A Run-time Assertion Checker for Java using JML

Abhay Bhorkar

TR #00-08
May 2000

Keywords: run-time assertion checking, precondition checking, model-based specification, behavioral interface specification language, behavioral subtyping, refinement, formal specification languages, Eiffel, JML, Java.

1999 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications — languages, tools, JML; D.2.4 [*Software Engineering*] Software/Program Verification — Assertion checkers, class invariants, formal methods, programming by contract, reliability, tools, JML; D.2.5 [*Software Engineering*] Testing and Debugging — Debugging aids; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs — Assertions, invariants, pre- and post-conditions, specification techniques;

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA

A Run-time Assertion Checker for Java using JML

Abhay Bhorkar
226, Atanasoff Hall, Ames, IA -50011
abhayb@cs.iastate.edu

May 1, 2000

Abstract

The Java Modeling Language (JML) is a behavioral interface specification language tailored for specifying Java modules. This paper describes a source-to-source translation tool that takes a JML specification and Java source code for a module and produces source code that checks assertions at run-time. It describes issues unique to JML, including specification-only variables, refinement, specification inheritance, and privacy. It also describes the design and implementation of the translation tool.

Contents

Chapter 1

Introduction

1.1 Specification and Assertion Checking

The Java Modeling Language (JML) [LBR99] is a model-based behavioral interface specification language (BISL) tailored to Java. It is based on Eiffel [Mey88] and Larch/C++ [LC92] (a BISL tailored to C++). It includes features like quantification, specification-only variables, etc., that makes it more powerful and expressive than Eiffel. It allows one to specify Java modules (classes, interface, etc.) in terms of annotated *assertion* expressions. Annotations are viewed by a standard Java compiler as comments, while the assertions describe a property of the module in terms of various program entities like local variables, instance variables, etc. [Mey88]. Assertions state "what" a module should do without stating "how" it should be done.

As a means of debugging and of partially checking correctness, one would like to see if a module satisfies these assertions at run-time. However, until recently, few programming languages supported assertions as a means of run-time program verification — Eiffel is one such language. Although Java lacks such support, new third-party tools like iContract [iCo00] are being developed for Java that enable one to specify, and later check, Java modules using annotated assertions. This document describes one such tool that takes Java source code, annotated with JML specifications, and performs source-to-source translation to generate an alternate source code, which, when executed, will check if the module adheres to its specification. This tool is integrated with existing JML type checker [Gan98] and uses ANTLR [Par99].

The current version of the tool only supports checking preconditions. This should not be a major limitation as it is suggested in [Mey88, pages 145–146] that as a good programming style and to avoid software failure, it is usually enough to check preconditions without degrading the performance appreciably. The tool can also demonstrate semantics of behavioral subtyping and use of specification-only variables by checking preconditions.

1.2 Goals

The goals of source-to-source translation are:

- Any type-correct source code, when translated, should remain type-correct.
- Any code that is being added should not alter the current state of the computation and thus affect the final result.

- Execution of the translated code with assertion checks turned off should be identical to the execution of the original source code. It may reduce the speed of execution, however.

- The source code added should not give rise to any name space conflicts with the existing variables. That is, if the new code uses some temporary variables, they should have fresh names.

- The tool should be able to translate specifications that use specification-only variables.

- The tool should work in presence of information hiding and inheritance. That is, one should be able to inherit the assertions from the superclass in a subclass in presence of information hiding in Java to enforce subcontracting (see Section 2.1.4).

- The tool should be maintainable given that both JML and Java are being constantly modified to include more and more language features.

- Compile-time performance of the tool — speed of translation, although not crucial, is important. This speed should be kept reasonable by sharing information among different phases of the tool and by reducing number of passes through the source code.

- In the event of assertion failure, the tool should give enough context information about the failure. Such context information could include, but is not limited to:

  1. Name of the file that contains this assertion,
  2. Name of the class that checks the assertion,
  3. Line number of the file,
  4. Name of the method that contains the code, etc.

  Note that in presence of behavioral subtyping (Section 2.1.4), refinement (Section 2.1.10) and interface implementation (Section 2.2), the actual assertion expression may be physically located in some other file or class. Context information should direct the user as close to the source of the failure as possible.

The next section gives an overview of the tool — what it does, how it is used, etc.

1.3 Overview of the tool

The basic usage of the tool is as shown in the figure 1.1. The tool has two components: a translator and a run-time system. The translator is a source-to-source tree transformer integrated with the JML type checker. It takes a file "Foo.java" annotated with JML specifications and transforms the generated abstract syntax tree (AST) to produce a file with the same name "Foo.java" that contains extra code to perform the assertion checks at run-time. The new code retains all the old source code so as to retain the functionality of the original source code. This new, augmented source code is then compiled using a standard Java compiler and executed with the run-time system. The run-time system helps the user control the assertion checking. By setting various options provided by the run-time system, the user can switch off the execution of the code that checks assertions, modify the

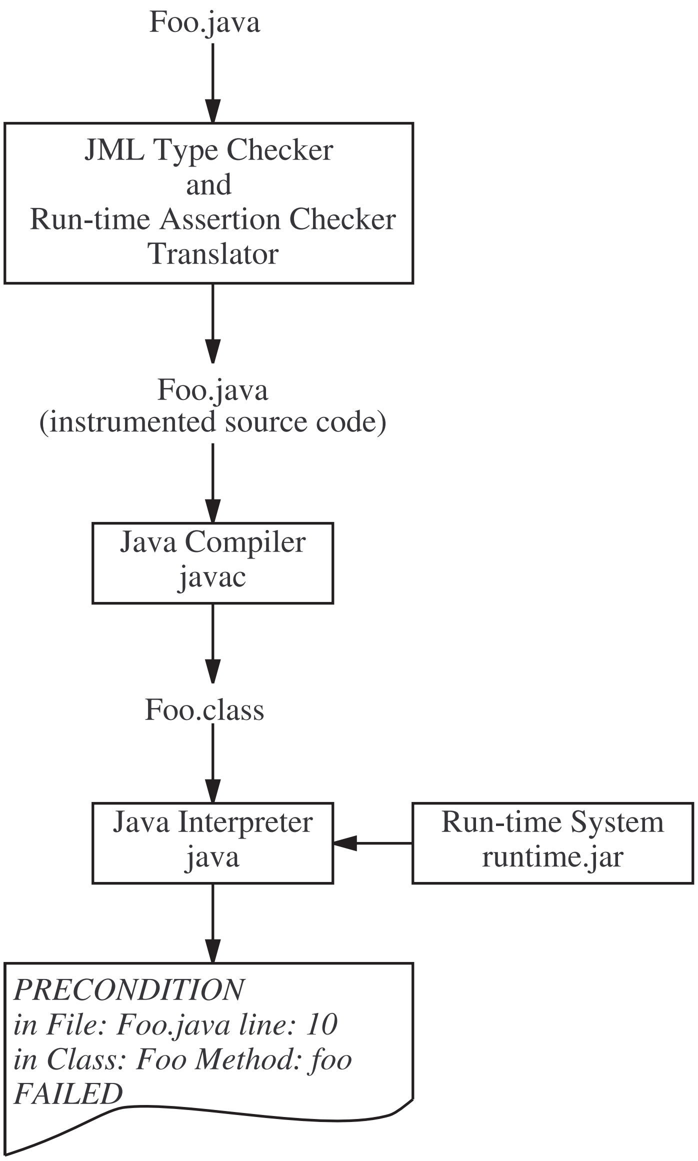

Figure 1.1: Usage of the tool.

types of assertions to be checked, and the way the tool should handle assertion failures, all without regenerating the code and without recompiling the entire source code[1].

