SRC Technical Note

2000-002

October 12, 2000

ESC/Java User's Manual

K. Rustan M. Leino, Greg Nelson, and James B. Saxe

Compaq Computer Corporation
Systems Research Center
130 Lytton Avenue
Palo Alto, CA 94301
http://research.compaq.com/SRC/

Abstract

The *Compaq Extended Static Checker for Java* (ESC/Java) is a programming tool that attempts to find common run-time errors in Java programs by static analysis of the program text. Users can control the amount and kinds of checking that ESC/Java performs by annotating their programs with specially formatted comments called *pragmas*. This manual is designed to serve both as an introduction to ESC/Java and as a reference manual. It starts by providing an overview of ESC/Java through an illustrative example of its use and a summary of its features, and then goes on to document all the pragmas supported by ESC/Java and all the kinds of warnings that it generates. Appendices provide a brief sketch of ESC/Java's implementation, information about obtaining ESC/Java, and some discussion of its limitations.

Preface

The *Compaq Extended Static Checker for Java* (ESC/Java) is a programming tool that attempts to find common run-time errors in Java programs by static analysis of the program text. Users can control the amount and kinds of checking that ESC/Java performs by annotating their program with specially formatted comments called *pragmas*. This manual starts by providing an overview of ESC/Java through an illustrative example of its use and a summary of its features, and then goes on to document all the pragmas supported by ESC/Java and all the kinds of warnings that it generates. It also provides basic information about running ESC/Java.

This manual documents Version 1.2.2 of ESC/Java, built on October 12, 2000. We sometimes speak of ``the current ESC/Java'--rather than just ``ESC/Java''--to emphasize that particular features, bugs, or limitations under discussion are artifacts of the Version 1.2.2 implementation and may be subject to change in future versions. Of course there is no guarantee that all such aspects of ESC/Java will in fact change (nor that other aspects will remain unchanged) as the tool evolves.

This manual is designed to serve both as an introduction to ESC/Java and as a reference manual. First-time readers may prefer to skip the portions marked ``**Fine point(s)**'' as well as some other parts that we have indicated. On the other hand, the extended example in section 0 should be particularly helpful to first-time readers.

For a much abridged treatment of the information in this manual, see the ``ESC/Java Quick Reference'' [SLS00]. The Quick Reference lists most of the specification language features described in sections 2 and 3 of this manual and all the warning types described in section 4, giving very brief descriptions of each. While it necessarily omits numerous examples, tips, motivating discussions, and technical details found in the present manual, it will still be of interest both the new reader seeking a whirlwind tour of ESC/Java's features and to the experienced user seeking a quick reminder about some ESC/Java feature.

Although ESC/Java contains a full Java program verifier, the goal of ESC/Java is not to provide formally rigorous program verification. Rather, it is to help programmers find some kinds of errors more quickly than they might be found by other methods, such as testing or code reviews. Consequently, ESC/Java embodies engineering trade-offs among a number of factors including the frequency of missed errors, the frequency of false alarms, the amount of time used by the tool, the effort required to learn and use the annotation language, and the effort required to implement the tool. In this manual we attempt to give a precise description of the syntax, type-checking, and other linguistic rules of the annotation language, as well as a clear though informal description of the meanings of the various pragmas. While we discuss potential sources of missed errors or false alarms at

various places in the manual, including a summary in appendix C, we do not attempt to characterize precisely the degree of (in)accuracy of ESC/Java's checking

In many places we cite sections of *The Java Specification Language*, by James Gosling, Bill Joy, and Guy Steele [*JLS*]. For example, the notation ``[*JLS*, 19.2]" refers to section 19.2 of *The Java Language Specification*. The list of references at the end of this manual gives bibliographic information for this and other works cited herein.

A specification notation and set of tools related to ESC/Java is the Java Modeling Language (JML) [LBR99, LBR00, LLPRJ00]. Through a collaborative effort, we have attempted to make the JML specification language and the ESC/Java annotation language as similar as practical, with JML providing a superset of the features of ESC/Java. The goals of the two languages are different, so differences in the notations remain. However, many programs annotated with ESC/Java annotations should be amenable to processing with other JML tools, and programmers who learn one language should have little trouble picking up the other as well.

Acknowledgments

ESC/Java was originally designed and implemented at the Compaq Systems Research Center (SRC) by Cormac Flanagan, Mark Lillibridge, Raymie Stata, and the authors. Todd Millstein implemented ESC/Java's execution trace facility and helped with other aspects of ESC/Java. Caroline Tice and Rajeev Joshi lent helping hands with miscellaneous aspects of the implementation, including reducing ESC/Java's memory footprint (Tice) and getting the build recipe for the Simplify theorem prover to work on multiple platforms (Joshi).

The ESC/Java project is a follow-on to an earlier extended static checking project at SRC targeting the Modula-3 language [DLNS98], and continues to use the Simplify theorem prover developed as part of that project.

The authors thank our colleagues who have offered comments on earlier versions of this manual. Allan Heydon, David Jefferson, Mark Lillibridge, Silvija Seres, and Caroline Tice have been particularly helpful in this regard. Silvija Seres compiled the initial version of the ESC/Java Quick Reference [SLS00] from an earlier version of this manual. We and the other implementers of ESC/Java have benefited from the feedback of early users, including Sanjay Ghemawat, Steve Glassman, Allan Heydon, David Jefferson, Marc Najork, Keith Randall, and Silvija Seres. Gary Leavens played a major role in our attempt to remove gratuitous incompatibilities between ESC/Java and JML, and helped us to improve the ESC/Java annotation language in the process.

Contents

---

0 An illustrative example of using ESC/Java

To give the reader a general idea of what ESC/Java does and how to use it, we begin with an example illustrating some salient features of ESC/Java through its application to a simple class declaration deliberately seeded with some errors.

0.0 Scene 0: We write a class declaration.

Our example is the class `Bag` declared in a file `Bag.java` as follows:

```
line
 1: class Bag {
 2:   int[] a;
 3:   int n;
 4:
 5:   Bag(int[] input) {
 6:     n = input.length;
 7:     a = new int[n];
 8:     System.arraycopy(input, 0, a, 0, n);
 9:   }
10:
11:   int extractMin() {
12:     int m = Integer.MAX_VALUE;
13:     int mindex = 0;
14:     for (int i = 1; i <= n; i++) {
15:       if (a[i] < m) {
16:         mindex = i;
17:         m = a[i];
18:       }
19:     }
```

```
20:       n--;
21:       a[mindex] = a[n];
22:       return m;
23:     }
24:   }
```

A copy of this program is in *escjavaRoot*`/examples/0.0/Bag.java`, where *escjavaRoot* denotes the (site-specific) top-level directory for ESC/Java (see appendix B). The idea is that an object `x` of class `Bag` represents a multiset (also known as a ``bag'') of integers in the form of integer array `x.a` together with an integer `x.n`, where the elements of the multiset are the first `x.n` elements of the array. In our example declaration, we define a constructor that creates a `Bag` from an array, and a single method `extractMin` that finds, removes, and returns the minimal element of a `Bag`. The `extractMin` method iterates over the first `this.n` elements of `this.a`, keeping track of the smallest one in the local variable `m`. After finding the smallest element of `this.a`, it copies the element at the highest used position of `x.a` into the position formerly occupied by the minimum element, decreases the count of used elements, and returns the value that was saved in `m`.

0.1 Scene 1: We run ESC/Java, and it warns of two potential `null` dereferences.

To compile `Bag.java`, we would type the command line:

```
javac Bag.java
```

To check our definition of class `Bag` with ESC/Java, we instead type the following similar command line:

```
escjava Bag.java
```

ESC/Java then produces the following output:

```
Bag.java:6: Warning: Possible null dereference (Null)
    n = input.length;
             ^
Bag.java:15: Warning: Possible null dereference (Null)
      if (a[i] < m) {
           ^
Execution trace information:
    Completed 0 loop iterations in "Bag.java", line 14, col 4.

Bag.java:15: Warning: Array index possibly too large (IndexTooBig)
      if (a[i] < m) {
           ^
Execution trace information:
    Completed 0 loop iterations in "Bag.java", line 14, col 4.

Bag.java:21: Warning: Possible null dereference (Null)
    a[mindex] = a[n];
                 ^
Execution trace information:
    Completed 0 loop iterations in "Bag.java", line 14, col 4.
```

```
Bag.java:21: Warning: Possible negative array index (IndexNegative)
    a[mindex] = a[n];
                 ^
Execution trace information:
    Completed 0 loop iterations in "Bag.java", line 14, col 4.

5 warnings
```

Remark: Actually, the above is the output generated by the command

```
escjava -quiet Bag.java
```

The normal output of ESC/Java also includes various progress messages and timing information which are omitted here.

Remark: Some of the messages above include a part marked as ``Execution trace information". We say more about execution traces below, particularly in section 4.0, but will not discuss them further in the course of this introductory example.

0.2 Scene 2: We write a `requires` (precondition) pragma for the `Bag` constructor.

ESC/Java's first warning is that the attempt on line 6 to access `input.length` might fail because `input` might be `null`. We now must decide what to do about this warning.

One approach would be to decide that the implementation of the constructor is incorrect. Following this approach, we would modify the constructor to test for a null argument and, for example, construct an empty multiset:

```
Bag(int[] input) {
  if (input == null) {
    n = 0;
    a = new int[0];
  } else {
    n = input.length;
    a = new int[n];
    System.arraycopy(input, 0, a, 0, n);
  }
}
```

This would indeed eliminate the first warning. Instead, however, we will continue our example by illustrating a second approach, in which we decide that the implementation of the constructor is correct, but that we do not intend for the constructor ever to be called with a null argument. We inform ESC/Java of this decision by adding an annotation to the constructor declaration:

```
//@ requires input != null;
Bag(int[] input) {
  n = input.length;
...
```

When the character `@` is the first character after the initial `//` or `/*` of a Java comment, as in the first line of the

program fragment above, ESC/Java expects the body of the comment to consist of a sequence of ESC/Java annotations, known as *pragmas*. The `requires` pragma above specifies a *precondition* for the constructor, that is, a boolean expression which must be `true` at the start of any call. When checking the body of a method or constructor that is annotated with a precondition, ESC/Java can assume the truth of the precondition to confirm the absence of errors in the body (for example, the absence of a null dereference during the evaluation of `input.length` in the code fragment above). When checking code that calls a method or constructor annotated with a precondition, ESC/Java will issue a warning if it cannot confirm that the precondition (with the values of the actual parameters substituted for the formal parameter names) would always evaluate to `true` at the call site.

0.3 Scene 3: We add a `non_null` pragma for the field `a`.

So much for the first warning. We now turn our attention to the second warning:

```
Bag.java:15: Warning: Possible null dereference (Null)
      if (a[i] < m) {
           ^
```

Here, ESC/Java is warning that the array variable `a` (actually `this.a`) might be `null`. We could deal with this warning by either of the approaches discussed above in connection with the first warning--that is, by adding the precondition `requires a != null` to the `extractMin` method, or by adding special code for the case `a == null`. However ESC/Java offers yet another choice, which is to specify that the `a` field of any `Bag` is always supposed to be non-null. To do this, we annotate the declaration of `a` with a `non_null` pragma:

```
class Bag {
  /*@ non_null */ int[] a;
  ...
```

This causes ESC/Java to assume that the `a` field of any `Bag` object is itself non-null (and thus can safely be dereferenced). Conversely, it causes ESC/Java to issue a warning for any assignment to the `a` field of a `Bag`, if it cannot confirm that the expression being assigned will have a non-null value at run time. Furthermore ESC/Java will check that every `Bag` constructor initializes the `a` field of the constructed object to a non-null value.

0.4 Scene 4: We correct a bug in `Bag.extractMin`.

We now consider the third warning:

```
Bag.java:15: Warning: Array index possibly too large (IndexTooBig)
      if (a[i] < m) {
           ^
```

Examining the program, we now find a genuine bug. The `for` loop starting on line 15 (in the original program) examines array elements `a[1]` through `a[n]`, but array indexing in Java is zero based. We correct the line to read

```
      for (int i = 0; i < n; i++) {
```

0.5 Scene 5: We take no action for a redundant warning.

The fourth warning

```
Bag.java:21: Warning: Possible null dereference (Null)
    a[mindex] = a[n];
                ^
```

requires no action, as the `non_null` pragma we added in section 0.3 already prevents `a` from being `null`. In other words, the second and fourth warnings complain about the same problem.

> **Remark:** ESC/Java often avoids issuing such redundant warnings. Note, for example, that it doesn't complain about the expression `a[m]` on (original) line 17, or the expression `a[mindex]` on the left hand side of the assignment on (original) line 21. However, it does not avoid them in all cases. A detailed description of the circumstances in which ESC/Java will or will not issue multiple warnings with the same underlying cause is beyond the scope of this manual.

0.6 Scene 6: We supply a precondition for `Bag.extractMin`.

We now consider the last of the five warnings:

```
Bag.java:21: Warning: Possible negative array index (IndexNegative)
    a[mindex] = a[n];
                ^
```

The problem is that the code in (original) lines 20-21 removes an element from the bag even when the bag is already empty (that is, when `this.n == 0` on entry to `extractMin`). ESC/Java has called our attention to the need for another design decision: do we add special code to handle the situation when `extractMin` is called on an empty bag, or do we add a precondition forbidding such calls? Let's try the latter course:

```
12:    //@ requires n >= 1;
13:    int extractMin() {
```

0.7 Scene 7: We run ESC/Java again and it still issues a warning.

With all the changes described above, our example program now reads:

```
 1: class Bag {
 2:   /*@ non_null */ int[] a;
 3:   int n;
 4:
 5:   //@ requires input != null;
 6:   Bag(int[] input) {
 7:     n = input.length;
 8:     a = new int[n];
 9:     System.arraycopy(input, 0, a, 0, n);
10:    }
11:
12:   //@ requires n >= 1;
12:   int extractMin() {
13:     int m = Integer.MAX_VALUE;
14:     int mindex = 0;
15:     for (int i = 0; i < n; i++) {
```

```
16:       if (a[i] < m) {
17:         mindex = i;
18:         m = a[i];
19:       }
20:     }
21:     n--;
22:     a[mindex] = a[n];
23:     return m;
24:   }
25: }
```

We now run ESC/Java again and it produces the following warning:

```
Bag.java:17: Warning: Array index possibly too large (IndexTooBig)
      if (a[i] < m) {
           ^
```

0.8 Scene 8: We supply an `invariant` pragma relating `n` to `a.lenght`.

It may appear that ESC/Java is still complaining about the bug we thought we'd fixed in section 0.4, but further study reveals a different problem: ESC/Java has no reason to expect that `n`, which we intend to be the length of the meaningful prefix of `a`, will in fact be at most `a.length`. We can express this intention with an `invariant` pragma:

```
1: class Bag {
2:   /*@ non_null */ int[] a;
3:   int n;
4:   //@ invariant 0 <= n & n <= a.lenght;
...
```

Roughly speaking, ESC/Java treats an invariant as an implicit postcondition of every constructor and as both a precondition and postcondition of every method. The semantics of `invariant` pragmas--and all other ESC/Java pragmas--are described in greater detail in section 2 below. The full rules about expressions that can occur in pragmas (called *specification expressions*) are given in section 3. For now, we remark that since specification expressions are not actually executed, the unconditional logical operators `&` and `|` are interchangeable (except for binding power) in specification expressions with the respective conditional operators `&&` and `||`.

