Contributions to the JML Project: Safe Arithmetic and Non-null-by-default

Hao Xi

A Thesis
in
The Department
of
Computer Science

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

September 2006

Your file Votre référence
ISBN: 978-0-494-20784-0
Our file Notre référence
ISBN: 978-0-494-20784-0

ABSTRACT

Contributions to the JML Project: Safe Arithmetic and Non-null-by-default

Hao Xi

The MultiJava Compiler (MJC) is an extension to the Java programming language that adds open classes and symmetric multiple dispatch. The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) that can be used to specify both simple DBC and full behavioral interface specifications. The JML toolset is based on MJC and contains tools such as the JML (type) checker and the JML Runtime Assertion Checker (RAC). JMLb, is a new version of JML that supports arbitrary precision integers and safe-arithmetic. In this thesis we present the implementation of (bytecode level) support for safe-math integral arithmetic in MJC as well as a performance evaluation of this new version of MJC, in comparison with other Java compilers. Another main enhancement presented in this thesis is the implementation of a non-null statistics gathering tool in the JML checker. An overview of the desugaring process for various kinds of JML method specifications is given. In addition, rules for judging non-null usage are described by presenting examples of different scenarios.

Acknowledgments

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 Perry James for his guidance and comments while reviewing my thesis.

Finally, I thank my families for their love and support and my parents for their encouragement and support.

Table of Contents

List of Figures

List of Tables

1 Introduction

One of the areas of activity of Concordia University's Dependable Software Research Group (DSRG) is in the formal specification and verification of Java programs. One active area of research is the Java Modeling Language (JML), a behavioral interface specification language (BISL) in which specifications are expressed in the form of contracts. The JML compiler is based on another project, the MultiJava Compiler (MJC), a Java compiler that supports open classes and symmetric multiple dispatch [Clifton01].

Chalin recently demonstrated that use of Java's arithmetic in specifications causes problems (e.g. even inconsistencies in specifications) [Chalin03]. To correct this problem, Chalin proposed JMLb, an extension to JML supporting arbitrary precision arithmetic and various math modes—including one called "safe math" in which any arithmetic overflows are reported as exceptions. In this thesis we describe the implementation of JMLb's safe math.

Also, Chalin and Rioux recently conjectured that Java programmers often want references to objects to be non-null [ChalinRioux05]; however, the default values of these references are `null` in Java and JML. In order to verify this idea, a tool that gathers statistics of non-null usage in JML specifications was developed.

The work described in this thesis is done in the context of overall research in formal program verification.

1.1 Contributions

The main objectives of the research reported in this thesis have been to:

- Add bytecode level support for "safe-math arithmetic" (arithmetical computations that throw exceptions when overflows occur) in MJC.
- Enhance the JML checker so that it can gather statistics concerning the frequency of occurrence of non-null declarations. The purpose of this enhancement is to support a study which has as objective to prove that the majority of declarations of reference types are meant to be non-null (based on design intent).

More specifically, the work reported in this thesis involved:

- Studying and understanding the design and source code of both MJC and JML projects. More than six hundred files are involved.
- Designing and implementing the two new features in MJC and the JML checker in accordance with existing development conventions established by the JML developer community.
- Reviewing desugaring processes for different kinds of method specification in JML and creating a tool for non-null statistics calculation and analysis of non-null usage in JML specifications by simulating the desugaring processes.
- Developing test cases and integrating them into the existing JML project testing framework.

1.2 Thesis Organization

In Chapter 2, we present a survey of tools that extend Java with support for Design by Contract (DBC) and a brief introduction of JML. In Chapter 3, we discuss integral arithmetic in JML and illustrate how the current semantics of integral arithmetic causes inconsistency in JML specifications. JMLa, an extension to JML that addresses some of the issues related to integral arithmetic is described. In Chapter 4, JMLb, a variant of JML that supports primitive arbitrary-precision numeric types in Java is introduced. In Chapter 5, we discuss design alternatives to support safe arithmetic in MJC and JML. In Chapter 6, the implementation of safe arithmetic in MJC and JML is given. In Chapter 7, we evaluate the performance of MJC after implementing safe arithmetic in MJC by comparing it to other Java compilers. In Chapter 8, we illustrate rules for non-null statistics and their implementation in JML. Chapter 9 presents a summary and discusses possible future work.

2 Design by Contract for Java and Related Work

Design by contract (DBC) is a software development technique whose main idea is that a class and its clients have a contract with each other. The client must guarantee certain conditions that are specified by the method before calling it, and in return the method guarantees that certain properties will hold after the call. The use of such pre- and post-conditions to design software dates back to Hoare's 1969 paper on formal verification [Hoare69]. The novelty of DBC lies in its contract-executing capability. The contracts are specified with program code and are translated into executable code by a compiler. Hence, any violation of the contract that occurs during program execution can be detected immediately.

Figure 2-1 presents an example of code written in the Java Modeling Language in the DBC style. It defines an insertion method for a HashMap class. The `requires` clause specifies an input condition, or precondition; the `ensures` clause introduces an output condition, or postcondition. Both of these clauses are called *assertions*. In the precondition, `count` is the current number of objects in `HashMap`, and `capacity` is the maximum number of objects the `HashMap` can contain. In the postcondition, `exist(x)` is a boolean query which tells whether a certain object is present, and the method `item` returns the object associated with a certain key. The notation \old(count) refers to the value of `count` when the program entered the method.

```
Class HashMap{
  /*@ spec_public */ private int count;
  /*@ spec_public */ private int capacity = 100;

 // Insert x so that it will be retrievable through key.
/*@ public normal_behavior
  @ requires count < capacity
  @ requires key != null;
  @ ensures exist(x);
  @ ensures item(key) = x;
  @ ensures count = \old(count) +1;
  @*/
 Public InsertHashMap (x: Object; key: STRING) {
  ...
 }
}
```

Figure 2-1 a DBC example

Preconditions and postconditions apply to an individual method. Other kinds of assertions characterize a class as a whole. An assertion describing a property which holds of all instances of a class is called a *class invariant* [Meyer02]. For example, an invariant of class `HashMap` could be

```
 //@ invariant count <= capacity;
```

Another important issue in DBC is contract inheritance. A class `B` that inherits from a class `A` may have a new specification for an inherited specification *r* of *A*, which is called a *subcontract.* For example a subclass B of `HashMap` might override the method `InsertHashMap` and specify different preconditions or postconditions. The subcontract is bound by the inherited contract. The principle of subcontracting is that a subcontract may keep or weaken the precondition and it may keep or strengthen the postcondition.

As a popular programming language, Java is criticized for its lack of support for DBC by many people [SDN]. Assertion support was voted second on Sun's 'Request for Enhancements' list [Jass]. This hinders Java from being a successful language for developing reliable software. Although a recent version of the Java specification (v1.4) adopts the keyword `assert` to declare assertions in Java code, it still has no direct support for

DBC. Like the successful usage of assertions in Eiffel, many tools such as the JML compiler, Jass, iContract, and Jcontract have been developed to introduce DBC to Java. In the following subsections, we briefly review each of these approaches.

2.1 Jass

Jass (Java with assertions) is a precompiler that supports assertions in Java [Bartetzko01, Jass]. It precompiles a Jass file to a normal Java file (it should come as no surprise that the resulting Java file is much bigger than Jass file). The decorated Java code can be compiled with a Java compiler like any other Java code.

Jass supports the following assertions: preconditions, postconditions, class invariants, loop invariants, and a *check* expressions. A *check* expression is an expression that can state an assertion at any place in a method. All assertions are Boolean expressions except for the loop invariant, which is of type int. They may contain variables and method calls. It also supports the universal quantifier (`Forall`) and existential quantifier (`Exists`). Furthermore, the recursive definition of assertions (i.e., assertions that include a method call in which at least one assertion is defined) is not allowed in Jass. No expression or method calls that may cause side effects (e.g., assignment and instance creation) can be used in assertions.

Preconditions and postconditions are used to specify the state of the system that should be satisfied before and after the execution of a method. In postconditions, to implement an `old` expression, Jass needs to store a copy of the object at the beginning of a method and implement the method `clone()` without throwing any exception. Another expression is `change only` expression that defines a list of variables that may be changed after executing the method.

```
public class firstJass implements Cloneable  {          //1
  protected int count=0;
  public int usePre(int i){
    /** require i>0; i<Integer.MAX_VALUE; **/           //4
    count=i+1;
    /** check count >= 0; **/                           //6
    return count;
    /** ensure Old.count == i-1;**/                     //8
    /** rescue catch (PreconditionException e) {        //9
      if (e.label.equals("valid_parameter")) {
        i = Integer.MIN_VALUE ; retry;
      } else {
        throw new RuntimeException("precondition is violated");
      }
    }**/
  }
  protected Object clone() {                            //10
    /* Use the Objects clone method*/
    Object b = null;
    try {
    b = super.clone();
  }catch (CloneNotSupportedException e){;}
    return b;
  }
  /** invariant 0 <= count ; **/                        //19
}
```

Figure 2-2 a sample Jass program

Moreover, Jass extends the exception mechanism in Java by adopting rescue and retry statements. For example, Figure 2-2 line 9 catches a precondition exception and reinitiates the method call with new parameter.

In Jass a subclass can refine the specification of its superclass. To realize refinement, a programmer must implement `jass.runtime.Refinement` to inform the precompiler to add extra refinement checks in the final Java code. In addition, the method `jassGetSuperState()`, which maps states of a subclass to corresponding states in its superclass, must be implemented. In [Abercrombie02], the author says it is a disadvantage of Jass that private variables in a superclass cannot be copied to its subclass. However, due to the mapping mechanism between a superclass and its subclass, the problem is safely solved.

In Eiffel and JML, if the evaluation of an assertion invokes a method that also declares assertions, these secondary assertions are not checked. Jass partly implements this principal, namely, if an assertion contains a call to a method from the same class, this method will be copied without any assertion checks; however, if an assertion contains a call to a method from the another class, then assertions in the called methods will be checked.

2.2 iContract

Like Jass, iContract also precompiles an iContract file to a Java file decorated with iContract assertions [Lackner02, Enseling01]. Moreover, all iContract notations also appear within Java as comments. Since the iContract notations are modeled after a subset of the Object Constraints Language (OCL), they look different from Java notations in style.

iContract supports preconditions, postconditions, class invariants, the universal quantifier (`forall`) and existential quantifier (`exists`), and implication (`implies`). Like Jass, it supports an *old* expression (written `expr@pre)` by implementing the method `clone()`. But its *old* expression is just a shallow copy of the variable `expr`, (i.e., there are no copies of other objects but only copies of all fields and references in an object).

In iContract, class invariants, preconditions, and postconditions can refer only to non-private instance variables unless the class is *final.* A subclass inherits all assertions from its superclass but may add more restricted assertions. Also, method calls with no side effects can be used in assertions. Furthermore, to automatically avoid non-terminating recursion, iContract instruments checking code by keeping track of the call chain at run time [Enseling01].

```
public class firstiContract implements Cloneable{
  public int count=0;
  public int incrementCountBy(int i){
  /**
   *@pre i>0
   *@pre i<Integer.MAX_VALUE
   *@post @return =count@pre + i;              //7
   */
    count+=i;
    return count;
  }
}
```

Figure 2-3 sample iContract program

In Figure 2-3, we present a sample iContract program. The notation `@pre` indicates a precondition assertion. Notation `@post` indicates a postcondition assertion. At line 7, `@return` represents the method return value (like `\result` in JML); `count@pre` stands for the *old* value of the variable `count`.

2.3 jContractor

Unlike Jass and iContract, jContractor [Abercrombie02] is not a precompiler. To use it, users just use a modified command line to launch the JVM, and jContractor replaces the default class loader with a specialized version that adds the defined contracts to the class bytecodes as they are loaded.

In jContractor, contracts are written as standard Java methods following a naming convention. It supports preconditions, postconditions, class invariants, universal quantifier (`forall`) and existential quantifier (`exists`), set comprehension (`suchThat`), and implication (`implies`). In jContractor, the *old* expression is represented as: `OLD.value(variable)` where `OLD` is a class with a static method `value()`. To use it, a class must explicitly declare a private instance variable OLD of the same type as the class (its subclass has its own private OLD variable). Hence, users may reference the OLD variable to access fields and methods in the postcondition.

Since jContractor works at the Java bytecode level without the presence of source code, it cannot adopt the techniques that simply copy the code of old reference to the top of the method and save the result [Abercrombie02]. For example, the clause `int OLD.count = getCount()` will store the old value of count derived from method `getCount()` in variable `OLD.count`. jContractor will create a clone of the object before executing the method body and redirect all reference to OLD to the cloned object. Therefore, it also needs to implement the `clone()` method in the class.

In Figure 2-4, a sample jContractor program is given. At line 2 it defines a private field OLD with the same type as the class. The precondition (line 8), postcondition (line 11), and class invariant (line 14) are all represented in the method definition.

```
public class firstJContractor implements Cloneable {
  private firstJContractor OLD;                            //2
  public int count=0;
  public int usePre(int i){
    count=i+1;
    return count;
  }
  protected boolean usePre_Precondition (int i) {          //8
    return (i>0) && (i<Integer.MAX_VALUE);
  }
  protected boolean usePre_Postcondition (int i, Void RESULT) {//11
    return  OLD.count == i -1;
  }
  protected boolean _Invariant () {                        //14
    return count >= 0;
  }
...
}
```

Figure 2-4 sample program of jContractor

In order to avoid infinite recursion, jContractor, like iContract, keeps track of the contract checking's progress with a shared hashtable of threads. jContractor implements contract inheritance only for non-private, non-constructor methods. The check of an overriden method's contract also redirects to the contract of the method of its superclass. Because interface contracts are stored in a separate file, jContractor is responsible for inserting contract code into classes that implement the interface.

2.4 Java Modeling Language (JML)

The Java Modeling Language (JML) is a Behavioral Interface Specification Language (BISL) [Leavens02] that can be used to specify the behavior of Java modules, i.e. either a Java class or interface. JML is like a model-based specification language, like VDM (The Vienna Development Method) [Jones90] or Larch [Wing87], and has some elements of a refinement calculus [Leavens02]. JML specifications are written with a syntax based on Java. It supports Design by Contract, quantifiers, specification-only variables, and other enhancements.

JML supports a Hoare-logic-like method contract using preconditions and postconditions. The postcondition can be divided into a normal postcondition and an exceptional post-condition. The former is used to indicate behavior of the method when it terminates normally, while the later is adopted when it terminates by throwing an exception.