To make full use of assertion inheritance, the user is required to generate instrumented source code for the superclass as well. This is a must especially if the the base class refers to specification-only variables (see Section 2.1.5) defined in the superclass.

In chapters to come, we describe the issues in the tool's design (Chapter 2) and the design and the implementation (Chapter 3); Chapter 4 discusses further enhancements to the tool. We end the discussion by describing related work in Chapter 5. Appendix A gives listing of the translated source code for class `BoundedColorPoint` shown in figure 3.3.

[1]Typically the part of the source that sets these options will need to be recompiled.

Chapter 2

Issues

This chapter discusses several issues that influenced the design the tool.

2.1 JML features

JML has rich syntax and features that allow one to specify Java modules with a lot of expressive power. This section describes those features that need to be taken into consideration in designing the run-time assertion checker.

2.1.1 Preconditions [JML keyword: requires]

Preconditions are assertions that must be satisfied when a routine is called. These are usually expressed in terms of state variables of a module and input parameters to the routine. These preconditions have to be checked before any of the code of the routine is executed and after checking the invariants (see Section 2.1.3), if any. The main function of the first version of the checker is to check preconditions.

2.1.2 Assertion Expressions [JML keyword: assert]

General assertion expressions can occur anywhere in the body of a routine, and are written using `assert` in JML. They describe assertions over the current state of computation. Such expressions can be handled in a way similar to preconditions, but their handling will differ in following cases:

1. As they can occur anywhere in the text of routine, translating these to source code will require us to make sure that the code is placed at the appropriate point in the body of the current method.

2. Expressions described using `assert` will not be inherited like preconditions.

3. They also require the checker to process the entire source code of the current method instead of just the method specifications.

2.1.3 Class Invariants [JML keyword: invariant]

Class invariants express global properties of an instance of a class, which must be preserved by all the routines of the class [Mey88]. These are logically factored out of preconditions and postconditions of all non-private methods of a module. Note that:

1. The effective class invariant is the conjunction of all the individual class invariants. Since individual invariants may be scattered all over the class, they will need to be collected first in order to form an effective invariant. This can be done by having a compiler pass collect all the invariants. Such a pass can be a separate one or it can be combined with other compiler passes like the pass that does type checking.

2. Effective preconditions and postconditions are to be formed by the conjunction with the class invariants. As a conjunction is short-circuited in Java, invariant expression should be checked before both the preconditions and the postconditions of the public methods are checked.

2.1.4 Behavioral Subtyping and Specification Inheritance

Through behavioral subtyping a BISL forces *subcontracting* [Mey88, DL96]. Syntactically subtyping implies that a subtype object should be able to replace a super-type object in a module without the client getting any type errors. Behavioral subtyping adds an additional constraint that such a replacement should not produce an unexpected behavior for the client that is expecting the super-type's behavior. This implies that any method redefined in the derived class, has the responsibility of carrying out the contract defined by the original method in the super class.

The rule of subcontracting, applied to inheritance, is stated in [Mey88, page 256] as:

> `Assertion redefinition rule` : Let r be a routine in class A and s a redefinition of r in a descendant of A, or an effective definition of r if r was deferred. Then pre_s must be weaker than or equal to pre_r, and $post_s$ must be stronger than or equal to $post_r$.

This rule is viewed to be stronger than necessary in JML. Therefore, JML uses following inheritance semantics (see [DL96]) instead of the one given above:

> *Effective precondition for a redefined routine is the disjunction of existing precondition and all preconditions existing in all previous definitions of the routine. Effective class invariants are formed by conjunction. Effective postconditions are formed using following rule:*

$$(\ pre_r \ \texttt{==>} \ post_r \ ) \ \texttt{\&\&} \ (pre_s \ \texttt{==>} \ post_s)$$

When inheriting the specifications the base class may not have any assertions at all. In such a case, the precondition of the super class method is considered to be `true` and hence after combining, the subclass method has precondition `true`.

2.1.5 Specification-only variables [JML keyword: model, depends, and represents]

Specification-only variables are the basis of model-based specification of abstract data types. These are the variables that are used to describe the model that is being implemented by a concrete variable(s), and are usually accompanied by an abstraction function. In JML, this abstraction function is described using a Java expression in a `represents` clause that evaluates to yield the specification-only variable. Such a description will look like:

`represents:` *model_var* `<-` *expression* ;

The steps that will be required are:

1. For every model variable encountered, record the name and the type of the variable in the type checker symbol table.

2. For every represents clause, lookup the symbol table for a corresponding model variable entry:

   (a) If an entry is found, note the abstraction function that is to the right of `<-` in the `represents` clause.

   (b) If entry is not found, just make an entry with this abstraction function for the model variable.

2.1.6 Privacy of specification

Privacy of specification enhances behavioral subtyping semantics for JML (see Section 2.1.4). By using this semantics, individual specifications, for example, invariants, can be declared as `public`, `protected`, or `private` or it can have the default visibility (i.e., a package visibility). Semantically it means that a `public` or `protected` specification should not have reference to `private` class members, and so on.

When translating code in presence of privacy of specifications we need to note the following:

1. Make sure that a `protected` specification does not have reference to any `private` data member or function and a `public` specification does not refer to any `protected` or `private` data member.

2. Specifications tagged `private` will not be subcontracted by a subclass and specifications with default visibility will not be subcontracted by a subclass that is not in the same package as the super class.

3. A `private` model variable or data member may be declared `spec_public` for specification purposes. While semantically, use of such a member is valid in a `public` or a `protected` assertion, when such an assertion is inherited in subclass, it will refer to the `private` member of the super class. Therefore, references to such a member will need to be replaced by a call to a new `protected` method added to the class to enclose reference to this member. The `protected` method will be inherited by the subclass and therefore can be referred by the inherited specification in the subclass.

