specified. For instance, the following example means that class A performs weak behavioral subtyping of its super-class B, but strong behavioral subtyping of its super-interface C.

```
public class A extends B /*@ weakly @*/ implements C { ... }
```

As both classes and interfaces can have specifications, if a class has both super class and an interface, it will become a case of multi-inheritance, which is a major difference from Java.

3 JML Runtime Assertion Checker

3.1 Introduction and Example

JML specifications are essentially captured in the form of assertions. The JML Runtime Assertion Checker (RAC) uses assertions to generate code that checks the consistency of Java programs and their specifications dynamically.

In the example shown in Figure 3.1, there are three JML assertions, that is, (1) is a class invariant; (2) is a precondition for method `m()`; (3) is a post-condition for `m()`. After the RAC has been used to compile this example, execution of a call to `m()` will cause the class invariant and the precondition to be checked (i.e. evaluated) before the method `m()` is invoked, and the post-condition and the class invariant after the method is executed. (A sample of code generated by the RAC is given in Chapter 9).

```
public class T {
  protected int nSize;

  //@ invariant nSize > 0; //(1)

  //@ requires a > 0;        //(2)
  //@ ensures \result > 0; //(3)
  public int m(int a) {
  ...
  }
}
```

Figure 3.1 A simple class with JML specification

3.2 Tool Behavior and Phases

The RAC is built upon the JML checker. In fact, it shares the initial five phases of the checker, and then adds two RAC specific phases in a technique called *double-round compilation* [3, Section 1.5.1], as is illustrated in Figure 3.2 (refer from [3, Figure 1.2]). The first round is the JML checker phases (indicated by the while ovals), and the second round consists of the MultiJava compilation phases (represented by the grey ovals) of the internally generated instrumented Java code.

The phase "parsing" is used to scan and parse source code, creating an abstract syntax tree (AST). The parser is created by ANTLR (ANother Tool for Language Recognition) Parser Generator [31]. The phase "internalization" is used to process imported modules so that all the methods which belong to these external modules can be recognized by later phases. The phase "interface checking" is used to check the legality of interfaces, such as imported classes, module signature, methods, fields and so on. The phase "initializer checking" is used to check initializers in the AST. The phase "type checking" is used to check whether all the other parts in the AST have well-formed types. The phase "RAC code generation" is used to translate JML specifications in the AST into runtime assertion checking code. The phase "RAC code printing" is used to output the new AST created by the previous phase to a temporary file. The phase "code generation" is used to generate Java byte code from the AST.

3.3 Runtime Assertion Checking: How it is Done

In this section we describe some of the techniques used by the JML RAC to accomplish runtime assertion checking of JML specifications.

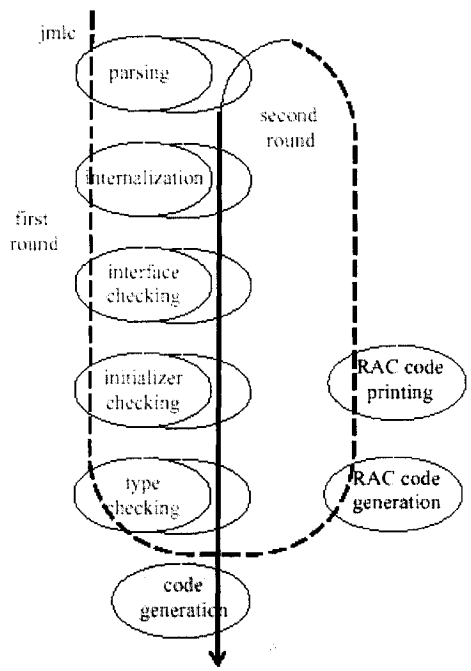

Figure 3.2 JML RAC phases

3.3.1 Undefinedness in Assertions

3.3.1.1 JML assertion semantics

In the current JML assertion semantics, assertions are interpreted as predicates in a *classical two-valued logic*. In such a logic,

partial functions are modeled as *underspecified total functions* [29, Chapter 3]. This means that partial functions like integer division and array element access are treated as total functions. Hence, for example, "3/0" is assumed to have an integer value, although which value it is remains unspecified. Similarly, if *a* is an array of Strings, then *a[0]* will be a value of type String when a is null, though we know nothing about the value. (Problems with this semantics have been identified by Chalin and an alternate semantics has been proposed [29, 35]. In this thesis, we mainly stick to the current JML semantics.)

3.3.1.2 Angelic and demonic undefinedness

In the context of the runtime checking of assertions, the application of partial functions to arguments outside its domain (e.g. like "3/x" when x is 0) result in runtime exceptions. In fact, when JML assertions are evaluated at runtime, two kinds of undefinedness (or exceptions) can arise: *demonic* and *angelic* [3, Section 3.2.2]. A *demonic* undefinedness is caused by exceptions such as the null pointer exceptions that might arise when, e.g., `theStack` is null in

```
//@ requires !( theStack.isEmpty() );
```

An *angelic* undefinedness is due to an attempt at evaluating non-executable specifications, such as informal assertion clause like

```
//@ requires !( theStack.isEmpty() ) && (* the stack depth >= 2 *);
```

In the sample shown above, if the subexpression *(* the stack depth >= 2 *)* is reached, an angelic undefinedness will be thrown.

The RAC does its best[1] to implement JML's classical (i.e. 2-valued) interpretation of assertions by using a technique called *local contextual interpretation*. With this

[1] See Section 6.1 on *neutral contexts*.

technique, any exception thrown during the evaluation of an assertion is caught. Then the smallest possible boolean subexpression, E_c, containing the call that caused the exception to be raised is determined. The RAC code then returns a boolean value for E_c that will allow the *overall* assertion containing the subexpression to be either false (in the case of demonic exceptions) or true (in the case of angelic exceptions) [29, Section 6.3].

In *local contextual interpretation*, expressions are interpreted *locally*, i.e. in the *context* of the smallest boolean expression that encloses an occurrence of an undefined subexpression. In some cases, undefinedness is meant to be mapped to true and in other cases to false. The key idea of this approach is to "think of runtime assertion checking as a game, and then apply an optimal strategy for selecting a value for undefinedness." [3, Section 3.2.3]. Let us consider another example. For instance, consider the effect of evaluating the assertions in the following two clauses:

```
//@ requires !( theStack.isEmpty() );                    //(1)
//@ ensures  theStack.isEmpty();                         //(2)
```

when `theStack` is null. Since `theStack` is null, the attempted method call of `isEmpty()` would result in a `NullPointerException`. This occurrence of demonic undefinedness is caught by the RAC code and the smallest boolean expression containing the exception is re-interpreted as either true or false depending on the context. For both assertions in our example, the smallest boolean expression enclosing the undefinedness is `theStack.isEmpty()`. But, `theStack.isEmpty()` is interpreted as true in (1) so as to render the precondition false; while in (2), the same sub-expression is interpreted as false so as to render the post-condition false. This approach is covered in detail in Yoonsik Cheon's thesis [3, Chapter 3].

Consider another sample assertion, namely `x.length > 5 || true` when x is null. When applying the approach of *local contextual interpretations,* the boolean subexpression `x.length > 5` will be interpreted as false. This will yield `false || true` leaving the overall assertion to be true. Similar remarks can be made of the expression `x.length > 5 | true`. This is simply a consequence of the choice of logical semantics currently adopted by JML. (We will return to this example in Chapter 6.)

3.3.2 Method Specifications

A "wrapper approach" is used by the RAC to handle method specifications. For example, let `m` be a method in a class `C` with a JML method specification, then effectively, `m` is renamed to `orig_m` and a new "wrapper" method is created. This wrapper method hides the actual method by having the same name and signature as the original one. The preconditions, post-conditions, invariants and constraints of the original method are encoded as special RAC methods called *assertion checking methods.* Even if a given specification element does not exist, an assertion checking method with an empty body will still be created. These checking methods are given names like `checkPre$m$C`, `checkPost$m$C`, etc.

Figure 3.3 (excerpt from [3, Figure 4.2, 5.1]) shows the order in which the wrapper method invokes the checking methods and the original method. Details of this approach are given in [3, Chapters 4-5].

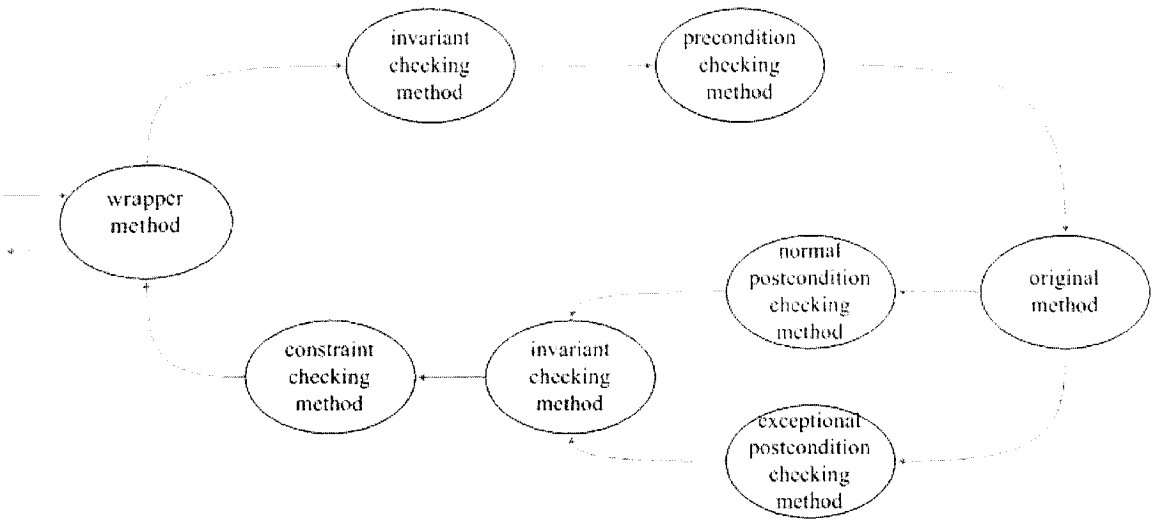

Figure 3.3 Order of method calls in a wrapper method

3.3.3 Specification Inheritance

In JML, a subclass inherits specifications from the super classes it extends as well as from the interfaces it implements. As both super classes and interfaces can have their own specifications (potentially for the same methods), it makes specification inheritance one of the challenges for run time assertion checking.

In the RAC, a *delegation* approach is used to support specification inheritance in JML [30]. This means that all super types create corresponding assertion checking methods for their specifications. These methods will be invoked by subtypes to achieve behavioral subtyping. The call to these methods needs to be done dynamically because the super types might not have specifications. Technically, Java reflection facility is used to implement the dynamic call. Further details can be found in [3, Chapter 6].

3.3.4 Specification-only Attributes

To provide a runtime realization of specification-only members, such as model fields and ghost fields in JML (see Section 2.4.3), the RAC makes use of *access methods*.

For a model field, an access method is generated and then a call to this access method is substituted for every occurrence of the model field in a JML expression. The access method body is defined by the model field's *represents* clause. For example, let's suppose a model field is represented as

```
represents m <- E;
```

in which m is a model field, S is the module which contains m, and *T* is the type of m. Then, the access method generated by the RAC is shown in Figure 3.4 (excerpt from [3, p.101]).

```
public T model$m$S() {
   T rac$v;
   C [[E]] (rac$v, false)
   return rac$v;
}
```

Figure 3.4 Model field access method

For a ghost field, two access methods are formed, one is called *ghost field getter method*, which returns the value of this field and replaces each appearance of the field, the other one is called *ghost field setter method*, which is created by the set statement to set the value of this field. For example, to process a ghost field

```
private T ghost$v;

public T ghost$v$S() {
return ghost$v;
}

public void ghost$v$S(T v) {
ghost$v = v;
}
```

Figure 3.5 Ghost field access methods

```
ghost T v;
```

which is owned by a module S, the RAC generates code as shown in Figure 3.5 (excerpt from [3, Figure 7.10]).

3.4 RAC Semantics

The interpretation of JML specifications during runtime checking is given by means of an (large step) operational semantics. In an operational semantics, language elements are most often mapped into "code" of an abstract machine [32]. In the case of the RAC, a JML specification element is mapped into the Java code that will be used to evaluated it at runtime.