All JML specifications are in comments that start with the character "@". Figure 2-5 gives a sample JML program.

```
public class Purse{
 final int MAX_BALANCE
 int balance;
 //@ invariant 0 <= balance && balance <= MAX_BALANCE;  //3
 /*@ requires amount >= 0;                              //4
   @ assignable balance;                                //5
   @ ensures balance == \old(balance) - amount          //6
   @&& \result == balance;                              //7
   @ signals (PurseException) balance == \old(balance); //8
   @*/
 int debit(int amount) throws PurseException{
    ...
 }
}
```

Figure 2-5 a sample JML annotated class

An important feature of JML is that it supports abstract specification written in terms of specification-only declarations such as model fields, ghost fields and model methods. Figure 2-6 presents an example using model fields that is from the JML package. The class `List` declares a public model field `listValue`, which describes the abstract value of a `List` object.

```
/@ model import edu.iastate.cs.jml.models.*;

 public abstract class List {

 //@ public model non_null JMLObjectSequence listValue;
 protected /*% rep %*7 Node first, last;

 //@ protected depends listValue <- first, first.values, last;

 /*@ protected represents listValue <-
   @  (first == null ? new JMLObjectSequence() : first.values);
   @*/

 /*@ public normal_behavior
   @  requires o != null;
   @  modifies listValue;
   @  ensures listValue.equals(\old(listValue.insertBack(o)));
   @*/
 public void append(/*% readonly %*/ Object o) {
   if (last==null) {
     last = new /*% rep %*/ Node(null, null, o);
     first = last;
   } else {
     last.next = new /*% rep %*/ Node(null, last, o);
     last = last.next;
   }
 }
 /* ... */
}
```

Figure 2-6 A JML specification of the Java class List

JML is supported by tools such as the JML Checker, which can parse and type check Java programs in which JML annotations are integrated. Another important tool is known as the JML RAC (Runtime Assertion Checker) that is an extension of the JML checker; the RAC can compile a JML-annotated Java file into Java bytecode. The compiled bytecode includes runtime assertion checking instructions that check JML specifications.

2.5 Summary of capabilities of DBC languages and tools for Java

Finally, we give a brief comparison of the above-mentioned DBC tools support for basic DBC features.

	Jass	iContract	jContractor	JML
Pre- /, Post- conditions	yes	yes	yes	yes
Invariant	class loop	class	class	class
Qualifier	\forall,\exists	\forall,\exists	\forall,\exists,\suchthat	\forall, \exists, \sum, \min
Contract inheritance	non-private,	non-private,	on-private, non-constructor	non-private
OLD expression	yes	yes	yes	yes
Abstract specification	no	no	no	yes
Location	Java comment	Java comment	separate file	Java comment
Pre- compiler	yes	yes	no, it is library based	yes
Other features	rescue/retry , trace assertion, refinement	no	name convention, recovery and exception handling	rich set of assertion, operators, support refinement, behavior, exception handling, etc

Table 2-1 Comparison among Java DBC Tools

In the next chapter, we will introduce the integral arithmetic in Java, illustrate a problem in JML that is caused by adopting Java semantics of integral arithmetic in JML and provide a solution to the problem.

3 Integral Arithmetic in Java and JML

Integral arithmetic in JML is based on that of Java. This makes it easy for Java developers to write JML specifications; however, this also introduces problems since the Java language was not designed to be a specification language. In this chapter, we briefly illustrate the integral arithmetic used in Java and JML and present the problem of inconsistencies in JML specifications caused by this.

First, a brief introduction to Java's integral arithmetic is given. Next, an example is employed to indicate that the semantics of current JML numeric expressions fail to meet specifier expectations. Then, JMLa, predecessor of JMLb, is introduced. This chapter is based on earlier work by Chalin [Chalin03, Chalin04].

3.1 Integral Arithmetic in Java

Since JML inherits Java's integral arithmetic, we briefly present Java's integral arithmetic here, concentrating on type conversion and promotion.

3.1.1 Integral types

In Java, the integral types are byte, short, int and long (in Java, char is also an integral type. Since our later discussion focuses on the others, we ignore char type here). Table 3-1 presents range information for these types.

	Bits	Min Value	Max Value
byte	8	-2^7	2^7-1
short	16	-2^{15}	$2^{15}-1$
int	32	-2^{31}	$2^{31}-1$
long	64	-2^{63}	$2^{63}-1$

Table 3-1 Ranges for Integral Types in Java

3.1.2 Type conversions

Occasionally, the integral type of a Java expression is not appropriate to its surrounding context and consequently a conversion, either explicit or implicit, is needed. After a conversion from type τ_1 to type τ_2, type τ_1 can be treated like a type τ_2. In some cases a runtime check is executed to ensure the validity of the conversion or a runtime action is taken to finish the type translation. For example, conversion from type `int` to `long` requires run-time sign-extension of a 32-bit value to its 64-bit representation. Next, some type conversions and promotions that are related to Java integral arithmetic are given.

3.1.3 Widening Primitive Conversion

The following three conversions on primitive types are called the widening primitive conversions:

- `byte` to `short`, `int`, `long`
- `short` to `int`, `long`
- `int` to `long`

A widening conversion from one integral type to another does not lose any information, (i.e., the numeric value is preserved). A widening conversion from a signed integer value τ_1 to another integral type τ_2 simply extends the sign bit in two's-Complement Representation (TCR) of the integer value τ_1 to fill the higher order bits [JLS00, §4.2.2].

3.1.4 Narrowing Primitive Conversions

In Java, in terms of integral types, the narrowing primitive conversions include:

- `short` to `byte`
- `int` to `byte` or `short`
- `long` to `byte`, `short`, or `int`

Narrowing conversions may cause loss of precision due to loss of value magnitude. A narrowing conversion from a signed integer τ_1 to another integral type τ_2 simply keeps the *n* low-order bits, where *n* is the number of bits used to represent type τ_2, and discards other bits. Consequently, in some cases, loss of magnitude information of the numeric value may cause the sign of the resulting value to be different from the sign of the input value—e.g., `(byte) 129` evaluates to `-127`.

Despite the possible overflow, underflow, or loss of information, narrowing conversions among primitive types never result in a run-time exception. The test program given in Figure 3-1 demonstrates narrowing conversions in which loss of precision occurs:

```
class Test {
 public static void main(String[] args) {
  // A narrowing of int to short loses high bits:
  System.out.println("(short)0x67890==0x" +
  Integer.toHexString((short)0x67890));
  // A narrowing of int to byte causes changes of sign and magnitude:
  System.out.println("(byte)129==" + (byte)129);
  // A long value that is too big to fit gives largest int value:
  System.out.println("(int) 2147483648L ==" + (int) 2147483648L);
  }
}
```

Figure 3-1 sample program of Narrowing Primitive Conversions

This test program produces the following output:

```
(short)0x67890==0x7890
(byte)129==-127
(int) 2147483648L== -2147483648L
```

3.1.5 Assignment Conversion

Assignment conversion occurs when a variable is assigned with the value of an expression; the type of the expression must be converted to the type of the variable. Towards integral primitive types, widening primitive conversion and narrowing primitive conversion are allowed within an assignment context.

If the type of an expression can be converted to the type of a variable by assignment conversion, we say the type of the expression is assignment compatible with the type of the variable. Otherwise, a compile-time error occurs. For example:

```
short s = 200;
s = s + s;
```

The expression `s + s` is automatically promoted to type `int`, and an `int` cannot be implicitly converted to type `short`; therefore a compile-time error that possible loss of precision is generated.

If a variable like the above variable `s` has a type `int`, the program is executed without any warning, even in those cases in which an incorrect result is computed. For example, the code

```
int i = Integer.MAX_VALUE + 1;
System.out.print("Integer.MAX_VALUE + 1= " + i);
```

produces this output

```
Integer.MAX_VALUE + 1= -2147483648
```

3.1.6 Numeric Promotion

Numeric promotion is used to convert the operands of a numeric operator to a common type so that an operation can be performed. There are two kinds of numeric promotion: unary numeric promotion and binary numeric promotion [JLS00, §5.6]

Unary numeric promotion

If a unary expression has type `byte` or `short`, unary numeric promotion promotes it to type `int` by widening conversion. Unary numeric promotion is performed on an expression in the following situations:

- Each dimension expression in an array creation expression
- Each index expression in an array access expression
- An operand of the unary plus operator +
- An operand of the unary minus operator -
- An operand of the bitwise complement operator ~
- Each operand, separately, of a shift operator >>, >>>, or << (note that a `long` shift distance (right operand) does not promote the value being shifted (left operand) to `long` [JLS00].)

The following is a test program that provides examples of unary numeric promotion:

```
class Test {
  public static void main(String[] args) {
    byte bValue = 2;
    int a1[]=new int[bValue];//dimension expression promotion
    short s1 = 2;
    a1[s1-1] = 1;               // index expression promotion
    a1[0] = -s1;                         // unary - promotion
    System.out.println("array a1: " + a1[0] + "," + a1[1]);
  }
}
```

Figure 3-2 sample program for unary numeric promotion

This test program produces the following output:

```
array a1: -2,1
```

Binary numeric promotion

Binary numeric promotion is performed on binary expressions in the following situations:

- If either operand is of type `long`, the other is converted to `long`.
- Otherwise, both operands are converted to type `int`.

Binary numeric promotion is performed on the operands of certain operators:

- The multiplicative operators `*`, `/`, and `%`
- The addition and subtraction operators for numeric types + and -
- The numerical comparison operators <, <=, >, and >=
- The numerical equality operators == and `!=`
- The integer bitwise operators `&`, `^`, and `|`
- In certain cases, the conditional operator `?:` [JLS00, § 5.6.2].

Here is an example in which binary numeric promotions are applied

```
class Test {
  public static void main(String[] args) {
    int i1 = 1;
  /* expression having types char & byte is promoted to types int
& int: */
    byte b1 = 0x1f;
    short c1 = 0x45;
    int bc = c1 & b1;
    System.out.println(Integer.toHexString(bc));
  }
}
```

Figure 3-3 sample program for binary numeric promotions

The output is: 5.

3.2 Integral Arithmetic in JML

Integral arithmetic in JML has the same definition as Java. Specifically, the above-mentioned conversions and promotions are also applicable to the integral arithmetic in JML specifications.

3.2.1 A semantic gap

Figure 3-4 gives the JML specification of method `negAdd`. In this method, assuming that the value of both arguments `a` and `b` are `0`, if the specification of `negAdd` is interpreted using Java semantics, a valid implementation of method `negAdd` will permit to return `Integer.MIN_VALUE`.

```
public class Test {

  /*@ normal_behavior
    @ requires a <= 0 && b <=0;
    @ ensures \result <= a && \result <= b
    @ && \result * \result / 4 >= a * b;
    @*/
public int negAdd (int a, int b){
  ...
  }

  public static void main(String[] args) {
    ...
  }
}
```

Figure 3-4 specification of method negAdd

The reason is that Java integral types have fixed precision, and operations over these types obey the rules of modular arithmetic (modular arithmetic is the arithmetic of congruences, sometimes known informally as "clock arithmetic". In modular arithmetic, numbers "wrap around" upon reaching a given fixed quantity, which is known as the modulus [Modulus]). For example, the following equations hold in Java [Chalin03]:

```
Integer.MIN_VALUE == Integer.MAX_VALUE +1;

Math.abs(Integer.MIN_VALUE) == Integer.MIN_VALUE

(Integer.MIN_VALUE +1) * (Integer.MIN_VALUE +1) == 1

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

Based on the examples above, we can find that since specifiers generally ignore the finiteness of numeric primitive types and think in terms of arbitrary precision arithmetic, problems of invalid and inconsistent JML specifications often occur. Hence, a semantic gap exists between specifier expectations and the current semantics of JML numeric types [Chalin03]. Next, we will try to fix the specification of `negAdd` within the current semantics of JML; then a method to close the semantic gap is proposed.

3.2.2 Fix the semantic gap

The problem with the specification of `negAdd` occurs because `Integer.MIN_VALUE` is not a valid result when both arguments are zero. This can be easily corrected by ensuring arithmetic overflow does not occur within an `ensures` clause expression. To preserve the previous form of the `ensures` predicate as much as possible, a sample solution is given in Figure 3-5 (differences between the current and previous specifications are underlined), in which explicit type widening ensures that all operators will be with long type.

```
public class Test {

  /*@ normal_behavior
    @ requires a <= 0 && b <=0;
    @ ensures \result <= a && \result <= b
    @ && (long)\result * \result / 4 >= (long) a * b;
    @*/
public int negAdd (int a, int b){
  ...
  }
```

Figure 3-5 specification of negAdd with cast to long

Although explicit type casting solves the problem in this particular case, it would be ineffective if the argument type of `negAdd` were changed to long. The only available solution in current JML semantics is to employ the `JMLInfiniteInteger` model class for arbitrary precision representation. The new specification of method `negAdd` is given in Figure 3-6 in which we see how the problem is solved at the cost of clarity.

```
public class Test {

  /*@ normal_behavior
    @ requires a <= 0 && b <=0;
    @ ensures \result <= a && \result <= b
    @ && (new JMLInfiniteInteger(\result)).multiply(
    @ new JMLInfiniteInteger(\result)).divide(
    @ new JMLInfiniteInteger((long) 4))).compareTo(
    @ new JMLInfiniteInteger(a).multiply(
    @ new JMLInfiniteInteger(b))) >= 0;
    @*/
public int negAdd (long a, long b){
   ...
  }
```

Figure 3-6 Specification of negAdd using JMLInfiniteInteger

Based on the above discussion, we can draw the conclusion that under current JML semantics, a general and practical solution is not available. Hence, two variants of JML named JMLa and JMLb [Chalin03] that support arbitrary-precision arithmetic in JML are illustrated in the next sections.

3.3 JMLa: supporting primitive arbitrary precision numeric types

3.3.1 Closing the semantic gap

Figure 3-6 indicates that in addition to primitive fixed-precision numeric types, JML, like Larch, should also support primitive arbitrary-precision numeric types. Therefore, in the JML variant JMLa, two primitive numeric types `\bigint` and `\real` that represent arbitrary precision integers and floating point numbers, respectively, are introduced. In order to avoid causing name collision with existing Java code, these two types start with a backslash character.

Also, a model class `org.jmlspecs.lang.JMLMath` is defined in which methods in `java.lang.Math` are presented, but defined over `\bigint` and `\real` types.