4. A specification-only (or model) variable (see Section 2.1.5) can be represented by expression that contains reference to `private` members of the class. Similar to the case above, a wrapper method with `protected` visibility will need to be added that evaluates this expression and returns its value.

5. In presence of Java information hiding it may not be possible to just inherit and conjoin (or disjoin) the text of assertion. If one just inherits the text, the text may refer to the private members of the super class. Such translation will result in compilation errors in the translated code. So the final scheme needs to find a way to semantically augment the assertions without disturbing the visibility.

2.1.7 Quantification [JML keyword: forall and exists]

JML increases expressive power of assertions by incorporating *universal* and *existential* quantifiers. The following issues need to be considered while tying to translate the quantifiers to suitable Java code:

1. If the quantification is over an infinite domain, we will need to map it to a bounded domain in Java depending on the type of domain. For example, an infinite domain of integers can be mapped using `Integer.MIN_VALUE` and `Integer.MAX_VALUE`. That is, a quantification given by:

   ```
   \forall (int i) i > 0 ==> Math.pow(i,2) > 0
   ```

   is over infinite domain of positive integers that can be translated to look like:

   ```
   \forall (int i) i > 0 && i <= Integer.MAX_VALUE ==> Math.pow(i,2) > 0
   ```

   But such a mapping may not be so obvious for complex data types, especially the user-defined ones.

2. In case of bounded quantification, JML syntax does not enforce any order on the bounding predicate. That is, a bounding predicate over the domain of integers may be written as *(lower_bound < var && var < upper_bound)* or *(upper_bound > var && var > lower_bound)*. For example, a bounded quantification using integers can be written in JML as:

   ```
   \forall (int i) 0 <= i && i < list.size() ==> list.elementAt(i)!= null
   ```

   or as :

   ```
   \forall (int i) i < list.size() && 0 <= i ==> list.elementAt(i)!= null
   ```

   where `list` is an object of type `java.util.Vector`. To construct an equivalent loop this order will have to be taken into account.

A reasonable scheme could be to construct a lazy enumeration of an object store (see [KC97]) using the type of the object and the bounding predicate. A *lazy enumeration* would be the one in which an object will not be constructed until its value is asked; it will be constructed only when the enumeration is asked to give the next element of the domain. This scheme still requires the enumeration to know the upper and the lower bounds in order to terminate normally.

2.1.8 Labeled Assertions [JML keyword: label]

Like in Eiffel, an expression in JML can be labeled for readability. This label can give contextual information in the event of failure. For example, one can print the label (if any) of the failed assertion in the error message. Therefore we should record and later use the label as a part of information to be displayed in the event of assertion failure.

2.1.9 Order of subexpressions and Run-time Exceptions

In JML, order of subexpressions in a predicate is immaterial. That is, a predicate can be written as *(((x/y) > 1) || (y == 0))* or *((y == 0) || ((x/y) > 1))*. Translated code for this predicate would, however, impose an order because of short-circuited operator '||' in Java. This creates a problem in presence of code that may throw a run-time exception. For example, in the predicate mentioned above, if (y = 0), subexpression (x/y) will throw `DivideByZero` exception at run-time in Java. Because of such cases, the exception needs to be caught and the value of the subexpression be set to `false`, so that when these expressions are evaluated and combined, the assertion evaluates without throwing any exceptions. This scheme also eliminates the order imposed by Java as both of the expressions given above will evaluate to `true` if (y = 0).

2.1.10 Refinement Syntax [JML keyword: refines]

JML permits a user to write specifications for a given module in several files. Use of such separate files allows the user to separate the concrete implementation of the module from its specifications. A `refines` clause in the corresponding Java file that implements the module tells the reader where to find its specifications [LBR99, Page 3]. Such a refinement can be done recursively using more than one file, where one file refines the other and so on. Therefore, a complete specification of the given module needs to be obtained by merging all the individual specifications from these files.

2.1.11 Non-executable assertions

Until now, we discussed various JML features and how they may be translated into to Java syntax. However, there are a lot of features that do not lend themselves easily to such translation. Assertions that use such features will essentially be considered non-executable. *Non-executable* assertions are the specification expressions that can not be translated to Java due to various factors:

1. Presence of certain JML features in the assertion. For example, quantifiers (see Section 2.1.7) will be considered non-executable in the first version of the tool, implying that any assertion with keywords `forall` or `exists` will be non-executable.

2. If assertion expressions contain subexpressions that are non-executable, then the subexpression will be considered as `\not_specified`.

3. A non-executable assertion will be considered `\not_specified` for the purpose of inheritance.

In the next section, we discuss some features of Java that require attention while during the process of translation.

2.2 Java features

This refers to Java language features and constructs that require special consideration.

Threads In Java, a multi-threaded program may spawn a number of threads of execution. These threads share the instance variables of the class but have different flow-of-control. This implies that each thread can invoke methods independently. It also means that each thread will check specifications for each method separately. So the generated source code needs to be thread-aware (re-entrant).

Interfaces Interfaces are Java modules that can have only `static` data members and method prototypes. Actual implementation of the methods have to be in a class that implements the particular interface. But, in JML, an interface method may have specifications that need to be satisfied by the implementation of that method. So we need to combine the specifications written for an interface with that of the class, which implements the interface, before generating the assertion checking code for the class.

Name conflicts Java allows one to define packages so that developers can avoid naming conflicts. Files in a package need to reside in a particular subdirectory of path reachable through `CLASSPATH` environmental variable. Also all the public classes and interfaces need to be in files with the same name, with a ".java" extension. When the tool inserts new code in a class, the combined code needs to be written to a file and then compiled using a standard Java compiler. This new file still needs to be in the same package and have the same file name so that it can be compiled and used by others. This results in naming conflict for the new source code.

Overwriting an original file is one way to avoid naming conflicts. But the new file may be highly unstructured as it is required to combine and carry a lot of information — for example, invariants from a super class or interface the class implements. One way to avoid overwriting, is to create a source code library structure similar to that of the existing code. By manipulating the `CLASSPATH` environment variable one may be able switch between the new and the old code.

2.3 User features

As mentioned in Chapter 1, a user is expected to use this tool to translate specifications written in JML into Java source code that will perform checks at run-time. Such a translation may add a lot of source code into the existing program. This may not only increase the size of the code but also slow its execution considerably. Therefore, the user needs to have fine control on the process of translation of the code as well as on the process of run-time checking. This control is provided through various options to the user. This section discusses various issues that need to be considered in order to provide such support.

2.3.1 Compile-time options

These are the options that the user requires to control the process of translation. These options will mainly affect the size of the code that will be generated upon translation. The following options are considered to be useful:

- The user should be able to generate the code only for selected types of assertions (preconditions, invariants, etc.). By default, the tool will generate the code only for preconditions.
- The user should be able to get warnings when the tool is not able to generate the code for a given specification, for example, if the specification is non-executable (see Section 2.1.11). Such a warning should have enough context information to help the

user identify the cause so that he or she may take measures to correct the problem if desired.