In this thesis, the semantics are presented using a format based on the one used by Cheon in his Ph.D. thesis on the RAC [3, Chapter 3]. We will be solely interested in translation rules / functions which, for our purposes, will all have the following signature:

$$C\ [\![\text{Expression}]\!] : \text{Identifier} \times \text{boolean} \rightarrow \text{Program}$$

where, *Expression* denotes a JML expression to be translated; *Identifier* is the name of the identifier in the resulting program that will be used to store the result of *Expression*; the *boolean* is a parameter which is used to provide contextual information about how the expression should be interpreted if its evaluation should result in demonic or angelic undefinedness (cf. Section 3.3.1).

For example, C ⟦ *E1* == *E2* ⟧ (*r*, *p*) represents a sequence of Java statements that will evaluate the JML assertion `E1 == E2`, in which the result will be saved to the variable *r*. The boolean variable *p* is used to indicate the positive or negative "contextual value",

```
C [[E1 == E2]] (r, p):=
try {
    int v1 = 0;
    int v2 = 0;
    C [[E1]] (v1, p)
    C [[E2]] (v2, p)
    r = v1 == v2;
} catch (JMLAngelicException e) {
    r = !p;
} catch (Exception e) {
    r = p;
}
```

Figure 3.6 Sample semantic function for '=='

i.e., p is the value that is to be returned when a demonic exception (cf. Section 3.3.1) occurs, so as to make the overall assertion containing the sub-expression *E1* == *E2* false. Similarly, $!p$ is the value that is returned when an angelic exception (cf. Section 3.3.1) occurs, so as to make the overall assertion containing the equality sub-expression true.

Figure 3.6, based on [3, Figure 3.1], shows the translation rule for == when the *Ei* are integer expressions. The main part of the translated program code is a try-catch block, which is only needed for a boolean expression. In the *try* block, two local variables *v1* and *v2* are declared and assigned to initial value *0*. The following two denotations are used to translate the sub-expressions *E1* and *E2*, whose results are stored in *v1* and *v2* correspondingly. The last statement in this block is to save the final result to *r*. The use of the first *catch* block is to catch any possible angelic undefinedness and return $!p$ in that case. The use of the second *catch* block is to catch a possible demonic undefinedness and make the overall assertion false.

In the rule shown above, *r* must be declared before this rule is applied and p's value is determined from the context of the expression *E1* == *E2*. For example, given the assertion

```
//@ assert !(E1 == E2);
```

then the value of p will be true. In this way, if any exception happens during calculating the value of *E1* and *E2*, the value of expression *E1* == *E2* will be true, and the value of the whole assertion will be false. On the other hand, given the assertion

```
//@ assert E1 == E2;
```

the value of p will be false. In the same way, when any exception happens, the value of the whole assertion will be false. An assertion being false indicates that a specification is violated. As a result, RAC code will raise an assertion violation exception.

As another example, consider a logical complement expression $!E$, its translation rule C $[\![!E]\!]$ (r, p) is shown in Figure 3.7. An important difference between Figure 3.6 and Figure 3.7 is the later one doesn't have a try-catch block in the translated code. That is because according to the approach

local contextual interpretation (see Section 3.3.1), only the smallest boolean expression needs to catch potential undefinednesses. Another difference in Figure 3.7 is that the sub translation rule `C [[E]] (b, !p)` uses $!p$ as the translation contextual variable, which can make the top-level expression false when any demonic exception happens. For changes to translation contextual variables, see [3, Table 3.1].

```
C [[!E]] (r, p):=
    boolean b = false;
    C [[E]] (b, !p)
    r = !b;
```

Figure 3.7 Semantic function for '!'

Note that such translation rules will be used extensively in Chapter 4.

3.5 RAC Design

This section presents an overview of the overall organization of the JML RAC. This will help to set the context for the explanation of the changes that were performed in the thesis work. The JML RAC is built upon the JML type checker. Both of these tools are based on the MultiJava compiler. MultiJava extends Java by adding open classes and

symmetric multiple dispatch [14]. Highlights of the overall design of the JML RAC are given in Figure 3.8 to Figure 3.14.

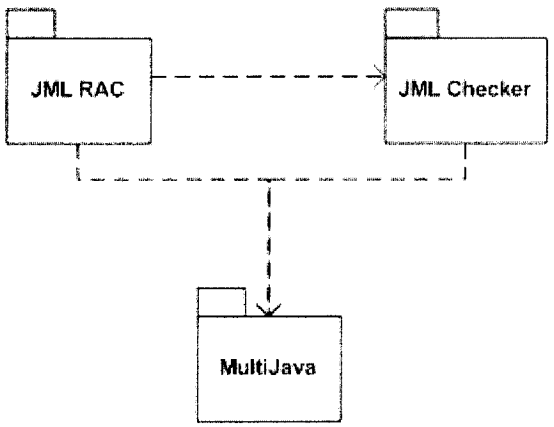

Figure 3.8 JML RAC: overall tool interdependencies

Figure 3.8 shows the relationship among the JML RAC, Checker and MultiJava compiler. The classes and interfaces in the RAC depend on many modules defined in MultiJava and the JML Checker. Furthermore, some classes in the RAC are subclasses of the ones declared in MultiJava.

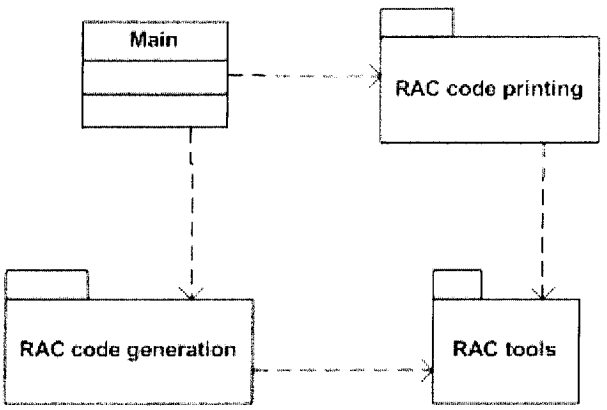

Figure 3.9 RAC: overall design

Figure 3.9 illustrates the overall organization of the RAC into its top-level packages. The two most important packages are "RAC code generation" and "RAC code printing", which correspond to the two RAC phases given in Figure 3.2 on page 20.