```
public class Test {

  /*@ normal_behavior
    @ requires a <= 0 && b <=0;
    @ ensures \result <= a && \result <= b
    @ && (\bigint)\result * \result / 4 >= (\bigint) a * b;
    @*/
public int negAdd (long a, long b){
}   ...
}        ...
}
```

Figure 3-7 JML specification of negAdd with casts to \bigint

In Figure 3-7, a specification of method `negAdd` is given in JMLa in which both accuracy and clarity (only two new expressions are added, which are underlined) of specification are achieved.

After introducing new primitive arbitrary-precision types, we need to define the arithmetic rules among `\bigint`, `\real` and other primitive types. Here, the general rule is to add implicit promotion to `\bigint` for integral expression.

3.3.2 Informal semantics

Figure 3-8 presents the JMLa numeric type hierarchy in which both the `\bigint` and `\real` types are defined as top elements [Chalin03]. Type widening and narrowing are defined as natural extensions of the rules of Java. One design goal of JMLa is to ensure that numeric operations that can cause overflow are performed over `\bigint` by default. Since unsafe operators, including unary -, binary +, -, *, and /, can cause overflow, they will unconditionally promote their integral operands to `\bigint`. However, this rule is not totally applicable to constant expressions involving unsafe operators. Java semantics are preserved if the operands are constant expressions and operator evaluation does not result in overflow (note that in JML, a constant expression will be folded to a constant using the `constantFolding` method during the `typechecking` stage of compilation).

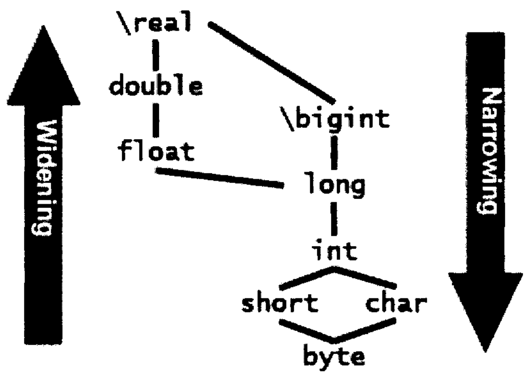


Figure 3-8 JMLa primitive numeric type hierarchy [Chalin03]

For example, in JMLa, for an `int` type variable `i`, the expression `-i` will be interpreted as `- (\bigint) i`; expression `-5` will be interpreted as `-5`; the expression `- Integer.MIN_VALUE` will be interpreted as `- (\bigint) Integer.MIN_VALUE` since the constant expression value is not in the range of `int`.

In the next chapter, we introduce a variant of JML called JMLb that employs the notions of integral arithmetic modes to indicate the context in which a numeric expression should be interpreted.

4 Safe Arithmetic in Java and JML: JMLb

Although JMLa solves the inconsistency in JML specifications caused by fixed-precision arithmetic in Java with either no syntactic changes or minor syntactic changes to the specifications [Chalin03]. However, JMLa breaks one basic design goal of JML: expressions that are valid in Java should remain valid in JML and with the same meaning. Therefore, a new variant of JML, JMLb, is introduced in which the notions of integral arithmetic modes (we refer to these as *math modes* for short) are employed to indicate the context math mode in which a numeric expression should be interpreted. In addition, the default math mode in JMLb is Java mode; this ensures that expressions in JML specification have the same semantics as those in Java by default.

4.1 Math modes

In JMLb, there are three integral arithmetic modes, namely

- *Java math*, which corresponds to Java semantics.
- *bigint math* corresponds to the implicit promotion to `\bigint` semantics of JMLa.
- *safe math* is like Java math except that arithmetic overflow is detected and an exception is thrown when it occurs (like checked mode in language C#).

A math mode can be applied to a class, a method or an expression. The modifiers `spec_java_math`, `spec_bigint_math` and `spec_safe_math` can be applied to the declaration of a class. As a result, all specification expressions within the class will have a default math mode of Java, bigint, and safe, respectively. These modifiers can also be applied to a method definition to set the math mode within the method scope. To implement finer control, JMLb provides operators \java_math(E), \bigint_math(E), and \safe_math(E) to restrict the math mode during the evaluation of an expression E.

A sample JMLb specification that adopts several math mode modifiers is given in Figure 4-1. At line 1, the spec_bigint_math modifier indicates that all specification expressions in the class are to be interpreted under bigint math mode by default. The first method specification is from the method negAdd given in Figure 3-4. Notice that under JMLb, the specification of method negAdd is consistent since the expressions are interpreted over \bigint rather than int. The second specification of method decrement_and_wrap, we use the \java_math modifier to specify that the expression i-1 should be interpreted under Java mode. That is to say, i-1 will be equal to Integer.MAX_VALUE if variable i is equal to Integer.MIN_VALUE. The specification of the model method cal illustrates the usage of \bigint in a model method.

If there is no explicit math mode is set, then Java math mode will be applied. However, as mentioned before, JML specifiers generally think in terms of arbitrary-precision arithmetic, (i.e., in a bigint math mode); therefore JML tools should produce a warning message if a math mode is not explicitly stated.

In JMLb, with modifiers code_java_math, code_bigint_math, and code_safe_math, math modes of Java, bigint, and safe can be respectively applied to Java code. For example, in Figure 4-1 the method useMathMode will use Java math mode. However, this only has effect when using the MJC or JML RAC compilers [Chalin04].

```
public /*@spec_bigint_math@*/ class Test {
        //1
  /*@ normal_behavior
    @ requires a <= 0 && b <=0;
    @ ensures \result <= a && \result <= b
    @ && \result * \result / 4 >= a * b;
    @*/
  public int negAdd (int a, int b){
     ...
  }

  /*@ public normal_behavior
    @ assignable \nothing;
    @ensures \result == \java_math(i-1);
    @*/
  public static int decrement_and_wrap (int i) {
    return i-1;
  }

  /*@ public normal_behavior
    @requires i >0;
    @assignable \nothing;
    @ensures \result == i*j+1;
    @*/
  /*@ public pure model static \bigint cal (\bigint i,\bigint j)    {
    @return i*j+1;
    @*/

  /*@
    @public normal_behavior
    @requires i < Integer.MAX_VALUE;
    @ensures \result = i +1;
    @*/

   public /*@ code_java_math @*/ /*@ spec_java_math @*/
   int UseMathMode (int i ){
    return i+1;
  }

  public static void main(String[] args){
    Test t = new Test();
    short uo =t.unaryOppose (Short.MIN_VALUE);
    int daw =t.decrement_and_wrap (Integer.MIN_VALUE);
  }
}
```

Figure 4-1 Sample JMLb specification

4.2 JMLb semantics

In this section we present the definition of JMLb semantics. Similar to what is done by the LOOP tool [vdBJ01], an example demonstrates how a JMLb program is translated into a corresponding PVS theory by applying JMLb elaboration rules.

4.2.1 The LOOP Tool and PVS

LOOP is a tool that can verify JML assertion. It translates code that is annotated with JML specifications into proof obligations that can be proved with the PVS theorem prover by defining a formal denotational semantics of both Java and JML [Berg01]. PVS, short for Prototype Verification System, consists of a specification language: PVS, several predefined theories, and a theorem prover [PVS]. In this section, we follow the approach of LOOP tool, namely, formalizing JMLb semantics and then embedding them in PVS. Here, we only focus on the difference between JMLb and JML, (i.e. on the semantics of arithmetic expressions under various math modes). We also ignore some important issues, such as abnormal termination in expressions, which are already effectively handled in LOOP.

4.2.2 Abstract syntax ands semantic objects

The semantics of JMLb expressions is defined by means of an "inference system" in a style referred to as natural semantics [Winskel93]. The inference rules allow us to establish the validity of elaboration predicates in the form (this definition is taken from [Chalin04]):

$\rho \quad H \quad \alpha \xrightarrow{A} x$

where A is generally stands for an abstract syntax phrase class. This predicate asserts that the syntactic element α will map (also called elaborate in [Chalin04]) to the semantic object x under the context ρ; α is a JMLb expression; and x is the counterpart expression of α in PVS with its type.

$e \in$ EXPR :: = c_{τ} | ι | op($e_1, \ldots, e_k$) | (τ) e | \old(e) | e_{ι} | e ι ($e_1, \ldots, e_k$) | (q τ ι ; e) |
\java_math(e) | \bigint_math(e) | \safe_math(e) | …
$\tau \in$ TYPENM :: = ι | …

$op \in OPNM$
$q \in QUANTNM ::= \backslash forall \mid \backslash exists$

Figure 4-2 Abstract syntax of JMLb expressions [Chalin04]

In Figure 4-2, the abstract syntax for JMLb expressions is given. The expression could be:

- An integral literal constant of type $\tau \in \{int, long\}$.
- An identifier representing a logical variable (`\result`, method parameters and quantifier variables). Note that we assume that all classes and instance members are expressed in the form e_ι so that they can be distinguished from the occurrence of other logical variables.
- An operator, it could be unary operator or binary one. it includes those of Java (e.g., +, -, *) and JML (e.g., ==>, <==>).
- A type cast expression.
- A pre-state expression.
- A field access expression.
- A method invocation expression.
- A quantified expression.
- Math mode expressions.

JMLb expressions are translated into corresponding PVS expressions, whose annotated abstract syntax is

$\varepsilon \in PVSEXPR ::= c : \tau \mid op\ (\varepsilon_1, \ldots, \varepsilon_k) : \tau \mid q\ (\iota : \tau) : \varepsilon$

Each PVS expression is annotated with its type, which ensures that overloaded operators have the same meaning as in JMLb. We define constants, operators (including logical connectives and equality), and quantifier (either *FORALL* or *EXISTS*) in PVSEXPR. Elaboration of expressions is done within the context of an environment: $\rho \in$ ENV that can be thought of as a mapping from the identifiers into their attributes. However, some JMLb identifiers have special meanings:

- `\result` is a JML logical variable that denotes the value returned by a method;
- `\mathMode` denotes the "currently active" math mode;
- `\state` denotes the evaluation state context and can be bound to either pre or post state. By default, `requires` clause expressions are evaluated in pre state and `ensures` clause expressions in post state.
- $\rho \oplus \{ \iota :\text{->} \alpha\}$ denotes the update of mapping from ι to α in context ρ. Note that in the initial JMLb context ρ_0, `\mathMode` is set to `Java`.

4.2.3 Primitive numeric types in PVS

Next, we discuss the relationship between JMLb primitive numeric types and their corresponding types in PVS. The JMLb arbitrary precision types `\bigint` and `\real` are mapped to the standard PVS types: integer and real. For convenience, a type: bigint, which actually is identical to type integer is also defined. In addition, for each of the bounded-precision integral types (i.e., `byte`, `short`, `int`, and `long`), we created simple theories, all of them in the same form. In Figure 4-3 [Chalin04], we present an excerpt of the theory for `int`. At line 3 and 4, existing theories from standard library files are imported. For example, `number_theory@mod_nt` indicates theories in file pvs/lib/number_theory/ mod_nt.pvs are imported; at line 10, int type is defined as the subtype of built-in type integer that contains values in the range of min to max inclusive. At line 12, the function narrow implements the narrowing conversion from integer to type int. All arithmetic operations of type int are defined based on the function narrow. Finally, at line 30, a lemma div_rem is defined with respect to two int variables i , j: if j is not zero, then the property i = i/j *j + rem(i , j) should hold.

All arithmetic operators are defined using their integer counterparts followed by an application of narrow. Thus, the addition of two values of type int is defined as the addition of their values interpreted as integers followed by a narrowing of the result to int: (i.e,. add(i,j) = narrow((i:int + j:int):integer):int).

```
int: THEORY
BEGIN
  IMPORTING number_theory@mod_nt, div@div,                        %3
         div@div_alt, div@rem                                     %4
...
  twoPn: posint = 4294967296                                      % ==2^32
  max : nat = 2147483647                           % ==Integer.MAX_VALUE
  min : negint = -2147483648                       % ==Integer.MIN_VALUE

  int: TYPE+ = {i: integer | min <= i AND i <= max} CONTAINING 0;  %10

  % narrowing primitive conversion to int
  narrow(i: integer): int =                                       %12
    LET b:nat = mod(i, twoPn) IN
    IF b <= max THEN b ELSE b - twoPn ENDIF
%------------------------------------------------
  neg(i:int): int = narrow(-i)                                    %16
  add(i,j:int): int = narrow(i + j)
  sub(i,j:int): int = narrow(i - j)
  mul(i,j:int): int = narrow(i * j)
%------------------------------------------------
%Division rounds towards zero (JLS 2.0, Section 15.17.2):
%
  div(i:int, j:{j:int|j /= 0}): int = narrow(div.div(i,j))

%Remainder satisfies (i/j)*j + (i%j) == i for all
%values except j=0, including  i= max and j=-1
%(JLS2.0, section 15.17.3).
%
  rem (i:int, j:{j:int | j/=0}): int = narrow (rem.rem( i,j))
  div_rem: LEMMA FORALL (i ,j:int):
    j /= 0 => div(i , j )* j + rem (i , j) =i
%------------------------------------------------
  bit_neg(i:int): int = -i - 1

END int
```

Figure 4-3 PVS theory for int

The definitions of elaboration rules for transforming JMLb expressions into corresponding PVS expressions are given in [Chalin04].

4.2.4 An Application Example of Elaboration Rules

As an example of the application of the elaboration rules, Figure 4-4 presents a resulting PVS translation of the specification of the method `inc`. The lemma `inc_consistent` is used to check the consistency of the specification and has been successfully proved.

```
/*@spec_bigint_math@*/public class Test{
  /*@ normal_behavior
    @ requires t <0;
    @ ensures \result == t+1 ;
    @*/
  public short inc (short t){
    return t+1;
  }
}

and the corresponding PVS translation is

IMPORTING short, ...
inc_requires(y: short): bool = y < 0;
inc_ensures(y, result: short): bool = result=y+1;
inc_consistent : LEMMA FORALL (y: short):
  EXISTS (result: short): inc_requires(y) => inc_ensures(y,result)
```

Figure 4-4 PVS definition of inc

In the next chapter, our work will focus on adding support to safe arithmetic in MJC and JML; and we will discuss two approaches: post-processing and redesign MJC to implement this.

5 Supporting Safe Arithmetic in MJC and JML

In this chapter, we will discuss methods of implementing support for the safe arithmetic mode (short for safe arithmetic) of JMLb. Since JML is established on the basis of the MultiJava Compiler (MJC) [MultiJava04], most of work is occurred in MJC.. The implementation is achieved at two levels: at the compiler (i.e. source) level and at the bytecode level (i.e., in Java class files). Here we mainly present the implementation at the bytecode level in MJC.

5.1 Relationship between MJC and JML

Firstly, we will briefly explain the relationship between MJC and JML. MJC extends Java by adding features of open classes and symmetric multiple dispatch [Clifton01]. The JML checker is built upon MJC; that is, it extends most of the MJC classes and also adds support for JML specification processing. As a result, if we add support for safe arithmetic in MJC, JML will automatically "inherit" the support.

5.2 Implementing support to safe arithmetic at the bytecode level

Next, we propose two approaches to support safe arithmetic at the bytecode level in MJC. One is called post processing (i.e., adding support to safe arithmetic by modifying class files generated from MJC). The other is to redesign MJC.

5.2.1 Post-Processing Method

A Post-Processing method is implemented using a Java class file disassembler and assembler.

```
public class TestDJava{
  static int i=1;
  public int getSum(int k, int j){
    long l = k+j ;
    if ( l < Integer.MAX_VALUE)
    return k+j;
    else
    return Integer.MAX_VALUE;
  }
  public static void main(String[] args) {
    TestDJava tdj = new TestDJava();
    int mi = tdj.getSum(i, 1);
  }
}
```

Figure 5-1 a sample Java file

There are many Java class file disassemblers, such as D-Java [DJava], Javad [JavaD], Neuron [Neuron]. Javap, a tool included with the Java SDK, is also a class file disassembler. Figure 5-1 shows a sample Java file. The corresponding JVM instructions disassembled from its class file are given in Figure 5-2 and Figure 5-3.

After using the command: `D-Java -o jasmin TestDJava.class`, the corresponding D-Java output is a given in Figure 5-2 and Figure 5-3 . There are corresponding JVM codes for each method. It also indicates the maximum usage of local variables and stack; therefore, JVM can reserve correct space for the program execution. The disassembled code also set label name for the conditional statements like `ifge`.