2.3.2 Run-time options

Run-time options let the user control the actual assertion checking process at run-time. These options can affect the speed of execution of the translated code. To be able to use some of the options at the run-time, the user must first generate the code with proper options (as described in previous section) at the compile-time. For example, for checking preconditions at the run-time, option to generate code for preconditions or all assertions must be used at the compile-time. The following options can give good control at run-time:

- The user should be able to turn execution of the code that checks assertions on and off.
- There are several ways to recover from failure when a particular assertion is not satisfied. Some of those include giving error messages, throwing run-time exception, and halting. The tool will have to provide such different ways for recovery and an option to the user to select one of them.

Next section discusses other issues that influenced the design of the tool.

2.4 Other issues

Assertion expressions may call other methods of the same or a different class. These methods themselves may have assertion checking code. Such a recursion may give rise to circular dependencies, therefore such recursion should be avoided [Mey88, pages 156–157]. To avoid this problem, assertion checking needs to be turned off once the execution is inside the code that is checking assertion expression for a given method. This means that the execution of the assertion checking code for the methods invoked as a part of assertion checking code will be turned off.

In the next chapter we discuss the design and implementation of the tool.

Chapter 3

Design and Implementation

This chapter discusses the design of the tool. The design is mainly based on issues discussed in Chapter 2. In Section 3.1 we discuss the transformations the original source code goes through and then in Section 3.2 we discuss the design of the translator that does the transformations.

3.1 Transformations

In this section we give a line-by-line explanation of code transformations using the examples in figures 3.1 and 3.2, where figure 3.1 is the original source code and figure 3.2 is the transformed code.

Consider figure 3.2. Lines 1–7 contain the class definition, the method header, and the method specification and are identical to lines 1–7 in the original source code (see figure 3.1). Lines 8–24 are the ones that are added by the translator to check the precondition of the method `isqrt` at run-time, while the block spanning lines 25–27 is the original body of the method `isqrt`.

The translator converts the precondition into an if-then block on lines 8–24. This block is inserted before the original method body so that the precondition is checked before the method is executed. The condition in the `if` statement is a call to `isActive()` (lines 8 and 9) to check if this code should indeed be executed at run-time. The method `isActive()` checks if:

1. this type of assertion is allowed to be checked at run-time. This is determined by passing the type of this assertion (`PRECONDITION` on line 9) to `isActive()`. `isActive()` checks if a run-time system option, called an *assertion level*, permits us to check this type of assertion. Any assertions above assertion level should not be checked. For example, a precondition can be checked if the level is `PRECONDITIONS_ONLY` (implies that check only preconditions) or `ALL` (implies that check all types of assertions), but checking a postcondition requires the level to be set to the later.

2. the call to this method at run-time is part of assertion checking code of any other method for the same thread of execution. If yes, then this code should not be executed to avoid recursions as described in Section 2.4.

   Possible recursion is checked by the run-time system by maintaining a per-thread flag in a hashtable. This table maps a `Thread` object to a `boolean` flag. Whenever a thread enters the code to check assertions in a method for the first time, an entry is

```
public class IntMathOps {                                    // 1
  public static int isqrt(int y)                             // 2
  /*@ normal_behavior                                        // 3
    @   requires: y >= 0;                                    // 4
    @   ensures: \result * \result <= y                      // 5
    @         && y < (\result + 1) * (\result + 1);          // 6
    @*/                                                      // 7
  { return (int) Math.sqrt(y); }                             // 8
}                                                            // 9
```

Figure 3.1: Original source code. (Adapted from [LBR99])

> created in this table and is set to `false`; a call to `enterAssertionCheck()` on line 10 and 11 does this. Whenever the thread tries to execute assertion check code for any other method while inside the first assertion check code, it checks the value of this flag in the table. If the flag is set to `false` then the assertions are not checked for this method. On exiting the first assertion check code, a call to `exitAssertionCheck()` (line 23 and 24) destroys the flag.
>
> The entry in the table is destroyed to keep the size of the table under control, as a large number of threads can exist and can be created at any time in the system. Also the table will be required to be cleaned up time-to-time by deleting entries corresponding to the terminated threads. To avoid the overhead of cleaning up, the entry corresponding to a thread object is created and destroyed on-the-fly.

As we continue to describe figure 3.2, lines 12–18 actually evaluate the precondition expression described by the `requires` clause on line 4. The value of the expression is assigned to a temporary `boolean` variable, `__jml0`. The whole evaluation is enclosed in a `try-catch` block to recover from any runtime exceptions thrown by this code so that it also takes care of the order of evaluation (see Section 2.1.9). The value of this variable along with some context information (in this order) like the type of the assertion — `PRECONDITION`, the name of the file — "IntMathOps.java", the line number of the assertion clause in the original definition — 4, the name of the class — "IntMathOps", and the name of the method — "isqrt" are passed to a method `assert()`[1] (lines 19–22). This method checks the value of the variable. If it is `false`, implying a failed assertion, then `assert()` combines the context information to create an error message and throws `AssertionException()` at run-time. The user can change the failure recovery mechanism to get an error message displayed instead of an assertion being thrown.

In the next section, we describe the design of the translator that actually transforms the code.

[1]The last parameter to `assert()` is currently `null` and is reserved for future enhancements to include support for `label` information.(see Section 2.1.8)

```
public class IntMathOps {                                         //1
 public static int isqrt(int y)                                   //2
  /*@ normal_behavior                                             //3
       requires: (y >= 0);                                        //4
       ensures: (((\result * \result) <= y)                       //5
                && (y < ((\result + 1) * (\result + 1))));        //6
  @*/                                                             //7
  {
     if (edu.iastate.cs.jml.checker.runtime.Checker.isActive(     //8
         edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION) //9
         ){
          edu.iastate.cs.jml.checker.runtime.Checker.             //10
                    enterAssertionCheck();                        //11
          boolean __jml0;                                         //12
          try  {                                                  //13
            __jml0 = (y >= 0);                                    //14
          }                                                       //15
           catch(Exception e) {                                   //16
            __jml0 = false;                                       //17
          }                                                       //18
          edu.iastate.cs.jml.checker.runtime.Checker.assert(      //19
            __jml0,                                               //20
          edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,//21
            "IntMathOps.java", 4, "IntMathOps", "isqrt", null );  //22
          edu.iastate.cs.jml.checker.runtime.Checker.             //23
                    exitAssertionCheck();                         //24
        }
        {                                                         //25
          return (int )Math.sqrt(y);                              //26
        }                                                         //27
   }
}
```

Figure 3.2: Generated source code.