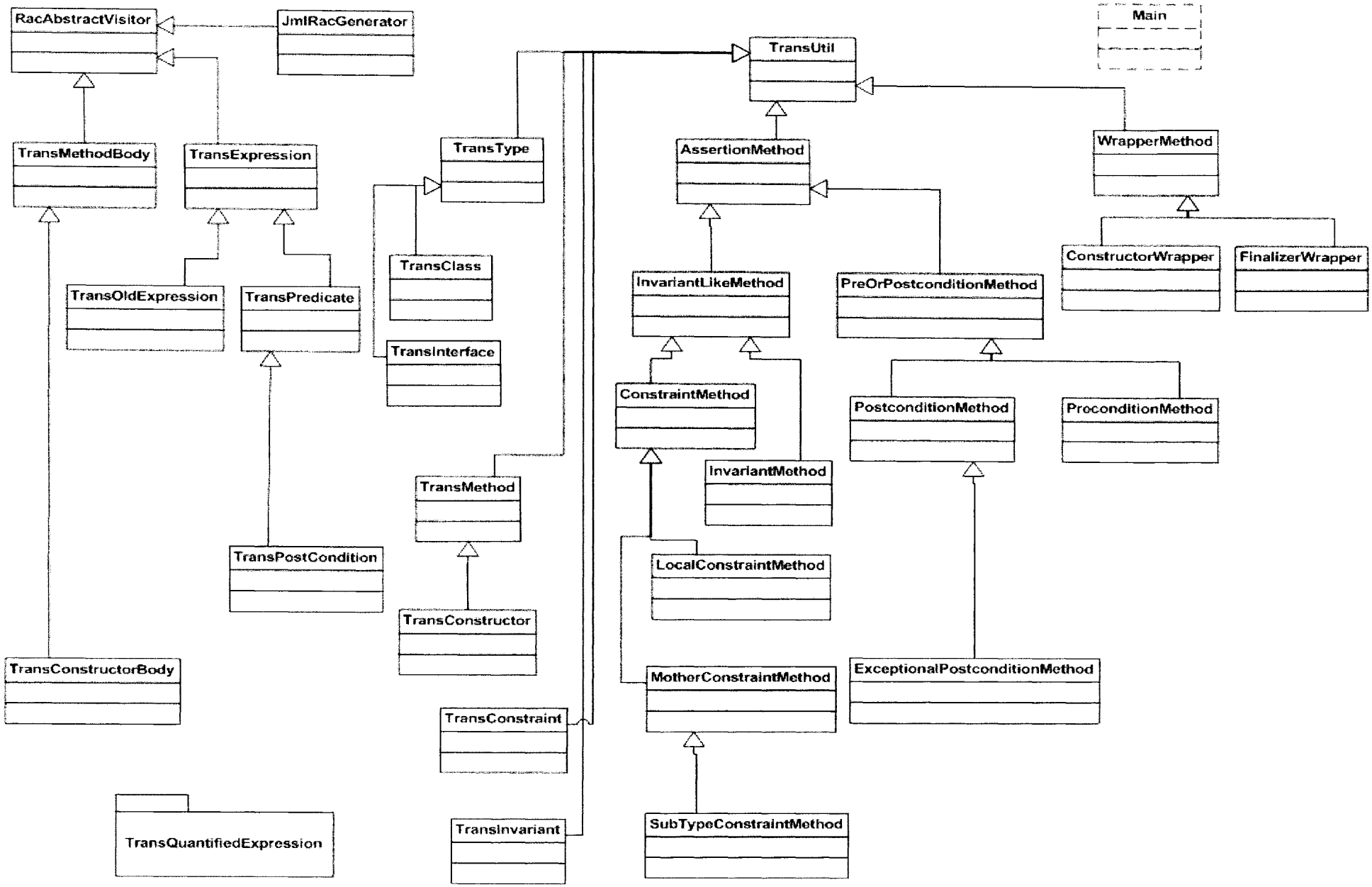

Figure 3.10 RAC code generation package inheritance hierarchy

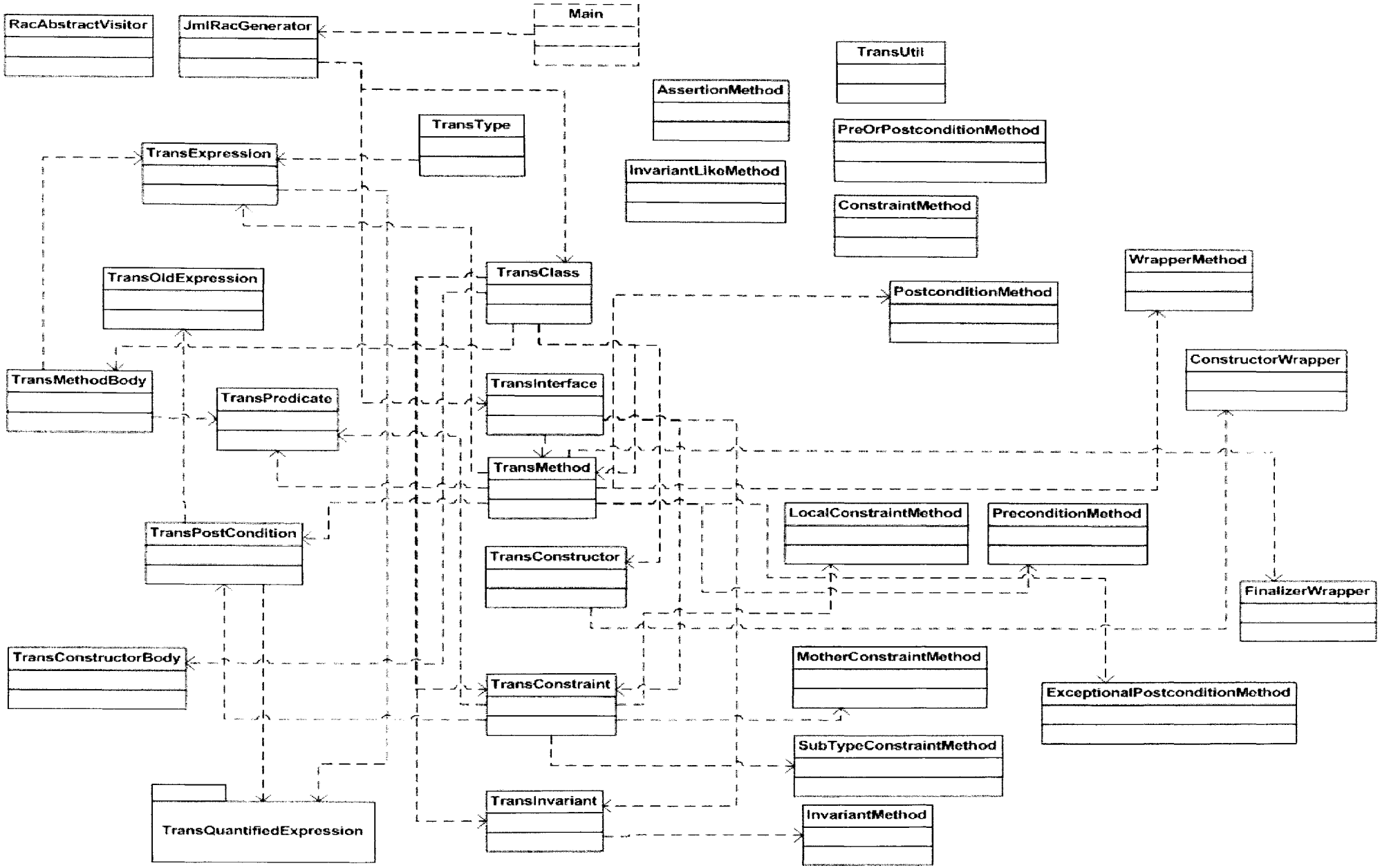

Figure 3.11 RAC code generation package: module interdependencies

The inheritance hierarchy and module interdependencies for the RAC code generation package are given in Figure 3.10 and Figure 3.11 respectively. The classes in these two figures can be divided into the following logical groups:

- `Trans*` classes which translate the various kinds of JML specification elements into their runtime representation. For example, the class `TransConstraint` is used to translate constraints and the class `TransMethod` is used to translate Java methods (together with their corresponding JML specifications).
- class `AssertionMethod` and all its subclasses are used to generate assertion checking methods for preconditions, post-conditions, class invariants and constraints (see Section 3.3.2).
- class `WrapperMethod` and all its subclasses are used for wrapping a method with assertion checking methods as was explain in Section 3.3.2.
- other miscellaneous classes.

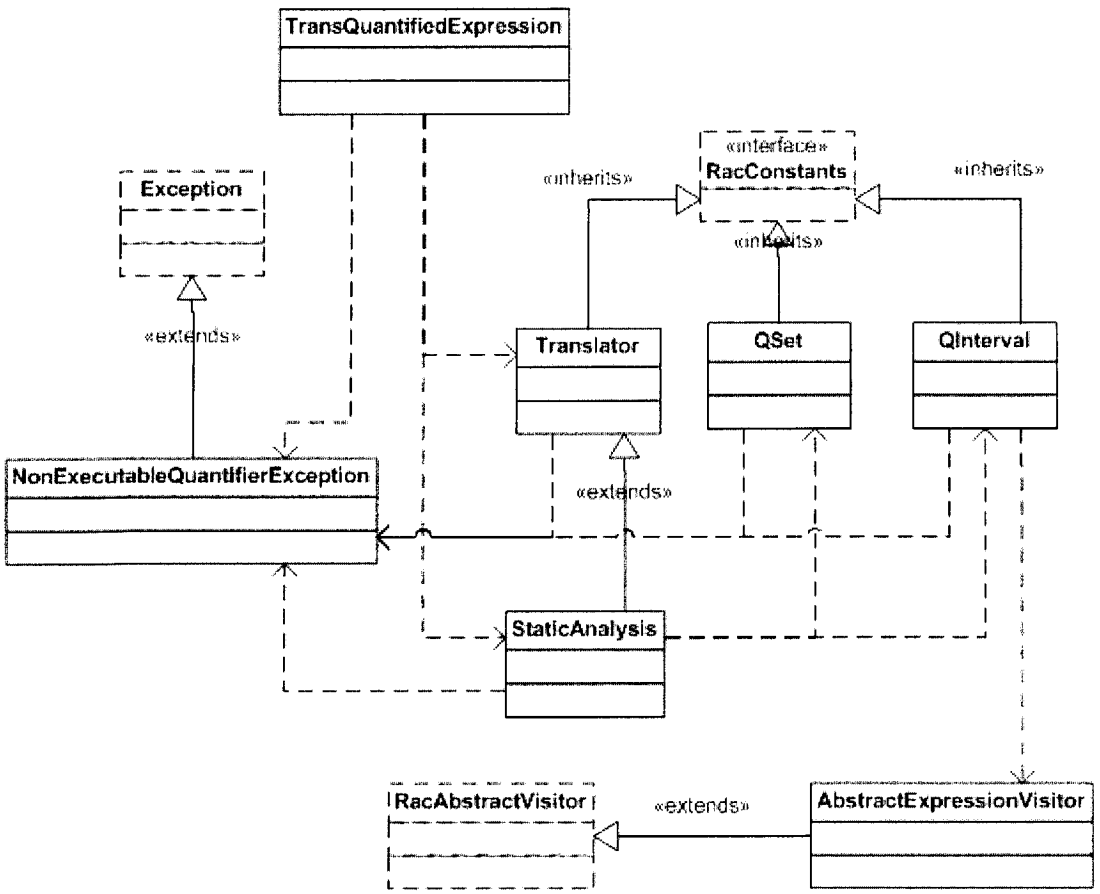

Figure 3.12 Class Diagram for the `qexpr` package

In the RAC, there is a package named *qexpr*, which is used to translate quantified expressions. Its classes and interfaces are illustrated in Figure 3.12. This package corresponds to the package `TransQuantifiedExpression` in Figure 3.10 and Figure 3.11. The main modules of this package are:

- `QSet` used to get the set of all objects which need to be checked while calculating the value of a quantified expression with reference variable(s).
- `QInterval` used for getting the interval while calculating the value of a quantified expression. Apparently, it is only used for integral types.

- `StaticAnalysis` to analyze the structure of a quantified expression using an approach named *pattern-based static analysis approach* [3, Section 3.3.3] to translate the expression into Java source code.

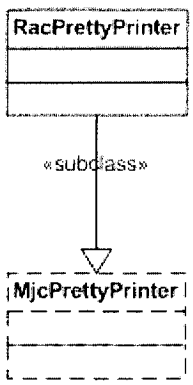

Figure 3.13 Class Diagram for the RAC code printing package

In Figure 3.13, we can see that the RAC code printing package contains only one class which is a subclass of the MultiJava `MjcPrettyPrinter` class. This class is a typical Visitor [33] for "visiting" JML-annotated Java abstract syntax trees.

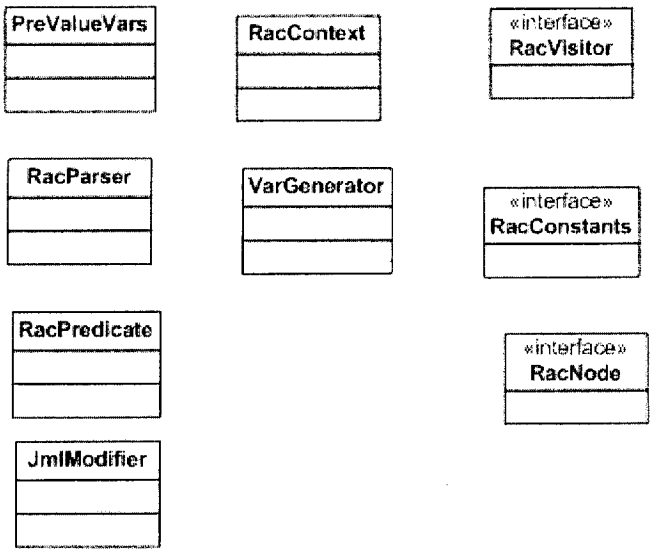

Figure 3.14 Classes & Interfaces of the design RAC tools package

The above figure shows the modules in the design RAC tools package.

4 Supporting Arbitrary Precision Integers

4.1 Introduction

Chalin recently proposed that JML be extended to support arbitrary precision numbers [21]. We briefly demonstrate the motivation behind this extension using a simple example.

```
/*@ public normal_behavior
  @ requires y >= 0;
  @ assignable \nothing;
  @ ensures 0 <= \result
  @     && \result * \result <= y
  @     && y < ((\result + 1) * (\result + 1));
  @*/
public static int isqrt(int y)
```

Figure 4.1 JML specification of isqrt(int)

The specification shown in Figure 4.1 for an integer square root function is actually invalid. That is, when the parameter of `isqrt(int)` is 0 then it permits the return value of `isqrt` to be `Integer.MIN_VALUE`. This is due to the modulo arithmetic of Java for integral types; e.g. the following expression holds in Java:

```
Integer.MIN_VALUE * Integer.MIN_VALUE == 0
```

The reason for this unexpected situation is that when an overflow happens to a integral number, Java just ignores the overflowing. One way to get rid of such potential errors in JML is to use the model type `JMLInfiniteInteger` provided in JML, but the specifications using it become verbose (see [21] Figure 4). The other way, which, was finally adopted by JML, is creating a new primitive numeric type `\bigint` which represents the type of arbitrary precision integers.

As the precision of `\bigint` is infinite, `\bigint` is placed at the top of JML integral numeric type hierarchy as shown in Figure 4.2 [21].

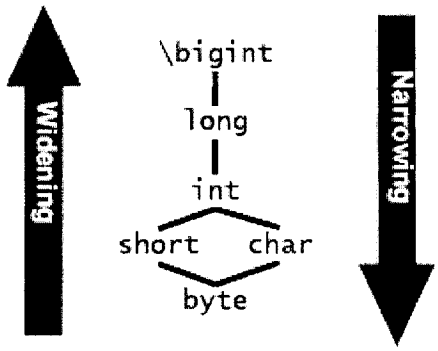

Figure 4.2 Integral numeric type hierarchy in JML

Next we explain the enhancement made to enable the JML checker and RAC to support `\bigint` by defining formal translation rules using the notation presented in Section 3.4.

4.2 Semantics of Primary Expressions

In summary, what needed to be done was to modify all the expressions whose sub-expression's type might be `\bigint`. In the rest of this section we present the adapted semantic rules used to support `\bigint`. All the expressions in the RAC are shown in Appendix 10.

4.2.1 `\bigint` Literal Expression

`\bigint` literal expression is a literal whose type is `\bigint`. Its semantic function is shown in Figure 4.3. Note that the lexeme "v" will be a legal Java integer constant (either in the format of an `int` or `long`).

```
C [[v]] (r) :=
    r = java.math.BigInteger.valueOf(v);
```

Figure 4.3 Semantic function for literals

4.2.2 Local Variable Expression

The semantic function for local variables is given in Figure 4.4 where the symbol := means "is defined as", and the symbol "*?* <condition>" is used to indicate a condition that must hold for that particular rule instance to be applicable.

```
C [[v]] (r) := ? if neither v's type nor r's type is \bigint || both v's type and r's type are \bigint
    r = v;

C [[v]] (r) := ? if v's type is \bigint  (but r's type is not)
    r = v.longValue();


C [[v]] (r):= ? otherwise          // r's type is \bigint although v's is not
    r = java.math.BigInteger.valueOf(v);
```

Figure 4.4 Semantic function for local variable expressions

4.2.3 Unary Expressions

The syntax of unary expressions is as follows.

UnaryExpression: UnaryOperator Expression
UnaryOperator: one of + - ~

The semantic function for unary expressions is given in Figure 4.5. In the function definition, *v1* is a local variable which will be used to store the result of *E1*; *T1* is the type of a variable *E1*.

```
C [[uop E1]] (r, p) :=
    Decl(v1, T1)
    C [[E1]] (v1, p)
    r = APP_UOP (uop, v1, T1)
```

Figure 4.5 Semantic function for unary expressions

where the definition of APP_{UOP} (uop, v1, T1) is shown in Table 4-1 and the definition of Decl(v,T) is given in Table 4-2.