```
;
; Output created by D-Java (mailto:umsilve1@cc.umanitoba.ca)
;

;Classfile version:
;    Major: 46
;    Minor: 0

.source TestDJava.java
.class  public synchronized TestDJava
.super  java/lang/Object


.field static i I

; >> METHOD 1 <<
.method public <init>()V
   .limit stack 1
   .limit locals 1
.line 1
   aload_0
   invokenonvirtual java/lang/Object/<init>()V
   return
.end method

; >> METHOD 2 <<
.method public getSum(II)I
   .limit stack 4
   .limit locals 5
.line 4
   iload_1
   iload_2
   iadd
   i2l
   lstore_3
.line 5
   lload_3
   ldc2_w 2147483647
   lcmp
   ifge Label1
.line 6
   iload_1
   iload_2
   iadd
   ireturn
.line 8
Label1:
   ldc 2147483647
   ireturn
.end method
```

Figure 5-2 sample output of D-Java disassembler

```
; >> METHOD 3 <<
.method public static main([Ljava/lang/String;)V
  .limit stack 3
  .limit locals 3
.line 11
  new TestDJava
  dup
  invokenonvirtual TestDJava/<init>()V
  astore_1
.line 12
  aload_1
  getstatic TestDJava/i I
  iconst_1
  invokevirtual TestDJava/getSum(II)I
  istore_2
.line 13
  return
.end method

; >> METHOD 4 <<
.method static <clinit>()V
  .limit stack 1
  .limit locals 0
.line 2
  iconst_1
  putstatic TestDJava/i I
  return
.end method
```

Figure 5-3 sample output of D-Java disassembler (cont.)

5.2.1.1 Java Class File Assembler

There are some assemblers for the JVM, such as Jasmin [Jasmin], Jamaica [Jamaica], and the Java Bytecode Assembler [JBA]. We chose D-Java and Jasmin because both support ASCII descriptions of Java classes that are written in a simple assembler-like syntax using the JVM instruction set. They can convert these descriptions to a binary Java class file, which can then be loaded by a Java runtime system. In addition, D-Java provides a specific option to customize output in Jasmin format.

5.2.1.2 Implementing Safe Arithmetic by Post-Processing

Post-processing a Java class file involves two phases. Firstly, a Java class is disassembled into a set of JVM instructions. Next, some modifications made to the instructions, and they are assembled into a new class file. For example, to enable the program in Figure

5-1 to throw a `RuntimeException` when the sum of two arguments is greater than `Integer.MAX_VALUE`, we replace previous JVM instructions with new instructions to throw a `RuntimException`. Figure 5-4 presents the new D-Java code for method `getSum`. The new code is underlined.

```
; >> METHOD 2 <<
.method public getSum(II)I
  .limit stack 4
  .limit locals 5
.line 4
  iload_1
  iload_2
  iadd
  i2l
  lstore_3
.line 5
  lload_3
  ldc2_w 2147483647
  lcmp
  ifge Label1
.line 6
  iload_1
  iload_2
  iadd
  ireturn
.line 8
Label1:
  new java/lang/RuntimeException
  dup
  ldc "sum is overflow"
  invokenonvirtual
java/lang/RuntimeException/<init>(Ljava/lang/String;)V
  athrow
.end method
```

Figure 5-4 example of post-processing

To implement safe arithmetic in MJC and JML using post-processing, the following steps are taken:

- Find the JVM instructions corresponding to unsafe integral arithmetic operations and replace these instructions with JVM instructions that perform safe integral arithmetic
- Since changing bytecodes introduces new parameters, stack limit and local-variable limit should be recalculated.

- Create new labels as necessary

5.2.1.3 Advantages and disadvantages

A post-processing method has the following advantages:

- No negative effect on current MJC and JML projects since there is no modification to their source code.
- No specific knowledge of MJC or JML is needed. Because new added codes are isolated from current projects, there is no need for the developer to be familiar with MJC or JML.
- It can be applied to .class files produced with any Java compiler.

However, this method finally proved to be unsatisfactory since MJC conducts constant folding during compilation. As an example using addition, if both operands are constants, MJC will calculate the result of the addition expression at *compile time*. As a result, some unsafe operations cannot be determined. For example, the assignment expression `int i = Integer.MAX_VALUE + 1` should throw an overflow exception under safe-math mode. However, if constant folding occurs, the assignment expression will be treated as `int i = Integer.MIN_VALUE` after narrowing primitive conversion is performed on `Integer.MAX_VALUE + 1`. The unsafe addition operation can therefore not be found in a D-Java file.

5.2.2 Redesign the MJC compiler

Another approach to implementing safe arithmetic in MJC is to adapt the compiler so that it can generate different bytecode. The work of redesigning the MJC compiler includes the following:

- An attribute is added to the abstract syntax tree (AST) classes that represent unsafe math operators. This attribute specifies in which mode the operator is to be interpreted.
- Integral arithmetic in Java is analyzed and the condition at which an overflow will occur towards unsafe operators is summarized.
- Provision of functions like check/uncheck in C# to implement finer control of safe arithmetic on expressions are created
- Method genCode() of each unsafe operator is enhanced. If an operator is interpreted in safe mode, then safe math bytecode will be generated.

To support safe math in MJC, in other words, we need to explore cases on which incorrect integral arithmetic operations take place in Java math mode and ensure these cases should throw an exception in safe math mode. Next, we present how the safe arithmetic is achieved in MJC by providing a new class that implements safe integral arithmetic.

5.2.2.1 Incorrect integral arithmetic operation cases

Based on above description, what we need to do is to find the cases in which the integral arithmetic operations are incorrect and replace them with code that throws an exception instead of giving an incorrect result. In the following, four kinds of basic integral arithmetic operations are discussed.

In the case of a addition expression, a + b, an exception should be thrown in the following cases: (a or b is integral type operator; R(E) stands for the result given by current Java operations for evaluating the expression E, max stands for the maximum value of the expression type, and min stands for its minimum value) [Winkler02]:

- if $a > 0 \wedge b > 0 \wedge max < a + b \leq 2 * max$,

  then $R(a + b) < 0 \wedge R(a + b) = (a + b) - (2 * max + 2)$
- if $a < 0 \wedge b < 0 \wedge 2 * min \leq a + b < min$,

  then $R(a + b) > 0 \wedge R(a + b) = (a + b) - 2 * min$

These two cases can also be represented as

$$(a > 0 \wedge b > 0 \wedge R(a + b) < 0) \vee (a < 0 \wedge b < 0 \wedge R(a+b) > 0)$$

which indicates that if $a > 0$, $b > 0$, and $a + b$ (as computed in Java math mode) is both greater than max and less than $2 * max$ then the actual result of $a + b$ will be both less than zero and equal to $(a + b) - (2 * max + 2)$. Similarly, if $a < 0$, $b < 0$ and $a + b$ is both greater than $2 * min$ and less than min then the actual result of $a + b$ will be both greater than zero and equal to $(a + b) - 2 * min$. In both of these cases, an overflow will occur, and they require that an exception be thrown.

In the case of a subtraction expression, $a - b$, an exception should be thrown in the following cases:

- if $a \geq 0 \wedge b < 0 \wedge max < a - b \leq max - min$,

  then $R(a - b) < 0 \wedge R(a - b) = a - b + 2 * min$
- if $a < 0 \wedge b > 0 \wedge min - max \leq a - b < min$,

  then $R(a - b) > 0 \wedge R(a - b) = a - b - 2 * min$.

These two cases can also be represented as:

$$(a \geq 0 \wedge b < 0 \wedge R(a - b) < 0) \vee (a < 0 \wedge b > 0 \wedge R(a - b) > 0)$$

In the case of negation, expression `- a`, an exception should be thrown in the case that `a == min`.

In the case of Division, expression `a / b`, an exception should be thrown in the case

- if `b = 0` then `R(a / b) = Infinity`
- if `(a == min ∧ b == -1)`, then `R(a / b) = min`

These two cases can also be represented as:

```
b == 0 ∨ (a == min ∧ b == -1)
```

Evaluation of the expression `a * b` should cause an exception to be thrown in the following case (`Round(E)` returns the closest value to `E` that can be represented by the integral type):

```
(a > 0 ∧ (b < Round(min / a) ∨ b > Round(max / a ))
∨ (a == -1 ∧ b == min )
∨ (a < -1 ∧ (b < Round(max / a) ∨ b > Round(min / a)))
```

which indicates that an exception should be thrown in three cases:

- when `a` is greater than zero and `b` is either less than `Round(min / a)` or greater than `Round(max / a)`, the actual result will be less than `min` or greater than `max`,
- when `a` is equal to -1 and `b` is equal to `min`, the actual result is still `min`, or
- when `a` is less than -1 and `b` is less than `Round(max / a)` or greater than `Round(min / a)`, the actual result will be out of range; therefore, an exception should be thrown.

5.2.2.2 Redesign the MultiJava Compiler (MJC)

Based on the previous discussion, there are two alternatives to the redesign of MJC:

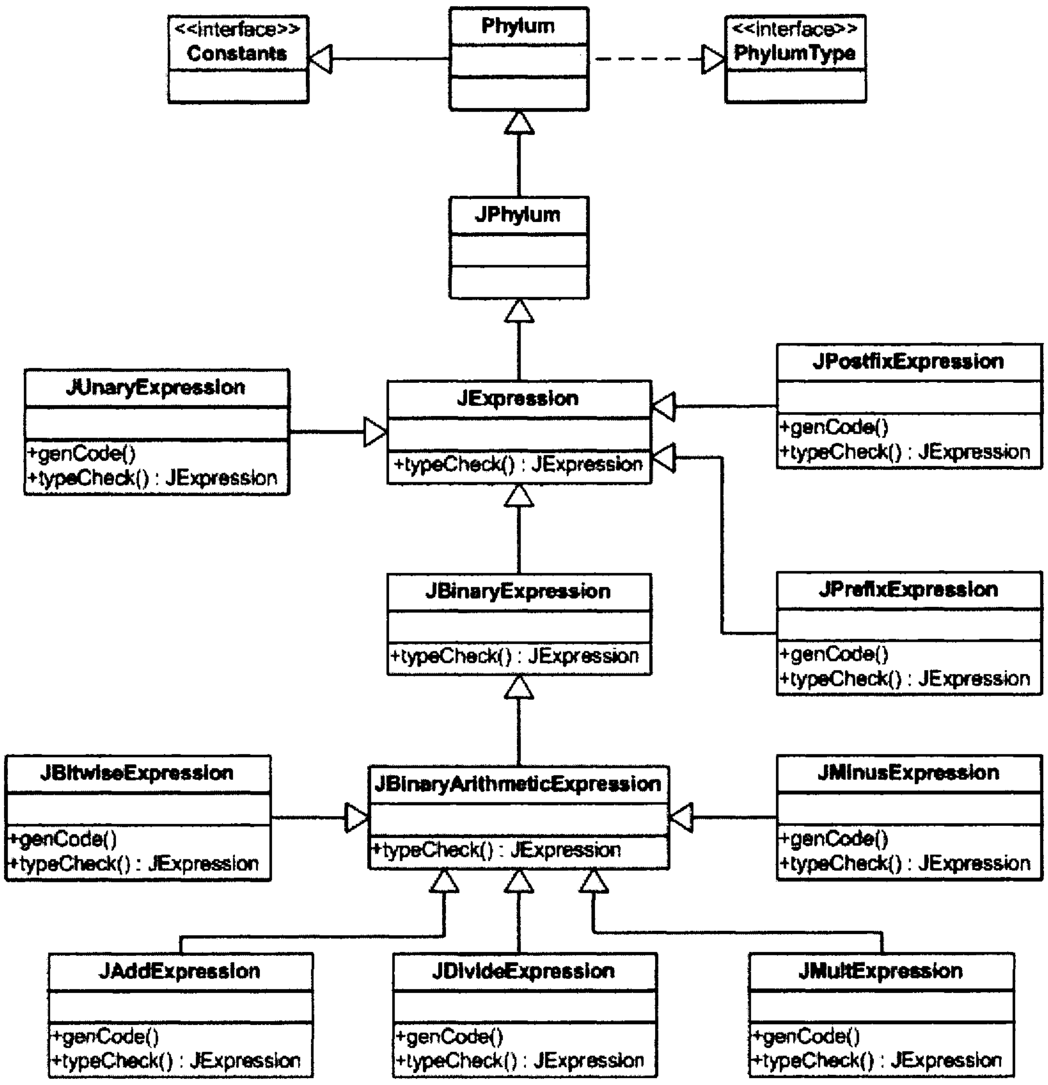
- In the first method, when each unsafe operation is executed in safe arithmetic mode, the above mentioned rules should be applied while generating bytecodes. Thus, we need to obtain JVM instructions representing the applied rules. Also, modification of the stack limit and local variable limit needs to be considered since that new bytecode probably increases the stack limit.
- The second method is to implement all of the above-mentioned rules in a separate class. When an operation is executed in safe arithmetic mode, bytecode calling the corresponding method in this class are generated. It is unnecessary to consider factors such as changes in stack limit or local variable limit here since replacing an arithmetic operation with a method call causes no change to the stack usage or the number of local variables. For example, both a JVM addition instruction (`iadd`) and the JVM instruction to invoke a method implementing safe arithmetic addition (invokestatic #13) have stack limit 2 and local limit 2.

We chose to implement the second approach since it exerts less effect on current projects than the first solution, and there is no extra workload for stack or local variable limit calculation. Thus, a new file `safeIntegralArithmetic.java` was created, in which methods for safe integral arithmetic operations are defined. For each kind of operation, two methods with different argument types are created (i.e., one for `int` operands and one for `longs`). Furthermore, methods named checked() and unchecked()—equivalent to C#'s `checked` and `unchecked` modes—are provided in the file. For example,

```
public int inc(int i) {
  int k;
  try {
      k = checked (i+1) ;                                    //4
  } catch(ArithmeticException e) {
      e.printStackTrace();
  }
  return k;
}
```

at line 4, we use `checked()` to explicitly enable overflow-checking on expression i+1.

Figure 5-5 demonstrates the class diagram of the Expression hierarchy in MJC [MultiJava04]. In MJC, the class `Phylum` represents a node in the AST; class `JPhylum` means a MJC node in the AST; class `JExpression` is the parent node of all kinds of expression in MJC; representations of all binary-operantor expressions are made with `JBinaryExpression` or its subclasses. Two methods of interest are `genCode`, which generates JVM bytecodes for the corresponding expression, and `typecheck`, in which the implementation of safe math at static checking level is achieved. Most of the work of implementing the redesign of MJC is contained in the method `genCode`; we provide details of the implementation in the next chapter.



<<interface>>
Constants
Phylum
<<interface>>
PhylumType
JPhylum
JUnaryExpression
+genCode()
+typeCheck() : JExpression
JExpression
+typeCheck() : JExpression
JPostfixExpression
+genCode()
+typeCheck() : JExpression
JBinaryExpression
+typeCheck() : JExpression
JPrefixExpression
+genCode()
+typeCheck() : JExpression
JBitwiseExpression
+genCode()
+typeCheck() : JExpression
JBinaryArithmeticExpression
+typeCheck() : JExpression
JMinusExpression
+genCode()
+typeCheck() : JExpression
JAddExpression
+genCode()
+typeCheck() : JExpression
JDivideExpression
+genCode()
+typeCheck() : JExpression
JMultExpression
+genCode()
+typeCheck() : JExpression

Figure 5-5 class diagram for safe arithmetic implementation

5.3 Implement support for safe arithmetic at the source code level

Before implementing safe math at the bytecode level in MJC, DSRG member Frederic Rioux had finished the work of supporting safe arithmetic in MJC at the source level. The following is a summary of his main work in this area (as it related to the work discussed in this thesis):

- Added support for the command-line option `-safemath` (abbreviated `-s`).
- Modified `java.JAddExpression`'s `constantFolding()`: if the operand type is an integral type (i.e., either `int` or `long`) and the arithmetic context is not `java_math` then do `safe_math` computations.
- These same modifications were made to
  - `JDivideExpression`,
  - `JMinusExpression`, and
  - `JMultExpression`.

In the following chapter, we will present the implementation of safe arithmetic in MJC and JML.

6 Implementation of Safe Arithmetic in MJC and JML

In this chapter, we describe the implementation of safe arithmetic in the MJC and JML tools.

6.1 Multijava Compiler

In MJC, a new file `safeIntegralArithmetic.java` is created in which methods for safe integral arithmetic operations are defined for all unsafe operators: binary +, -, * and /, and unary -. For each operator, methods supporting operands of each of the two integral types (i.e., `int` and `long`) are created. For example, the methods corresponding to the addition operator are defined as:

```
public static final int add(int arg1, int arg2) {
  long result = (long) arg1 + (long) arg2;
  if (result < Integer.MIN_VALUE || result > Integer.MAX_VALUE)
    throw newArithException("int addition", arg1, arg2);
  return (int) result;
  }
public static final long add(long arg1, long arg2) {
  long result = arg1 + arg2;
  if ((arg1 > 0 && arg2 > 0 && result < 0)
      || (arg1 < 0 && arg2 < 0 && result >= 0))
    throw newArithException("long addition", arg1, arg2);
  return result;
}
```

Figure 6-1 add methods in `safeIntegralArithmetic.java`

To enforce safe arithmetic in MJC, bytecode that implement integral arithmetic operations are replaced with bytecode instructions that invoke the corresponding methods in the class `safeIntegralArithmetic`. For example, the following JVM instructions are employed to implement a Java addition operation in a Java class:

```
0:iload_1
1:bipush  7
3:iadd
```

In `safe math` mode, this code is replaced with

```
0:iload_1
1:bipush  7
3:invokestatic #13; //Method org/multijava/mjc/SafeIntegralArithmetic.add:(II)I
```

6.2 JML Checker and Runtime Assertion Checker

Support for JMLb was added to the

- JML checker by Patrice Chalin and Frederic Rioux. This support consisted of the necessary updates to the grammar of JML as well as added type checking.
- JML Runtime Assertion Checker (RAC) by Kui Dai. Dai added run-time checking support for bigint math only (and not for safe math).

6.3 Test Cases

The implementation of support for safe math in MJC at the bytecode level is tested using JUnit, a unit-testing tool.

6.3.1 JUnit

JUnit is a framework for writing unit-tests in Java. It was developed by Erich Gamma and Kent Beck [JUnit]. It provides a simple way to test individual methods in a program. With JUnit, writing a test case is as simple as writing a method. In test cases, a programmer just sets the expected results and execution route, and Junit sets the run-time context and runs the test cases automatically. If the results match those expected then nothing happens; otherwise, JUnit reports an error.

In JUnit, test cases can be integrated into test suites, and test suites can be organized hierarchically. JUnit can be used to run regression tests for either the entire system or for specific subsystems. This is useful in project development. We can write test cases for all core components of the project, and when a modification is made, we can run the test cases and see results immediately: if no errors are shown, the modification (is likely to have) had no negative effect on the existing system.

6.3.2 Test cases

The MJC and JML projects are covered by a large number of JUnit test cases. After a modification is made to these projects, existing test cases are run again to ensure the new modification does not break any existing functionality. In our implementation, we provide abundant test cases to ensure our new functionality works properly and does not cause any negative effect on existing projects.

6.3.2.1 Test case: Java Math

In Figure 6-2, we present a sample JUnit test class illustrating how unsafe operations can be tested with JUnit.

```
//Test case for java  math
package org.multijava.mjc.testcase.runtime;

import junit.framework.TestCase;

public class TestMath extends TestCase {

  public void testAddByte() {
    byte i = Byte.MAX_VALUE;
    assertEquals((byte) (i+1), Byte.MIN_VALUE);          //10
  }

  // ... test for short, ...

  public void testAddInt() {
    int i = Integer.MAX_VALUE;
    assertEquals(i + 1, Integer.MIN_VALUE);              //17
  }


  ...// test cases for Long,
  ...// test cases for other integral operations

  public static void main(String[] args) {
    junit.textui.TestRunner.run(TestMath.class);
  }
}
```

Figure 6-2 sample JUnit test class

The program provides test cases for integral addition operations in Java mode for the types `byte` and `int`. Since the statement `assertEquals(i+1,Integer.MIN_VALUE)` is interpreted in Java mode, the expression `Integer.MAX_VALUE+1` should be equal to the value `Integer.MIN_VALUE`. Also, the structure of the program shows how JUnit provides an easy and concise way to implement unit testing. If we want to implement testing on the type `long`, we just need to add the following method to the class `TestMath`:

```
public void testAddLong() {
  long i = Long.MAX_VALUE;
  assertEquals(i + 1, Long.MIN_VALUE);
}
```

The class in Figure 6-2 is compiled with MJC and is executed during the project build procedure, i.e., with a `make` command.

1. Test case: Java Safe Math

The program in Figure 6-3 implements safe math unit tests for types `byte` and `int`. Since the program is compiled under safe math mode, it should be compiled with MJC in `safe math` mode.

```
//testcase for safeMath
package org.multijava.mjc.testcase.runtime;

import junit.framework.TestCase;

public class TestSafeMath extends TestCase  {

  public void testAddByte() {
    byte i = Byte.MAX_VALUE;
    try{
      assertEquals((byte) (i+1), Byte.MIN_VALUE);
    } catch(ArithmeticException  e){
      fail();
    }
  }
  public void testAddInt() {
    int i = Integer.MAX_VALUE;
    try {
        assertEquals(i + 1, Integer.MIN_VALUE);
      fail();
    } catch(ArithmeticException  e){
    }
  }

  //… testcases for short, long and other integral operations

  protected void setUp() throws Exception {
    super.setUp();// no setup needed in this test file.
  }

  public static void main(String[] args) {
    junit.textui.TestRunner.run(TestSafeMath.class);
  }
}
```

Figure 6-3 sample JUnit program for the types `byte` and `int`

We claim that in safe-math mode, an overflow exception must be checked, i.e. the catch blocks should be entered after doing the addition operations `Byte.MAX_VALUE+1` and `Integer.MAX_VALUE+1`. If not, the method `fail()` will be executed, which indicates that a unit test has failed. The class will be compiled in safe math mode and is executed during project rebuilding with a `make` command.

6.3.2.2 Testcase: for the methods **`checked()`** and **`unchecked()`**

The classes in Figure 6-4 and Figure 6-5 are used to conduct unit testing of method `checked()` and `unchecked()` on the types `byte`, `short`, and `int` for the addition operations. From these tests, we can see that the methods `checked()` and `unchecked()` provide finer control over the math mode used in an application, i.e., arithmetic modes are only applicable to the expressions that are the actual parameters of the methods `checked()` and `unchecked()`.

```
//testcase for _checked method in SafeIntegralArithmetic
package org.multijava.mjc.testcase.runtime;

import junit.framework.TestCase;

public class TestChecked extends TestCase  {

   public void testAddByte() {
            byte i = Byte.MAX_VALUE;
      try{
            assertEquals((byte)
   org.multijava.mjc.SafeIntegralArithmetic.checked(i+1), Byte.MIN_VALUE);
         } catch(ArithmeticException  e){
            fail();
         }
   }

   public void testAddShort() {
      short i = Short.MAX_VALUE;
      try{
            assertEquals((short)
   org.multijava.mjc.SafeIntegralArithmetic.checked(i+1), Short.MIN_VALUE);
         } catch(ArithmeticException  e){
            fail();
         }
   }

   public void testAddInt() {
      int i = Integer.MAX_VALUE;
            try {
                        assertEquals(
org.multijava.mjc.SafeIntegralArithmetic.checked(i+1), Integer.MIN_VALUE);
            fail();
      } catch(ArithmeticException  e){
      }
   }

   //… testcases for long and other integral operations

   public static void main(String[] args) {
      junit.textui.TestRunner.run(TestChecked.class);
   }

}
```

Figure 6-4 sample JUnit program for `checked`

Figure 6-4 shows that safe math mode is applied to addition expressions by using the static `checked()` method of class `SafeIntegralArithmetic`. For example, in method `testAddInt()`, the expression `i+1` (i.e., `Integer.MAX_VALUE+1`) will throw an overflow exception. If `Integer.MAX_VALUE+1` is interpreted in Java mode—and the expression therefore equal to `Integer.MIN_VALUE`—then the method `fail ()` will be called and an error thrown.

```
//Test case for unchecked method in SafeIntegralArithmetic
package org.multijava.mjc.testcase.runtime;

import junit.framework.TestCase;

public class TestUnchecked extends TestCase {

   public void testAddByte() {
      byte i = Byte.MAX_VALUE;
      assertEquals((byte)
   org.multijava.mjc.SafeIntegralArithmetic.unchecked(i+1), Byte.MIN_VALUE);
   }

   public void testAddShort() {
      short i = Short.MAX_VALUE;
      assertEquals((short)
   org.multijava.mjc.SafeIntegralArithmetic.unchecked(i+1), Short.MIN_VALUE);
   }

   public void testAddInt() {
      int i = Integer.MAX_VALUE;
      assertEquals(org.multijava.mjc.SafeIntegralArithmetic.unchecked(i+1),
   Integer.MIN_VALUE
   );
   //… testcases for long and for other integral operations
   public static void main(String[] args) {
      junit.textui.TestRunner.run(TestUnchecked.class);
   }
}
```

Figure 6-5 sample JUnit program for unchecked

Figure 6-5 gives an example of the application of the method `unchecked()`, in which Java math mode is applied to addition expressions using the static `checked()` method from class `SafeIntegralArithmetic`. For example, in the method `testAddInt()`, the expression `i+1` (i.e., `Integer.MAX_VALUE+1`) should not throw an overflow exception. If the expression `Integer.MAX_VALUE+1` is interpreted in Java mode, the result will be equal to `Integer.MIN_VALUE`. Otherwise, if the expression is not equal to `Integer.MIN_VALUE`, an error will be thrown during the testing.

In the next chapter, we will employ three benchmarks to evaluate the new redesigned MJC.

7 Benchmarks

In this chapter we present some benchmarks used to evaluate the new version of MJC that supports safe arithmetic. In general our approach has been to compile two versions of a given benchmark, one with MJC and the other with MJC having the safe-math option on (we call it SMJC for short).

7.1 Java Grande Benchmark

7.1.1 Introduction

Java Grande Benchmark is a suite of benchmark tests for measuring and comparing different Java execution environments. The benchmark applications uses large amounts of CPU processing, I/O, network bandwidth, and memory. They include not only applications in science and engineering but also, for example, corporate databases and financial simulations [JGrande].

The benchmark applications provide three different versions of test suites:

- sequential, suitable for single processor execution
- multi-threaded, suitable for parallel execution on shared memory multiprocessors
- MPJ, suitable for parallel execution on distributed memory multiprocessors

Our testing made use of the sequential benchmarks. The sequential benchmarks can be divided into three areas

- Low-level operations (section 1), evaluating the performance of low-level operations such as arithmetic operations, method calls, and casting.
- Kernels (section 2), measuring specific complex operations such as Heap Sorting,

Fast Fourier Transform (FFT), Sparse Matrix multiplication.

- Large-scale applications (section 3), including applications that require significant resources at runtime.

More information about each section can be found in Appendix A.

7.1.2 Evaluation process

In this section, we compare two versions of JGrande: one is JGrande compiled with MJC and the other with SMJC (i.e., MJC with the safe-math option on). The evaluation process has two phases. First, we obtain two versions of the JGrande bytecode by compiling with the two compilers. Then we run a built-in tool called JGFNumber to generate performance indexes for each section and for the overall package.

7.1.3 Results

The overall performance indices of the two versions of MJC are presented in Table 7-1. The numbers represent the comprehensive index of processing performance on each section. The greater the number is, the better performance the compiler has. Table 7-1 indicates there is no big difference between MJC and SMJC. The result is: 0.6% increase in section1, -1.8% decrease in section2, -0.9% decrease in section3 and overall -0.8% decrease. The reason is the class generated with SMJC implements bytecode safe math. During execution of these classes, extra check for overflow exception cost extra time and hence lowers the performance of SMJC.

Compiler/run		section 1	section 2	section 3
MJC	run 1	4.85	6.15	8.60
	run 2	4.91	6.12	8.49
	run 3	4.80	6.21	8.50
	run 4	4.78	6.25	8.53
	run 5	4.82	6.30	8.55
	average	4.83	6.21	8.53
SMJC	run 1	4.81	6.20	8.37
	run 2	4.81	6.15	8.42
	run 3	4.89	6.12	8.44
	run 4	4.92	5.98	8.50
	run 5	4.98	6.06	8.53
	average	4.86	6.10	8.45

Table 7-1 MJC and SMJC Java Grande Benchmark results

7.2 ESC/Java2 as a Benchmark

ESC/Java2 is an Extended Static Checker for Java, i.e., a programming tool for finding errors in Java programs by examining the source text [Cok+04]. In performing its verification, ESC/Java2 checks code against specifications that are written in the Java Modeling Langauge (JML). ESC/Java2 is distributed as part of the `ESCTools` package which consists of approximately 124 KLOC of code and it contains an extensive test suite. In this section, we compare the run-time efficiency of two versions of ESC/Java2, one compiled with MJC and the other with SMJC. We ran the ESCTools complete test suite (via `make test`) to check all of the test files in ESC/Java and recorded the time for each ESC/Java tool.

Results are shown in Table 7-2. ESC/Java2 compiled with SMJC takes a little bit longer time (3.5% increases) than the one compiled with MJC. Since there are not many arithmetic operations within source code of ESC/Java2 and test codes, the cost of calling safe math method in class `SafeIntegeralArithmetic` and doing safe math operations is small.

ESC/Java2 version		Testing time (sec)
MJC compiled ESC/Java2	run 1	1249.3
	run 2	1227.3
	run 3	1250.3
	run 4	1221.7
	run 5	1245.2
	average	1238.8
SMJC compiled ESC/Java2	run 1	1275.3
	run 2	1280.4
	run 3	1270.3
	run 4	1295.9
	run 5	1289.2
	average	1282.2

Table 7-2 Comparison of execution efficiency of ESC/Java2 (MJC and SMJC compiled version)

7.3 MJC as a benchmark

We decided to use MJC as a benchmark itself. I.e., what we did was:

- As usual, created an MJC binary by using `javac`, the standard Sun Java compiler. We call this version of MJC the Sun MJC. (Note: the Sun MJC is the version we used in the previous section.)
- Next, a version of MJC was compiled with the Sun MJC. We will refer to this version as MJC2.
- Finally, we compiled the MJC source using the Sun MJC but with safe math enabled. We call this the SMJC2.

Then we compiled JGrande with MJC2 and SJMJC2 and compare the time it takes for the compilation. Table 7-3 presents the comparison of compilation time between MJC2 and SMJC2. The difference (only 0.4% increase) can be ignored, which means generating bytecode of calling a method does not cost much more time than primitive arithmetic operations in terms of JVM instructions using these two compilers.

Run	MJC2(s)	SMJC2(s)
1	5.975	6.054
2	5.969	5.991
3	5.984	6.005
4	5.971	5.976
5	5.992	5.991
average	5.978	6.003

Table 7-3 Compile times of Java Grande using MJC2 and SMJC2

We also conduct the same comparison towards ESC/Java2. Table 7-4 presents the result of compilation time of ESC/Java2 using MJC2 and SMJC2. The result shows only 0.08% increase in compilation time.

Run	MJC2(s)	SMJC2(s)
1	8.160	8.146
2	8.162	8.171
3	8.159	8.224
4	8.131	8.148
5	8.205	8.159
average	8.163	8.170

Table 7-4 Compile times of ESC/Java2 using MJC2 and SMJC2

7.4 Comparison of bytecode size

We also collected metrics on bytecode size of all class files generated with MJC and SMJC in each above-mentioned benchmarks. Table 7-5 presents the result:

benchmark	MJC (bytes)	SMJC (bytes)	% (increase)
JGB	246639	248639	0.81%
ESC/Java2	683984	691157	1.05%
MJC	3246026	3270030	0.74%

Table 7-5 bytecode size of benchmarks using MJC and SMJC

The result shows only 0.81%, 1.05% and 0.74% increase in each benchmark respectively. The reason is there is no big difference in bytecode size between JVM instructions of conducting primitive arithmetic operations and calling a safe math method in class `SafeIntegeralArithmetic`.

7.5 Summary

Based on above-mentioned experiments, we can draw the conclusion:

- Cost of generating bytecode for a Java file under different math modes: safe math mode or Java math mode does not have great difference. The reason is: writing a JVM instruction like `iadd` costs similar time as writing a JVM instruction like `invokevirtual` to a class file.
- Cost of conducting safe math operations is a little bit larger than conducting java math operations (on average, around 2% increase based on all conducted testing). The reason is: compared with doing one JVM operation like `iadd`, doing safe math operations require JVM to find the entrance address for class `SafeIntegeralArithmetic` and execute extra operations to make judgment on the result. Then it will cost more time.
- For the most products that have no intensive application of arithmetic operations like FFT, there are no significant performance differences between applying Java math and safe math.
- In terms of bytecode size, the bytecode generated from SMJC has similar size to the bytecode from MJC since the size of JVM instructions between conducting primitive arithmetic operations and calling corresponding methods in class `SafeIntegeralArithmetic` is almost the same.

8 Non-null-by-default in JML: Tool Support for an Empirical Study

8.1 Introduction

Chalin and Rioux recently hypothesized that by design, most declarations of reference types in Java and JML are meant to be non-null [ChalinRioux05]. However, the default in both Java and JML is to allow any reference to hold the value `null`; therefore, developers have to add JML `non_null` annotations to all declarations that are not to contain `null`. In order to verify their hypothesis, Chalin and Rioux conducted an empirical study of the actual number of declarations that are non-null, deriving nullity information from the Java source and JML specifications. In support of this study, my contribution was the creation of an enhancement to the JML checker that gathers statistics of nullity of declarations. In the remainder of this thesis we will sometimes refer to this JML checker feature as the "non-null statistics tool".

As shall be seen, determining if a declaration is meant to be non-null is much more involved than it might at first appear. Hence, in the sections that follow we take the time to describe the general rules that have been used for collecting nullity statistics in JML annotated Java files. This involves defining rules for three constructs:

- class fields,
- method parameters, and
- methods (having a non-`void` return type).

Some program samples are given to help illustrate the rules; the samples are excerpt from an extensive suite of tests developed in conjunction with the non-null statistics feature.

8.2 Nullity Statistics of Class Fields

Field declarations (static or non-static) are counted as non-null when they are either explicitly marked with the `non_null` annotation or constrained to not be `null` by means of a class `invariant` (static or non-static, respectively). The following gives some examples of how to determine if a class field is to be considered non-null.

8.2.1 Explicit non_null modifier