3.2 The Design of the Translator

This section describes various phases of the translator and interaction between these phases. This design discussion assumes the translation of only preconditions although the design can be extended to include other types of assertions.

For describing the design, we use the class `BoundedColorPoint` (figure 3.3) as an example. This class extends class `Point` (figure 3.4) while implementing interface `ColorPoint` (figure 3.5) and refining file `BoundedColorPoint.jml-refined` (figure 3.6).

3.2.1 Phase 1 - Type Checking

The JML type checker forms the first phase of the translator for the reasons explained below.

Specification Inheritance and Refinement

In general, in JML, a complete specification of a method need not be attached to the method's implementation. The specification can exist in an interface that this class implements (for example, method `setColorPoint` on lines 33–43 in class `BoundedColorPoint` in figure 3.3 implements method `setColorPoint` on lines 2 and 3 in interface `ColorPoint` in figure 3.5), it can be described in another file using refinement (for example, method `setResolution` on lines 22–31 in class `BoundedColorPoint` in figure 3.3 refines method `setResolution` on lines 6–9 in file `BoundedColorPoint.jml-refined` in figure 3.6), and there can be multiple levels of such a refinement (see Section 2.1.10). Also to enforce behavioral subtyping (see Section 2.1.4), a specification of an overriding method in a subclass needs to be augmented with the specification of the method it is overriding (for example, method `setPoint` on lines 11–21 in class `BoundedColorPoint` in figure 3.3 overrides method `setPoint` on lines 6–14 in class `Point` in figure 3.4). Therefore, the translator also needs to get the specifications from a superclass. This requires the translator to collect all the specifications for a method from all of these possible sources.

Specification-only Variables

Also, the specifications in the given file may refer to model variables (for example, variable `resolution` on line 25 of class `BoundedColorPoint` in figure 3.3). These model variables can be inherited from a superclass, declared in an interface, or declared in a file that this class refines. The user also needs to specify a `represents` clause that describes how to construct the value of these model variables (for example, `represents` clause on line 9 of class `BoundedColorPoint` in figure 3.3). To generate the code that checks assertions that refer to model variables, one needs to collect these declarations along with their corresponding `represents` clauses.

Type Checking Original Code

Finally, the code that the translator generates is going to be compiled using a standard Java compiler. Therefore, as mentioned in Section 1.2, the extra code that the translator adds should be free of type errors. As the translator makes use of the information stored in the original source to generate this extra code, it is a good idea to type check the original code before using it to do any manipulations.

```java
//@ refine: BoundedColorPoint <- "BoundedColorPoint.jml-refined";
public class BoundedColorPoint extends Point implements ColorPoint  {

  protected int color;

  private int maxx = 640;
  private int maxy = 480;

  //@ private represents: resolution <- maxx * maxy ;

  public void setPoint(int _x, int _y )
  /*@ also
    @   private_normal_behavior
    @     requires: _x <= maxx && _y <= maxy;
    @     ensures: x == _x && y == _y;
    @*/
  {
      x = _x;
      y = _y;
      color = 0;
  }
  public void setResolution(int _maxx, int _maxy)
    /*@ also
      @  private_normal_behavior
      @     requires:     resolution >= 0;
      @     ensures: maxx == _maxx && maxy == _maxy;
      @*/
    {
        maxx = _maxx;
        maxy = _maxy;
    }

  public void setColorPoint(int _x, int _y, int _color )
  /*@ also
    @   private_normal_behavior
    @     requires: _x <= maxx && _y <= maxy;
    @     ensures: x == _x && y ==_y && color == _color;
    @*/
  {
      x = _x ;
      y = _y ;
      color = _color;
  }
}
```

Figure 3.3: Original Source Code- BoundedColorPoint.java

```
public class Point {

    protected int x;
    protected int y;

    protected void setPoint (int _x, int _y)
    /*@  protected_normal_behavior
      @     requires: _x >= 0 && _y >= 0;
      @     ensures: x == _x && y == _y;
      @*/
    {
        x = _x;
        y = _y;
    }
}
```

Figure 3.4: Super class - Point.java

```
public interface ColorPoint  {
    public void setColorPoint(int _x, int _y, int _clr );
    //@ requires: _x >= 0 && _y >= 0 && _clr >= 0;
}
```

Figure 3.5: Interface - ColorPoint.java

```
public class BoundedColorPoint  extends Point implements ColorPoint  { // 1
    //@public model int resolution;                                      // 2
                                                                         // 3
    public void setPoint(int _x, int _y );                               // 4
                                                                         // 5
    public void setResolution(int _maxx, int _maxy) ;                    // 6
    protected_normal_behavior                                            // 7
        requires: _maxx >= 0 && _maxy >=0;                               // 8
        ensures: resolution <= Integer.MAX_VALUE;                        // 9
                                                                         // 10
    public void setColorPoint(int _x, int _y, int _color );              // 11
}                                                                        // 12
```

Figure 3.6: Refinement - BoundedColorPoint.jml-refined

Keeping these issues in mind we note that the JML type checker already has multiple phases that traverse the abstract syntax tree to collect type information from all the files that relate to the given class [Gan98]. These files include the files that this class refines, superclass of this class, and the interfaces that this class implements. Therefore the JML type checker is a good place to look for the information that we need for generating the code besides getting the original code type checked.

But the current design of the type checker just collects the type attributes relevant for type checking, stores them in a symbol table, type checks the code using the symbol table and throws the table away. To use the type checker to collect the information we need, the following modifications have been made:

- Currently, as mentioned above, the type checker stores the type information in a symbol table in organized as levels of environments (see [Gan98]). An *environment* essentially defines a scope — local, class, package, etc., in a file. To get type information about a particular symbol one has to lookup the symbol table recursively through these environments. This could be very time consuming. So, the type checker was modified to attach the type information to the corresponding AST nodes besides storing it in the symbol table. This *annotated* AST can then be passed to the later phases where they can easily extract the type information from the node itself.

- A complete specification of a method is a specification augmented from three possible sources — superclass, interfaces and any file that this file refines. This requires us to lookup the superclass to find the method that is being overridden by this method, lookup the interfaces to find the methods that are being implemented by this method and lookup the files to find the method that is being refined by this method. After we find these methods, we need to extract their specifications and augment them with the current specification of the method. The type checker currently just locates these methods for us without extracting any information. To store this information, a method's type attribute is modified to include a list of definitions of these methods that it overrides, implements and refines. This list is updated every time a previous definition of the method is found. Such a list can be used later to augment the specifications and then generate the code. Also note that a method overrides another method if it has same return type, same name, same number of formal parameters, and all the formal parameters have the same type in the same order. This implies that actual names of the formal parameters may not match even though the types match. As these parameters may be referred to by the precondition of a given method, renaming of formal parameters may be required before actually storing the definition of overridden method. For example, the formal parameter *_clr* in the definition of `setColorPoint` in interface `ColorPoint` in figure 3.5 will need to be renamed to *_color*.