Uop	T1	Program
+	Any	v1
-	\bigint	v1.negate()
	Else	-v1
~	\bigint	v1.not()
	Else	~v1

Table 4-1 Definition of APP_{UOP}

T	Program
\bigint	Java.lang.BigInteger v = java.lang.BigInteger.ZERO;
other	T v = 0;

Table 4-2 Definition of Decl(*v*,*T*)

4.2.4 Binary Expressions

The syntax of binary expressions is:

BinaryExpression: ExpressionName BinaryOperator ExpresssionName
BinaryOperator: one of > < >= <= == != + - * / % << >> >>> & | ^

The semantic function for binary expressions is given in Figure 4.6. As was explained previously, the v*i* are local variables used to hold the values of the E*i*. T*i* are the types of E*i*.

```
C [[E1 bop E2]] (r,p) :=
    Decl(v1, T1)
    Decl(v2, T2)
    C [[E1]] (v1, p)
    C [[E2]] (v2, p)
    r = APP_BOP (bop, v1, v2, T)
```

Figure 4.6 Semantic function for binary expression

where the definition of APP_{BOP}(*bop, v1, v2, T*) is given in Table 4-3.

Bop	T	Program
>	\bigint	v1.compareTo(v2) > 0
	Else	v1 > v2
<	\bigint	v1.compareTo(v2) < 0
	Else	v1 < v2
>=	\bigint	v1.compareTo(v2) >= 0
	Else	v1 >= v2
<=	\bigint	v1.compareTo(v2) <= 0
	Else	v1 <= v2
==	\bigint	v1.compareTo(v2) == 0
	Else	v1 == v2
!=	\bigint	v1.compareTo(v2) != 0
	Else	v1 != v2
+	\bigint	v1.add(v2)
	Else	v1 + v2
-	\bigint	v1.add(v2 .negate())
	Else	v1- v2
*	\bigint	v1.multiply(v2)
	Else	v1 * v2
/	\bigint	v1.divide(v2)
	Else	v1 / v2
%	\bigint	v1.remainder(v2)
	Else	v1 % v2
<<	\bigint	v1.shiftLeft(v2)
	Else	v1 << v2
>>	\bigint	v1.shiftRight(v2)
	Else	v1 >> v2
>>>	\bigint	fail()
	Else	v1 >>> v2
&	\bigint	v1.and(v2)
	Else	v1 & v2
|	\bigint	v1.or(v2)
	Else	v1 | v2
^	\bigint	v1.xor(v2)
	Else	v1 ^ v2

Table 4-3 Definition of APP_{BOP}

4.2.5 Cast Expression:

The abstract syntax and semantic function of cast expressions are illustrated in the following.

CastExpression: (Type)ExpressionName

```
C [[(T)E1]] (r, p):=
    Decl(v1, T1)
    C [[E1]] (v1, p)
    r = APP_CAST (T, v1)

APP_CAST (T, v1) := ? if both T and T1 are \bigint
    v1;

APP_CAST (T, v1) := ? if neither T nor T1 is \bigint
    (T)v1;

APP_CAST (T, v1):= ? else if T1 is \bigint
    (T)v1.longValue();

APP_CAST (T, v1):= ? otherwise
    java.math.BigInteger.valueOf(v1);
```

Figure 4.7 Semantic functions for cast expressions

4.2.6 Unary Promotion Expression

Unary Promotions are one of the following widening primitive conversions:

- byte to short, int, long, `\bigint`, float, or double
- short to int, long, `\bigint`, float, or double
- char to int, long, `\bigint`, float, or double
- int to long, `\bigint`, float, or double
- long to `\bigint`, float or double
- float to double

Figure 4.8 shows the semantic function for unary promotion expression where E1 is being promoted to type T. This function differs from that of cast expression because the type *T* must be widen than the type of *E1*.

```
C [[(T)E1]] (r, p) :=
    Decl(v1, T1)
    C [[E1]] (v1, p)
    r = APP_UPROM (T, v1)


APP_UPROM (T, v1):= ? if T is not \bigint
    (T)v1;

APP_UPROM (T, v1):= ? otherwise
    java.math.BigInteger.valueOf(v1);
```

Figure 4.8 Semantic function for unary promotion expressions

4.3 Semantics of Quantified Expressions

Quantified expressions have a form like "Q T v1...vn; E1; E2" in which Q represents the quantifier or another form of binding operator. In JML, the binding operators are `\forall`, `\exists`, `\max`, `\min`, `\product`, `\sum`, and `\num_of`. Such expressions can be divided into two categories, one is for reference types, and the other is for numeric types. Only the one for numeric types need be changed for supporting `\bigint`. The semantic function for quantified expressions is shown in Figure 4.9[2]. *Tq* is the type of the quantified expression (i.e. `boolean` for \forall and \exists and `int` otherwise). The value of *loopCondPrefix*(*qop*) is "r" for \forall and \exists and "true" otherwise.

[2] Derived from comments in file "JML2/org/jmlspecs/jmlrac/qexpr/StaticAnalysis.java".

```
C ⟦(qop T v1,...vn; E1; E2)⟧  (r, p) := ? if T is one of byte, char, short, int, long, \bigint
try {
    Tq r = initResVal(qop, Tq);
    boolean bFirstCompare = true;          // used only for \min and \max.
    boolean bEmptyRange = true;T lB1 = initBoundVal(T);            //lBi: the
    lower bound of variable i
    T uB1 = initBoundVal(T);               //uBi: the upper bound of variable i

     ⟦calculate lB1 and uB1 by E1 ⟧

    while(loopCondPrefix(qop) &&  lessThan(lB1, uB1, T) ) {
        bEmptyRange = false;
        v1 = lB1;
        ...
        T lBn = initBoundVal(T);           //lBi: the lower bound of variable i
        T uBn = initBoundVal(T);           //uBi: the upper bound of variable i

         ⟦calculate lBn and uBn by E1⟧

        while(loopCondPrefix(qop) && lessThan(lBn, uBn, T) ) {
             vn = lBn;
             APP_QOP (qop, E1, E2, r, p)
             lBn = (T is \bigint then lBn.add(java.math.BigInteger.ONE)  else lBn +
             1);
        }
        ...
        lB1 = (T is \bigint then lB1.add(java.math.BigInteger.ONE) else lB1 + 1);
    }
    if(bEmptyRange) { throw empty range exception.}
} catch (JMLAngelicException e) {
    r = !p;
} catch (Exception e) {
    r = p;
}
```

Figure 4.9 Semantic function for quantified expressions

The definitions of *initBound* and *lessThan* are shown in Figure 4.10, the definition of *initResVal* is given in Table 4-4, and the definition of APP_{QOP} is illustrated in Table 4-5 to Table 4-8.

```
initBoundVal(T) := T is \bigint ?
    java.math.BigInteger.ZERO
    : 0;

lessThan(l, u, T) := T is \bigint ?
    l.compareTo(u) < 0 : l <
    u
```

Figure 4.10 Definition of initBound and lessThan

Qop	Tq	Program
\forall		True
\exists		False
\sum	\bigint	java.math.BigInteger.ZERO
	Else	0
\product	\bigint	java.math.BigInteger.ONE
	Else	1
\min	\bigint	java.math.BigInteger.ZERO
	Else	0
\max	\bigint	java.math.BigInteger.ZERO
	Else	0
\num_of	\bigint	java.math.BigInteger.ZERO
	Else	0

Table 4-4 Definition of initResVal

Note in Figure 4.9, a flag variable *bEmptyRange* is used to decide if the range is empty. One more thing is that the initial value for \min and \max is actually irrelevant.

qop	Program
\forall	C ⟦E1=>E2⟧ (r, p)
\exists	C ⟦E1&&E2⟧ (r, p)

Table 4-5 Definition of APP_{QOP} for \forall and \exists

qop	Program

\sum	boolean b = false;
	C ⟦E1⟧ (b, p)
	if(b) {
	  T2 x2 = initT2;
	  C ⟦E2⟧ (x2, p)
	  r = r + x2;
	}

\product	boolean b = false;
	C ⟦E1⟧ (b, p)
	if(b) {
	  T2 x2 = initT2;
	  C ⟦E2⟧ (x2, p)
	  r = r * x2 ;
	}

Table 4-6 Definition of APP_{QOP} for \sum and \product

qop	Program

\min	boolean b = false;
	C ⟦E1⟧ (b, p)
	if(b) {
	T2 x2 = initT2;
	C ⟦E2⟧ (x2, p)
	if(bFirstCompare) {
	r = x2 ;
	} else {
	r = r's type is \bigint ?
	r.min(x2); :
	java.lang.Math.min(r, x2);
	}
	bFirstCompare = false;
	}

\max	boolean b = false;
	C ⟦E1⟧ (b, p)
	if(b) {
	T2 x2 = initT2;
	C ⟦E2⟧ (x2, p)
	if(bFirstCompare) {
	r = x2 ;
	} else {
	r = r's type is \bigint ?
	r.max(x2); :
	java.lang.Math.max(r, x2);
	}
	bFirstCompare = false;
	}

Table 4-7 Definition of APP_{QOP} for \min and \max

qop	Program

\num_of	boolean b1 = false;
	C ⟦E1⟧ (b1, p)
	boolean b2 = false;
	C ⟦E2⟧ (b2, p)
	if(b1 && b2) {
	r's type is \bigint ?
	r.add(java.math.BigInteger.ONE) :
	r++;
	}

Table 4-8 Definition of APP_{QOP} for \num_of

For \min expression (see Table 4-7), as there is no minimum integral value, when the body of the while loop is executed at the first time, I set *r* with the value of *E2*. In the following execution of this body, I compare *r* and *E2*, then set *r* with the value of *E2* if *E2* is less than *r*. In this way, when the while loops end, the value of *r* will be the minimum value in the range. A flag variable *bFirstCompare* is used to indicate the first time when the while loop should be entered. The \max expression works in the same way.

4.4 Semantics of Model Fields

Model fields are explained in Section 2.4.3. The representation of a model field `f` is given in the form of a represents clause like this

```
represents f <- E
```

The semantic function defining the body of the field's access method (Section 3.3.4) is shown in Figure 4.11. T_f stands for the type of `f`, and T_E means the type of E..

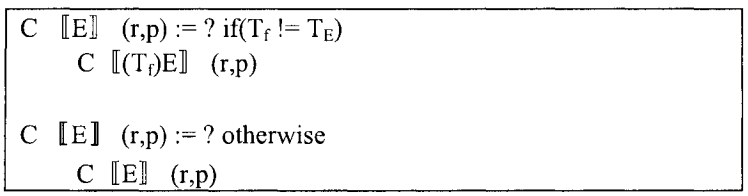

Figure 4.11 Semantic function for model field access method body

4.5 Testing

One or two test cases are created per kind of expression to test their support for `\bigint`. Some of the test cases are created in a new test file, while others are added to existing files. All the test files are put in subdirectories of *JML2/org/jmlspecs/jmlrac/testcase/*. There are three subdirectories in all. One is *racgen* containing files which are used to test Java code generated by the RAC. Another one is *racopt* grouping files used to test command-line options. The last subdirectory is *racrun* containing files used test behaviors of the byte code created by the RAC during runtime. One test case is given in Figure 4.12.

```
import org.jmlspecs.jmlrac.runtime.JMLAssertionError;

/*@spec_bigint_math@*/
public class arithmetic_bigint_exp
{
    /*@ normal_behavior
    @  requires Long.MAX_VALUE < Long.MAX_VALUE + 1;
    @  assignable \nothing;
    @  ensures  true;
    @*/
    void skipInt1pre() { }

    /*@ normal_behavior
    @  assignable \nothing;
    @  ensures  Long.MIN_VALUE - 1 < Long.MIN_VALUE;
    @*/
    void skipInt1post() { }

    /*@ normal_behavior
    @  requires Long.MIN_VALUE != Long.MAX_VALUE + 1;
    @  assignable \nothing;
    @  ensures  !(Long.MAX_VALUE == Long.MIN_VALUE - 1);
    @*/
    void skipInt2() { }

    public static void main(String args[]) {
        arithmetic_bigint_exp exp = new arithmetic_bigint_exp();
        int fcnt = 0;

        try {
            exp.skipInt1pre(); exp.skipInt1post(); exp.skipInt2();
        }
        catch (JMLAssertionError e) { fcnt++; }

        if (fcnt > 0) {
            System.out.println(fcnt + "F(arithmetic_bigint_exp.java)");
        }

    }
```

Figure 4.12 Sample test case for supporting \bigint

Test cases like these are compiled using the RAC. Execution of the test case invokes the main method which instantiates the class and then calls the test methods. If errors happen in the code, then JMLAssertionErrors will be raised. These are caught and the number of errors are counted in the counter *fcnt*; if non-zero on termination then an error message is printed – such messages are interpreted by the testing framekwork..