```
public class Field02b {
    public  /*@non_null@*/ static Object so;                //2
    private /*@non_null@*/ Object o;                        //3
}
```

At lines 2 and 3, static and non-static fields are declared with an explicit `/*@non_null@*/`, therefore, the fields `so` and `o` are counted as non-null.

8.2.2 Implicitly non-null

```
public class Field02d {
    static Object so;
    Object o;
    //@ static invariant so != null;                //4
    //@ invariant o != null;                        //5

}
```

At lines 4 and 5, static and non-static fields are declared to be non-null using an expression of the form "*x* `!= null`".

```
public class Field03d {
    static Object so;
    Object o;
    /*@ invariant so != null && o != null; @*/          //4
}
```

The static field `so` is constrained by a non-static invariant, so it cannot be counted as non-null. Only the instance variable `o` should be considered as non-null.

```
/*@ vismod behavior
  @   requires P;
  @   diverges T;
  @   assignable SR;
  @   ensures Q;
  @   signals (Exception e) Ex;
  @*/
vismod T m(T1 p1, T2 p2, …, Tn pn);
```

Figure 8-1 Basic form of a JML method specification (main clauses).

8.3 Method Specifications, Core Syntax and Sugars

Before we describe how to determine nullity information for method parameters and method return types, we begin by describing how JML method specifications can be defined. The first thing that we note is that the grammar of JML is defined in two parts:

- there is a base (or core) subset of JML on which
- so-called syntactic sugars are defined.

Syntactic sugars (e.g., extended and nested specifications) simplify the writing of method specifications in JML. In this section, we present the process for desugaring some syntactic sugars that are involved in our non-null statistics analysis. Here only the parts of specification desugaring methods that are related to non-null statistics are given. Readers can obtain more details from [Cheon03].

8.3.1 The Basic (Core) Form of a JML Method Specification

The basic form of a JML method specification is illustrated in Figure 8-1—`vismod` is one of the visibility modifiers, `private`, `protected` and `public`. Such a basic grouping of clauses (`requires`, `assignable`, `ensures`, etc.) is called a *specification case*. In general, a method specification can have multiple specification cases separated by the keyword `also` (as we shall describe at further length in the next section), but the basic

```
  requires PreCond1;
  assignable Var1;
  ensures PostCond1 ;
  signals (Exception1 e1 ) C1;
also
  requires PreCond2;
  assignable Var2;
  ensures PostCond2;
  signals (Exception2 e2 ) C2;
```

(a) sugared

```
requires PreCond1 || PreCond2;
assignable Var1 , Var2;
ensures (\old(PreCond1 ) ==> PostCond1 )
     && (\old(PreCond2 ) ==> PostCond2 );
signals (Exception1 e1 ) \old(PreCond1 ) ==> PostCond1;
signals (Exception2 e2 ) \old(PreCond2 ) ==> PostCond2;
```

(b) desugared

Figure 8-2 desugaring specification cases combined with "also"

form of method specification has only one specification case. While the basic form is the simplest, however, it can not effectively represent complicated method specifications, e.g., that would naturally be captured by cases, as we clarify next.

8.3.2 Desugaring Specification cases (separated using `also`)

In JML, a method specification can be composed of one or more specification cases separated using `also` as is illustrated in Figure 8-2(a). The specification cases can, but need not, be disjoint. This kind of specification is desugared into a single method specification case.

Figure 8-2 illustrates the desugaring of a method having two specification cases: the desugared precondition is the disjunction of the preconditions of the specification cases; the desugared postcondition is the conjunction of postconditions of specification cases, each guarded with its corresponding precondition. Since all preconditions are evaluated at prestate, they are wrapped with `\old()`.

```
//@  requires index > 0;
public Object getElement(Stack s, int index) throws EmptyException;
)
```

(a) sugared

```
/*@ public behavior
  @   requires  index > 0;
  @   diverges \not_specified;
  @   assignable \not_specified;
  @   ensures \not_specified ;
  @   signals (Exception1) \not_specified;
  @*/
public Object getElement(Stack s, int index) throws EmptyException;
)
```

(b) desugared

Figure 8-3 Desugaring a lightweight specification (main clauses)

8.3.3 Desugaring Lightweight, Normal, and Exceptional Specifications

8.3.3.1 Lightweight

A lightweight method specification case is a method specification case in which

- does not start with a "behavior" keyword;
- only selected clauses are written.

Lightweight specifications are desugared by adding clauses that are omitted each with the value `\not_specified`. The resulting block of clauses is prefixed with

```
        //@ vismod behavior
```

where `vismod` is the visibility of the method being specified. Figure 8-3 illustrates the desugaring of a lightweight specification.

8.3.3.2 `normal_behavior`

A method spec case that starts with

```
//@ vismod normal_behavior
```

is desugared to a `vismod behavior` spec case that has the following `signals` clause added:

```
signals (Exception e) false
```

This clause indicates that no exceptions can be thrown by the method.

8.3.3.3 `exceptional_behavior`

Similarly, a spec case that starts with

```
vismod exceptional_behavior
```

is desugared to a spec case that starts with `vismod behavior` and has the following `ensures` clause inserted:

```
ensures false
```

```
/*@  public normal_behavior
  @     requires !empty(s);
  @     ensures \result !=null;
  @ also
  @  public exceptional_behavior
  @     requires empty(s);
  @     signals_only EmptyStackException;
  @*/
public Object getTop(Stack s) throws EmptyException;
)
```

(a) sugared

```
/*@ public behavior
  @     requires !empty(s);
  @     ensures \result !=null;
  @     signals (Exception e) false;
  @  also
  @ public behavior
  @     requires empty(s);
  @     ensures false;
  @     signals_only EmptyStackException;
  @*/
public Object getTop(Stack s) throws EmptyException;
```

(b) desugared

Figure 8-4 desugaring Normal and Exceptional Specification

This clause indicates that the method cannot return normally. Figure 8-4 illustrates the desugaring of a normal and exceptional specification cases.

8.3.4 Desugaring of Extended Specifications

In JML, a method's specification can be inherited or refined. A subtype inherits all non-private members from its supertypes as well as all non-private method specification. In addition to method specification subtyping, JML's refinement files can extend the corresponding refined method specifications and have no visibility restrictions. For example, the specification in file `IntMathOps2.jml-refined` (shipped with the standard distribution of JML) inherits specifications with all four visibility levels (viz., *private*, *protected*, *public*, and *default*) from file `IntMathOps2.java`.

JML method specifications starting with the keyword `also` are referred to as *extending specifications*. Such an annotation can only be applied to overriding or refining methods, regardless of whether the overridden and refined methods have specification or not. When gathering non-null statistics on a method's extending specification, all of the specifications inherited from its supertypes must be considered.

Figure 8-5 illustrates an example of specification inheritance and its desugaring result. The inherited specification (class CH01A) and extending specification (class CH01AE) are combined with "`also`".

```
public class CH01A{

  String str;

  //@ public behavior
  //@   requires !hasElement(s) ;
  //@   ensures \result !=null;
  public Object getTop(object s){ … }
}

public class CH01AE extends CH01A {
  /*@ also
    @  public behavior
    @    requires !hasElement(s) and s.length() > 1;
    @    ensures \result !=null;
    @    ensures Q;
    @*/
  public Object getTop(object s){ … }
}
```

(a) sugared

```
public class CH01AE extends CH01A {
  /*@ public behavior
    @   requires !hasElement(s) ;
    @   ensures \result !=null;
    @ also public behavior
    @   requires !hasElement(s) and s.length() > 1;
    @   ensures \result !=null;
    @   ensures Q;
    @*/
  public Object getTop(object s){ … }
}
```

(b) desugared version of class CH01AE

Figure 8-5 specification inheritance and its desugaring

8.3.5 Desugaring Nested specifications

It is easiest to explain what a nested specification cases are by providing an example, hence see Figure 8-6. Nested specification cases are written inside "{|" and "|}" brackets. Desugaring of nested specifications is handled by copying all of the clauses in the top spec-header into each of the nested spec cases, as is illustrated in Figure 8-6.

```
requires PreCond0 ;
 {|
   requires PreCond1 ;
   assignable Var1 ;
   ensures PostCond1 ;
   signals (Exception1 e1 ) C1 ;
 also
   requires PreCond2 ;
   assignable Var2 ;
   ensures PostCond2 ;
   signals (Exception2 e2 ) C2 ;
 |}
```

(a) sugared

```
  requires PreCond0 ;
  requires PreCond1 ;
  assignable Var1 ;
  ensures PostCond1 ;
  signals (Exception1 e1 ) C1 ;
also
  requires PreCond0 ;
  requires PreCond2 ;
  assignable Var2 ;
  ensures PostCond2 ;
  signals (Exception2 e2 ) C2 ;
```

(b) desugared

Figure 8-6 desugaring nested method specification

8.3.6 Desugaring empty specifications

If a method not overriding an existing method (or a constructor) has an empty specification, all specification clauses are implicitly taken to be `\not_specified`. The meaning of `\not_specified` may vary between different tools that use JML specifications. For example, a static checker might treat a `requires` clause that is `\not_specified` as `true`, while a verification tools may treat it as `false`. For the purpose of our nullity processing, we treat `\not_specified` as `true`.

However, if a method whose specification is empty overrides another method, then its specification should be considered to be the lightweight specification:

```
also
requires false;
```

Such a specification case actually has no effect on the overall specification of overriding methods—that is, it is a "unit" of also-joined specification cases. For example, the precondition of a desugared specification is in the form

```
requires (p1) || (p2) || … || (pn)
```

where each P_K (k=1..n) is the precondition from each specification case, then, e.g., if p1 is `false`, then p1 can be ignored in this expression.

8.4 Nullity Statistics of Method Parameters

A method parameter is counted as non-null if all `requires` clauses the desugared method specification constrain the parameter to be non-null; i.e. the parameter is defined in one of following forms:

1. a single assertion expression of the form
   - `parameter !=null`, or `null != parameter`, or `!(parameter == null)`, etc.
   - `parameter instanceof T,`
   - `\nonnullelements(parameter)` where parameter is of an array type.
2. a single assertion expression consisting of the application of the conditional (&&) or logical (&) conjunction, where one of the argument expressions is of the form listed in (1).
3. a single assertion expression is a conditional disjunction (`||`) or logical (`|`) disjunction where all of the argument expressions contain an expression of the form listed in (1).

Of course, a method parameter can be declared as explicit non-null with modifier `/*@non_null@*/`

Here are some examples written in the form of sugared specs (since this is the form most familiar to JML developers).

```
public class Param01b {
    String m(/*@non_null*/String s) { return s; }
}
```

Method parameter `s` should be counted as `non_null` since it is declared with an explicit `/*@non_null@*/` modifier.

```
public class Param01c {
    //@ requires s != null;
    String m(String s) { return s; }
}
```

Method parameter `s` should be counted as non-null since it is constrained to be non-null in a precondition clause with `!= null` expression.