- The type checker uses the same type of attributes for a model variable and a non-model variable. But we require a symbol table entry for a model variable to also have an attribute that contains information from its corresponding `represents` clause.

After the type checking, the next phase takes the annotated AST and the symbol table as input, and uses the information to generate the actual code. In case there are any type errors, the translator does not generate any assertion checking code for the given class.

3.2.2 Phase 2 - Code Generation

This phase walks the annotated AST and modifies the AST to actually generate transformed code to check the assertions. Before it can do that it needs to take care of another issue.

A class may have `spec_public` members and `private represents` clauses[2] that describe some model variables. When another class inherits such a class in Java, it inherits all the public and protected (and package visible, if both are in the same package) data members but not the private ones. But in JML, such a subclass can contain specifications that refer to `spec_public` members of a superclass and public or protected model variables with private represents clause. Moreover, as a complete specification of an overriding method also includes the specification from the superclass (see Section 2.1.4), the specification borrowed from the superclass can also refer to these variables. Therefore, if we try to generate code to check augmented assertions for the method in the subclass, the generated code will end up referring to `private` data members of the super class.

To get around this visibility problem, such specification-only variables (model and spec_public) are wrapped inside a `protected` method in the superclass. That is, this phase creates one `protected` method per specification-only variable. For a model variable, the body of the method is essentially the expression in the `represents` clause that evaluates its value. For a `spec_public` variable, it is the variable itself. Since these methods have `protected` visibility, they can be inherited without any visibility errors in Java.

To create such a method, this phase generates a unique name using the class name and the variable name itself; the return type of the method is based on type of the variable this method encloses. After creating these methods, this phase actually inserts these methods as if they are members of this class. For example, for public model variable `resolution` defined in `BoundedColorPoint.jml-refined`, a method `__jmlresolutionBoundedColorPoint()` will be created as shown in figure 3.7 using `private represents` clause on line 9 of class `BoundedColorPoint` in figure 3.3.

[2] A `private represents` clause implies that the expression that describes a `public` (or `protected`) model variable refers to private members of the class. For details, see Section 2.1.5 and Section 2.1.6.

```
protected int __jmlresolutionBoundedColorPoint() {
        return (maxx * maxy);
}
```

Figure 3.7: Model variable method for `resolution`

Also, to handle all model variables symmetrically, the tool inserts these methods even for the model variables whose corresponding `represents` clause has public or protected visibility. We require these methods for only those model variables which have a `represents` clause in the current class, interface that this class implements and the file that this class refines. In other words, we do not add these methods for the model variables which have a `represents` clause in the super class, we assume that the instrumented super class would contain these methods.

After inserting such methods, phase 2 walks the AST. For every method node it encounters, it retrieves the type attribute that was attached in phase 1. This type attribute contains a list of previous definitions of this method. It, then, extracts the specifications from the previous definitions and combines them with the current specification. For example, augmented precondition expression for method `setPoint` on lines 11–21 in class `BoundedColorPoint` in figure 3.3) will be:

```
requires: (_x <= maxx && _y <= maxy) || (_x >= 0 && _y >= 0);
```

Such augmented specification is stored back in the type attribute so that it can be retrieved later. While augmenting the specification, the AST walker replaces references to specification-only variables by calls to the corresponding methods. For example, for method `setResolution` on lines 22–31 in class `BoundedColorPoint` in figure 3.3, the precondition would now be:

```
requires: (_maxx >= 0 && _maxy >= 0) ||
(__jmlresolutionBoundedColorPoint() >=0);
```

It also eliminates the specifications that are tagged `redundant`, meaning that these specifications just restate some fact.

As it continues to walk the AST, whenever it encounters a method definition with a method body, it extracts the augmented specifications for that method from the type attribute.

It then extracts the precondition expression and walks it to check if the assertion expression is executable. As described in Section 2.1.11, a non-executable expression would be the one that can not be converted to Java. This is checked by looking for certain keywords in the assertion expression. If these keywords are present in the assertion expression subtree, then it is flagged as non-executable.

If the precondition expression is executable, it wraps the code in an `if-then` block AST. This block is then inserted at the front the existing method body as mentioned in Section 2.1.1.

Finally, this modified AST is then walked by a JML unparser. This unparser takes the AST and *unparses* it. That is, it generates the equivalent Java code back such that parsing the generated Java code gets an AST with same structure back. As this unparser is an independent component that assumes the AST to have a proper syntactic structure for Java and JML, care needs to taken while modifying the AST to maintain the structure of legitimate Java code.

Appendix A shows the code generated for class `BoundedColorPoint`[3]. Note how it retains all of the code from original `BoundedColorPoint` shown in figure 3.3 while adding the new code.

In the next chapter we discuss further enhancements to this scheme so as to include more JML and Java features, to check more types of assertions, etc.

[3]The formatting of the code is a bit modified in order to fit it in the width of the paper

Chapter 4

Future Work

This section discusses possible enhancements to the existing tool. Because JML has a large number of features, all of the features of JML could not be taken into account while designing this tool.

The tool currently can not handle quantification. As mentioned in Section 2.1.7, to support quantification, one can try construction of enumeration over sets at run-time.

The amount of context information given to the user (see Section 1.2) is currently limited. Also this context information relates to the modified source code and can not effectively give clues about the original location of the assertion. By modifying symbol table, one can attempt to carry more information about the original location of the failed assertions. Mainly, the labels of labeled assertions (see Section 2.1.8) can be effective in giving context information. But note that, in JML, even the subexpressions in an assertion can have labels. Hence one needs to consider the scope of a label. Therefore, to give fine-grained context information, one can check the run-time value the subexpressions evaluates to; if it evaluates to `false` (and it occurs in a context that needs a `true` value), then one can include its label as a part of context information. This can tell the user what was exactly responsible for the failed assertion.

One can also look at optimizing performance by doing some amount of static analysis. For example, if an assertion expression evaluates to `true` at compile-time, the tool should avoid generating the code. This can result in reduction in both space and time.

One more small enhancement possible is that instead of throwing exception or displaying error message, the tool could log the failures in a text file so that the user can monitor the progress remotely.

Finally, the tool could be extended to support more types of assertions like invariants, general assertions, etc.