0.9 Scene 9: ESC/Java notices a typographical error.

When we run ESC/Java again, the result is:

```
Bag.java:4: Error: No such field in type int[]
  //@ invariant 0 <= n & n <= a.lenght;
                                ^
Caution: Turning off extended static checking due to type error(s)
1 caution
1 error
```

0.10 Scene 10: Our efforts come to a happy conclusion.

Whoops! Our `invariant` pragma had a spelling error. We correct it to read

```
4:   //@ invariant 0 <= n & n <= a.length;
```

and try again. This time ESC/Java runs without reporting any further complaints.

The file *escjavaRoot*`/examples/0.10/Bag.java` contains a copy of the final version of the `Bag` class.

---

1 An overview of ESC/Java and of this manual

In this section we summarize the principal features of ESC/Java. While some of the things we say here reiterate points made in our example of section 0, we also describe a number of features of ESC/Java that are not discussed at all in section 0. Moreover, we address a number of points that were glossed over in section 0, and that possibly raised questions in the mind of the perceptive reader. Throughout this section we refer the reader to later parts of this manual where various topics are discussed in more detail.

- **ESC/Java is a checker for Java programs, optionally annotated with user-supplied *pragmas*.**
  - **ESC/Java pragmas must occur within *pragma-containing comments*.**
    We showed some examples of pragma-containing comments in section 0. Section 2 includes all the rules about where the various kinds of pragmas are allowed.
  - **ESC/Java pragmas can contain expressions that are similar to Java expressions.**
    The ESC/Java *specification language*--that is, the language of ESC/Java pragmas--includes expressions, which we call *specification expressions*, or *SpecExpr*'s. While the syntax, name-resolution, and type-checking rules for specification expressions are similar to those for ordinary Java expressions, there are inevitably some differences. The rules for specification expressions are described in section 3 of this manual.
  - **ESC/Java pragmas record programmer design decisions.**
    In addition to giving the user control over ESC/Java, pragmas serve to record formally some of the programmer's intentions about the function and use of methods, constructors, and variables (for example, that the `Bag` constructor of our example in section 0 was meant never to be called with a `null` argument). These are the same sorts of intentions that good programmers may already record informally in natural language comments.
  - **ESC/Java's pragmas support *modular* checking.**
    The checking done by ESC/Java is *modular*. When checking the body of a *routine* (that is, a method or constructor) `r`, ESC/Java does not examine bodies of routines called by `r`. Rather, it relies on the *specifications* of those routines, as expressed by `requires` pragmas, by other pragmas described below, and by Java constructs such as signatures, return types, and `throws` clauses. Conversely, to check the body of `r`, ESC/Java does not examine callers of `r`. It does, however, assume that `r` is called only in accordance with its own specification.
  - **When Java sources are unavailable, users can supply pragmas in `.spec` files.**
    When the file declaring a class `T` uses a type `U`, ESC/Java may need data invariants of `U` and/or specifications of `U`'s routines in order to check the routines of `T`. If ESC/Java can find only a binary (`.class`) file and no source file declaring `U`, then it will assume simple default specifications for the routines of `U` based on their signatures. The user can supply additional specifications for the routines of `U`, and also invariants for `U`, by putting them in pragmas in a file `U.spec`. A `.spec` file is like a

`.java` file except that it is allowed to omit method bodies. Section 5.1 tells more about `.spec` files.

Section 2 includes descriptions of all ESC/Java pragmas.

- **ESC/Java is checker for (almost all of) Java 1.2.**
  ESC/Java is targeted for Java 1.2, as described in *The Java Specification Language*, by James Gosling, Bill Joy, and Guy Steele [*JLS*], supplemented by the ``Inner Classes Specification" [*ICS*], except for some limitations described in section 6 of this manual.

- **ESC/Java has a command-line interface like the Java compiler's.**
  An invocation of ESC/Java has the form:

  `escjava` [*options*] *sourcefiles*

  The `escjava(1)` man page in the ESC/Java release (see appendix B) includes descriptions of ESC/Java's command-line options and of environment variables that affect ESC/Java's operation. For now we mention only the `CLASSPATH` environment variable, whose effect on ESC/Java is similar to its effect on `javac(5)` and the `-suggest` command-line option, which causes ESC/Java to offer suggestions of pragmas that might eliminate certain kinds of warnings. Section 5 contains further description of these features (but not all command-line options and environment variables).

- **ESC/Java issues *warning* messages for potential run-time errors.**
  - **The current ESC/Java checks only method and constructor bodies.**
    The current ESC/Java provides no warnings for potential run-time errors in static initializers [*JLS*, 8.5] or in initializers for `static` fields or in anonymous classes.
  - **ESC/Java treats exceptions thrown by the Java run-time system as run-time errors.**
    Some of the potential ``error" conditions detected by ESC/Java are conditions that would be detected by the Java run-time system and give rise to exceptions (specifically, `NullPointerException`, `IndexOutOfBoundsException`, `ClassCastException`, `ArrayStoreException`, `ArithmeticException`, and `NegativeArraySizeException` [*JLS*, 11.5.1.1]). The current ESC/Java treats these conditions as errors, and generates warnings for them even if the program actually catches the resulting exceptions. Accordingly, our use of the word ``error" in this manual includes such conditions. Future versions of ESC/Java may provide support for checking programs that catch exceptions thrown by the Java run-time system. The current ESC/Java version does support checking of programs that catch explicitly-thrown exceptions (including those just listed).
  - **ESC/Java does not warn of potential errors in the evaluation of specification expressions.**
    Specification expressions are never actually evaluated, and (with one exception described in section 2.6.1) ESC/Java will not produce specific warnings about specifications expressions whose evaluation might dereference `null`, access arrays out of bounds, etc. Rather, the meanings of such expressions (for example, *`O`*`.f` where *`O`* , if it were actually evaluated, might evaluate to `null`) are an unspecified function of (the meanings of) their subexpressions. In most cases, attempts to prove things about such unspecified values will fail, thus giving rise to warnings of some sort (though alas not the clearest warnings that one could hope for).
  - **ESC/Java warning messages may include *execution traces*.**
    Associated with each ESC/Java warning message is some execution path that--so far as ESC/Java can determine--may plausibly lead to the run-time error mentioned in the warning. If certain kinds of ``interesting" events occur on this execution path, the message will contain an *execution trace*

listing those events. See section 4.0 for details about which events are considered ``interesting''. Section 4 of this manual includes descriptions of all ESC/Java warning messages.

- **ESC/Java issues *error* messages for badly formed programs**.
  Before ESC/Java can analyze a program for potential run-time errors, it must first perform operations such as parsing, name resolution, and type-checking both of the Java code and of any pragmas. When ESC/Java detects an illegal construct (such as the syntactically incorrect pragma shown in section 0.8 above) during this preliminary processing, it issues an *error* message. Error messages are distinguished from warning messages by the occurrence of the word `Error` instead of `Warning` near the beginning of the message. Only when its input is free of such errors can ESC/Java go on to look for potential run-time errors and generate warnings (just as a compiler generates object code only when the source code is free of compile-time errors).
  - **ESC/Java error messages are like compiler error messages.**
    ESC/Java error messages are similar to compiler error messages, and we hope they will be self-explanatory. Thus, they are not fully enumerated or described in this manual. We believe that all ESC/Java error messages arise either (1) from circumstances that would cause the Java compilers to report compile-time errors or (2) from violations of the rules for writing pragmas as given in section 2 and section 3.
  - **ESC/Java doesn't detect all compile-time errors that Java compilers detect.**
    A number of conditions that give rise to Java compile-time errors are not detected by ESC/Java. For example, the current ESC/Java does not enforce Java's definite assignment rules [*JLS*, 16] or all of Java's restrictions on access to `protected` variables [*JLS*, 6.6.2]. Thus it is wise to run your code through a Java compiler at least once before trying to run it through ESC/Java. (**Tip:** Sometimes the nature of a syntax error in your program may not be immediately clear from an ESC/Java error message. In such cases, a compiler may detect the same error and offer a better--or at least different--description of the problem.)
- **ESC/Java's extended static checking isn't perfect.**
  - **ESC/Java can miss errors.**
    When ESC/Java processes a program and produces no warnings, it is not necessarily true that the program is free of all errors. For example, ESC/Java never checks for some kinds of errors, such as arithmetic overflow, or infinite looping. Also ESC/Java doesn't check programs for functional correctness properties other than those given by the user in pragmas. Finally, even when ESC/Java checks for a particular kind of errors, there may be situations in which it erroneously concludes that the error cannot occur, and therefore fails to issue a legitimate warnings. In the jargon of proof theory, we say that ESC/Java--viewed as a system for proving program correctness--is *unsound*. Section C.0 describes the known sources of unsoundness in ESC/Java.
  - **ESC/Java can give spurious warnings.**
    Conversely, when ESC/Java issues a warning, it doesn't necessarily indicate the presence of an error; it merely means that ESC/Java is unable to conclude that the indicated error will never occur, given the annotations that the user has supplied. In the jargon, ESC/Java--viewed as a system for proving program correctness--is *incomplete*. Section C.1 describes the main sources of incompleteness in ESC/Java.
  - **Pragmas give the user some control over ESC/Java's unsoundness and incompleteness.**
    ESC/Java provides pragmas that let the user eliminate spurious warnings--that is, reduce ESC/Java's incompleteness--either without loss of soundness (as, for example, the `requires` pragma we wrote in section 0.2 eliminated the warning about a potential dereference of `null` in the `Bag` constructor by imposing a precondition on calls) or, if need be, at some risk of lost soundness (as in the case of the `nowarn`, `assume`, and `axiom` pragmas respectively described in sections

2.1.0, 2.1.2, and 2.4.2 below).

- **ESC/Java issues *caution* messages in some (but not all) cases where unsound checking may occur.**

---

2 ESC/Java pragmas

In this section we describe all the kinds of ESC/Java pragmas, the places where they can occur, and (informally) what they mean. We begin by giving some general information about where pragmas can occur, and then go on to describe the individual pragmas.

Many pragmas can contain expressions--called specification expressions--which are similar to Java expressions, but with a few constructs legal in Java expressions being forbidden and a number of added constructs being permitted. We mention some of the added constructs in connection with pragmas where they are of use, but defer a detailed description of specification expressions to section 3.

Most pragmas are ways of asking ESC/Java to produce warnings if certain user expectations about the behavior of the annotated program may be wrong. We will often say that ESC/Java ``checks" that some condition *x* holds at a particular control point when it would be more precise to say that ESC/Java issues a warning message if it cannot prove that *x* holds at that point. Keep in mind that in addition to the possibility that *x* might fail, the warning may also be issued because the annotations are inadequate to imply that *x* holds, because the theorem-prover's deductive power is inadequate to complete the proof, or because ESC/Java's model of Java semantics is incomplete.

2.0 Rules about where pragmas may occur

2.0.0 ESC/Java pragmas must occur within *pragma-containing comments.*

ESC/Java looks for pragmas within certain specially formatted comments. Specifically:

- When the character `@` is the first character after the initial `//` or `/*` of a Java comment, ESC/Java expects the rest of the comment's body to consist entirely of a sequence of (zero or more) ESC/Java pragmas.
- Inside a documentation comment [*JLS*, 18], a sequence ESC/Java pragmas can be bracketed by `<esc>` and `</esc>`.

Many pragmas end with an optional semi-colon ($;_{opt}$). If such a pragma is followed by another pragma in the same pragma-containing comment, then this semi-colon is required.

2.0.1 There are four *syntactic categories* of pragmas.

Pragmas are divided into four *syntactic categories* according to the places in a program where they can sensibly be used. All pragmas in any single pragma-containing comment must be of the same syntactic category.

- *Lexical pragmas* may occur anywhere that ordinary Java comments may occur.
  [It would be more accurate to say that ESC/Java allows the occurrence of a pragma-containing comment whose body is a sequence of zero or more lexical pragmas at any point where Java allows a comment; we henceforth take the liberty to eschew this degree of pedantry.]

The current ESC/Java includes only one kind of lexical pragma:
    - `nowarn` pragma (section 2.1.0).
- *Declaration pragmas*--such as, for example, the `invariant` pragma in section 0.8 in our introductory example--are analogous to Java declarations, and may occur only where declarations of class members or interface members may occur. The current ESC/Java includes the following kinds of declaration pragmas:
    - `invariant` pragma (section 2.4.1)
    - `axiom` pragma (2.4.2)
    - `ghost` pragma (2.6.0)
- *Statement pragmas* are analogous to Java statements. They may occur only where Java statements may occur. The current ESC/Java includes the following statement pragmas:
    - `assert` pragma (2.1.1)
    - `assume` pragma (2.1.2)
    - `unreachable` pragma (2.1.3)
    - `loop_invariant` pragma (2.4.3)
    - `set` pragma (2.6.1)
- *Modifier pragmas* are analogous to Java modifiers such as `private` and `final`. Generally, modifier pragmas are allowed in the same positions as Java modifiers, but they are also allowed in a few other places, as described in the fine points below. Some modifier pragmas (for example, the `non_null` pragma in section 0.3 in the introductory example) modify variable declarations; other modifier pragmas (for example, the `requires` pragmas in sections 0.2 and 0.6 in the introductory example) modify declarations of *routines* (methods or constructors). The current ESC/Java includes the following modifier pragmas for variable declarations:
    - `non_null` pragma (2.4.0)
    - `spec_public` pragma (2.5.0)
    - `readable_if` pragma (2.5.1)
    - `uninitialized` pragma (2.5.2)
    - `monitored_by` pragma (2.7.0)
    - `monitored` pragma (2.7.1)

  The current ESC/Java includes the following modifier pragmas for routine declarations:
    - `requires` pragma (2.3.0)
    - `modifies` pragma (2.3.1)
    - `ensures` pragma (2.3.2)
    - `exsures` pragma (2.3.4)
    - `also_ensures` pragma (2.3.5)
    - `also_exsures` pragma (2.3.6)
    - `also_modifies` pragma (2.3.8)

  In the current ESC/Java there are no pragmas that can modify both variable and routine declarations.