```
/*@ public model void m(int i, non_null Integer x){}
  @*/
```

Model method (a "specification-only" method that can only be referred to within JML specifications) parameter `s` should be counted as non-null since it is declared with explicit `non_null` modifier.

```
//@ requires x != null && o != null;
public void m(int i, Object o){}
```

Method parameter `i` and `o` should be counted as non-null since they are constrained to be non-null in a precondition clause with a conjunction of two `!=null` expressions.

```
//@ requires x != null || o != null;
public void m(int i, /*@non_null@*/ Integer x, Object o){}
```

Method parameter `i` and `o` should not be counted as non-null since they are specified in a precondition clause with a disjunction of two not equal to null expressions.

```
public class CF_HP_b {
    //@ requires o != null;
    public void m(int i, Integer x, /*@ non_null @*/ Object o){}
}
public class CF_HC_pb_b extends CF_HP_b {
    //@ also
    //@ requires o != null;
    public void m(int i, Integer x, /*@ non_null @*/ Object o){}
}
```

After desugaring, the method specification of `m()` in class `CF_HC_pb_b` can be written as:

```
requires o!=null || o!=null;
```

which always constrains `o` to be not equal to null. Therefore, `o` should be counted as non-null.

```
public class CF_HC_pb_e extends CF_HP_e {

    //@ requires o != null || x !=null;
    //@ ensures \result !=null;
    public Object m(int i, Integer x, Object o){
      if (o!=null)
        super(i , x, o);
      else if (x!=null)
        return x;
    }
}
public class CF_HP_e {
    //@ also
    //@   requires o != null;
    //@   ensures \result !=null;
    public Object m(int i, Integer x, /*@ non_null @*/ Object o){return o;}
}
```

After desugaring, the method specification of method `m()` in class `CF_HC_pb_e` can be written as:

```
//@ requires (o != null || x !=null) || (o != null);
//@ ensures \old(o != null || x !=null) =>
//@         \result !=null && \old(o != null)=> \result !=null;
```

In the desugared precondition clause, `o` is not always required to be not equal to null. Therefore, it should not be counted as non-null.

8.5 Nullity Statistics of Method Return Types

A method's return type is counted as non-null if all `ensures` clauses constrain `\result` to not be equal to `null`. That is, its return type is counted as non-null if after desugaring of its specification, all postconditions have one of the following forms:

1. a single assertion expression is of one of the following forms:
   - `o != null`, reference `o` is not equal to `null`
   - `o instanceof C`, reference `o` is an instance of class `C`, which states `o` is not equal to `null`
   - `\fresh(o)`, reference `o` is to a new object (this can be found only in an `ensures` clause)
   - `\nonnullelements(a)`, for an array reference `a`
2. a single assertion expression is a `&&` or `&` (i.e., a *conditional* or *logical* conjunction) expression that contains `\result != null`;
3. a single postcondition statement is a `||` or `|` (i.e., a *conditional* or *logical* disjunction) expression that contains `\result != null` in both left- and right-hand side expressions.

Also, a method's return type can be inferred to be non-null when its declaration explicitly contains the `/*@non_null@*/` modifier.

Some examples follow.

```
public class Method01b {
  /*@non_null@*/ String m1() { return ""; }          //2
}
```

Method `m1()` should be counted as non-null since it is declared with an explicit non-null modifier.

```
public class Method01c {
  //@ ensures \result != null;
  String m2() { return ""; }
}
```

Method `m2()` should be counted as non-null since it is constrained to be non-null in a postcondition clause with the single expression `\result != null`.

```
public class CHPla {
    public String m3() { return ""; }
}
public class CH01d extends CHPla {
    public /*@non_null@*/ String m3() { return ""; }
}
public class CH01dS1 extends CH01d {
    public /*@non_null@*/ String m3() { return ""; }
}
```

The method specification of `m3()` in class `CH01dS1` can be desugared to:

```
  requires \not_specified
also
  ensures \result != null;
also
  ensures \result != null;
```

Tthe above specification can be further desugared to

```
requires \not_specified || \not_specified|| \not_specified;
ensures \old(\not_specified) =>\not_specified &&
\old(\not_specified)=>\ result != null &&
\old(\not_specified)=>result != null;
```

Therefore, the method `m3()` in class `CH01dS1` should not be counted as non-null.

```
public class CH01k_e extends CH01k {

   //@ requires s!=null;
   //@ ensures \result !=null && str!=null;
   public String m(String s) {
      str=" ";
      return s+str;
   }
}
public class CH01K {

   String str;

   //@ also
   //@   ensures \result !=null;
   public String m(String s){
      str=" ";
      return s+str;
   }
}
```

After desugaring, the method specification of `m()` in class `CH01K_e` can be written as