4.6 Implementation

The places modified to implement the support for `\bigint` are the packages *RAC code generation* and *RAC code printing* (see Figure 3.9). In the first package, the files impacted are `Trans*` ones (see Figure 3.10, Figure 3.11) which are used to translate all

kinds of JML expressions into Java code. In the second package, the file (see Figure 3.13) used to output the RAC generated Java code is impacted.

5 Supporting \bigint in Model Method Bodies

This chapter explains the changes made to the RAC to enable support for \bigint in model method bodies. A sample model method is shown in Figure 5.1. A model method can only be declared and invoked in specifications. In the following figure, the model method is *testPlusPlusI* which is declared by the keyword *model* and used in the precondition of method *mPlusPlusI*. To process a model method, the RAC will simply remove the annotation marks, and treat it as a member method in the generated code. If there are \bigint types used in its body, some changes have to be done, as shown herein. For more details, see [3, Section 7.5].

```
public class PrePostfixExp {

    /*@
    @ public static pure model int testPlusPlusI(int i) {
    @ return ++i;
    @ }
    @*/

    //@ requires testPlusPlusI(I) == Iplus1;
    public void mPlusPlusI(int I, int Iplus1) { }
}
```

Figure 5.1 A sample for model method

5.1 Field Expressions

The syntax of field expressions is as follows

Field Expression:
Identifier
ClassName . Identifier
super . Identifier
ClassName . super . Identifier

The semantic function for a field expression in a model method is demonstrated in Figure 5.2 where *S* is the module defining the model method.

```
C [[id]] (r) :=? id is a model field of the class S
     r = model$id.$S()"
C [[prefix.id]] (r) := ? id is a model field of the class S
     r= prefix.model$id. $S()
```

Figure 5.2 Semantic function for field expression in model method

The semantic function translates a given field expression into a call of the appropriate access method. For example, as results of this translation rule, C [[f]] will be *model$f.$S()*; C [[S2.f]] will be *S2.model$f.$S()*; C [[super.f]] will be *super.model$f.$S()*, if *f* is a model field.

5.2 Assignment Expressions

The abstract syntax of assignment expressions in model method is illustrated in the following:

Assignment Expression: LefHandSideExpression = Expression

The semantic function for assignment expressions is shown in Figure 5.3, and the definition of APP_{ASSMM} (v1, v2, T1, T2) is given in Table 5-1.

```
C [[E1 = E2]] (r, p) :=
     Decl(v1, T1)
     Decl(v2, T2)
     C [[E1]] (v1, p)
     C [[E2]] (v2, p)
     r = APP_ASSMM (v1, v2, T1, T2) ;
```

Figure 5.3 Semantic function for assignment expressions in model method

T1	T2	Program
\bigint	Not \bigint	v1 = BigInteger.valueOf(v2)
int	\bigint	v1 = v2.intValue()
long	\bigint	v1 = v2.longValue()
Else		v1 = v2

Table 5-1 Definition of APP_{ASSMM}

5.3 Compound Assignment Expressions in model method

The abstract syntax of compound assignment expressions in model method is shown as:

CompoundAssignmentExpression:
LeftHandSideExpression CompoundAssignmentOperator Expression
CompoundAssignmentOperator: one of
*= /= %= += -= <<= >>= >>>= &= ^= |=

The semantic function for compound assignment expressions in model method is given in Figure 5.4 where *preOp* is the corresponding operator in compound assignment.

```
C [[E1 preOp= E2]] (r, p) :=
    C [[E1 preOp E2]] (E1, p)
    r = E1;
```

Figure 5.4 Semantic function for compound assignment expressions in model method

5.4 Unary Expressions in model method

The abstract syntax of unary expressions in model method is:

UnaryExpressionMM:
PrefixUnaryOperatorMM ExpressionName
ExpressionName PostUnaryOperatorMM
PrefixUnaryOperatorMM: one of
++ -- + - ~
PostfixUnaryOperatorMM: one of
++ --

Notice that unary operators in method bodies (as opposed to assertion expressions) can contain prefix and postfix increment and decrement operators. The semantic function for unary expressions is shown in Figure 5.5, and the definition of $APPU_{OPMM}$ (uop, v1, T1) is shown in Table 5-2.

```
C [[uop E1]] (r, p) :=
    Decl(v1, T1)
    C [[E1]] (v1)
    r = APP_UOPMM (uop, v1, T1)
```

Figure 5.5 Semantic function for unary expressions in model method

uop	T1	Program
Prefix ++	\bigint	JMLRacBigIntegerUtils.value(v1 = v1.add(java.math.BigInteger.ONE));
	Else	++v1
Prefix --	\bigint	JMLRacBigIntegerUtils.value(v1 = v1. subtract(java.math.BigInteger.ONE));
	Else	--v1
Postfix ++	\bigint	JMLRacBigIntegerUtils.first(v1, v1 = v1.add(java.math.BigInteger.ONE));
	Else	v1++
Postfix --	\bigint	JMLRacBigIntegerUtils.first(v1, v1 = v1. subtract(java.math.BigInteger.ONE));
	Else	v1--
The definition of other unary expressions are the same as the ones in Table 4-1		

Table 5-2 Definition of APP_{UOPMM}

For this case, a runtime class `JMLRacBigIntegerUtils` was created, which has two major methods *value*() and *first*(). The function of method *value*() is to return its parameter. The function of method *first*() is to return its first parameter. The creation of this class is necessary, because an expression cannot be used where a statement is expected while an invocation to a method can. For example, the prefix expression *++bi,* where bi's type is `\bigint`, should be translated to *(bi = bi.add(BigInteger.ONE)).* But if ++bi itself is a statement, this translation will be wrong, while a method invocation (like value()) is OK. As a result, the class `JMLRacBigIntegerUtils` is created, and the expression *++bi* is translated into *JMLRacBigIntegerUtils.value(bi = bi.add(BigInteger.ONE)).* For the detail of class `JMLRacBigIntegerUtils`, see Figure 5.6.

Actually, there is still a defect in this approach. For example, in the following codes

```
int i;
\bigint a[];
...
a[i++]++;
...
```

the statement `a[i++]++` will be translated to

```
a[i++] = JMLRacBigIntegerUtils.first(
                        a[i++],
                        a[i++].add(java.math.BigInteger.ONE) );
```

after this translation, the index *i* is added twice, which is not correct at all. One solution to fix this issue is to report an error when a pre(post)-fix expression contains a pre(post)-fix sub-expression. But it needs to change a lot in Multi Java; it is not adopted now.

```
//This file is in the directory of "JML2/org/jmlspecs/jmlrac/runtime"

package org.jmlspecs.jmlrac.runtime;
import java.math.BigInteger;

public class JMLRacBigIntegerUtils {

    // ---------------------------------------------------------------------
    // CONSRUCTORS AND FACTORY METHODS
    // ---------------------------------------------------------------------

    /**
    * Constructs a new instance. */
    private JMLRacBigIntegerUtils() {
    }

    // ---------------------------------------------------------------------
    // ACCESSORS
    // ---------------------------------------------------------------------

    /**
    * This method is used in the RAC implementation of prefix ++ and
    * -- over \bigint's. Given \bigint bi, we translate ++bi into
    * value(bi = bi.add(BigInteger.ONE)).  The reason that ++bi is
    * not simply transated into (bi = bi.add(BigInteger.ONE)) is that
    * expressions cannot be used where a statement is expected where
    * as a method call like value(...) can.
    */
    //@ ensures \result == i;
    public /*@pure@*/ static BigInteger value(BigInteger i) { return i; }

    /**
    * This method is used in the RAC implementation of postfix ++ and
    * -- over \bigint's. E.g. given \bigint bi, we translate bi++ into
    * first(bi, bi = bi.add(BigInteger.ONE)).
    */
    //@ ensures \result == i;
    public /*@pure@*/ static BigInteger first(BigInteger i, BigInteger j) {
        return i;
    }
}
```

Figure 5.6 Definition of class JMLRacBigIntegerUtils

5.5 Tests

For the changes in this chapter, one or two test cases were also created per kind of expression. The following figure shows one of the test cases.

```
import org.jmlspecs.jmlrac.runtime.*;

public class ModelMethod_bigint1 {
    /*@  public static pure model \bigint binc(\bigint i) {
    @    return i + 1;
    @ }
    @*/

    /*@ public normal_behavior
    @  assignable \nothing;
    @  ensures  \result == binc(i);
    @*/
    public static int inc(int i) {
        return i+1;
    }

    /*@ public normal_behavior
    @  assignable \nothing;
    @  ensures  \result == binc(i);
    @*/
    public static long inc(long i) {
        return i+1;
    }

    public static void main(String args[]) {
        ModelMethod_bigint1 test = new ModelMethod_bigint1();
        int fcnt = 0;

        // no assertion violation
        try { test.inc(0); }
        catch (JMLAssertionError e) { fcnt++; }

        // postcondition violation
        try { test.inc(Integer.MAX_VALUE); fcnt++; }
        catch (JMLPostconditionError e) {}
        catch (JMLAssertionError e) { fcnt++; }

        // postcondition violation
        try { test.inc(Long.MAX_VALUE); fcnt++; }
        catch (JMLPostconditionError e) {}
        catch (JMLAssertionError e) { fcnt++; }

        if (fcnt > 0) {
            System.out.println(fcnt + "F(ModelMethod_bigint1.java)");
        }
    }
}
```

Figure 5.7 Sample test case for model method supporting \bigint

This test case is used in the same way as the one given in Figure 4.12.

5.6 Implementation

To implement the changes mentioned in this chapter, I modified a file used to translate model field in package *RAC code generation* (see Figure 3.9). Another file changed is in package *RAC code printing* (see Figure 3.9). The use of this file is to translate model method.

6 Neutral Contexts: An Experiment

6.1 Introduction

As was explained in Section 3.3.1, the semantics of assertions in JML are currently based on two-valued logic, in which partial functions are modeled as underspecified total functions [Section 3.3.1.1]. In order to implement such semantics, the RAC uses an approach named *local, contextual interpretation* (see Section 3.3.1.2) to translate JML expressions to Java program code. However, this approach doesn't work well for the equality expression(==). For example, let's consider a JML expression *(x.size() > 0) == (y.size() > 0)* (from [3, Section 3.5.1]). If both x and y are null, this expression will be translated to *false == false*, whose value is true. In this case, two null-pointer exceptions are not reported. To solve this anomaly, the RAC then uses an approach called *neutral context* which means "temporarily disable the contextual interpretation" [3, section 3.5.1].

Chalin recently proposed an alternate semantic for assertions, in which partial functions are modeled explicitly—i.e. any demonic exceptions simply result in the top level assertion being falsified [29]—hence there is no local contextual interpretation. As an experiment in enabling the RAC to partially emulate this approach, The use of "neutral contexts" was extended to conditional or, "||".

For example, an expression like "`x.length >0 || true`" will be true when `x == null` while using the *local, contextual interpretation* approach under the current JML semantics. It does obey the logic rules of JML, but it also hides a potential error. If the RAC concept of *neutral context* is used then the expression will be false. This seems to be the behavior that most practitioners want [29] .

When it's indicated, the RAC will use the *neutral context* approach to translate conditional OR (||) expressions and logical OR (|) expressions to Java program code.

The rest of this chapter shows how to implement the *neutral context* for conditional OR and logical OR expressions.