Fine points

A modifier pragma that modifies a routine (method or constructor) declaration may appear either in any of the following places:

- near the beginning of the declaration, in the usual place where Java modifiers such as `private` or `final` may occur,
- just before the opening left brace of the routine's body, or
- just before the final semicolon of a routine declaration if there is no body (as in an interface, in an `abstract` class, or sometimes in a `.spec` file (see section 5.1.2)

The semantics of a routine modifier pragma is independent of whether the pragma appears lexically before or after the signature of the routine. Routine modifier pragmas are described further in section 2.3.

A modifier pragma that modifies a variable declaration may appear either in the usual place for modifiers near the beginning of the declaration or just before the final semicolon. The pragma applies to all variables declared in the declaration, and its semantics is independent of its position within the declaration. For convenient reference, here is a table listing ESC/Java pragmas that can modify variable declarations, the sections of this manual where they are described, and the kinds of declarations they are allowed to modify:

	instance variable	static field	local variable	formal parameter
`non_null` (2.4.0)	yes	yes	yes	yes
`spec_public` (2.5.0)	yes	yes	no	no
`readable_if` (2.5.1)	yes	yes	yes	no
`uninitialized` (2.5.2)	no	no	yes	no
`monitored_by` (2.7.0)	yes	yes	no	no
`monitored` (2.7.1)	yes	no	no	no

Note that ESC/Java sometimes allows modifier pragmas in declarations of local variables (including those declared in the *ForInit* of a `for` statement [*JLS*, 14.12]) and formal parameters.

Particular pragmas may have further restrictions on where they may occur beyond those given above. These restrictions are described in the sections describing the respective pragmas.

2.0.2 ESC/Java sometimes allows pragmas to contain nested pragmas or comments.

Situations sometimes arise where it is convenient to nest a comment inside a pragma, or where it is convenient--or even necessary--to nest a pragma inside another pragma. In particular, this is the only way to annotate a ghost field (see section 2.6.0) with a modifier pragma. We describe here the rules governing such nesting. [*Readers may wish to skip the remainder of this section on first reading of the manual, or until occasion for using such nesting presents itself.*]

In the current ESC/Java, a pragma-containing comment (call it ***`inner`***) may be nested inside pragma-containing comment (***`outer`***) only in the following cases:

(1) each pragma in ***`inner`*** is a lexical (`nowarn`) pragma (see section 2.1,0), or

(2) each pragma in ***`inner`*** is a modifier pragma for a ghost variable declared in ***`outer`*** (see section 2.6.0 for further details).

Furthermore, ***`outer`*** must not itself be nested inside another pragma-containing comment. (However, ***`outer`*** may be of the form `<esc>...</esc>`, which must of necessity occur inside a Java documentation comment).

Any pragma-containing comment ***`outer`***--even a nested one--may have an ordinary (i.e., non-pragma-containing) comment ***`inner`*** nested inside it.

Also, there are some restrictions concerning the syntactic forms of a nested (ordinary or pragma-containing) comment *inner* and the enclosing pragma-containing comment *outer*, as given in the following table and the associated notes. These restrictions ensure that the portions of the input file that ESC/Java regards as comments or pragmas will be precisely those regarded as comments by Java compilers. Each row of the table corresponds to a syntactic form of *outer* and each column corresponds to a syntactic form of *inner*. Entries tell whether each specific form of nesting is allowed and reference notes giving any additional restrictions on allowed forms of nesting and explanations of why forbidden forms of nesting are forbidden.

outer \ *inner*	//...	//@...	/*...*/	/*@...*/	/**...*/
//@...*inner*...	ok(1)	ok(1)	ok(2)	ok(2)	no(3)
/*@...*inner*...*/	ok(4)	ok(4)	no(5)	no(5)	no(5)
<esc>...*inner*...</esc>	ok(6)	ok(6)	no(5)	no(5)	no(5)

Notes:

(1) ESC/Java considers *inner* to extend to the end of the line on which it begins.
(2) *inner* must be entirely on one line (because Java defines *outer* to end at the end of the line).
(3) The current ESC/Java might accept this form of nesting, but it is strongly deprecated. There is no good reason to use it, and `javadoc(5)` will not recognize *inner* as a documentation comment.
(4) ESC/Java considers `inner` to extend either to the end of the line or up to the closing `*/` of *outer*, whichever is earlier.
(5) These forms of nesting are forbidden because Java does not allow comments of the form `/*...*/` to be nested inside each other. Thus Java compilers would interpret the closing `*/` of *inner* as ending *outer*.
(6) ESC/Java considers `inner` to extend either to the end of the line or up to the closing `</esc>` of *outer*, whichever is earlier.

Examples

Here are some examples of legal nesting of comments and pragmas inside pragmas:

- `//@ ghost public /*@ non_null // comment */ Object o;`

- ```
  /*@ requires a > 0;                 // comment 1
      requires b > 0;  //@ nowarn Pre // comment 2
      requires c > 0;                 // comment 3 */
    void m(int a, int b, int c) { …
  ```

Here are some examples of illegal nesting:

- `/*@ ghost public /*@ non_null */ Object o; */`
- ```
  //@ requires a > 0;  /* multi-line "nested"
                           comment */
  ```

Tip: ESC/Java's warning messages for these two examples of illegal nesting fail to give the line number on which

the error occurs (but do indicate that the problem is an "unterminated comment"). You can localize the error by running a Java compiler on your program (which is generally a good thing to do before running ESC/Java, anyway).

2.1 The most basic pragmas

In this section we describe the simplest ESC/Java pragmas: `nowarn`, `assume`, `assert`, and `unreachable`. The `nowarn` pragma is essentially a blunt instrument for getting ESC/Java to shut up about uninteresting warnings, thus helping to prevent ESC/Java's known imperfections and limitations from becoming major sources of user annoyance. The `assert` and `assume` pragmas are the fundamental pragmas of which most others are simply more elaborate forms (see section 2.2.1). They may also be useful in their own right (see particularly section 2.2.3).

2.1.0 `nowarn` pragma

A `nowarn` pragma is a lexical pragma. It has the form:

`nowarn` *L* `;`$_{opt}$

where *L* is a (possibly empty) comma-separated list of warning types from the following list:

```
ArrayStore      Invariant       Post
Assert          LoopInv         Pre
Cast            NegSize         Race
Deadlock        NonNull         Reachable
Exception       NonNullInit     Unreadable
IndexNegative   Null            Uninit
IndexTooBig     OwnerNull       ZeroDiv
```

The pragma suppresses any warning messages of the types in *L* that are associated with the line on which the pragma appears. If *L* is empty, all warning types are suppressed. See section 4 for descriptions of the different types of warnings.

Fine points

Some ESC/Java warning messages refer to two source code locations, namely (1) a location indicating the control point where an error could potentially occur at run-time, and (2) the location of a pragma (or, occasionally, a Java declaration) associated with the warning. In such cases the warning can be suppressed by a `nowarn` pragma on either of the indicated source lines.

The `nowarn` pragma is potentially unsound, and should be used only in cases where the programmer is willing to take responsibility that the suppressed warnings are really false alarms. The primary intended use of `nowarn` pragmas is where the suppressed warnings concern situations that are impossible in practice, but for reasons beyond ESC/Java's ability to discover. Another use would be to suppress warnings for circumstances that are actually harmless (and where the programmer is willing to take responsibility that they are harmless). For example, a `nowarn` pragma might be used to suppress a warning for a null dereference if the resulting exception would be caught by a handler (but in such a case the current ESC/Java will not check that there actually is a handler, nor will it check for any errors that might occur during or after execution of the handler).

The `nowarn` pragma suppresses warnings on a line-by-line basis. ESC/Java also provide command-line options that enable and disable checking at a much coarser grain (see the descriptions of the `-nowarn`, `-warn`, `-start`, `-routine`, and `-routineIndirect` options on the `escjava(1)` man page).

Unlike warning messages, ESC/Java error messages cannot be suppressed by `nowarn` pragmas. Error messages report conditions that prevent ESC/Java from making enough sense of the program to do further checking.

Bug: The current ESC/Java does not allow another pragma to follow a `nowarn` pragma in the same pragma-containing comment. Since the only lexical pragma in the current ESC/Java is `nowarn`, and since you can say with one `nowarn` pragma anything that you can say with two, this should not cause problems in practice.

2.1.1 `assert` pragma

An `assert` pragma is a statement pragma. It has the form

`assert` *E* `;`$_{opt}$

where *E* is boolean specification expression. The pragma causes ESC/Java to issue a warning if it cannot establish that *E* is true whenever control reaches the pragma.

2.1.2 `assume` pragma

An `assume` pragma is a statement pragma. It has the form

`assume` *E* `;`$_{opt}$

where *E* is boolean specification expression. The pragma causes ESC/Java to assume that *E* is true whenever control reaches the pragma. In other words, for any execution path in which *E* is false when control reaches the pragma, ESC/Java ignores the path from that point on.

Example

The usual purpose of an `assume` pragma is to supply ESC/Java with some piece of information that is incapable of deducing on its own, thereby preventing ESC/Java from generating spurious warnings. In the code fragment

```
22:       // start complicated computation guaranteed to leave i != 0
...      ...
146:      // end of complicated computation
147:      //@ assume i != 0;
148:      if (b) {
149:        g = h/i;
150:      } else {
151:        h = g/i + g/j;
152:      }
```

the `assume` pragma at line 147 prevents ESC/Java from warning that the division by `i` in lines 149 and 151 may

give rise to an `ArithmeticException`, but ESC/Java will still generate a warning about the division by `j` in line 151 unless it can deduce that `j` will never be zero when control reaches that point.

Fine points

Like the `nowarn` pragma, the `assume` pragma is potentially unsound, and should be used only if the programmer is willing to take responsibility that *E* holds whenever control reaches the pragma (or at least is willing to give up further checking for any execution paths on which *E* is false). When faced with the choice of using either an `assume` pragma or a `nowarn` pragma to suppress a spurious warning, it is preferable to use an `assume` pragma (if it is practical to do so), since the `assume` pragma more explicitly documents your assumptions about the behavior of the program.

The sentence ``In other words, for any execution path ... from that point on." may have seemed unclear to some readers. For some more examples of its meaning, consider the following code fragment:

```
31:     if (u != null | v != null) e = 10;
32:     if (v == null) {
33:       w = a.f;
34:       //@ assert b != null;
35:       //@ assume a != null & b != null & c != null & d != null & e > 0;
36:       x = c.f;
37:       //@ assert d != null;
38:     } else {
39:       c = new …;
40:       d = v;
41:     }
42:     y = a.g;
43:     //@ assert b != null;
44:     z = c.g;
45:     //@ assert d != null;
46:     //@ assert e > 0;
```

When ESC/Java checks this code:

- ESC/Java will not warn of a possible dereference of `null` at line 36 or of a possible assertion failure at line 37, because the only way control can reach those lines is by first reaching the `assume` pragma at line 35.
- The `assume` pragma at line 35 will not prevent ESC/Java from issuing warnings about line 33 and/or 34, because control can reach those lines before reaching line 35.
- ESC/Java might issue warnings about line 42 and/or 43, because control might reach those lines by some path that does not first reach line 35. This can happen if control can reach line 32 with `a` or `b` being `null` and `x` being non-null.
- ESC/Java will not warn of a possible dereference of `null` at line 44 or of a possible assertion failure at line 45 or 46. Every execution path that reaches line 44 first either reaches line 35 (in which case ESC/Java considers further execution of that path only for cases where the expression in the `assume` pragma would evaluate to `true`) or reaches lines 39 and 40 (in which case the `c` and `d` are assigned non-null values, and `e` must already have been assigned a nonnegative value at line 31).

For a concise formal description of the semantics of `assume`, see [LSS99].

2.1.3 `unreachable` pragma

An `unreachable` pragma is a statement pragma. It has the form

```
unreachable ;opt
```

The pragma is semantically equivalent to

```
assert false;
```

except for giving rise to a different warning message.

2.2 Some remarks concerning `assert` and `assume`

[*Readers anxious to ``cut to the chase'' may skip this section on first reading. Others, however, may find that this material aids their intuition about how ESC/Java works.*]

The `assert` and `assume` pragmas introduced in section 2.1 are in some sense the most basic ESC/Java pragmas. We make some remarks about them here before going on to describe the rest.

2.2.0 The `assume` pragma should be used with judgment.

An `assume` pragma resembles an `assert` pragma in that each states a condition that the programmer believes to hold whenever control reaches a certain point in the program. The difference is that ESC/Java checks--that is, issues a warning if it cannot establish--the condition (called an *asserted condition* or *assertion*) in an `assert` pragma, but ESC/Java takes the *assumed condition* (or *assumption*) in an `assume` pragma for granted. That is, the programmer takes responsibility that assumed conditions will hold.

Put another way, the `assume` pragma allows the programmer to trade spurious warnings (incompleteness) for possible missed warnings (unsoundness). Thus, you should use `assume` pragmas with care, lest by supplying an incorrect assumption you suppress warnings of genuine errors. For example, ESC/Java will not warn of any error that might occur downstream in the execution path from the line:

```
x = null; //@ assume x != null;
```

since there can be no execution path where the assumption holds after the assignment is performed.

On the other hand, there will be cases where `assume` pragmas (or `nowarn` pragmas, which call for similar caution in their use) will be the only practical means to eliminate spurious warnings. Also, while it is sometimes possible to eliminate spurious warnings by means that don't carry the same risks of missed warnings, some judgment must be exercised as to whether the improved assurance of correctness will be worth the increased effort in any particular case.