Chapter 5

Related Work

Idea of checking assertions at run-time is not new. In the best-known example, Meyer [Mey88] pioneered the support for *programming by contract* in Eiffel. Eiffel takes a pragmatic approach to run-time assertion checking by not providing features like quantification due to difficulty in providing run-time support for checking them. Eiffel also follows the principle of *subcontracting*. It uses the same rules for augmenting the preconditions. However, the subcontracting is aided by the lack of inheritance barriers. As Eiffel supports information hiding through `export`, independent of inheritance, it is possible for the redefined routines to simply inherit the clauses of antecedents. However, Java's information hiding semantics is much complex than Eiffel's. In Java, a `private` member of the class can not be referred by the subclass directly. Therefore, in JML, one can not simply restate the assertion in the subclass as it may refer to `private` members of the super class. Therefore, the run-time checker semantically augments the specifications as oppose to just restating them. Eiffel also lacks support for model-based specifications. The checker handles model-based specifications in JML to allow the programmers to use assertions to describe the behavior of interfaces and abstract classes, which is difficult otherwise. Sather [Gea96], is a new language developed at the International Computer Science Institute (ICSI) at Berkeley. It looks similar to Eiffel but has a lot of features that are different than Eiffel. It does include support for assertions through what they call "Safety Features". These features enjoy built-in support but with few restrictions. Sather has interesting limitations — for example, it insists that the preconditions can not impose conditions on internal state of the class as the client can not be assumed to be aware of this state. So the client should not be expected to satisfy the condition that may have internal state description. This implies that the preconditions are allowed to be stated only in terms of input parameters to the routine. This eases the specification inheritance as the input parameters to a routine are not hidden in the subclass. JML, on the other hand, permits one to write the preconditions that use the internal state of the module and hence the `private` members. Sather also suffers from the drawback of not supporting model-based specifications like Eiffel.

Work of Porat and Fertig, `exlC++` [PF95], tries to extend C++ — not just syntactically but semantically. It tries to take into account behavioral subtyping — inheritance of assertions. It gives users a lot of flexibility, allowing them to select the extent and the type of assertion check (for example, preconditions only, all assertions, etc.) using compiler options. This is a source-level translator which imposes some restrictions: the user can not control the way recovery is handled, it does not support quantification, and it restricts the use of assertions to non-static member functions of a class. The Run-time Assertion

Checker for Java using JML derives a lot of concepts and ideas from this work. The main difference between the checker and exlC++ is that the checker needs to handle refinement and has to do type checking before it can generate the code, while the exlC++ does not have features like refinement and also it assumes correct typing for the specifications. The checker also provides an option to the user that lets him or her select the way the program should recover from assertion failures, and it also supports the assertions written for static functions. Also the checker supports model-based specifications that exlC++ does not.

Welch and Strong [WS98], in their article, discuss the shortcomings of the C/C++ assertion mechanism through its `assert()` macro. The article then suggests an extension of this mechanism through use of exceptions in C++. However the mechanism is a pure extension of macro processing. It does not take care of the semantic implications of various features like behavioral subtyping for subclasses. It just defines a pure macro translator through a preprocessor. More expressive power, however, is given to the programmer through additional macro definitions and by adding a wrapper to the existing `assert()` macro. The default `assert()` macro now is wrapped inside an `Assertion` exception class. More expressive power is imparted through additional macros like `REQUIRE`, `ENSURE`, `INVARIANT`, `THERE_EXISTS` and so on. It still abides by Meyer's ([Mey88]) programming by contract idea. On the other hand, Java language does not have a macro pre-processor. This makes it difficult to use a simplistic macro translation scheme as is used by this work. Also, as mentioned, this work does not support subcontracting, which is an essential feature for object-oriented languages like Java and C++. The run-time assertion checker is more than just a source-to-source translator — it enhances the semantics of specification through the use of inheritance and model-based specifications.

A few third-party tools have been designed to add programming by contract support to Java; iContract ([iCo00]), by Reliable-Systems is one such tool. iContract is a source code preprocessor that identifies annotated Javadoc-style assertion expressions that use tags like `@pre`, `@post`, etc., and converts these into assertion check code. It supports only class invariants, preconditions and postconditions but provides support for quantification for enumerators, and also propagation of assertions via inheritance. As the assertions are annotations, they go undetected by a standard Java compiler. iContract also lacks support for model-based specifications.

Rosenblum [Ros95], in his article, describes a tool called APP, an Annotation Pre-Processor for C. The work mainly strives to add *annotations* to the existing C programs to describe the assertions. This description follows a certain syntactic structure. For all practical purposes, the work recognizes only four different types of assertions through four different keywords. The specifications are embedded through annotations that look like comments to a normal C compiler but help APP identify the specification structure. This also is a source-level translator that works by using additional variables, buffers, etc. It gives additional power by using looping syntax to permit *existential* and *universal* quantification over finite domains of any type of data structure (array, linked list etc.). However, the use of the tool is limited to C. Therefore it is does not need to support subcontracting and does not need to worry about privacy of variables. It also lacks support for model-based specifications.

Conclusions

The important contribution of the “Run-time Assertion Checker” are supporting model-based specifications that are not provided by most of the specification languages. It also handles refinement and behavioral subtyping in presence of complex information hiding semantics of Java. Also the checker is easy to use and gives the user control over generating and executing the extra code to check the preconditions.

Acknowledgements

This work was supported in part by NSF grant CCR-9803843.

Thanks to Clyde Ruby and Curtis Clifton for discussions about the design of the tool. Thanks to my colleagues in the “Writer’s Workshop” seminar course of Spring 2000 that helped me improve my writing and hence this design document. Thanks to Dr. Doug Jacobson for serving on my POS committee and also to Dr. Albert Baker to take time out of his work schedule to serve on the POS committee. Thanks to Dr. Gary Leavens for his guidance in designing the tool and for discussions to clarify the semantics of JML and of course, to agree to guide me as a major professor through my Masters’ degree in Computer Science.

Bibliography

[DL96] Krishna Kishore Dhara and Gary T. Leavens. Forcing Behavioral Subtyping Through Specification Inheritance. March 25–29 1996. 18th International Conference On Software Engineering.

[Gan98] Ananad Ganapathy. Design and Implementation of a JML Type Checker. Master's thesis, Iowa State University, 1998.

[Gea96] Benedict Gomes and David Stoutamire et al. *A Language Manual for Sather 1.1*, October 1996.

[iCo00] iContract,The Java(tm) Design by Contract(tm) Tool, February 2000. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[KC97] Miguel Katrib and Jesús Coira. Improving Eiffel Assertions Using Quantified Iterators. *JOOP*, 10(7):35–43, November 1997.

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Technical report, Iowa State University, December 1999.

[LC92] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. Technical Report 92-16, Iowa State University, Department of Computer Science, May 1992.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Par97] Terence Parr. Exactly 1800 words on languages and parsing. Excerpt from: Language Translation Using PCCTS & C++, January 1997.