6.2 Semantics

6.2.1 Conditional OR Expression

The semantic function for conditional OR expression that is implemented using neutral context is given in Figure 6.1.

```
C [[E1 || E2]] (r, p) :=
try {
    boolean v1 = false;
    boolean v2 = false;
    C_NT [[E1]] (v1)
    C_NT [[E2]] (v2)
    r = v1 || v2;
} catch (JMLAngelicException e) {
    r = !p;
} catch (Exception e) {
    r = p;
}
```

Figure 6.1 Semantic function for conditional OR expressions

In the rule a special semantic function C_{NT} [[E]] (v) is used. It is like the C function, but it does not wrap the translated code with a try-catch block. It also doesn't have the boolean variable *p* which is used to indicate the positive or negative "contextual value". As an example, we provide the by C_{NT} [[E]] rule for "==" in Figure 6.2 . This rule, when compared with Figure 3.6, one can see the difference between C_{NT} [[E]] (v) and C [[E]] (v, p).

```
C_NT [[E1 == E2]] (r):=
    int v1 = 0;
    int v2 = 0;
    C_NT [[E1]] (v1)
    C_NT [[E2]] (v2)
    r = v1 == v2;
```

Figure 6.2 Sample translation rule for '==' by C_{NT} [[E]] (v)

6.2.2 Figure Logical OR Expression

The abstract syntax of logical OR expression is shown as:

Logical OR Expression : ExpressionName | ExpressionName

The semantic function for logical OR expression is given in Figure 6.3.

```
C [[E1 | E2]] (r, p) := ? if E1's type is boolean
try {
    boolean v1 = false;
    boolean v2 = false;
    C_NT [[E1]] (v1)
    C_NT [[E2]] (v2)
    r = v1 | v2;
} catch (JMLAngelicException e) {
    r = !p;
} catch (Exception e) {
    r = p;
}
```

Figure 6.3 Semantic function for logical OR expressions

6.3 Testing

For the changes in this chapter, one test case was created; it is shown in the following figure

```
import org.jmlspecs.jmlrac.runtime.JMLAssertionError;
import org.jmlspecs.jmlrac.runtime.JMLInvariantError;

public class NeutralContext{
    /*@
    @ requires (a.length > 0) || true;
    @*/
    void m1(int a[]){
    }

    /*@
    @ requires (a.length > 0) | true;
    @*/
    void m2(int a[]){
    }

    public static void main(String args[]) {
        NeutralContext o = new NeutralContext();
        int fcnt = 0;

        try {
            o.m1(null);
            fcnt++;
        }
        catch (JMLAssertionError e) {  }

        try {
            o.m2(null);
            fcnt++;
        }
        catch (JMLAssertionError e) {  }

        if (fcnt > 0) {
            System.out.println(fcnt + "F(NeutralContext.java)");
        }
    }
}
```

Figure 6.4 Test case for neutral context

For how to examine this test case, see the explanation following Figure 4.12.

6.4 Implementation

The area modified to implement the contributions mentioned in this chapter are the *RAC code generation* package (see Figure 3.9). The files impacted are `Trans*` ones (see Figure 3.10, Figure 3.11) used to translate logical OR and conditional OR expressions.

7 Conclusion

The main contribution of this thesis is to make the JML RAC support Arbitrary Precision Integers, which is represented in JML as the type `\bigint`. The reason to import this new type in JML is based on the extensions made by Chalin [21]. To implement this type in the assertion checking code generated by the RAC, the Java class `java.math.BigInteger` is used. Of all the kinds of expression used in the JML (see Appendix 9), only the ones whose subexpression types might be `\bigint` needed to be changed. However, model method bodies can contain arbitrary Java code and hence they may also contain expressions/statements, like assignment expression, compound assignment expression, pre(post)-fix expressions and so on; `\bigint` support for these kinds of statement was also implemented. The planned changes were first described by means of semantic functions based on the ones used in Cheon's Ph.D. thesis [3].

Another contribution in this thesis is to exdend *neutral contextual interpretation* (see Section 6.1) to all other kinds of expressions, like logical OR expression and conditional OR expression. The main reason for doing this was to experiment with alternate semantics for assertion expressions that makes it possible to find more assertion violations.

7.1 Future Work

A few more extensions need to be implemented in the RAC. One of them is to make reference types non-null by default. This is an extension proposed by Chalin and Rioux, which was motivated by the fact that the majority of reference types are intended to be non-null in Java [27]. If this proposal is implemented, it can both reduce the code work of programmers and remove potential errors from the code.

Another extension that needs to be implemented is the newly proposed semantics for assertions. Currently, the JML assertions are interpreted in a classical two-valued logic. Such a logic has some basic problems, which have been identified by Chalin, and an alternate semantics (three-valued logic) has been proposed [29, 35].

8 References

[1] Yoonsik Cheon and Gary T. Leavens: *A Contextual Interpretation of Undefinedness for Runtime Assertion Checking*

[2] Gary T. Leavens, Albert L. Baker, and Clyde Ruby: Preliminary Design of JML. TR #98-06w, Iowa State University, September 2003

[3] Y. Cheon. *A runtime assertion checker for the Java Modeling Language.* Technical Report 03-09, Department of Computer Science, Iowa State University, Ames, IA, Apr. 2003.

[4] Patrice Chalin. Improving JML: *For a Safer and More Effective Language.* In Stefania Gnesi, Keijiro Araki, and Dino Mandrioli (Eds.), FME 2003, International Symposium of Formal Methods Europe, Pisa, Italy, Sept. 8-14, 2003.

[5] Patrice Chalin. Back to Basics: *Language Support and Semantics of Basic Infinite Integer Types in JML and Larch.* Technical Report 2002-003.4, Computer Science Department, Concordia University, October 2002, revised March , May and July 2003.

[6] Jeannette M. Wing. Writing *Larch interface language specifications. ACM Transactions on Programming Languages and Systems*, 9(1):1–24, January 1987.

[7] K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. *Extended Static Checking*, October 12, 2000. Web page at 'http://research.compaq.com/SRC/esc/Esc.html'

[8] Alex Borgida, John Mylopoulos, and Rayomnd Reiter. *On the frame problem in procedure specifications.* IEEE Transactions on Software Engineering, 21(10):785-798, October 1995.

[9] Richard Allen Lerner. *Specifying Objects of Concurrent Systems School of Computer Science*, Carnegie Mellon University, CMU-CS-91-131, May 1991. URL: 'ftp://ftp.cs.cmu.edu/afs/cs.cmu.edu/project/larch/ftp/thesis.ps.Z'.

[10] Gowri Sivaprasad. *Larch/CORBA: Specifying the behavior of CORBA-IDL interfaces.* Technical Report 95-27a, Department of Computer Science, Iowa State University, Ames, Iowa, 50011, December 1995.

[11] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Joseph Kiniry: *JML Reference Manual.* 11 October 2004

[12] Krishna Kishore Dhara and Gary T. Leavens. *Forcing behavioral subtyping through specification inheritance.* In Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pages 258-267. IEEE Computer Society Press, March 1996.

[13] Bertrand Meyer. *Eiffel: The Language.* Object-Oriented Series. Prentice Hall, New York, NY, 1992

[14] The MultiJava Team: The MultiJava Project, December 2004.Webpage: 'http://multijava.sourceforge.net/index.shtml'.

[15] The JML Team: *The Java Modeling language* (JML), August, 2005. Home Page 'http://www.cs.iastate.edu/~leavens/JML/index.shtml'.

[16] Esc/Java 2 *KindSoftware ESC/Java2.* Webpage: 'http://secure.ucd.ie/products/opensource/ESCJava2/readme.html'

[17] B.P.F. Jacobs, E. Poll Nijmegen, *Java Program Verification at Nijmegen: Developments and Perspective,* Institute for Computing and Information Sciences. NIII-R0318 September 2003

[18] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T.A. Henzinger, editors, *Computer Aided Verification*, number 1102 in Lect. Notes Comp. Sci., pages 411–414. Springer, Berlin, 1996.

[19] Patrice Chalin. *On the Language Design and Semantic Foundation of LCL, a Larch/C Interface Specification Language.* PhD thesis, Computer Science Department, Concordia University, October 1995. (Also CU/DCS TR 95-12).

[20] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll, *An Overview of JML Tools and Applications,* International Journal on Software Tools for Technology Transfer (STTT), vol. 7, no. 3, pp. 212-232, (first published online on Dec. 14, 2004) June, 2005.

[21] Patric Chalin. *JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics*, in Journal of Object Technology, vol. 3, no. 6, June 2004, pp. 57-79.

[22] Cliff B. Jones. *Systematic Software Development Using VDM.* International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.

[23] John Fitzgerald and Peter Gorm Larsen. *Modelling Systems: Practical Tools and Techniques in Software Development.* Cambridge University Press, Cambridge, UK, 1998.

[24] John V. Guttag, James J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, New York, N.Y., 1993.

[25] Jeannette M. Wing. *Writing Larch interface language specifications.* ACM Transactions on Programming Languages and Systems, 9(1):1–24, January 1987.

[26] Jeannette M. Wing. *A specifier's introduction to formal methods.* Computer, 23(9):8–24, September 1990.

[27] P. Chalin and F. Rioux, *"Non-null References by Default in the Java Modeling Language." In Proceedings of the Workshop on the Specification and Verification of Component-Based Systems* (SAVCBS'05), Lisbon, Portugal, Sept., 2005. (Preliminary version available as ENCS-CSE TR 2005-004. June, 2005.)

[28] JML website: *www.jmlspecs.org*, August 2005.

[29] P. Chalin, "*Logical Foundations of Program Assertions: What do Practitioners Want?"* In Proceedings of the 3rd International Conference on Software

Engineering and Formal Methods (SEFM'05), Koblenz, Germany, September 5-9, 2005 (to appear). Preprint available as ENCS-CSE TR 2005-002.

[30] Craig Larman, *Applying UML and Patterns* : An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition), Prentice Hall PTR. Oct 20, 2004

[31] Terence Parr, *ANTLR Parser Generator,* http://www.antlr.org/. Aug, 2005

[32] Peter Grogono, Lecture notes for COMP7451, Programming Language Semantics. CSE Department, Concordia University. May 2002

[33] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley Professional. Oct 31, 1994

[34] Richard Paige and Jonathan Ostroff. A Seamless Eiffel-Based Refinement Calculus for Object-Oriented Systems (Extended Abstract) http://www.inf.ethz.ch/personal/daniekro/classes/se-sem/ss2005/papers/Paige.pdf. Apr, 2005

[35] P. Chalin. *Ensuring Continued Mainstream Use of Formal Methods: An Assessment, Roadmap and Issues*. Dependable Software Research Group, Concordia University. ENCS-CSE TR 2005-001. February 2005

[36] R. J. R. Back. A calculus of refinements for program derivations. Acta Informatica, 25(6):593--624, August 1988.

[37] L. Friendly. *Design of Javadoc*. International Workshop on Hypermedia Design '95, 1995

[38] Cheon Y. and Leavens G. T., *A simple and practical approach to unit testing: The JML and JUnit* way, In Proc of 16th European Conference Object-Oriented Programming (ECOOP02), pp. 231-255., 2002.

[39] JACK: Java Applet Correctness Kit. http://www-sop.inria.fr/everest/soft/Jack/jack.html. June 2005

[40] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[41] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. *Dynamically discovering likely program invariants to support program evolution.* IEEE Transactions on Software Engineering, 27(2):1–25, 2001.

[42] Cormac Flanagan and K. Rustan M. Leino. Houdini,*an annotation assistant for ESC/Java*. In J. N. Oliveira and P. Zave,ed itors, FME 2001,vo lume LNCS 2021,p ages 500–517. Springer, 2001.

9 Appendix: Example of the RAC generated code

A very simple Java class with a two JML assertions is given in Figure 9.1. The class contains a method *mTrue()* whose body has an inline assertion clause. No explicit method specification is provided hence, the RAC treats it as equivlant to "requires true" and "ensures true".

```
class A0 {
    public static void mTrue() {
        //@ assert true;
    }
}
```

Figure 9.1 A Sample for inline assertion

To translate this method, the RAC creates a wrapper method (Section 3.3.2) and it renames the "original" method to *internal$mTrue*. Actually, the "original" method also contains RAC generated code due to the presence of an inlined assertion. The rest of this section provides a brief explanation of the RAC code in these two figures.