2.2.1 Most ESC/Java pragmas are just fancy forms of `assert` and `assume`.

The ESC/Java annotation language includes over twenty different kinds of pragmas, but to a first approximation they are mainly more or less elaborate ways of adding assertions (i.e., claims about the program state that are

checked by the ESC/Java) and/or assumptions (i.e., claims that are taken for granted by ESC/Java) at different points in the program. Indeed one can also think of the checking that ESC/Java does in the absence of any user annotations as simply checking implicit assertions before each pointer dereference, array access, etc., so that, for example, the program fragment

```
x = a[i];
```

is treated as if it were

```
//@ assert a != null; assert 0 <= i; assert i < a.length;
x = a[i];
```

There are several valuable differences between the explicit introduction of assertions through the use of `assert` pragmas and their implicit introduction through other ESC/Java pragmas and through built-in checking rules:

- Warnings arising from `assert` pragmas all produce the same kind of warning message:

  ```
  ...: Warning: Possible assertion failure (Assert)
  ```

  Implicitly introduced assertions, on the other hand, give rise to a wide variety of more specific messages.
- An `assert` pragma can occur only where a Java statement can occur, but implicitly introduced assertions are not limited to statement boundaries. For example, given the statement

  ```
  p = m(q) + q[++i]
  ```

  ESC/Java makes sure that the implicit assertion that array `q` is non-null is introduced at a control point after the call to method `m`, and that the implicit assertion that index `i` is in bounds is introduced at a control point after `i` is incremented. For the user to introduce explicit assertions at these points, it would be necessary to modify the Java source code, breaking the statement into several parts and introducing a temporary variable for the result of the call to `m`.
- An `assert` pragma introduces a single assertion at the control point where it occurs. In contrast, a single other pragma (or built-in checking rule) can introduce many assertions throughout the program--for example, at every call of a given method, or at every access to a given variable.
- The systematic introduction of assertions at certain points in a program sometimes makes it possible for ESC/Java safely to introduce assumption at other points.

In short, the various pragmas (and built-in checking rules) provided by ESC/Java enable the introduction of collections of assertions and assumptions in a manner that would be quite tedious, awkward, and error-prone to accomplish with explicit `assert` and `assume` pragmas alone.

2.2.2 A `nowarn` pragma suppresses warnings by turning assertions into assumptions.

Recall from the previous section (2.2.1) that ESC/Java's built-in checking for null dereferences works, in effect, by implicitly putting an assertion before each pointer dereference in the program, so that the line

```
209:      z = x.f;
```

gets treated as if it were

```
209:      /*@ assert x != null; */ z = x.f;
```

except that the precise text of the warning message, if any, is different. The way that a `nowarn` pragma suppresses warnings is by turning the (implicit or explicit) assertions that would generate the warnings into assumptions. Thus, for example, the code fragment

```
209:      z = x.f; //@ nowarn Null;
210:      w = x.g;
```

is treated by ESC/Java like

```
209:      /*@ assume x != null; */ z = x.f;
210:      /*@ assert x != null; */ w = x.g;
```

By changing the implicit assertion that `x` is non-null on line 209 into an assumption, the `nowarn` pragma not only prevents ESC/Java from warning of a possible dereference of `null` on line 209, but also satisfies the implicit assertion on line 210, thus preventing ESC/Java from warning that the evaluation of the expression `x.g` might dereference `null`.

It should be clear from the above that the comments in section 2.2.0 about the need for judgment regarding the use of `assume` pragmas are (at least) equally applicable to `nowarn` pragmas.

2.2.3 A helpful tip: Experiments with `assert` and `assume` pragmas can help you understand ESC/Java's behavior and debug your annotated code.

As illustrated in the example of section 0, using ESC/Java will often be an iterative process: You run ESC/Java on your program; it reports some warnings; you address the warnings by changing either the Java code itself or the annotations; you run ESC/Java again; and so on until ESC/Java reports no warnings. At some point in this process, you may find that you can't figure out why ESC/Java is issuing some warning, or why the change you made to address some warning isn't making the warning go away. In such cases experiments with `assert` and `assume` pragmas can be useful in the same way that displaying intermediate results is useful in ordinary debugging.

Suppose, for example, that the problem seems to be that ESC/Java is missing some critical fact that ``should be obvious." You might try adding an `assert` pragma for the (supposedly obvious) fact and see whether ESC/Java really warns that the `assert` could fail. Or you might try adding an `assume` pragma to see whether supplying the (supposedly critical) missing fact really eliminates the warning.

Such experiments can also clarify your own understanding of your program. Consider, for example, the situation described in section 0.7, where ESC/Java continued to complain of a possible array bounds error even after we fixed the bug in the surrounding `for` loop:

ESC/Java input from file Bag.java:

```
      ...
16:     for (int i = 0; i < n; i++) {
17:       if (a[i] < m) {
      ...
```

ESC/Java output:

```
Bag.java:17: Warning: Possible array index too large (IndexTooBig)
      if (a[i] < m) {
           ^
1 warning
```

If we found this behavior puzzling we might consider the experiment of adding an `assert` pragma between lines 16 and 17. The outcome of the experiment would depend on which of the two ``obvious'' assertions we chose:

```
//@ assert i < n;
```

or

```
//@ assert i < a.length;
```

In either case, observing the outcome might take us a step closer to understanding the situation.

Note: Some earlier versions of ESC/Java had a bug that sometimes resulted in highly counterintuitive failure to warn of the first potential error on an execution path. For example, if a program contained the lines

```
23:     //@ assert x < 10 & y < 10;
...     ...   // statements not modifying x
33:     //@ assert x < 10;
```

then ESC/Java might have produced an `Assert` warning for line 33, but not for line 23 (even though the assertion at line 33 could be false after execution of lines 24-32 only if the assertion at lines 23 had been false beforehand, and even though ESC/Java would have warned about line 23 if the assertion in line 33 were not present). This behavior could be quite confusing for a user attempting to use `assert` pragmas as a debugging aid, as suggested in this section. We have since made changes to ESC/Java to prevent such behavior, or at least greatly reduce its likelihood. If you observe a case where addition of an assertion inhibits ESC/Java from warning about a potential error earlier in the execution path, we would like to know about it. See the instructions on reporting bugs in appendix B.

2.3 Pragmas for specifying routines

In this section, we describe those pragmas, called *routine modifier pragmas*, that explicitly supply specifications for individual routines.

Fine points

The reader should be aware that these pragmas are not the only ones that give rise to routine specifications. In this regard, we direct the reader's attention particularly to the descriptions of the `non_null` and `invariant` pragmas in sections 2.4.0 and 2.4.1, as well as to the description of the `axiom` pragma in section 2.4.2.

All routine modifier pragmas have a number of properties in common:

ESC/Java allows modifiers for a routine declaration to appear not only in the usual place for modifiers near the beginning of the declaration, but also just before the opening left brace of the routine's body, or before the final semicolon if there is no body (as in an interface, `abstract` class, or sometimes in a `.spec` file (see section 5.1.2)). For example, the `requires` pragma that we introduced for the `Bag` constructor in

section 0.2

```
//@ requires input != null;
Bag(int[] input) {
...
```

might equally well have been written after the signature of the constructor

```
Bag(int[] input)
  /*@ requires input != null; */ {
...
```

with identical semantics (except, of course, that warning messages referring to the pragma would indicate a different source file location).

- Regardless of where a routine modifier pragma appears, the parameters of the routine are in scope in any specification expression in the pragma.
- In an `ensures` (section 2.3.2) pragma modifying a constructor and in any pragma modifying an instance method, specification expressions may mention `this`, denoting the constructed object or the object whose method is being invoked. As usual in contexts where `this` may occur, it may occur implicitly, a field access `this.f` being written simply as *f* (unless the name *f* is hidden, for example by a parameter name).
- The pragmas in this section specify preconditions, modification targets, and normal and exceptional postconditions of routines. When a method of a class or interface *S* inherits or overrides [*JLS*, 8.4.6] a method *m* from a class or interface *T*, the method *S.m* inherits all the preconditions, modification targets, and postconditions of *T.m*, with the formal parameter names of *S.m* being substituted for those of *T.m*. (ESC/Java desugars implicit references to `this`, as described in the previous bullet, before doing the formal parameter substitution, so nothing funny happens if only one of the two formal parameter lists includes a name conflicting with a field of `this`.) This treatment of inheritance sometimes leads to unsound checking in the presence of multiple inheritance (see section C.0.5).
- When checking code that contains a call to a routine, ESC/Java interprets the routine's preconditions, modification targets, and postconditions with the actual parameters values substituted for the formal parameter names, and in the case of an instance method call *E*`.`*m*`(...)`, with the value of *E* substituted for `this` (including, of course, implicit occurrences of `this`). In cases where evaluation of the actual parameters may have side-effects or may raise exceptions, ESC/Java does the right thing: It checks the preconditions of the routine for the program state after the side-effects, and only for cases in which evaluation of all parameters would terminate normally.
- When checking code that contains a method call *E*`.`*m*`(...)`, ESC/Java determines the specification (preconditions, modification targets, and postconditions) of *m* based on the static type *T* of *E*, even if *E* can be proven always to be of some subtype *S* of *T*. (To get ESC/Java to use additional specifications of *S*`.`*m* beyond those inherited from *T*`.`*m*, you could rewrite the Java code to cast *E* to type *S* before invoking *m*.)

2.3.0 `requires` pragma

A `requires` pragma is a routine modifier pragma. It has the form

`requires` *E* `;`$_{opt}$

where *E* is a boolean specification expression. The pragma makes *E* a precondition of the routine the pragma modifies. When checking the body of the routine, ESC/Java assumes that *E* holds initially. When checking a call to the routine, ESC/Java issues a warning if it cannot establish that *E* holds at the call site.

Fine points

If the routine is `synchronized`, then *E* is assumed to hold before acquisition of the lock. If the routine is a constructor, then *E* is assumed to hold before the implicit superclass constructor call, if any, and thus also before execution of instance variable initializers.

Except for the formal parameters of the routine, the variables mentioned in *E* must be spec-accessible (see section 3.3) anywhere the routine itself is accessible. For example, a precondition of a `public` method may not mention a `private` variable (unless the variable is declared `spec_public`, see section 2.5.0).

A method declaration that overrides another method declaration cannot be modified with a `requires` pragma, but inherits the overridden method's preconditions as described above. Multiple inheritance can lead to unsoundness in some cases, as discussed in section C.0.5.

A single routine declaration may be modified with any number of `requires` pragmas. The effective precondition is the conjunction of all the preconditions given, but any resulting warning message indicates the specific `requires` pragma giving rise to the warning, and warnings arising from each pragma can be suppressed individually.

2.3.1 `modifies` pragma

A `modifies` pragma is a routine modifier pragma. It has the form

`modifies` *L* `;`$_{opt}$

where *L* is a nonempty, comma-separated list of specification designators. A specification designator designates a mutable component of the state. It is very much like a Java *LeftHandSide* [*JLS*, 19.12], but generalized as described below. The pragma specifies that the routine is allowed (but not required) to modify any of the state components listed in *L*.

The state components named in modifies pragmas of a routine are called *modification targets* of the routine. When checking code that calls a routine, ESC/Java assumes that the call modifies only the routine's modification targets (with the usual substitutions) and possibly also any freshly allocated state, regardless of whether the call terminates normally or abruptly. However, the current ESC/Java does not enforce `modifies` pragmas when checking a routine's implementation.

Fine points

Permissible forms of specification designators are:

- *a simple name* *n*. The name must denote a non-`final` field (possibly a ghost field, see section 2.6.0). This form allows modification of `this.`*n* if the routine is an instance method, or of *T*`.`*n* if the routine is a static method of class *T*.
- *a field access* of the form *O*`.`*f*, where *O* is a specification expression of a reference type `T` and *f* denotes one of the fields (possibly a ghost field) of `T`. This form allows modification of *O*`.`*f*. If *f* is a static field, *O* is used only in that its static type disambiguates *f*.
- *an array access* of the form *A*`[`*I*`]`, where *A* is a specification expression of an array type, and *I* is a specification expression of an integral type other than `long`. This form allows modification of *A*`[`*I*`]`.
- *an array range* of the form *A*`[*]`, where *A* is a specification expression of an array type.

This form allows modification of all elements of `A` (but not of `A` itself).

A routine may be annotated with multiple `modifies` pragmas, in which case a call is assumed possibly to modify any state component listed in any of the `modifies` pragmas. If no modification targets are specified for a routine, then ESC/Java will assume that calls to the routine modify only freshly allocated state, if any.

A method declaration that overrides another method declaration cannot be annotated with a `modifies` pragma, but inherits the modification targets of the overridden method. Note that this forbids the overriding method from modifying additional state, but see the description of `also_modifies` below (section 2.3.8).

2.3.1.0 target fields

When a modification target of a routine has the form `E.f` (or simply `f`, meaning `this.f`), the field `f` is said to be a *target field* of the routine. (Note that a modification target of the form `E.g.f` makes `f` but not `g` be a target field).

2.3.2 `ensures` pragma

An `ensures` pragma is a routine modifier pragma. It has the form

```
ensures E ;opt
```

where `E` is a boolean specification expression. The pragma makes `E` a normal (that is, non-exceptional) postcondition of the routine the pragma modifies. When checking the body of the routine, ESC/Java issues a warning if it cannot establish that `E` holds whenever the routine terminates normally. When checking code that calls the routine, ESC/Java assumes that `E` holds just after the call if the call terminates normally.

In a postcondition of a non-`void` method, the special ESC/Java identifier `\result` denotes the result of the method. (For constructors, the constructed object may be denoted only by `this`, not by `\result`.) The static type of `\result` is the result type of the method.

Within `E`, an expression of the form `\fresh(R)` where `R` is a specification expression of a reference type is true if the object denoted by `R` in the post-state is allocated in the post-state (implying that `R != null` in the post-state) and was not allocated in the pre-state. The static type of `\fresh(R)` is boolean.

A postcondition `E` may contain expressions of the form `\old(X)`. Roughly speaking, `\old(X)` means the value of `X` in the pre-state. The static type of `\old(X)` is the same as the static type of `X`. An expression `X` used as an argument of `\old` may not itself contain applications of `\old` or `\fresh`. More precise details are given below.

Fine points

Postconditions of a `synchronized` method apply to the state after the release of the lock.

Except for formal parameters, identifiers used in postconditions of a routine (and not within `\old`) denote their values in the post-state. While Java allows a routine body to include assignment to the routine's formal parameters (thus using the parameters as local variables), such assignments have no effect as seen by the caller, since parameters are passed by value. Therefore ESC/Java interprets occurrences of formal parameters in postconditions as denoting the original (pre-state) actual parameter values.

A single routine declaration may be modified with any number of `ensures` pragmas. The effective postcondition is the conjunction of all the postconditions given, but any resulting warning message indicates the specific `ensures` pragma giving rise to the warning, and warnings arising from each pragma can be suppressed individually.

In a postcondition, an expression of the form `\old(`*X*`)`, where *X* is a specification expression, denotes the value denoted by *X*, except that (1) any occurrence in *X* of a target field (see section 2.3.1.0) of the routine is interpreted according to the pre-state value of the field, and (2) if any modification target of the routine has the form *A*`[`*i*`]` or *A*`[*]`, then *all* array accesses within *X* are interpreted according to the pre-state contents of arrays. Note that in the normal postcondition of a non-`void` method, `\result` always refers to the method's result, even when `\result` occurs in an argument to `\old`. Similarly occurrences of `this` in a normal postcondition of a constructor always refer to the constructed object. See sections 2.3.3and 3.2.15 for further discussion of the semantics of `\old`.