```
//@ requires s!=null || \not_specified;
//@ ensures \old(s != null) =>
//@          (\result != null && str != null)
//@           && \old(\not_specified) =>\result !=null;
```

Regardless of the precondition that holds, the `m`'s return type is constraint to be not equal to `null`; therefore, the return type of method `m()` in class `CH01K_e` should be counted as non-null.

9 Non-null Statistics Feature: Implementation and Results

The non-null statistics tool looks up existing specifications in a file and gives counts of non-null annotated references that occur in:

- normal, ghost and model class fields,
- normal and model method return references,
- normal and model method parameter references.

In addition, the tool provides separate counts for references that are explicitly declared with a `/*@non_null@*/` modifier and references that are implicitly (via assertion expressions as explained in the previous section).

9.1 Some Important Methods in the Implementation

Some of the more interesting methods in the implementation of the non-null statistics tool are discussed in the following sections.

9.1.1 Method CheckExpressions

The analysis of expressions such as the not-equal-to expression (`!=`) or the conditional and expression (`&&`) plays an important role in determining implicit non-null declarations. In the implementation, the handling of expressions ultimately ends by processing one of following expressions: `JInstanceofExpression`, `JmlEqualityExpression`, `JmlFreshExpression`, or `JmlNonNullElementsExpression`. The method `checkExpression()` recursively parses an expression until it reaches a basic expression and transfers the handling of parsed expressions to corresponding processing methods. For example, expression `a != null && b = null` will be considered as a JML conditional-conjunction expression and parsed to two JML equality expressions: `a != null` and `b =null`, then both expressions will be separately processed with a method that handles an equality expression. Figure 9-1 shows the hierarchy of JML expression classes that are involved in our expression processing

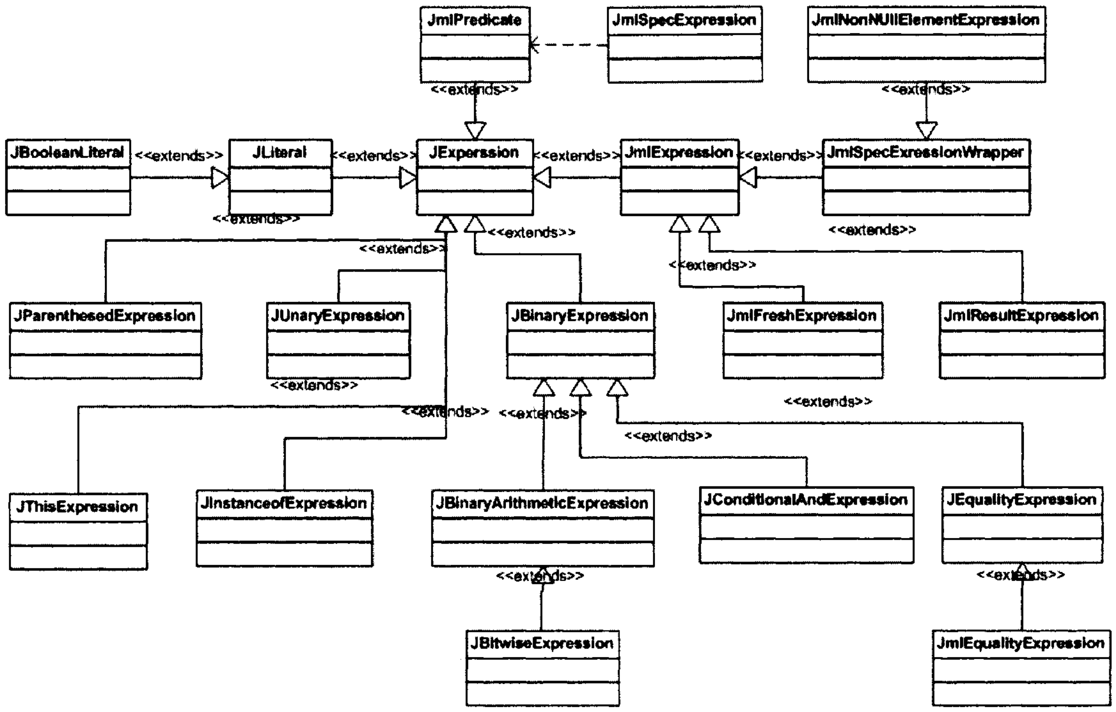


Figure 9-1 Class diagram of method CheckExpression

9.1.2 Method CheckSpecification

The basic idea of processing a method specification is to simulate its desugaring. Therefore, a recursive traversal of the inheritance tree is required. In the implementation, an array is employed to store the specification of each class node of the inheritance tree.

Elements of the array are then parsed and analyzed. In the implementation, the specification is divided into lightweight specifications and heavyweight specifications.

For lightweight specifications, the grammar is:

lightweight-spec-case ::= generic-spec-case

generic-spec-case ::= [spec-var-decls] spec-header [generic-spec-body]

| [spec-var-decls] generic-spec-body

spec-header ::= requires-clause [requires-clause] . . .

generic-spec-body ::= simple-spec-body | '{|' generic-spec-case-seq '|}'

generic-spec-case-seq ::= generic-spec-case [also generic-spec-case] . . .

simple-spec-body ::= simple-spec-body-clause [simple-spec-body-clause] . . .

simple-spec-body-clause ::= diverges-clause | assignable-clause | when-clause

| working-space-clause | duration-clause | ensures-clause

| signals-only-clause | signals-clause

The grammar for heavyweight specifications is

heavyweight-spec-case ::= behavior-spec-case | exceptional-behavior-spec-case

| normal-behavior-spec-case

behavior-spec-case ::= [privacy] [code] behavior-keyword generic-spec-case

behavior-keyword ::= behavior | behavior

normal-behavior-spec-case ::= [privacy] normal-behavior-keyword normal-spec-case

normal-behavior-keyword ::= normal_behavior | normal_behavior

normal-spec-case ::= generic-spec-case

exceptional-behavior-spec-case ::= [privacy] exceptional-behavior-keyword

exceptional-spec-case | exceptional_behavior

exceptional-spec-case ::= generic-spec-case

The grammar for top-level method specifications is

method-specification ::= non-extending-specification | extending-specification

non-extending-specification ::= spec-case-seq *spec-case-seq ::= spec-case [also spec-case] . . .*

spec-case ::= lightweight-spec-case | heavyweight-spec-case

extending-specification ::= also spec-case-seq

From the above grammar definitions, the handling of specifications entails handling spec case sequences, specification bodies, and specification body clauses. Therefore, in the implementation, four methods (viz., `checkSpecification()`, `checkSpecCase()`, `checkSpec-Body()`, and `checkSpecBodyClause()`) are handle specification, spec case sequences, spec body and spec body clauses, respectively.

In order to exert as little impact on current JML projects as possible, a new file called `NonNullStatistics.java` was created, in which most of processing work is done. Figure 9-2 gives the class diagram of the JMLNode class hierarchy illustrating all node types that need to be handled by `checkSpecification()`.

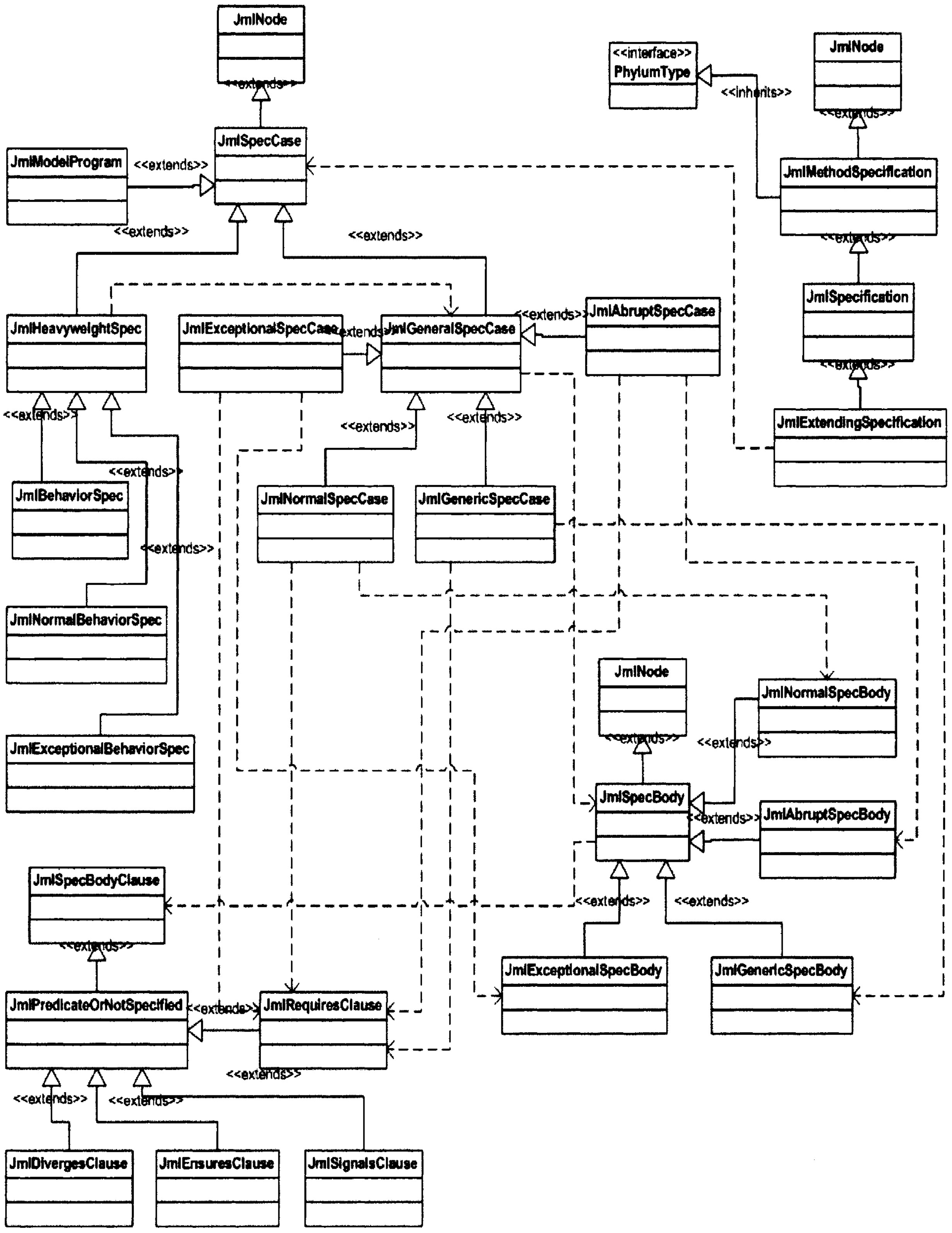

JmlNode
<<extends>>
JmlSpecCase
JmlModelProgram
<<extends>>
<<interface>>
PhylumType
<<inherits>>
JmlMethodSpecification
JmlSpecification
JmlExtendingSpecification
JmlHeavyweightSpec
JmlExceptionalSpecCase
JmlGeneralSpecCase
JmlAbruptSpecCase
JmlBehaviorSpec
JmlNormalBehaviorSpec
JmlExceptionalBehaviorSpec
JmlNormalSpecCase
JmlGenericSpecCase
JmlSpecBody
JmlNormalSpecBody
JmlAbruptSpecBody
JmlExceptionalSpecBody
JmlGenericSpecBody
JmlSpecBodyClause
JmlPredicateOrNotSpecified
JmlRequiresClause
JmlDivergesClause
JmlEnsuresClause
JmlSignalsClause

Figure 9-2 `JMLNode` class hierarchy.

9.2 Results

Chalin and Rioux made use of the nullity statistics features of the JML checker. They included in their study a random sampling of 161 KLOC out of a total 457 KLOC. They were able to demonstrate (with 95% certainty) that over 60% of declarations of reference types are meant to be non-null.

10 Conclusions and Future Work

The main contribution of this thesis is the implementation of bytecode level support for safe-math arithmetic in MJC. The work is based on a work by Chalin [Chalin03]. To implement this in MJC, cases in which incorrect integral arithmetic operations occurred are found, and a new class is generated based on these cases.

Another contribution of this thesis is the implement of a non-null statistics gathering tool in JML. Setting reference types as non-null by default in JML is proposed by Chalin, an idea which was motivated by the fact that the majority of reference types are intended to be non-null in Java. The implementation of this tool is based on the simulation of the desugaring process of different kinds of method specifications in JML and its corresponding non-null judgement rules.

Although the work to define all reference types as non-null by default has been implemented in the JML checker and RAC, it is based on detecting violations at runtime. Future work should consider extending the JML checker to use non-null types, guided by the work of Fähndrich and Leino [Fähndrich03]. In addition, the work presented here on safe-math arithmetic could be incorporated into ESC/Java2 to allow potential coding errors to be detected sooner.

11 References

[Abercrombie02]	Parker Abercrombie; Murat Karaorman, jContractor: Bytecode Instrumentation Techniques for Implementing Design by Contract in Java, Electronic Notes in Theoretical Computer Science, volume 70, Elsevier, 2002.
[Bartetzko01]	Detlef Bartetzko, M. M., Clemens Fischer and H. Wehrheim, Jass - java with assertions, (2001).
[Berg01]	Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In: T. Margaria and W. Yi editors, Tools and Algorithms for the Construction and Analysis of Software (TACAS), LNCS 2031, pp. 299-312, Springer, 2001.
[Breunesse02]	C.-B. Breunesse, J. van den Berg, and B. Jacobs. Specifying and verifying a decimal representation in Java for smart cards. In H. Kirchner and C. Ringeissen, editors, AMAST'2002, LNCS, pp. 304-318. Springer-Verlag, 2002. The Decimal class specification is available at www.cs.kun.nl/indexes/~ceesb/decimal/Decimal.java.
[Burdy03]	Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll, An overview of JML tools and applications, May 2003.
[Chalin02]	Patrice Chalin. Back to Basics: Language Support and Semantics of Basic Infinite Integer Types in JML and Larch. Technical Report 2002-003.3, Computer Science Department, Concordia University, October 2002, revised March, May 2003.
[Chalin03]	Patrice Chalin, Improving JML: For a Safer and More Effective Language, FME 2003, Proceedings of the International Symposium of Formal Methods Europe, Pisa, Italy, Sept. 8-14, 2003.
[Chalin04]	Patrice Chalin, JML Support for Primitive Arbitrary Precision Numeric Types: Definition and Semantics, Journal of Object Technology, June 2004.
[ChalinRioux05]	P. Chalin and F. Rioux, "Non-null References by Default in the Java Modeling Language". Workshop on the Specification and Verification of Component-Based Systems (SAVCBS'05), Lisbon, Portugal, ACM Press, Sept. 2005.
[Cheon02]	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.
[Cheon03]	Yoonsik Cheon, A Runtime Assertion Checker for the Java Modeling Language, Ph.D thesis, April 2003.

[Clifton01]	Curtis Charles Clifton, MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch, November 2001.
[Cok+04]	D. R. Cok and J. R. Kiniry, "ESC/Java2: Uniting ESC/Java and JML". In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean editors, Proceedings of the *International Workshop on the Construction and Analysis of Safe, Secure, and Interoperable Smart Devices* (CASSIS'04), Marseille, France, March 10-14, vol. 3362 of LNCS, pp. 108-128, Springer, 2004.
[DJava]	http://mrl.nyu.edu/~meyer/jvm/djava/.
[Enseling01]	Oliver Enseling, iContract: Design by Contract in Java, http://www.javaworld. com /javaworld/jw-02-2001/jw-0216-cooltools.html.
[Fähndrich03]	M. Fähndrich and K. R. M. Leino, "Declaring and Checking Non-null Types in an Object-Oriented Language," in Proceedings of the 18th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. OOPSLA'03: ACM Press, 2003, pp. 302-312.
[Hoare69]	C. A. R. Hoare. An axiomatic basis for computer programming. Communications of the ACM, 12(10), pp.576-583, October 1969.
[Jamaica]	http://www.javaworld.com/javaworld/jw-05-2004/jw-0503-jamaica.html
[JBA]	http://tinf2.vub.ac.be/~dvermeir/courses/compilers/javaa/
[Jacobs03]	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.
[Jasmin]	http://jasmin.sourceforge.net/
[Jass]	The Jass Team: The Jass Project, Mar 2004, Home Page http://csd.informatik.uni-oldenburg.de/~jass/.
[JavaD]	http://www.bearcave.com/software/java/javad/.
[JGrande]	Java Grande Forum Benchmark, http://www.epcc.ed.ac.uk/-javagrande/javag.html
[JLS00]	Sun Microsystem Ltd The Java Language Specification (Second Edition), 2000, Web Page http:// java.sun.com/docs/ books /jls/
[JML05]	The Java Modeling language (JML), August, 2005. Home Page 'http://www.jmlspecs.org'
[Jones90]	Cliff B. Jones. Systematic Software Development Using VDM. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition, 1990.
[JUnit]	http://www.junit.org.

[Lackner02]	Martin Lackner, Andreas Krall, Franz Puntigam, Supporting Design by Contract in Java, Journal of Object Technology,Vol 1, No 3, 2002.
[Larman04]	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.
[Leavens02]	Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. Department of Computer Science, Iowa State University,TR #98-06t, December 2002.
[Leavens04a]	Gary T. Leavens and Yoonsik Cheon, Design by Contract with JML, March 4, 2004.
[Leavens04b]	Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Joseph Kiniry: JML Reference Manual. 11 October 2004.
[Linpack]	Linpack benchmark, http://www.netlib.org/benchmark/linpack-java/.
[Meyer92]	Bertrand Meyer. Eiffel: The Language. Object-Oriented Series. Prentice Hall, New York, NY, 1992.
[Meyer02]	Bertrand Meyer, *Applying "Design by Contract*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992, pp. 40-51
[MultiJava04]	The MultiJava Team: The MultiJava Project, December 2004.Webpage: 'http://multijava.sourceforge.net/index.shtml'.
[Neuron]	http://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/Neuron-Java-Disassembler.shtml
[Owre96]	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., pp. 411–414, Springer, Berlin, 1996.
[PARR05]	Terence Parr, ANTLR Parser Generator, http://www.antlr.org/. Aug, 2005
[PVS]	The PVS Specification and Verification System. http://pvs.csl.-sri.com.
[Raghavan03]	Arun D. Raghavan and Gary T. Leavens, Desugaring JML Method Specifications, June 2003.
[SciMark]	SciMark Benchmarks, http://math.nist.gov/scimark2/.
[SDN]	http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4449383.
[vdBJ01]	Joachim van den Berg and Bart Jacobs. "The LOOP compiler for Java and JML". In *Tools and Algorithms for the Construction and*

Analysis of Software (TACAS), LNCS 2031, pp. 299-312. Springer, 2001.

[Wing87] Jeannette M. Wing. Writing Larch interface language specifications. ACM Transactions on Programming Languages and Systems, 9(1):1–24, January 1987.

[Wing90] Jeannette M. Wing. A specifier's introduction to formal methods. Computer, 23(9), pp. 8–24, September 1990.

[Winkler02] Jurgen F. H. Winkler, A safe variant of the unsafe integer arithmetic of Java, SOFTWARE—PRACTICE AND EXPERIENCE, pp. 669–701, 32, 2002.

[Winskel93] Glynn Winskel. The Formal Semantics of Programming Languages: An Introduction. Foundations of Computing Serices, MIT Press, 1993.

Appendix A – Java Grande Benchmark

Low-level operations:

Benchmarks in this section are designed to measure the performance of the Java compilers towards low-level operations. These benchmarks are designed to run for a fixed period of time and the number of executed operations during this time is recorded. The performance reported as operations/second.

Arithmetic

This benchmark measures the performance of arithmetic operations including addition, multiplication, and division on the primitive data types int, long, float, and double. The performance index is the number of operations performed per second.

Assignment

This benchmark measures the cost of assignment to different types of variables, which can be scalars or array elements, local variables, instance variables, or class variables. The performance index is the number of assignments performed per second.

Casting

This benchmark measures the performance of casting between different primitive types including int⇔float, int⇔double , long⇔float, and long⇔double. The performance index is the number of casts performed per second. Other type pairs could also be tested (e.g., byte⇔int), but sometimes the measured results are meaningless due to the swiftness of the casting.

Create

This benchmark tests the performance of creating objects and arrays. Arrays are created of types int, long, float, and Object, and in different sizes. Both complex and simple objects, with and without constructors, are created during the test. The performance index is the number of arrays or objects created per second.

Loop

This benchmark determines the performance of looping constructs, which include a for loop, a reversed for loop, and a while loop. The performance index is the number of iterations executed per second

Math

This benchmark measures the performance of math methods from java.lang.Math. The performance index is the number of operations performed per second. For some methods (e.g., `exp`, `log`, the inverse trig functions), the cost also includes the cost of any arithmetic operations (addition or multiplication) that are necessary to provide a stable iteration without overflow. In addition, if the cost of these additional operations is significant, the result can be corrected by using the relevant result from the Arithmetic benchmark.

Method

This benchmark measures the cost of method calling. Methods can be instance, final instance, or class methods, and the calling can be made from within the same or different classes. The performance index is the number of methods call per second.

Note: since both final instance and class methods can be statically linked, a high performance figure for these tests generally indicates that the compiler successfully inlined these methods[JGrande].

Serial

This benchmark measures the performance of object serialization. Both the writing and reading of objects to and from a file are involved. The types of tested objects are arrays, vectors, linked lists, and binary-trees. The performance index is the number of bytes per second converted.

Exception

This benchmark measures the performance of exception handling. The cost of creating, throwing, and catching exceptions, which can occur within a method or further down the inheritance tree, is measured. The performance index is the number of exceptions handled per second.

Kernels

Benchmarks in this section are some applications which conduct numerically intensive tests. For each benchmark, small (size A), medium (size B), and large (size C) versions are supplied. The performance index is the number of operations executed per second.

Series

This benchmark computes the first N Fourier coefficients of the function `f(x) = (x+1)^x` over the interval [0,2]. Performance units are coefficients calculated per second. This benchmark heavily employs transcendental and trigonometric functions. Since there is little array activity involved in the test, it should not be dependent on cache or memory architecture.

Size	N
A	10,000
B	100,000
C	1,000,000

LUFact

This benchmark solves an N x N linear system using LU factorization of linear equations with one right-hand side, `Ax=b`. The matrix is generated randomly and the right-hand side is constructed so that the solution has all components equal to one. The method of solution is based on a Gaussian elimination with partial pivoting [Linpack]. Performance units are Mflops/s (Millions of floating point operations per second)

Size	N
A	500
B	1,000
C	2,000

SOR (Jacobi Successive Over-Relaxation)

Jacobi Successive Over-Relaxation of an N x N grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Drichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns, where each `A(i,j)` is assigned an average weighting of its four nearest neighbors.

The inner loops of the kernel look like

```
for (int i=1; i < Mm1; i++)
{
        double[] Gi = G[i];
        double[] Gim1 = G[i-1];
        double[] Gip1 = G[i+1];
        for (int j=1; j < Nm1; j++)
                Gi[j] = omega_over_four * (Gim1[j] + Gip1[j] + Gi[j-1]
                + Gi[j+1]) + one_minus_omega * Gi[j];
}
```

Some hand optimizing is done by aliasing the rows of `G[][]` to streamline the array accesses in the update expression. The data size for the LARGE version of the benchmark uses a 1,000x1,000 grid.

The SOR benchmark performs 100 iterations of successive over-relaxation on an NxN grid. The performance is measured in iterations per second.

Size	N
A	1000
B	1500
C	2000

HeapSort

This benchmark measures memory moving performance by sorting an array of integers with size N using a heap-sort algorithm. The algorithm can test non-sequential performance of cache, with added burden that moves are byte-wide and can occur at odd address boundaries. This may exert extra weight on cell-based processors, in which additional shift operations are performed to deal with bytes. Performance is measured in units of items per second.

Size	N
A	1,000,000
B	5,000,000
C	25,000,000

Crypt

This benchmark performs IDEA (International Data Encryption Algorithm) encryption and decryption on an array of bytes with size N. Performance is measured in kilobytes per second.

Size	N
A	3,000,000
B	20,000,000
C	50,000,000

FFT

This performs a one-dimensional, forward transform of N complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references, and trigonometric functions[SciMark].

Size	N
A	2097152
B	8388608
C	16777216

Sparse Matrix Multiplication

This benchmark uses sparse matrices stored in compressed-row format with a prescribed sparsity structure. For example, a 1,000 x 1,000 sparse matrix with 5,000 nonzero elements has following storage pattern,

```
 **------------.
 ***           |
 * * *         |
** *  *        |
** **   *      |
** * *    *    |
* *  * *    *  |
*  *  *  *    *|
*   *  *   *   *|
*---*---*----*---*
```

in which each row has approximately 5 nonzero elements and evenly distributes between the first column and the diagonal column. This benchmark exercises indirection addressing and non-regular memory references [SciMark]. An N x N sparse matrix is used for 200 iterations. Performance is measured in iterations per second.

Size	N
A	50000
B	100000
C	500000

Large-scale applications

The benchmarks are intended to be large-size applications that are suitably modified by eliminating any I/O and graphical components. For each benchmark, small (size A) and large (size B) versions are supplied. The performance is measured in operations per second.

Search

The search benchmark solves a game of connect-4 on a 6 x 7 board using an alpha-beta pruning technique. The problem size is determined by the initial position from which the game in analyzed. The number of positions evaluated (N) is recorded, and the performance reported in units of positions per second (Memory and integer intensive).

Size	N
A	7321073
B	34517760