[Par99] Terence Parr. *ANTLR Reference Manual*. http://www.antlr.org/doc/index.html, 2.6.0 edition, May 1999.

[PF95] Sara Porat and Paul Fertig. Class assertions in C++. *JOOP*, 8(2):30–37, May 1995.

[Ros95] David S. Rosenblum. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, January 1995.

[Sch99] Gary L. Schaps. Compiler Construction with ANTLR and Java. *Dr. Dobb's Journal*, March 1999. http://www.ddj.com/articles/1999/9903/9903h/9903h.htm.

[WS98] David Welch and Scott Strong. An Exception-Based Assertion Mechanism for C++. *JOOP*, 11(4):50–60, Jul 1998.

Appendix A

Generated Code

The following code is generated by the tool for class BoundedColorPoint in figure 3.3. (Note: The code is formatted to fit the width of the paper).

```
//@ refine: BoundedColorPoint <- "BoundedColorPoint.jml-refined";
public class BoundedColorPoint extends Point implements ColorPoint
{
    protected int __jmlresolutionBoundedColorPoint()
    {
        return (maxx * maxy);
    }

    protected int color;
    private int maxx = 640;
    private int maxy = 480;

  //@private represents: resolution <- (maxx * maxy);
    public void setPoint(int _x, int _y)
      /*@
      also
        private_normal_behavior
          requires: ((_x <= maxx) && (_y <= maxy));
          ensures: ((x == _x) && (y == _y));
      @*/

      {
        if (edu.iastate.cs.jml.checker.runtime.Checker.isActive(
          edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))        {
          edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();
          boolean __jml0;
          try           {
              boolean __jml1 ;
              try    {
                      boolean __jml3;
```

```
            try             {
                __jml3 = (_x <= maxx);
            }
            catch(Exception e)            {
                __jml3 = false;
            }
            boolean __jml4;
            try             {
                __jml4 = (_y <= maxy);
            }
            catch(Exception e)            {
                __jml4 = false;
            }
            __jml1 = (__jml3 && __jml4);
        }
    catch(Exception e)          {
        __jml1 = false;
    }
    boolean __jml2;
    try {
        boolean __jml5;
        try {
            __jml5 = (_x >= 0);
        }
        catch (Exception e) {
            __jml5 = false;
        }
        boolean __jml6;
        try {
            __jml6 = (_y >= 0);
        }
        catch (Exception e){
            __jml6= false;
        }
        __jml2 = (__jml5 && __jml6);
    }
    catch (Exception e) {
        __jml2= false;
    }
    __jml0 = (__jml1 || __jml2);

}
catch (Exception e){
    __jml0= false;
}
```

```
      edu.iastate.cs.jml.checker.runtime.Checker.assert(__jml0,
      edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,
      BoundedColorPoint.java", 16, "BoundedColorPoint", "setPoint"
      , null);
      edu.iastate.cs.jml.checker.runtime.Checker.exitAssertionCheck();
    }
         {
      x = _x;
      y = _y;
      color = 0;
    }
  }

public void setResolution(int _maxx, int _maxy)
  /*@
  also
    private_normal_behavior
      requires:   (resolution >= 0);
      ensures: ((maxx == _maxx) && (maxy == _maxy));
  @*/

  {
    if (edu.iastate.cs.jml.checker.runtime.Checker.isActive(
        edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))        {
      edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();
      boolean __jml0;
      try          {
          boolean __jml1 ;

          try   {
                  __jml1 = (__jmlresolutionBoundedColorPoint() >= 0);
          }
          catch(Exception e)          {
              __jml1 = false;
          }
          boolean __jml2;
          try    {
                  boolean __jml3;
                  try             {
                      __jml3 = (_maxx >= 0);
                  }
                  catch(Exception e)            {
                      __jml3 = false;
                  }
                  boolean __jml4;
                  try             {
                      __jml4 = (_maxy >= 0);
                  }
```

```
                catch(Exception e)           {
                    __jml4 = false;
                }
                __jml2 = (__jml3 && __jml4);
            }
        catch (Exception e) {
            __jml2= false;
        }
        __jml0 = (__jml1 || __jml2);
    }
    catch (Exception e ){
        __jml0 = false;
    }

    edu.iastate.cs.jml.checker.runtime.Checker.assert(__jml0,
    edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,
    "BoundedColorPoint.java", 28, "BoundedColorPoint","setResolution"
    , null);
    edu.iastate.cs.jml.checker.runtime.Checker.exitAssertionCheck();
   }

          {
    maxx = _maxx;
    maxy = _maxy;
   }
  }
public void setColorPoint(int _x, int _y, int _color)
  /*@
  also
   private_normal_behavior
    requires: ((_x <= maxx) && (_y <= maxy));
    ensures: (((x == _x) && (y == _y)) && (color == _color));
  @*/

  {

   if (edu.iastate.cs.jml.checker.runtime.Checker.isActive(
       edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION))        {
    edu.iastate.cs.jml.checker.runtime.Checker.enterAssertionCheck();

    boolean __jml0;
    try           {
        boolean __jml1;

        try    {
                boolean __jml3;
                try             {
                    __jml3 = (_x <= maxx);
```

```
        }
        catch(Exception e)            {
            __jml3 = false;
        }
        boolean __jml4;
        try            {
            __jml4 = (_y <= maxy);
        }
        catch(Exception e)            {
            __jml4 = false;
        }
        __jml1 = (__jml3 && __jml4);
    }
catch(Exception e)          {
    __jml1 = false;
}
boolean __jml2;
try {
    boolean __jml5;
    try {
        boolean __jml7;
        try {
            __jml7 = (_x >= 0);
        }
        catch (Exception e) {
            __jml7 = false;
        }
        boolean __jml8;
        try {
            __jml8 = (_y >= 0);
        }
        catch (Exception e){
            __jml8 = false;
        }
        __jml5 = (__jml7 && __jml8);

    }
    catch (Exception e) {
        __jml5 = false;
    }
    boolean __jml6;
    try {
        __jml6 = (_color >=0);
    }
    catch (Exception e){
        __jml6= false;
    }
    __jml2 = (__jml5 && __jml6);
```

```
            }
            catch (Exception e) {
                __jml2= false;
            }
            __jml0 = (__jml1 || __jml2);

        }
        catch (Exception e){
            __jml0= false;
        }

        edu.iastate.cs.jml.checker.runtime.Checker.assert(__jml0,
        edu.iastate.cs.jml.checker.runtime.TypeCode.PRECONDITION,
        "BoundedColorPoint.java", 39, "BoundedColorPoint","setColorPoint"
        , null);
        edu.iastate.cs.jml.checker.runtime.Checker.exitAssertionCheck();
      }
              {
        x = _x;
        y = _y;
        color = _color;
      }
    }
}
```