It is a source of potential unsoundness for a postcondition to mention a variable that might not be spec-accessible (section 3.3) to an override of that method, and ESC/Java may forbid such postconditions. In particular ESC/Java forbids postconditions of a method that mention `private` variables except when the routine is `static`, is `final`, is `private`, or is declared in a `final` class, or when the `private` variables mentioned are declared `spec_public` (section 2.5.0). The current ESC/Java doesn't forbid, for example, postconditions of `public` methods from mentioning `package` variables, but future versions of ESC/Java may not be so lenient.

A method declaration that overrides another method declaration cannot be modified with an `ensures` pragma, but inherits the postconditions of the overridden method. (See also the `also_ensures` pragma described in section 2.3.5.)

Since Java guarantees that a constructor call returns a newly allocated object, ESC/Java automatically supplies the postcondition `\fresh(this)`for every constructor.

2.3.3 A note on the interaction of `modifies` and `\old`

[*This section may be skipped on first reading.*]

The current ESC/Java does not check that the body of a routine actually obeys the constraint expressed by the routine's `modifies` pragmas. This lack of checking is one of several potential sources of missed warnings (unsoundness). The potential for missed warnings is mitigated somewhat by a fact that may have seemed surprising when we mentioned it in the previous section (2.3.2): If a particular field (either a static field or an instance variable) is not specified as a target field (section 2.3.1.0) of a routine, then occurrences of that field within arguments to `\old` in the routine's postconditions are taken to refer to post-state values.

Consider, for example a class with an integer field `f` and a method `incf` declared as follows, with no `modifies` pragma:

```
//@ ensures f == \old(f) + 1;
void incf() {
  this.f++;
}
```

Since `f` is not specified as a modification target of `incf`, ESC/Java will interpret both occurrences of `f` in the ensures pragma as referring to the post-state value of `this.f`. Consequently ESC/Java will be unable to show that the method establishes the specified postcondition, and will issue a warning to that effect.

While this warning may seem surprising, the result of interpreting the second occurrence of `f` as the pre-state value of `this.f` would be even worse. Under the latter interpretation ESC/Java would issue no warnings about the body of `incf`, but would assume after a call `x.incf()` both (1) that `x.f` had been incremented in accordance the postcondition), and (2) that `x.f` was left unchanged in accordance with the (unchecked) empty set of modification targets. Since these assumptions are mutually contradictory, the result would be equivalent to assuming `false`, and ESC/Java would silently omit all checking after the call.

As an additional guard against omission of `modifies` pragmas, ESC/Java issues a caution message for any occurrence of `\old(`*`X`*`)` in a postcondition of a method *`m`* unless (1) the expression *`X`* mentions some target field of *`m`*, or (2) the expression *`X`* includes an array access and *`m`* has some modification target of the form `A[`*`I`*`]` or `A[*]`.

Of course, the interactions of `modifies` and `\old` described above do not entirely make up for the fact that the current ESC/Java does no checking of `modifies` pragmas. A method declaration like

```
//@ modifies someOtherObject.f; //instead of this.f
//@ ensures f == \old(f) + 1;
void incf() {
  this.f++;
}
```

can still effectively disable checking of code following calls to incf.

2.3.4 `exsures` pragma

An `exsures` pragma is a routine modifier pragma. It has the form

`exsures (`*`T t`*`) `*`E`*` ;`$_{opt}$

or

`exsures (`*`T`*`) `*`E`*` ;`$_{opt}$

where *`T`* is a subtype of `java.lang.Throwable`, *`t`* (if included) is a an identifier, and *`E`* is a boolean specification expression. The identifier *`t`* (if included) is in scope in *`E`*, where it has type *`T`*. The pragma makes *`E`* an exceptional postcondition of the routine the pragma modifies. That is, it specifies that *`E`* holds whenever the routine completes abruptly by throwing an exception *`t`* whose type is a subtype of *`T`*.

When checking the body of the routine, ESC/Java checks that *`E`* holds whenever the routine completes abruptly by throwing an exception *`t`* whose type is a subtype of *`T`*. When checking code that calls the routine, ESC/Java assumes that the *`E`* holds just after the call if the call completes abruptly with an exception whose type is a subtype of *`T`*.

Fine points

Like normal postconditions, exceptional postconditions of `synchronized` methods apply to the state after the release of the lock.

The identifier *`t`* may not be the same as any formal parameter of the routine, and quantified expressions (see sections 3.2.10 and 3.2.11) within *`E`* may not use *`t`* as a bound variable name.

The expression *`E`* can include uses of `\fresh` and `\old`, which have the same semantics as in an `ensures` pragma. However, *`E`* cannot mention `\result`, since an abruptly-terminating routine invocation returns no result. Similarly *`E`* cannot mention `this` if the `exsures` pragma modifies a constructor, since we take the view that the object being constructed should be discarded. (This view is potentially unsound; see section C.0.8.)

A single routine declaration may be modified with any number of `exsures` pragmas. ESC/Java checks that the body obeys each `exsures` pragma and assumes that calls obey each `exsures` pragma. For example, if a routine is modified by the pragmas

```
exsures (T1 t) E1; exsures (T2 t) E2;
```

where *`T2`* is a subtype of *`T1`*, then ESC/Java checks (if checking the body of the routine) or assumes (if checking a caller) that *`E1`* holds whenever the routine completes abruptly by throwing an exception that is an instance of *`T1`*, and that both *`E1`* and *`E2`* hold whenever the routine completes abruptly by throwing an exception that is an instance of *`T2`*.

A method declaration that overrides another method declaration cannot be modified with an `exsures` pragma, but inherits the exceptional postconditions of the overridden method. (See also the `also_exsures` pragma described in section 2.3.6.)

2.3.5 `also_ensures` pragma

An `also_ensures` pragma is a routine modifier pragma. It has the form

```
also_ensures E ;opt
```

where *`E`* is a boolean specification expression. An `also_ensures` pragma has the same semantics as an `ensures` pragma, but may appear as a modifier only of a method declaration that overrides another method declaration (while overriding method declarations are forbidden to have `ensures` pragmas).

2.3.6 `also_exsures` pragma

An `also_exsures` pragma is a routine modifier pragma. It has the form

```
also_exsures (T t) E ;opt
```

or

```
also_exsures (T t) E ;opt
```

where *`T`* is a subtype of `java.lang.Throwable`, *`t`* (if included) is a an identifier, and *`E`* is a boolean specification

expression. An `also_exsures` pragma has the same semantics, and must obey the same syntactic restrictions, as an `exsures` pragma, but may appear as a modifier only of a method declaration that overrides another method declaration (while overriding method declarations are forbidden to have `exsures` pragmas).

The relationship between `also_exsures` and `exsures` is exactly analogous to that between `also_ensures` and `ensures`.

2.3.7 `also_requires` pragma

An `also_requires` pragma is a routine modifier pragma. It has the form

```
also_requires E ;opt
```

where *E* is a boolean specification expression. An `also_requires` pragma has the same semantics as a `requires` pragma, but the declaration of a method *C.m* may be modified by an `also_requires` pragma only if all three of the following conditions hold:

- *C* is a class (not an interface),
- the declaration of *C.m* overrides some method declaration in a superinterfaces of *C*, and
- the declaration of *C.m* does *not* override any method declaration in a superclass of *C*.

(By contrast, a `requires` pragma may only modify a method [or constructor] declarations that does not override any other method declaration, and neither a `requires` pragma nor an `also_requires` pragma may modify a method declarations in classes that overridden declaration in a superclasses, or a method declaration in an interface that overrides a method declaration in a superinterface.)

Fine points

The `also_require` pragma is a potential source of unsoundness. Suppose method `C.m` of class `C` overrides method `I.m` of interface `I`. When checking a call of the form *E*`.m(...)`, where *E* is an expression of type `I`, ESC/Java only enforces the preconditions of `I.m` and not any preconditions given by `also_requires` pragmas modifying the declaration of `C.m`. However, the expression *E* might evaluate to a value of type `C`, causing the call to invoke `C.m`, and the correctness of `C.m`'s implementation may depend on preconditions given in such `also_requires` pragmas. The reason that ESC/Java includes an `also_requires` pragma, despite its unsoundness, is that it is often essential for preconditions of a method `C.m` to mention instance variables of class *C*, and Java does not allow instance variables to be declared in interfaces. [Note, however, that ESC/Java does allow declarations of ghost variables (section 2.6.0) in interfaces.]

In addition to the potential unsoundness just described, the `also_requires` pragma shares the potential unsoundness of the `require` pragma in the presence of multiple inheritance (see section C.0.5).

2.3.8 `also_modifies` pragma

An `also_modifies` pragma is a routine modifier pragma. It has the form

```
also_modifies L ;opt
```

where *L* is a nonempty, comma-separated list of specification designators (see section 2.3.1). An

`also_modifies` pragma has the same semantics as a `modifies` pragma (section 2.3.1), but may appear only as a modifier of a method declaration that overrides another method declaration (while overriding method declarations are forbidden to have `modifies` pragmas).

An overriding method declaration may modify both the targets named in its own `also_modifies` pragmas and any modification targets it inherits from the overridden method.

Fine points

Like the `also_require` pragma (section 2.3.7), the `also_modifies` pragma is a potential source of unsoundness. When writing code that follows a call *`E`*`.m(...)` where *`E`* has static type `T`, a programmer may reasonably be expected to cope with the possibility that the call has modified parts of the state named in `modifies` pragmas that annotate the declaration of `T.m`, but it seems unreasonable to expect the programmer to deal with the possibility that the call might modify other parts of the state (perhaps parts mentioned in `also_modifies` pragmas of yet-to-be-written overrides) as well.

The reason that ESC/Java includes an `also_modifies` pragma is that an overriding method may need to modify fields that are not in scope at the point where the overridden method is declared, such as `private` variables of the class declaring the overriding method.

To reduce the likelihood of unsoundness, `also_modifies` pragmas ought not to name modification targets that are accessible [*JLS*, 6.6] from the scope of the overridden method. However, the current ESC/Java does not enforce any such restriction. For more discussion of the problem of specifying modification targets in the presence of subclassing, and for a sound solution thereof, see [Leino98].

2.4 Pragmas for specifying data invariants

This section describes several pragmas that ESC/Java provides for specifying properties of variables and data structures.

2.4.0 `non_null` pragma

A `non_null` pragma is a modifier pragma. It may modify the declaration of a variable of a reference type. The variable may be a static field, instance variable, local variable, or parameter. It has the form

```
non_null
```

The pragma causes ESC/Java to check, at each assignment to the variable, that the value assigned is not `null`, and to assume at each use (except in one case, described below) that the value is not `null`. When a formal parameter declaration is annotated with a `non_null` pragma, ESC/Java checks at each call site that the corresponding actual is not `null`.

Fine points

In the case that a `non_null` instance variable *`x`* is declared in a class *`C`* and a read access of *`x`* occurs in a constructor of *`C`* that does not call a sibling constructor, then ESC/Java does not automatically assume that the value read will be non-null.

ESC/Java allows a `non_null` pragma to modify a formal parameter, even though Java 1.0 syntax does not allow modifiers on parameter declarations.

A `non_null` pragma may not modify a parameter declaration of a method that overrides another method.

Sometimes the same design decision might be expressed either by a `non_null` pragma or by some other kind of pragma, such as a `requires` pragma or an `invariant` pragma. For example, instead of using a `requires` pragma in section 0.2

```
//@ requires input != null;
Bag(int[] input) {
...
}
```

we might have written

```
Bag(/*@ non_null */ int[] input) ...
```

and instead of using a `non_null` pragma in section 0.3

```
class Bag {
  /*@ non_null */ int[] a;
  ...
```

we might have written

```
class Bag {
  int[] a;
  //@ invariant a != null;
  ...
```

In each of these cases, the alternative annotations have slightly different semantics, but either alternative would be adequate to enable checking of our example program. (The differences are that the `non_null` pragmas are checked at every assignment to `input` or `a`, whereas the `requires` pragma would allow assignments of `null` to `input` within the body of the `Bag` constructor, and the `invariant` pragma would be checked only at routine calls, as explained in section 2.4.1.)

We recommend that `non_null` pragmas be used in preference to semantically similar `requires` and `invariant` pragmas except in cases where the somewhat stricter semantics of `non_null` makes its use untenable.

Limitation: Since the current ESC/Java does not check static bodies and static initializers, it is entirely the user's responsibility to ensure that static fields declared as `non_null` are in fact initialized to have non-null values.

2.4.1 `invariant` pragma

An `invariant` pragma is a declaration pragma. It has the form

```
invariant E ;opt
```

where `E` is a boolean specification expression. The pragma declares *E* to be an *object invariant* of the class within whose declaration the pragma occurs. If `E` mentions `this`, either explicitly or implicitly, then `E` is said to be an *instance invariant*; otherwise `E` is a *static invariant*. Roughly speaking, all object invariants are supposed to hold at all routine call boundaries. That is (1) if `E` an instance invariant of class `T`, then

```
(\forall T t; E[this=t])
```

should be true at all routine calls and returns, where `t` is a variable not occurring in `E`, the universal quantification ranges over all allocated instances `T`, and $E_{\texttt{this=}t}$ is the result of substituting `t` for all (explicit and implicit) occurrences of `this` in `E`, and (2) if `E` is a static invariant of class `T`, then `E` should be true at all routine call boundaries, regardless of whether or not any allocated objects of class `T` exist.

Fine points

ESC/Java does not fully enforce the discipline just described, partly because it would be too strict for many programs (which may have legitimate reasons for temporarily breaking object invariants) and partly because such checking would be very expensive. Instead, it performs less expensive (and potentially unsound) checking. Essentially, ESC/Java assumes object invariants for all objects on entry to a routine body, and checks objects invariants for all objects at the end of the body; however, at call sites, ESC/Java checks object invariants only for parameters and static fields, and assumes of the call only that it doesn't break any object invariants and that it establishes object invariants for freshly allocated objects. More precisely, when checking the body of a routine `R`:

> ESC/Java chooses, from all available object invariants, a set of invariants to be considered relevant to the checking of `R`. [The heuristic used to choose the ``relevant" object invariants is fairly complicated and subject to change, so we don't explain it in this manual.]
>
> ESC/Java assumes that at the start of `R`'s body, all relevant static invariants hold, and all relevant instance invariants hold for all allocated objects.
>
> For every routine call in the body of `R`:
>
> > ESC/Java checks that all relevant (to `R`) static invariants hold before the call.
> >
> > ESC/Java checks that the value of each actual parameter of the call, including the implicit `this` parameter of an instance method call, satisfies every relevant (to `R`) instance invariant of every supertype of the static type of the corresponding formal parameter.
> >
> > ESC/Java checks that the value of each static field `f` (actually each of a heuristically selected set of ``relevant" static fields) satisfies every relevant (to `R`) instance invariant of every supertype of the static type of `f`.
> >
> > ESC/Java assumes that all relevant (to `R`) static invariants hold after the call.
> >
> > ESC/Java assumes that, after the call, each relevant (to `R`) object invariant `E` of type `T`

holds for all instances of *T* for which *E* held before the call, and for all instances of *T* allocated by the call (except that when a constructor calls a superclass constructor, ESC/Java assumes that the constructed object satisfies the instance invariants of the supertype but not necessarily any additional invariants of the subtype).