9.1 Inline assertion

Figure 9.2 shows the code generated by the RAC for the body of method *mTrue*. Its main part is a do-while block which is used to enclose the assertion checking code generated for the body of *mTrue* in Figure 9.1. Line (1) declares a variable *rac$where* which is used to save the information concerning the location of the assert statement in the body of *mTrue*. Line (2) declares a variable *rac$v0* which is used to store the value of the whole inline assertion. In the translation rule demonstrated in Section 3.4, *rac$v0* corresponds to the parameter *Identifier*. Lines (3) to (4) are a try-catch block which is generated of the inline assertion clause "*assert true*". This block corresponds to the translation rule shown in Section 3.4. Lines (5) to (6) are used to throw an exception if the inline assertion is violated, that is, the value of *rac$v0* is false.

```
private static void internal$mTrue() {

  /*@ assert true;
    @*/
  do {
    if (JMLChecker.isActive(JMLChecker.INTRACONDITION) && rac$ClassInitialized) {
      JMLChecker.enter();
      java.util.Set rac$where = new java.util.HashSet();                    (1)
      boolean rac$v0 = false;                                               (2)
    try {                                                                   (3)
      rac$v0 = true;
      if (!rac$v0) {
        rac$where.add("File \"A0.java\", line 3, character 25");
      }
    }
    catch (JMLNonExecutableException jml$e0) {
      rac$v0 = true;
    }
    catch (java.lang.Exception jml$e0) {
      rac$v0 = false;
    }                                                                       (4)
      if (!rac$v0) {                                                        (5)
        JMLChecker.exit();
        throw new JMLAssertError("ASSERT", "A0", "mTrue", rac$where);
      }
      JMLChecker.exit();
    }                                                                       (6)
  }
  while (false);
}
```

Figure 9.2 Code generated by the RAC for internal$mTrue().

9.2 Wrapper method

Figure 9.3 presents the definition of the wrapper method created for *mTrue*. Lines (1) to (2) are used to execute the original method during class initialization; in such situations, assertion checking is *not* done. Lines (3) to (4) are used to execute the original method when the preconditions don't need to be tested. Lines (5) to (6) are used to calculate the value of any possible old expressions (see Section 2.2). In this example, no assertions contain old expressions, but still a method with an empty body is created to calculating old expressions. Between line (5) and (6), there is a pair of statements *JMLChecker.enter()* and *JMLChecker.exit()* which are widely used. The function of their two statements are to turn on/off a flag indicating if the current thread is in assertion checking mode. In this way, it can avoid any possible recursive assertion checking because in statement *JMLChecker.enter()*, it will check if the flag is already on, if so, it will report error. Lines (7) to (8) and (9) to (10) are used to check the invariant and precondition in this method. Line (11) is used to invoke the original method. Lines (12)

to (13) are used to check normal post-conditions. Lines (14) to (17) are several catch blocks used to catch any exceptions that might be raised during the execution of statements in the try block. Lines (15) to (16) are used to check the exceptional post conditions. At last, lines (18) to (19) and (20) to (21) are used to check the class invariant and class constraint separately.

```
  public static void mTrue() {
    // skip assertion checks during initialization
    if (!rac$ClassInitialized) {                                          (1)
      internal$mTrue(); return;
    }                                                                     (2)
    if (!(JMLChecker.isActive(JMLChecker.PRECONDITION))) {                (3)
      internal$mTrue(); return;
    }                                                                     (4)
    // eval old exprs in constraint
    if (JMLChecker.isActive(JMLChecker.CONSTRAINT)) {                     (5)
      JMLChecker.enter();  evalOldExprInHC$static(); JMLChecker.exit();
    }                                                                     (6)
    // check static invariant
    if (JMLChecker.isActive(JMLChecker.INVARIANT)) {                      (7)
      JMLChecker.enter();
      checkInv$static("mTrue@pre<File \"A0.java\", line 2, character 11>");
      JMLChecker.exit();
    }                                                                     (8)
    // check precondition
    if (JMLChecker.isActive(JMLChecker.PRECONDITION)) {                   (9)
      JMLChecker.enter();  checkPre$mTrue$A0();  JMLChecker.exit();
    }                                                                     (10)
    boolean rac$ok = true;
    try {
      // execute original method
      internal$mTrue();                                                   (11)
      // check normal postcondition
      if (JMLChecker.isActive(JMLChecker.POSTCONDITION)) {                (12)
        JMLChecker.enter();  checkPost$mTrue$A0(null);
        JMLChecker.exit();
      }                                                                   (13)
    }
    catch (JMLEntryPreconditionError rac$e) {                             (14)
      rac$ok = false;
      throw new JMLInternalPreconditionError(rac$e);
    }
    catch (JMLExitNormalPostconditionError rac$e) {
      rac$ok = false;
      throw new JMLInternalNormalPostconditionError(rac$e);
    }
    catch (JMLExitExceptionalPostconditionError rac$e) {
      rac$ok = false;
      throw new JMLInternalExceptionalPostconditionError(rac$e);
    }
    catch (JMLAssertionError rac$e) {
      rac$ok = false;  throw rac$e;
    }
    catch ( java.lang.Throwable rac$e) {
      try {
        // check exceptional postcondition                                (15)
        if (JMLChecker.isActive(JMLChecker.POSTCONDITION)) {
          JMLChecker.enter();  checkXPost$mTrue$A0(rac$e);
          JMLChecker.exit();
        }                                                                 (16)
      }
      catch (JMLAssertionError rac$e1) {
        rac$ok = false; throw rac$e1;
      }
    }                                                                     (17)
    finally {
      if (rac$ok) {
        // check static invariant
        if (JMLChecker.isActive(JMLChecker.INVARIANT)) {                  (18)
          JMLChecker.enter();
          checkInv$static("mTrue@post<File \"A0.java\", line 2, character 11>");
          JMLChecker.exit();
        }                                                                 (19)
        // check static constraint
        if (JMLChecker.isActive(JMLChecker.CONSTRAINT)) {                 (20)
          JMLChecker.enter();
          checkHC$static("mTrue@post<File \"A0.java\", line 2, char...11>", "mTrue", new
java.lang.Class[] { });
          JMLChecker.exit();
        }                                                                 (21)
      }
    }
  }
```

Figure 9.3 Code generated by the RAC for mTrue

9.3 A More Complete Sample

This section will show some examples of the RAC generated code of the JML example shown in Figure 1.1.

Lines (a) and (b)

```
        //@ protected invariant nJump >= 0;                    //(a)
        //@ protected invariant nLife >= 0;                    //(b)
```

are generated to:

```
public void checkInv$instance$Player(java.lang.String rac$msg) {
  java.util.Set rac$where = new java.util.HashSet();
  boolean rac$b = true;
  try {
    // eval of &&
    boolean rac$v0 = true;
    boolean rac$v1 = false, rac$v2 = false;
    // arg 1 of &&
    try {
      boolean rac$v3 = false, rac$v4 = false;
      int rac$v5 = 0;
      int rac$v6 = 0;
      try {
        rac$v5 = this.nJump;
      }
      catch (JMLNonExecutableException jml$e0) {
        rac$v4 = true;
      }
      catch (java.lang.Exception jml$e0) {
        rac$v3 = true;
      }
      if (!rac$v3) {
        try {
          rac$v6 = 0;
        }
        catch (JMLNonExecutableException jml$e0) {
          rac$v4 = true;
        }
        catch (java.lang.Exception jml$e0) {
          rac$v3 = true;
        }
      }
      if (rac$v3) { rac$v0 = false; }
      else if (rac$v4) { rac$v0 = true; }
      else try {
       rac$v0 = rac$v5>=rac$v6;
      }
      catch (JMLNonExecutableException jml$e0) {
        rac$v0 = true;
      }
      catch (java.lang.Exception jml$e0) {
        rac$v0 = false;
      }
      if (!rac$v0) {
        int rac$v8 = JMLChecker.getLevel();
        JMLChecker.setLevel(JMLChecker.NONE);
        java.lang.String rac$v7 = "";
        rac$v7 += "\n\t'nJump' is " + JMLChecker.toString(nJump);
        rac$v7 += "\n\t'this' is " + JMLChecker.toString(this);
        JMLChecker.setLevel(rac$v8);
        rac$where.add("File \"Player.java\", line 8, character 39 when" + rac$v7);
      }
    }
    catch (JMLNonExecutableException jml$e0) {
      rac$v2 = true;
    }
    catch (java.lang.Exception jml$e0) {
      rac$v1 = true;
    }
    if (rac$v0) {
      // arg 2 of &&
      try {
        boolean rac$v9 = false, rac$v10 = false;
        int rac$v11 = 0;
        int rac$v12 = 0;
        try {
          rac$v11 = this.nLife;
        }
        catch (JMLNonExecutableException jml$e0) {
          rac$v10 = true;
        }
        catch (java.lang.Exception jml$e0) {
          rac$v9 = true;
        }
        if (!rac$v9) {
          try {
            rac$v12 = 0;
          }
          catch (JMLNonExecutableException jml$e0) {
            rac$v10 = true;
```

```
            }
            catch (java.lang.Exception jml$e0) {
              rac$v9 = true;
            }
          }
          if (rac$v9) { rac$v0 = false; }
          else if (rac$v10) { rac$v0 = true; }
          else try {
           rac$v0 = rac$v11>=rac$v12;
          }
          catch (JMLNonExecutableException jml$e0) {
            rac$v0 = true;
          }
          catch (java.lang.Exception jml$e0) {
            rac$v0 = false;
          }
          if (!rac$v0) {
            int rac$v14 = JMLChecker.getLevel();
            JMLChecker.setLevel(JMLChecker.NONE);
            java.lang.String rac$v13 = "";
            rac$v13 += "\n\t'nLife' is " + JMLChecker.toString(nLife);
            rac$v13 += "\n\t'this' is " + JMLChecker.toString(this);
            JMLChecker.setLevel(rac$v14);
            rac$where.add("File \"Player.java\", line 9, character 37 when" + rac$v13);
          }
        }
        catch (JMLNonExecutableException jml$e0) {
          rac$v2 = true;
        }
        catch (java.lang.Exception jml$e0) {
          rac$v1 = true;
        }
      }
      if (rac$v0) {
        if (rac$v1) { throw JMLChecker.DEMONIC_EXCEPTION; }
        if (rac$v2) { throw JMLChecker.ANGELIC_EXCEPTION; }
      }
      rac$b = rac$v0;
    }
    catch (JMLNonExecutableException jml$e0) {
      rac$b = true;
    }
    catch (java.lang.Exception jml$e0) {
      rac$b = false;
    }
    boolean rac$bSuper = true;
    if (!rac$b) {
      JMLChecker.exit();
      throw new JMLInvariantError("Player", rac$msg, rac$where);
    }
  }
```

- Method move() and its specifications (line (f) to (j))

```
    /*@ public normal_behavior                                          //(f)
      @ requires nLife > 0;
        //(g)
      @ assignable rcImage;
        //(h)
      @ ensures rcImage.height == \old(rcImage).height &&               //(i)
      @         rcImage.width == \old(rcImage).width;
        //(j)
      @*/
    public void move() {/*...*/}
```

are generated to:

- Line (g) are translated to the pre-condition method *checkPre$move$Player()* shown in the following:

```
/** Generated by JML to check the precondition of
 * method move. */
public boolean checkPre$move$Player() {
  try {
    java.awt.Rectangle rac$v6 = null;
    rac$v6 = this.rcImage;
    rac$old0 = JMLRacValue.ofObject(rac$v6);
  } catch (JMLNonExecutableException jml$e0) {
    rac$old0 = JMLRacValue.ofNonExecutable();
  }catch (java.lang.Exception jml$e0) {
    rac$old0 = JMLRacValue.ofUndefined();
  }
  try {
    java.awt.Rectangle rac$v7 = null;
    rac$v7 = this.rcImage;
    rac$old1 = JMLRacValue.ofObject(rac$v7);
  } catch (JMLNonExecutableException jml$e0) {
    rac$old1 = JMLRacValue.ofNonExecutable();
  }catch (java.lang.Exception jml$e0) {
    rac$old1 = JMLRacValue.ofUndefined();
  }
  java.util.Set rac$where = new java.util.HashSet();
  boolean rac$b = false;
  try {
    boolean rac$v0 = false, rac$v1 = false;
    int rac$v2 = 0;  int rac$v3 = 0;
    try {
      rac$v2 = this.nLife;
    } catch (JMLNonExecutableException jml$e0) {
      rac$v1 = true;
    } catch (java.lang.Exception jml$e0) {
      rac$v0 = true;
    }
    if (!rac$v0) {
      try { rac$v3 = 0;
      } catch (JMLNonExecutableException jml$e0) {
        rac$v1 = true;
      } catch (java.lang.Exception jml$e0) {
        rac$v0 = true;
      }
    }
    if (rac$v0) { rac$pre1 = false; }
    else if (rac$v1) { rac$pre1 = true; }
    else try {
     rac$pre1 = rac$v2>rac$v3;
    } catch (JMLNonExecutableException jml$e0) {
      rac$pre1 = true;
    } catch (java.lang.Exception jml$e0) {
      rac$pre1 = false;
    }
    JMLChecker.addCoverage("File \"Player.java\", line 18, character 25",rac$pre1);
    if (!rac$pre1) {
      int rac$v5 = JMLChecker.getLevel();
      JMLChecker.setLevel(JMLChecker.NONE);
      java.lang.String rac$v4 = "";
      rac$v4 += "\n\t'nLife' is " + JMLChecker.toString(nLife);
      rac$v4 += "\n\t'this' is " + JMLChecker.toString(this);
      JMLChecker.setLevel(rac$v5);
      rac$where.add("File \"Player.java\", line 18, character 25 when" + rac$v4);
    }
  } catch (JMLNonExecutableException jml$e0) {
    rac$pre1 = true;
  } catch (java.lang.Exception jml$e0) {
    rac$pre1 = false;
  }
  rac$b = rac$b || rac$pre1;
  if (!rac$b) {
    if (JMLChecker.getLevel() > JMLChecker.PRECONDITION) {
      saveTo$rac$stack1();
    }
    JMLChecker.exit();
    throw new JMLEntryPreconditionError("Player", "move", rac$where);
  }
  if (JMLChecker.getLevel() > JMLChecker.PRECONDITION) {
    saveTo$rac$stack1();
  }
  return true;
}
```

- Line (i) and (j) are translated to the post-condition method *checkPost$move$Player()* shown below:

```
public void checkPost$move$Player(java.lang.Object rac$result) {
    restoreFrom$rac$stack1();
    java.util.Set rac$where = new java.util.HashSet();
    boolean rac$b = true;
    boolean rac$b0 = true;
    if (rac$pre1) {
      try {
        // eval of &&
        boolean rac$v0 = true;
        boolean rac$v1 = false, rac$v2 = false;
        // arg 1 of &&
        try {
          boolean rac$v3 = false, rac$v4 = false;
          int rac$v5 = 0;
          try {
            java.awt.Rectangle rac$v6 = null;
            rac$v6 = this.rcImage;
            rac$v5 = rac$v6.height;
          }
          catch (JMLNonExecutableException jml$e0) {
            rac$v4 = true;
          }
          catch (java.lang.Exception jml$e0) {
            rac$v3 = true;
          }
          int rac$v7 = 0;
          if (!rac$v3) {
            try {
              java.awt.Rectangle rac$v8 = null;
              rac$v8 = ((java.awt.Rectangle)rac$old0.value());
              rac$v7 = rac$v8.height;
            }
            catch (JMLNonExecutableException jml$e0) {
              rac$v4 = true;
            }
            catch (java.lang.Exception jml$e0) {
              rac$v3 = true;
            }
          }
          if (rac$v3) { rac$v0 = false; }
          else if (rac$v4) { rac$v0 = true; }
          else {
           rac$v0 = rac$v5==rac$v7;
          }
        }
        catch (JMLNonExecutableException jml$e0) {
          rac$v2 = true;
        }
        catch (java.lang.Exception jml$e0) {
          rac$v1 = true;
        }
        if (rac$v0) {
          // arg 2 of &&
          try {
            boolean rac$v9 = false, rac$v10 = false;
            int rac$v11 = 0;
            try {
              java.awt.Rectangle rac$v12 = null;
              rac$v12 = this.rcImage;
              rac$v11 = rac$v12.width;
            }
            catch (JMLNonExecutableException jml$e0) {
              rac$v10 = true;
            }
            catch (java.lang.Exception jml$e0) {
              rac$v9 = true;
            }
            int rac$v13 = 0;
            if (!rac$v9) {
              try {
                java.awt.Rectangle rac$v14 = null;
                rac$v14 = ((java.awt.Rectangle)rac$old1.value());
                rac$v13 = rac$v14.width;
              }
              catch (JMLNonExecutableException jml$e0) {
                rac$v10 = true;
              }
              catch (java.lang.Exception jml$e0) {
                rac$v9 = true;
              }
```

```
            }
            if (rac$v9) { rac$v0 = false; }
            else if (rac$v10) { rac$v0 = true; }
            else {
             rac$v0 = rac$v11==rac$v13;
            }
          }
          catch (JMLNonExecutableException jml$e0) {
            rac$v2 = true;
          }
          catch (java.lang.Exception jml$e0) {
            rac$v1 = true;
          }
        }
        if (rac$v0) {
          if (rac$v1) { throw JMLChecker.DEMONIC_EXCEPTION; }
          if (rac$v2) { throw JMLChecker.ANGELIC_EXCEPTION; }
        }
        rac$b0 = rac$v0;
        if (!rac$b0) {
          int rac$v16 = JMLChecker.getLevel();
          JMLChecker.setLevel(JMLChecker.NONE);
          java.lang.String rac$v15 = "";
          rac$v15 += "\n\t'" + JMLChecker.toString("\\old(rcImage)") + "' is " +
JMLChecker.toString(rac$old0);
          rac$v15 += "\n\t'" + JMLChecker.toString("\\old(rcImage)") + "' is " +
JMLChecker.toString(rac$old1);
          rac$v15 += "\n\t'rcImage' is " + JMLChecker.toString(rcImage);
          rac$v15 += "\n\t'this' is " + JMLChecker.toString(this);
          JMLChecker.setLevel(rac$v16);
          rac$where.add("File \"Player.java\", line 20, character 31 when" + rac$v15);
        }
      }
      catch (JMLNonExecutableException jml$e0) {
        rac$b0 = true;
      }
      catch (java.lang.Exception jml$e0) {
        rac$b0 = false;
      }
    }
    rac$b = rac$b && rac$b0;
    if (!rac$b) {
      JMLChecker.exit();
      throw new JMLExitNormalPostconditionError("Player", "move", rac$where);
    }
  }
```

- The wrapper method is generated as:

```
public void move() {
  // skip assertion checks during initialization
  if (!rac$ClassInitialized || !rac$initialzed || !rac$dented()) {
    internal$move();
    return;
  }
  if (!(JMLChecker.isActive(JMLChecker.PRECONDITION))) {
    internal$move();
    return;
  }
  // eval old exprs in constraint
  if (JMLChecker.isActive(JMLChecker.CONSTRAINT)) {
    JMLChecker.enter();
    evalOldExprInHC$instance$Player();
    JMLChecker.exit();
  }
  // check static invariant
  if (JMLChecker.isActive(JMLChecker.INVARIANT)) {
    JMLChecker.enter();
    checkInv$static("move@pre<File \"Player.java\", line 17, character 18>");
    JMLChecker.exit();
  }
  // check instance invariant
  if (JMLChecker.isActive(JMLChecker.INVARIANT)) {
    JMLChecker.enter();
    checkInv$instance$Player("move@pre<File \"Player.java\", line 17, character 18>");
    JMLChecker.exit();
  }
  // check precondition
  if (JMLChecker.isActive(JMLChecker.PRECONDITION)) {
    JMLChecker.enter();
    checkPre$move$Player();
    JMLChecker.exit();
  }
  boolean rac$ok = true;
  try {
    // execute original method
    internal$move();
    // check normal postcondition
    if (JMLChecker.isActive(JMLChecker.POSTCONDITION)) {
      JMLChecker.enter();
      checkPost$move$Player(null);
      JMLChecker.exit();
    }
  }
  catch (JMLEntryPreconditionError rac$e) {
    rac$ok = false;
    throw new JMLInternalPreconditionError(rac$e);
  }
  catch (JMLExitNormalPostconditionError rac$e) {
    rac$ok = false;
    throw new JMLInternalNormalPostconditionError(rac$e);
  }
  catch (JMLExitExceptionalPostconditionError rac$e) {
    rac$ok = false;
    throw new JMLInternalExceptionalPostconditionError(rac$e);
  }
  catch (JMLAssertionError rac$e) {
    rac$ok = false;
    throw rac$e;
  }
  catch ( java.lang.Throwable rac$e) {
    try {
      // check exceptional postcondition
      if (JMLChecker.isActive(JMLChecker.POSTCONDITION)) {
        JMLChecker.enter();
        checkXPost$move$Player(rac$e);
        JMLChecker.exit();
      }
    }
    catch (JMLAssertionError rac$e1) {
      rac$ok = false;
      throw rac$e1;
    }
  }
  finally {
    if (rac$ok) {
      // check static invariant
      if (JMLChecker.isActive(JMLChecker.INVARIANT)) {
        JMLChecker.enter();
```

```
            checkInv$static("move@post<File \"Player.java\", line 17, character 18>");
            JMLChecker.exit();
          }
          // check instance invariant
          if (JMLChecker.isActive(JMLChecker.INVARIANT)) {
            JMLChecker.enter();
            checkInv$instance$Player("move@post<File \"Player.java\", line 17, character 18>");
            JMLChecker.exit();
          }
          // check static constraint
          if (JMLChecker.isActive(JMLChecker.CONSTRAINT)) {
            JMLChecker.enter();
            checkHC$static("move@post<File \"Player.java\", line 17, character 18>", "move", new
java.lang.Class[] { });
            JMLChecker.exit();
          }
          // check instance constraint
          if (JMLChecker.isActive(JMLChecker.CONSTRAINT)) {
            JMLChecker.enter();
            checkHC$instance$Player("move@post<File \"Player.java\", line 17, character 18>",
false, "move", new java.lang.Class[] { });
            JMLChecker.exit();
          }
        }
      }
    }
```

10 Appendix: JML expressions translated in RAC

Expressions	Syntax	Needs to be changed for supporting \bigint
conditional expression	B ? E1 : E2	No
subtypeof expression	T1 <: T2	No
forward implication expression	E1 ==> E2	No
reverse implication expression	E1 <== E2	No
equivalence expression	E1 <==> E2	No
inequivalence expression	E1 <!=> E2	No
less than expression	E1 < E2	Yes
less equal than expression	E1 <= E2	Yes
greater than expression	E1 > E2	Yes
greater equal than expression	E1 >= E2	Yes
conditional-and expression	E1 && E2	No
conditional-or expression	E1 || E2	No
bitwise-and expression	E1 & E2	Yes
bitwise-or expression	E1 | E2	Yes
xor expression	E1 ^ E2	Yes
equality expression	E1 == E2	Yes
inequality expression	E1 != E2	Yes
instanceof expression	E instanceof T	No
add expression	E1 + E2	Yes
minus expression	E1 - E2	Yes
multiply expression	E1 * E2	Yes
divide expression	E1 / E2	Yes
modulo expression	E1 % E2	Yes
shift-left expression	E1 << E2	Yes
shift-right expression	E1 >> E2	Yes
boolean-shift-right expression	E1 >>> E2	Yes
positive expression	+E	No
negative expression	-E	Yes
bitwise-complement expression	~E	No
logical-not expression	!B	No
cast expression	(T)E	Yes
unary promote expression	(T)E	Yes
method call expression	M(arg0, arg1, ... argn)	No
type name expression	O	No
this expression	This	No
super expression	Super	No

class expression	E.class	No
array length expression	A.length	No
array access expression	E1[E2]	No
name expression	s	No
local variable expression	v	Yes
parenthesed expression	(E)	No
new object expression	new T(...)	No
new anonymous class expression	new T(...) {...}	No
new array expression	new T[...]	No
class field expression	T.f|f	No
boolean literal	true|false	No
char literal	'[a-zA-Z]'	No
ordinal literal	[0-9]+	Yes
real literal	[0-9]+.[0-9]+	No
string literal	"[a-zA-Z]+"	No
null literal	null	No
set comprehension notation	new T1 {T2 v1...vn | E1.has(v) && E2}	No
quantified expression	Q T v1...vn; E1; E2	Yes

- some Java expressions can't be used as JML expressions because they are not pure. E.g. assignment, compound-assignment, post(pre)fix expression, and explicit constructor invocation.
- array dimension and array initialization expressions are not supported in RAC yet.