ESC/Java checks that at the end of *R*'s body all relevant static invariants hold, and all relevant instance invariants hold for all allocated objects (except that if *R* is a constructor, ESC/Java assumes that the constructed object satisfies all relevant object invariants of *R*'s type, including those inherited from supertypes, but not necessarily those of any subtypes of *R* whose constructors may call *R* as a superclass constructor, and furthermore, when considering abnormally-terminating executions of the constructor *R*, ESC/Java makes no assumptions about the constructed object).

It is a potential source of unsoundness for an object invariant to mention fields other than those declared in the class declaring the invariant. For example, suppose that `S` is a subclass of `T` and that an instance invariant of `S` mentions a field `f` declared in `T`, and consider a routine `R` containing an assignment `x.f = ...` where `x` has static type `T`. If `x` has allocated type `S`, the assignment might break the instance invariant declared in `S`. But neither the programmer of `R` nor ESC/Java can reasonably be expected to enforce the invariants of `S`, since `S` might not be in scope in `R`, and indeed might be written after `R`. The current ESC/Java does not attempt to detect declarations of such unenforceable invariants, but future versions may bring some such declarations to the programmer's attention. (For a more detailed examination of the unenforceable invariant problem, see [LS97].)

When a `final` field *f* is declared with a *VaribleInitializer* [*JLS*, 19.8.2], ESC/Java may infer some simple invariants automatically. For instance, if the initializer is a constant expression [*JLS*, 15.27] *C* not of type `String`, then ESC/Java infers the invariant *f* `==` *C*; and if the initializer is a string literal, an array constructor, or an invocation of `new`, then ESC/Java infers the invariant *f* `!= null`.

Limitation: The static bodies and static initializers of a class are supposed to establish the static invariants declared in the class, but the current ESC/Java does not check this. Thus the responsibility for initial establishment of static invariants lies entirely with the user.

2.4.2 `axiom` pragma

An `axiom` pragma is a declaration pragma. It has the form

`axiom` *E* `;`$_{opt}$

where *E* is a boolean specification expression. The pragma causes ESC/Java to assume that *E* (which we call an *axiom*) is true at the start of every routine body that it checks.

Fine points

Since the `axiom` pragma introduces assumptions without introducing reciprocal checks, it is potentially unsound and programmers should use it carefully.

An obvious use of an `axiom` pragma would be to state some universally true fact. For example, the built-in capabilities of Simplify (`Simplify(1)`), the theorem prover used by ESC/Java, include a decision procedure for linear arithmetic but no rules about multiplication other than by constants. Thus there may be cases where ESC/Java would be helped by an annotation like

```
//@ axiom (\forall int x, y; x >= 0 & y >= 0 ==> x*y >= 0);
```

(recall that ESC/Java doesn't claim to check for arithmetic overflow, so an axiom like this is consistent with the view that Java `int`'s behave like the mathematical integers, although this view is not true). Beware that profligate introduction of axioms (in the hope that they may occasionally be useful) can have a serious impact on performance.

Less obviously, axioms mentioning Java variables are sometimes useful, and ESC/Java allows such axioms despite the unsoundness inherent in assuming them when checking method bodies without also checking them at call sites. For further discussion of this point, see the example in section 2.7.2 involving an axiom about the lock order. Axioms may not mention `this` or `\lockset`.

When checking a constructor body, ESC/Java assumes that axioms hold before the implicit superclass constructor call, if any, and thus also before execution of instance variable initializers.

When it is clear that the correctness of the program depends on a particular property holding for a particular expression *E* of type *T* at a particular point in the program, it may be better to write an `assume` pragma

```
//@ assume ...E...;
```

at that point in the program, in preference to introducing an axiom pragma

```
//@ axiom (\forall t T; ...t...);
```

stating that the property holds for all values of the type (even if it does). For one thing, the Simplify theorem prover used by ESC/Java is incomplete, and may not discover the relevant instance of the axiom to apply. On the other hand, regardless of whether Simplify discovers the relevant instance of the axiom, it may spend a lot of time considering many irrelevant instances.

Limitation: Just as for invariants (section 2.4.1), ESC/Java limits its use of axioms to a restricted set of axioms considered ``relevant'' to the checking of any particular routine *R*. In the current version of ESC/Java, the only axioms considered relevant are those declared in the same class as the implementation being checked. Consequently, there is no way for the user to declare ``libraries'' of potentially useful axioms.

2.4.3 `loop_invariant` pragma

A `loop_invariant` pragma is a statement pragma. It has the form

```
loop_invariant E ;opt
```

where *E* is a boolean specification expression. A `loop_invariant` pragma must appear before a Java loop statement--that is, a Java `for`, `while`, or `do` statement [*JLS*, 14.10, 14.11. 14.12], or a Java labeled statement [*JLS*, 14.6] *L* : *S* such that *S* is a Java labeled statement. Between the `loop_invariant` pragma and the associated loop statement, there may be no intervening Java code and no intervening pragmas, except for `nowarn` pragmas and other `loop_invariant` pragmas. The pragma causes ESC/Java to check that *E* holds at the start of each iteration of the loop.

Fine points

For the purposes of name-scoping, a loop invariant is treated as if it occurred just inside the associated loop statement. That is, if the associated loop statement is a `for` statement (or a `for` statement wrapped within one or more labeled statements), then the variables declared by the *ForInit* [*JLS*, 19.11] are in scope in the loop invariant.

An ``iteration of a loop" includes the termination test, and also includes the update code in a `for` loop. Thus

```
//@ loop_invariant E;
while (B) {
  S
}
```

intuitively means

```
while (true) {
  //@ assert E;    // but giving a LoopInv warning
  if (!(B)) break;
  S
}
```

(however, see the comments below about loop unrolling). Note that the checking of the loop invariant `E` applies to the state before the test of `B` (and before any side effects in the evaluation of `B`). Likewise,

```
//@ loop_invariant E;
do {
  S
} while (B);
```

intuitively means

```
while (true) do
 //@ assert E;    // but giving a LoopInv warning
 S
 if (!(B)) break;
}
```

and

```
//@ loop_invariant E;
L: for(I1, …, Im; B; U1, …, Un) {
     S
   }
```

intuitively means

```
{ I1, …, Im;
  L: while (true) {
    //@ assert E;    // but giving a LoopInv warning
    if (!(B)) break;
```

```
    S
    U1, …, Un;
   }
}
```

but with any occurrence within *S* of

```
continue L;
```

(or simply of `continue;` appearing outside of any nested loop and thus meaning `continue L;`) transferring control only to the end of *S* rather than to the end of *Un*.

In ESC/Java, loop invariants are optional. The checker considers only execution paths in which the loop body is executed at most once (and the test for being finished is executed most twice), rather than the potentially infinite number of paths that are really possible. Because of this simplification the checker doesn't need an invariant to analyze the loop.

If you do include a loop invariant, ESC/Java will check that it holds both initially and after the single loop iteration that the current checker considers. Consequently, the checking performed on

```
//@ loop_invariant E;
while (B) {
  S
}
```

is actually the same as would be done for

```
//@ assert E;    // but giving a LoopInv warning
if (B) {
  S
  //@ assert E;  // but giving a LoopInv warning
  //@ assume !B; // (don't check later code for case B == true)
}
```

(where execution of a `continue` in *S* results in normal termination of *S* and execution of a `break` in *S* results in normal termination of the entire code fragment above). You can make ESC/Java check more (or fewer) iterations of loops by using the `-loop` command-line option (see section C.0.1).

2.5 Pragmas affecting conditions under which variables may be referenced

2.5.0 `spec_public` pragma

A `spec_public` pragma is a modifier pragma. It may occur as a modifier only of a non-`public` field declaration. It has the form

```
spec_public
```

The effect of the pragma is to make the fields declared in the declaration be as spec-accessible (that is accessible

in pragmas, see section 3.3) as they would have been if the declaration had been `public`.

For example, if a `private` field of a class is declared with a `spec_public` pragma, then the field can be mentioned in pre- and postconditions of a `public` method, but clients of the class cannot modify the field directly.

Example

Consider the `Bag` example form section 0. In presenting this example, we glossed over the issue of accessibility. To allow clients of the `Bag` class to make use of the routines of the class (`Bag`,`extractMin`, and others that we might add), these routines should be declared `public`. On the other hand, the fields `a` and `n` used in the implementation ought not to be `public`. For example, arbitrary clients ought not to be able to write these fields except by calling routines of the `Bag` class.

If we simply declare the routines, but not the fields of `Bag` to be `public`, ESC/Java will complain:

ESC/Java input from Bag.java

```
 1: class Bag {
 2:   /*@ non_null */ int[] a;
 3:   int n;
...  ...
13:   //@ requires n >= 1;
14:   public int extractMin() {
...  ...
```

ESC/Java output:

```
Bag.java:13: Error: Fields mentioned in this modifier pragma must be at
least as accessible as the field/method being modified (perhaps try
spec_public)
  //@ requires n >= 1;
              ^
Caution: Turning off extended static checking due to type error(s)
1 caution
1 error
```

If programmers using the `Bag` class are not supposed to know that its implementation includes the field `n`, then it is unreasonable to expect them to establish a precondition involving `n` before calling `extractMin`. We can prevent ESC/Java from complaining by declaring the field `n` to be `spec_public`:

```
 3:   /*@ spec_public */ int n;
```

This will allow `n` to be mentioned in `requires` pragmas of `public` routines, as well as in pragmas occurring in other packages that import the `Bag` class. Of course, actual Java code in other packages will not be able to read or write the `n` field directly (as would be allowed if `n` were declared with a Java `public` modifier).

Fine point

By declaring `n` with a `spec_public` pragma, the implementer of the `Bag` class expresses a design decision that users of `Bag` are supposed to know about the `n` field, even if their Java code cannot access it directly. This

design decision could cause problems if implementer later decided to change the implementation to represent a `Bag` using a data structure that did not include an explicit count--for example, a linked list. See the first example in section 2.6.2 for further discussion of this issue.

2.5.1 `readable_if` pragma

A `readable_if` pragma is a modifier pragma. It can occur only as a modifier of a field declaration or of a local variable declaration. It has the form

`readable_if` *E* `;`$_{opt}$

where *E* is a boolean specification expression.

The pragma causes ESC/Java to check that *E* is true just before any read access of any of the variable(s) declared in the declaration. The pragma thus expresses the programmer's intention that the variable modified by the pragma has a meaningful value only when *E* is true.

The specification expression *E* is allowed to mention `this` if the pragma modifies an instance field declaration, or if the pragma modifies a local variable declaration within an instance method or a constructor. If the pragma modifies the declaration of an instance field *f*, then for purposes of checking a read access *o*`.`*f*, occurrences of `this` in *E* are taken to denote the value of *o*.

Fine points

If the pragma modifies a local variable declaration then the variables declared by the declaration are not in scope in *E*, even if the pragma occurs just before the final semi-colon instead of before the type.

```
class C {
  boolean b;
  void m() {
    boolean b /*@ readable_if b */;
    // "b" in the pragma above means this.b, not the local b
    // "b" in the pragma below means the local b
    /* readable_if b */ int c;
    ...
  }
}
```

If the pragma modifies a field declaration, then all fields of the containing class are in scope in *E*, even those that are declared in textually later declarations.

```
class C extends B {
  /*@ readable_if b */ int a;
  boolean b;
  // "b" in the pragma means this.b, not ((B)this).b
  ...
}
```

If the pragma modifies a field declaration, then the free variables of *E* must be spec-accessible wherever fields

declared by the declaration are accessible according to Java's access control rules [*JLS*, 6.6]

> Remark: For the picky, a more precise name for this pragma might be ``meaningful_only_if".
>
> Remark: Perhaps unfortunately, ESC/Java does not provide a way to say when particular array elements are readable (meaningful). This could be useful since one could then express, for example, ``a[i] is meaningful only if 0 <= i & i < n", which says that only the first n elements of array a are in use.

2.5.2 `uninitialized` pragma

An `uninitialized` pragma is a modifier pragma. An `uninitialized` pragma can occur as a modifier only of a local variable declaration that has an initializer. The pragma causes ESC/Java to check that no execution path reads the variable without first performing an assignment (other than the initializer) to the variable.

The intended use of the `uninitialized` pragma is for situations in which the conservative nature of Java's ``definite assignment" rules [*JLS*, 16] has forced the programmer to supply an irrelevant initial value.

2.6 Pragmas concerning ghost variables

Unlike ESC/Modula-3 (see SRC Report 159, ``Extended static checking") ESC/Java does not support data abstraction. But ESC/Java does provide a poor man's version of abstract variables, called *ghost* variables. We describe here the `ghost` pragma, which declares ghost variables, and the `set` pragma, which modifies them, and then give an example of their use.

2.6.0 `ghost` pragma

A `ghost` pragma is a declaration pragma. It is allowed where the declaration of a class or interface member is allowed. It has the syntax

> `ghost` *M T VD* `;`$_{opt}$

where *T* is a specification type (see section 3.0), *VD* is a Java *VariableDeclarators* [*JLS*19.8.2], and *M* is a sequence of modifiers. (In other words, the pragma is like an ordinary Java variable declaration preceded by the word `ghost`.) In the current ESC/Java, *M* must include the modifier `public`, and *VD* can declare only one identifier. The only other modifier allowed in *M* is `static`. No initializers are allowed in *VD*.

The pragma is like the Java declaration

> *M T VD* `;`$_{opt}$

except that it is visible only to ESC/Java, not to the compiler. The variables declared by a `ghost` pragma are called ghost variables.

Fine points

No field declared in *VD* may have the same name as a field (including a ghost field or a field declared in a supertype) already declared for the type in whose declaration the pragma occurs.

The current ESC/Java does not implement local ghost variables.

A ghost field declared in an interface *I* is multiply inherited by classes that implement *I* and interfaces that extend *I*. (If a ghost variable *f* is declared in an interface *I*, and *I* is extended by interfaces *J* and *K*, and a class *C* implements *I*, *J*, and *K*, and if *x* is a variable of type *C*, then the expressions ((*I*)*x*).*f*, ((*J*)*x*).*f*, ((*K*)*x*).*f*, and *x*.*f* all denote the same ghost field. That is, *C* gets only one copy of *f*.)

For information about resolution of name conflicts involving ghost variables, see section 3.3.

In some cases, it is useful to declare a ghost variable with a modifier pragma, such as `non_null`. In this case, the modifier pragma must occur within a nested pragma-containing comment, for example:

```
//@ ghost public /*@ non_null */ T t;
```

More precisely, within a `ghost` pragma

M T VD ;*opt*

an inner pragma-containing comment containing modifier pragmas is allowed either as part of *M* or just after *VD*. The only modifier pragmas that can usefully modify a ghost variable are `non_null`, `monitored`, and `monitored_by` (the current ESC/Java may accept others, but if it does they will have no effect on the checking performed). In addition, these pragmas are allowed only in the case where they would be allowed for a normal variable according to the table in section 2.0.1. For example, a `static` ghost variable cannot be declared with the pragma `monitored`, or with the pragma `monitored_by` *E* where the expression *E* mentions `this`. ESC/Java checks `monitored` (section 2.7.1) and `monitored_by` (section 2.7.0) pragmas for a ghost variable *g* only where *g* is assigned to by a `set` pragma (section 2.6.1), not at other places where *g* is used in pragmas.

2.6.1 `set` pragma

A `set` pragma is a statement pragma. It has the form

`set` *D* `=` *E* ;*opt*

where *D* is a ghost designator and *E* is a specification expression containing no quantifiers or labels. The pragma has the same semantics as the Java assignment statement *D*= *E* would have if *D* and *E* were in Java.

A ghost designator can have one of the following forms:

O.*f* , where *O* is specification expression of an object type *T* and *f* is a ghost field of *T*.

f , where *f* is a static ghost field or `this`.*f* is a legal ghost designator of the preceding form.

Fine point

If the field *f* assigned to by a `set` pragma is declared with a `non_null`, `monitored`, or `monitored_by` pragma, then ESC/Java will perform the usual checking implied by the modifier pragma, generating warnings in case the value being assigned may be `null`, or in case the specified lock may not be held. This is the only circumstance in which the current ESC/Java generates warning of possible run-time errors in the evaluation of pragmas.

2.6.2 Examples using ghost variables

Example: specifying `Bag` without revealing its implementation

As pointed out in section 2.5.0, there is a scoping problem with the `Bag` example from section 0: In a realistic situation, the implementer of the `Bag` class may want to make `public` the routines of the class, but not the fields. How then are clients of the `public` method `extractMin` to discharge the precondition `requires n >= 1`, which mentions the non-`public` field `n`?

In section 2.5.0, we showed how to address the problem by declaring `n` with a `spec_public` pragma. But what if the implementer of the `Bag` class wanted to leave open the possibility of switching to a different implementation (say, a linked list) in which the representation of a `Bag` did not include an explicit count of the elements?

To resolve this annotation question, we must first make a design decision about the actual Java code: How, in fact, *are* the clients of the `Bag` class supposed to avoid ever calling the `extractMin` method of an empty `Bag`?

One approach--the one that we will illustrate here--would be for the implementer of the `Bag` class to provide a method `isEmpty` that clients could use to test a `Bag` for emptiness before calling its `extractMin` method. To specify the semantics of the `Bag` class, we would use a boolean ghost field `empty`, that would be `true` precisely for those `Bag`'s that contain no elements. Here's how that annotated implementation of `Bag` might look like under this approach:

```
1:  class Bag {
2:    /*@ non_null */ int[] a;
3:    int n;
4:    //@ invariant 0 <= n && n <= a.length;
5:    //@ ghost public boolean empty;
6:    //@ invariant empty == (n == 0);
7:
8:    //@ requires input != null;
9:    //@ ensures this.empty == (input.length == 0);
10:   public Bag(int[] input) {
11:     n = input.length;
12:     a = new int[n];
13:     System.arraycopy(input, 0, a, 0, n);
14:     //@ set empty = n == 0;
15:   }
16:
17:   //@ ensures \result == empty;
18:   public boolean isEmpty() {
19:     return n == 0;
20:   }
21:
22:   //@ requires !empty;
23:   //@ modifies empty;
24:   //@ modifies n, a[*];
25:   public int extractMin() {
```

```
26:       int m = Integer.MAX_VALUE;
27:       int mindex = 0;
28:       for (int i = 0; i < n; i++) {
29:         if (a[i] < m) {
30:           mindex = i;
31:           m = a[i];
32:         }
33:       }
34:       n--;
35:       //@ set empty = n == 0;
36:       //@ assert empty == (n == 0);
37:       a[mindex] = a[n];
38:       return m;
39:     }
40:   }
```

The precondition for `extractMin` on line 22 specifies that it is incorrect to invoke the `extractMin` method of an empty `Bag`. The postcondition for `isEmpty` on line 17 allows clients (and ESC/Java) to determine that a code fragment like

```
if (!x.isEmpty()) i = x.extractMin();
```

will never violate the precondition for `extractMin`. The invariant on line 6 relates the ghost field `empty` to the actual field `n` in the implementation of a `Bag` and is of interest only to the implementer of the `Bag` class and not to clients. ESC/Java uses this invariant to verify that the implementation of `isEmpty` satisfies its postcondition, and to verify that the implementation of `extractMin`, when called in accordance with its precondition, will not attempt to access `a[-1]` in line 37. The `set` pragmas on lines 14 and 35 guarantee that the invariant on line 6 is established by the `Bag` constructor and preserved by the `extractMin` method.

Example: specifying a generic queue

Consider a class that implements queues of `Object`'s. There is a constructor that creates an empty queue, and there are instance methods for enqueueing and dequeueing elements and for testing whether a queue is empty:

```
public class Queue {
  …
  public class Queue() {…}
  public enqueue(Object e) {…}
  public boolean isEmpty() {…}
  public Object dequeue() throws {…}
}
```

In the absence of any annotations, a number of problems come up when checking clients of the `Queue` class.

First, there is no check against calling `dequeue` on an empty queue.

Second, a user may wish to have a queue whose elements are all of some type *`T`* (a proper subtype of `Object`). Since the result type of `dequeue` is `Object`, the user will frequently have to cast the result of dequeue to a `T`.

```
T t = ..., u;
Queue q = new Queue();
...
q.enqueue(t);
...
u = (T)q.dequeue();
```

Given the code fragment above, ESC/Java has no way to know that the cast will succeed.

Finally, it is common for queues (and other such ``container'' objects) to contain only non-null elements. When a queue is intended to contain only non-null elements it would be nice to have ESC/Java check that only non-null elements are enqueued, and on the other hand not issue spurious warnings about potential null dereferences on dequeued objects.

Here is a `.spec` file (see section 5.1.2) for the `Queue` class with annotations addressing the issues described above.

```
public class Queue {

  //@ ghost public int size;
  //@ invariant size >= 0;
  //@ ghost public \TYPE elementType;
  //@ ghost public boolean canHoldNull;

  //@ ensures elementType == \type(Object);
  //@ ensures canHoldNull;
  //@ ensures size == 0;
  public Queue();

  /*@ requires \typeof(e) <: elementType |
               (e == null & canHoldNull); */
  //@ modifies size;
  //@ ensures size == \old(size) + 1;
  public void enqueue(Object e);

  //@ ensures \result == (size == 0);
  public boolean isEmpty();

  //@ requires size >= 1;
  //@ modifies size;
  /*@ ensures \typeof(\result) <: elementType |
              (\result == null & canHoldNull);
  */
  //@ ensures size == \old(size) - 1;
  public Object dequeue();
}
```

(The specification expression constructs `\TYPE`, `\type`, `\typeof`, and `<:` are explained in sections 3.0, 3.2.0, 3.2.1, and 3.2.3, respectively.) Given the specification above, the user can express the intention to use a queue `q`

to hold only non-null elements of type `T` by writing

```
Queue q = new Queue();
//@ set q.elementType = \type(T);
//@ set q.canHoldNull = false;
```

ESC/Java will then warn of a possible precondition violation on any call `q.enqueue(`*`E`*`)` where *`E`* may not be an instance of `T`. On the other hand it will not issue a (spurious) warning of possible cast failure on `(T)q.dequeue()`. Also ESC/Java will warn of a possible precondition failure on `q.dequeue()` if it cannot establish, based on the number of elements enqueued and dequeued done since the allocation of `q` or the most recent call to `q.isEmpty`, that `q` is nonempty (`q.size >= 1`) at the point of the call.

For more examples of the use ghost variables in specifying container classes, read the files `Dictionary.spec`, `Enumerator.spec` and `Vector.spec` in *escjavaRoot*`/lib/specs/java/util` (see appendix B).

2.7 Pragmas for specifying synchronization

ESC/Java provides support for checking that multi-threaded programs respect a locking discipline that prevents race conditions and simple deadlocks.

A race condition is a situation in which two threads access a variable simultaneously and the accesses are not both read accesses. To prevent race conditions, the locking discipline requires that every shared variable is *monitored* by one or more locks. If a variable is monitored by a lock, a thread is not allowed to access the variable unless it holds the lock (and this discipline is enforced by ESC/Java).

A deadlock occurs if there is a cycle of threads, each holding a lock that some other thread in the cycle is waiting to acquire. To prevent deadlocks, the locking discipline requires that the programmer declare a partial order in which locks are to be acquired. ESC/Java will then check that each thread does in fact acquire locks in the given order. But the checker trusts the programmer that the declared locking order is actually a partial order.

2.7.0 `monitored_by` pragma

A `monitored_by` pragma is a modifier pragma. It can occur only as a modifier of a field declaration. It has the form

`monitored_by` *`L`* `;`$_{opt}$

where *`L`* is a nonempty, comma-separated list of specification expressions.

The pragma declares that the modified field is a shared variable monitored by the locks in *`L`*. That is, it causes ESC/Java to check that (1) the field is never read except by a thread holding at least one non-null lock in *`L`* and (2) that the field is never written except when at least one lock in *`L`* is non-null and the writing thread holds all non-null locks in *`L`*. (See the first fine points below for an exception the preceding statement.)

If the field declaration modified by the pragma declares an instance field *`f`*, then the expressions in *`L`* may mention `this` (explicitly or implicitly). When ESC/Java checks an access to *`o.f`*, occurrences of `this` within *`L`* are considered to denote the value of *`o`*.

Fine points

When checking a constructor body ESC/Java does not require that any lock be held in order to accesses a field of `this`, even if the field is declared with at `monitored_by` or `monitored` (section 2.7.1) pragma. (In some cases, this can result in unsoundness; see section C.0.8.)

All fields of the class containing the pragma are in scope in `E`, even those that are declared in textually later declarations.

```
class C extends B {
  /*@ monitored_by g */ S f;
  T g;
  // "g" in the pragma means this.g, not ((B)this).g
  …
}
```

The variables mentioned in `L` must be spec-accessible (section 3.3) anywhere the field itself is accessible [*JLS*, 6.6]. For example, a `public` field may not be monitored by a `private` lock (unless the lock is declared `spec_public`, see section 2.5.0).

A field declaration may be modified by multiple `monitored_by` pragmas, in which case the effect is as if there were a single `monitored_by` pragma listing all the locks mentioned in any of the actual `monitored_by` pragmas. (**Bug/limitation:** When ESC/Java issues a `Race` warning for such a field, it will mention only one of the `monitored_by` pragmas, and possibly not the right one, as the annotation associated with the warning.)

2.7.1 `monitored` pragma

A `monitored` pragma is a modifier pragma. It can occur only as a modifier of an instance variable declaration. It has the form

```
monitored
```

and is semantically equivalent to

```
monitored_by this;
```

2.7.2 Examples illustrating race and deadlock checking

Annotation of programs for race and deadlock checking typically requires use not only of `monitored` and/or `monitored_by` pragmas but of other pragmas as well, most commonly `requires` pragmas (section 2.3.0) and `axiom` pragmas (section 2.4.2) involving the locking order < (section 3.2.7) and the lock set `\lockset` (section 3.2.5). Rather than leave it to the reader to assemble the big picture from fragments of information scattered over many sections of this manual, we present here a few examples to show how the pieces fit together.

In specification expressions the special ESC/Java identifier `\lockset` refers to a special variable, called the lock set, denoting the set of all objects held as locks by the current thread. For testing membership of locks in the lock set, we overload the array subscripting notation, so that `\lockset[X]` is true when object `X` is in the lock set `\lockset`.

Thus, given the instance field declarations

```
T f;
U g;
/*@ monitored_by f, g; */ V h;
```

the code fragment

```
x.h = y.h
```

is treated roughly like

```
/*@ assert y != null; */   // Check for null dereference
/*@ assert (y.f != null && \lockset[y.f]) |
            (y.g != null && \lockset[y.g]); */
            // Check for race (section 4.17) on read
/*@ assert x != null; */   // Check for null dereference
/*@ assert (x.f != null | x.g != null) &
            (x.f != null ==> \lockset[x.f]) &
            (x.g != null ==> \lockset[x.g]); */
            // Check for race on write
x.h = y.h;
```

The necessary conditions to establish such locking assertions can be established explicitly, for example by a `requires` pragma as in

```
//@ requires f != null && \lockset[f];
void m() {
  … h …   // read access produces no Race warning
}
```

Also, a `synchronized` routine body and a `synchronized` statement always begin by adding a lock to the lock set.

```
class C {

  monitored T t;

  synchronized void m() {
    T x = t; /* no race because starting a synchronized
                method establishes \lockset[this] */
    …
  }

  void n() {
    synchronized (this) {
      T x = t;  // no race
      …
    }
```

```
  }
}
```

The preceding examples deal only with race detection. We now turn to the issue of deadlock detection.

ESC/Java supports a synchronization discipline in which deadlocks are avoided by imposing a partial ordering on locks and acquiring locks only in increasing order. The arithmetic ordering relations < and <= are overloaded to compare objects in the locking order. The special function `\max` yields the maximum lock in a lock set.

Without some help in the form of user-supplied pragmas, it is never possible for ESC/Java to prove the absence of deadlock in a program that ever acquires a lock. Deadlock checking is therefore disabled by default. When deadlock checking is enabled (by use of the command-line option `-warn Deadlock`), ESC/Java checks `synchronized` statements and `synchronized` methods and issues warnings if they might acquire locks out of order.

A `synchronized` statement

```
synchronized (E) {...}
```

is treated roughly like

```
Object e = E; // where e is a variable name not already used
//@ assert e != null;  // but with Null warning on failure
//@ assert \lockset[e] | \max[\lockset] < e;  // but with Deadlock warning
Insert e into \lockset;
try
  {...}
finally
  Restore former value of \lockset;
```

The disjunct `\lockset[e]` in the test for potential deadlocks reflects the fact that Java locks are ``reentrant'', that is, that a thread that already holds a lock may acquire the lock again without deadlocking [*JLS*, 17.5].

When checking a `synchronized` method declaration, ESC/Java acts as if the body of the method were wrapped in `synchronized` statement, as described in [*JLS*, 8.4.3.5]. That is, ESC/Java checks an instance method

```
synchronized T m(parameters) {body}
```

as if it had been written

```
T m(parameters) {synchronized (this) {body}}
```

and ESC/Java checks a `static` method

```
static synchronized T m(parameters) {body}
```

in a class `C` as if it had been written

```
static T m(parameters) {synchronized (C.class) {body}}
```

When a routine that already holds a lock acquires another one, ESC/Java needs some way of knowing that the first lock precedes the second in the lock order. One way to give ESC the needed information is by supplying an `axiom` pragma (see section 2.4.2) about the lock order, as in the following example:

```
public class Tree {
  public /*@ monitored */ Tree left, right;
  public /*@ monitored non_null */ Thing contents;

  //@ axiom (\forall Tree t; t.left != null ==> t < t.left);
  //@ axiom (\forall Tree t; t.right != null ==> t < t.right);

  Tree(/*@ non_null */ Thing c) {
    contents = c;
  }

  //@ requires \max(\lockset) <= this;
  public synchronized void visit() {
     contents.mungle();
     if (left != null) left.visit();
     if (right != null) right.visit();
  }
}
```

Fine points

Note that the axioms above are inconsistent if a (so-called) `Tree` can in fact be cyclic. Note also that inconsistency or incompleteness can arise from the possible mutation of variables mentioned in the axioms, namely the fields `left` and `right`. For example, given the axioms above, ESC/Java will generate a spurious `Deadlock` warning for the following method:

```
  //@ requires \max(\lockset) <= this;
  public synchronized void wiggleWoggle() {
    // Perform a rotation on this.right (but give up and just
    // return if this.right or this.right.left is null):
    //
    //   this              this
    //   /  \              /  \
    // ...   x           ...   v
    //      / \    -->        / \
    //     v   y             u   x
    //    / \                   / \
    //   u   w                 w   y
    //
    { Tree x = this.right;
      if (x == null) return;
      synchronized (x) {
```

```
        Tree v = x.left;
        if (v == null) return;
        synchronized (v) {
          x.left = v.right;
          v.right = x;
          this.right = v;
        }                       // line (a)
      }
    }
    // Undo the rotation:
    { Tree v = this.right;
      synchronized (v) {            // line (b)
        Tree x = v.right;
        if (x != null) {            // line (c)
          synchronized (x) {        // line (d)
            v.right = x.left;
            x.left = v;
            this.right = x;
          }
        }                           // line (e)
      }
    }
  }
```

The problem is that the axiom is assumed to apply at the start of the routine, and thus to apply to the values of `.left` and `.right` at the start of the routine. According to the lock order thus defined, the lock acquired at the line `(d)` would precede that acquired at `(b)`. [Despite these caveats, our experience with ESC for Modula 3 suggests that axioms like the ones above will do the right thing surprisingly often and rarely cause problems.]

The preceding example also illustrates a possible source of unsoundness in ESC/Java's treatment of race detection. If the lines marked `(c)` and `(e)` are deleted, and if deadlock checking is left disabled, then ESC/Java will accept line `(d)` without complaint, ignoring the possibility that some other thread might have taken advantage of the window between lines `(a)` and `(b)` to synchronize on ``v'' and set its `.right` field to `null`.

---

3 Specification expressions

While specification expressions are generally similar to Java expressions, there are a number of differences, described below. Section 3.0 describes a slight extension to the Java type system. Section 3.1 describes notations allowed in Java expressions but forbidden in specification expressions. Section 3.2 describes additional notations allowed only in specification expressions. Section 3.3 describes how the rules for scoping, name resolution and access control in specification expressions differ from those in Java.

Specification expressions are never actually evaluated when a Java program is run, and ESC/Java will not produce specific warnings about specification expressions whose evaluation (if they were evaluated) might dereference `null`, access arrays out of bounds, etc. Rather, the meanings of such expressions (for example, *`E`*`.f` where *`E`* , if it were actually evaluated, would evaluate to `null`) are unspecified functions of (the meanings of) their subexpressions. In most cases, attempts to prove things about such unspecified values will fail, thus giving

rise to warnings of some sort, though alas not the clearest warnings that one could hope for. For example, given the program fragment

```
//@ assume \nonnullelements(a);
//@ assert a[j] != null;
```

in a place where `j` might be negative, ESC/Java will produce an `Assert` warning (section 4.2) rather than an `IndexNegative` warning (section 4.6).

3.0 Specification types

Like Java expressions, specification expressions have types (which we call *specification types*). A specification type is either a Java type, or one of the special types `\TYPE` or `\LockSet` (or an array of special types, for example `\TYPE[][]`). (In the current ESC/Java, the programmer cannot mention `\LockSet` explicitly.)

3.1 Restrictions

Specification expressions must be free of subexpressions that, in general, may have side effects. In particular, specification expressions may not contain any:

- assignment (=, +=, etc.),
- pre-increment, pre-decrement, post-increment, post-decrement (`++` or `--`),
- array or object creation (`new`), or
- method invocation.

Method invocations are forbidden in specification expressions even when the method is known to have no side effects.

The next section describes additional constructs that are allowed in specification expressions beyond those allowed in Java expressions. Some of these constructs have restrictions on their use. We describe these restrictions together with the descriptions of the constructs to which they apply.

3.2 Additions

3.2.0 `\type`

An expression of the form `\type(`*`T`*`)` where *`T`* is a specification type (see section 3.0), is a specification expression of type `\TYPE`. It denotes the type *`T`*.

3.2.1 `\typeof`

A specification expression of the form `\typeof(`*`E`*`)` where *`E`* is a specification expression whose type is a reference type is a specification expression of type `\TYPE`. It denotes the dynamic type of the value of *`E`*. The value of `\typeof(`*`E`*`)` is unspecified when *`E`* evaluates to `null`.

3.2.2 `\elemtype`

An expression of the form `\elemtype(`*`E`*`)` where *`E`* is a specification expression of type `\TYPE` is a specification

expression of type `\TYPE`. If *E* denotes an array type *T*`[]`, then `\elemtype(`*E*`)` denotes *T*. Otherwise the value of `\elemtype(`*E*`)` is unspecified.

3.2.3 Subtype: `<:`

An expression of the form *S* `<:` *T* where *S* and *T* are specification expressions of type `\TYPE` is a specification expression of type `boolean`. It denotes that *S* is a subtype of *T* (including the case where *S* is equal to *T*). The operator `<:` has the same binding power as the relational operators `<`, `>`, `<=`, and `>=`.

3.2.4 Examples involving `\TYPE`, `\type`, `\typeof`, `\elemtype`, and `<:`

Suppose file `T.java` contains the following declaration

```
class T {

  //@ requires a != null && 0 <= i & i < a.length;
  static void storeObject(T[] a, int i, T x) {
    a[i] = x;
  }
  ...
```

When checking the body of the `storeObject` method, ESC/Java will produce an `ArrayStore` warning (section 4.1) for the assignment

```
    a[i] = x;
```

The problem is that, so far as ESC/Java can tell, the actual type of `a` at the time of the call might be `S[]`, where `S` is some proper subtype of `T`, and `x` might not be of type `S`. Consequently, the attempt to store `x` into `a[i]` might give rise to an `ArrayStoreException` [*JLS*, 10.10, 15.25.1], which ESC/Java treats as an error.

The `ArrayStoreException` cannot arise in the method if the parameter `a` always has actual type exactly `T[]`. The programmer can express this intention with the pragma

```
//@ requires \elemtype(\typeof(a)) == \type(T);
```

or, equivalently, with the pragma

```
//@ requires \typeof(a) == \type(T[]);
```

Note that

```
//@ requires \typeof(a) == \type(T)[];
```

would not be legal, since the form *N*`[]` makes sense only when *N* is a specification type, not a specification expression of type `\TYPE`.

A weaker, but still sufficient, precondition for avoiding the `ArrayStoreException` would be to require that the array `a` merely have an actual element type adequate to hold the value of `x`. This precondition is expressed by the pragmas

```
//@ requires x == null || \typeof(x) <: \elemtype(\typeof(a));
```

Note that the pragma

```
//@ requires x == null || x instanceof \elemtype(\typeof(a));
```

is not legal, since the right hand argument of `instanceof` must be a type, not a specification expression of type `\TYPE`.

3.2.5 `\lockset`

The special identifier `\lockset` is a specification expression of type `\LockSet`. It denotes the set of locks held by the current thread.

3.2.6 Membership in lock sets: `[ ]`

An expression of the form *`S`*`[`*`L`*`]` where *`S`* is a specification expression of type `\LockSet` and *`L`* is a specification expression of a reference type is a specification expression of type `boolean`. It denotes that *`L`* is a member of *`S`*.

3.2.7 Lock order: < and <=

Within specification expressions, the relations < and <= are extended to apply to locks as well as numbers. The order they refer to is called the lock order. For some examples of use of the lock order, see section 2.7.2.

3.2.8 `\max`

An expression of the form `\max(`*`S`*`)` where *`S`* is a specification expression of type `\LockSet` is a specification expression of type `Object`. It denotes the maximum element of *`S`* in the lock order.

To insure that `\max` is always well defined, ESC/Java assumes that lock sets are always nonempty and that their elements are always totally ordered by the lock order <. Since locks must be acquired in increasing order, and since there is no way to write a program that releases the fictitious maximum element of the lock set of a thread that really holds no locks, the preceding assumptions are invariantly true if they are true initially.

3.2.9 Implication: ==>

An expression of the form *`E`* `==>` *`F`* where *`E`* and *`F`* are specification expressions of type `boolean` is a specification expression of type `boolean`. It denotes the condition that *`E`* implies *`F`*, that is, `(!(`*`E`*`)) | `*`F`*. The operator `==>` has less binding power than `&&` and `||`, but binds more strongly than the ternary conditional operator `? :t`. (The binding precedence of operators in Java is implicit in the grammar for expressions [*JLS*, 19.12].)

3.2.10 `\forall`

An expression of the form `(\forall` *`T V`*`;` *`E`*`)` where *`T`* is a specification type (see section 3.0), *`V`* is a nonempty comma-separated list of identifiers (called *bound variables*), and *`E`* is a specification expression of type `boolean` is a specification expression of type `boolean`. It denotes that *`E`* is true for all substitutions of values of type *`T`* for the bound variables. If *`T`* is a reference type, the quantification ranges only over allocated

objects that are instances of *T*. Note that this excludes `null`. If *T* is either of the types `int` or `long`, then the quantification ranges over all mathematical integers, regardless of whether they are in the ranges of possible values for Java variable of those types.

Fine points

Just as Java forbids declaration of an identifier as a local variable within the scope of a parameter or local variable of the same name [*JLS*, 14.3.2], so ESC/Java forbids declaration of an identifier as a bound variable within the scope of a parameter, local variable, or bound variable of the same name. ESC/Java also forbids declaration of `\lockset` or `\result` as a bound variable. The same restriction applies to variables bound by `\exists` (section 3.2.11).

If a specification expression *E* has an application of `\forall` as a (not necessarily proper) subexpression, then *E* may occur only in one of the following contexts:

- as an (entire) argument to one of the following operators:
  - the ESC/Java implication operator ==> (see section 3.2.9)
  - the Java conditional operators `&&` and `||` [*JLS*, 15.22, 15.23]
  - the Java logical operators `&`, `^`, and `|` [*JLS*, 15.21.2]
  - the Java boolean equality operators == and `!=` [*JLS*, 15.20.2]
  - the Java logical complement operator `!` [*JLS*, 15.14.6]
- as the (entire) body of
  - a parenthesized expression `(`*E*`)`,
  - a quantified expression `(\forall ...; `*E*`)` or `(\exists ...;`*E*`)` (see section 3.2.11), or
  - a labeled expression `(\lblpos n `*E*`)` or `(\lblneg n `*E*`)` (see section 3.2.16)
- as a top-level boolean specification expression (that is, not properly contained by another specification expression) in a pragma.

The same restriction applies to specification expressions containing applications of `\exists` (section 3.2.11) or of `\lblneg` or `\lblpos` (section 3.2.16). In particular, a quantified or labeled expression may not occur in an argument to the ternary conditional operator `? :` and this restriction cannot be evaded by use of the `(boolean)` cast operator. For example, specification expressions of the form

```
((boolean)(\exists ...)) ? ... : ...
```

are not allowed. A consequence of these restrictions is that all legal specification expressions containing quantified or labeled subexpressions are of type `boolean`.

Note that the preceding rule forbids quantified expressions (and labeled expressions) within arguments to `\old`. [Future versions of ESC/Java may liberalize this restriction, with quantifications on reference types inside `\old` ranging over objects allocated in the pre-state.]

3.2.11 `\exists`

An expression of the form `(\exists `*T V*`; `*E*`)` where *T* is a specification type (see section 3.0), *V* is a nonempty comma-separated list of identifiers (called *bound variables*), and *E* is a specification expression of type `boolean` is a specification expression of type `boolean`. It denotes that *E* is true for some substitution of values of type *T* for the bound variables. If *T* is a reference type, the quantification ranges only over allocated

objects that are instances of *T*.

Fine points

See section 3.2.10 for restrictions on bound variable names and restrictions on places where quantified formulas may appear.

3.2.12 `\nonnullelements`

An expression of the form `\nonnullelements(`*A*`)` where *A* is a specification expression of an array type is a specification expression of type `boolean`. It is equivalent to

```
A != null && (\forall int i; 0 <= i & i < A.length ==> A[i] != null)
```

Expressions of the form above came up fairly frequently in our early experiments with ESC/Java, and we found writing them sufficiently tedious to justify the introduction of a special notation.

Example

Consider the `main` method of a program:

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

The usual way for `main` to be invoked is with the value of `args` derived from the command line by the Java interpreter, in which case `args` will be non-null, and all its elements will be non-null. However, it is legal for a Java program to contain explicit calls to `main`, and the value of `args` supplied by such a call might in some cases be either `null` or an array containing `null` as an element.

It is often helpful to annotate `main` as follows:

```
//@ requires \nonnullelements(args);
static public void main (String[] args) {...}
```

Given this annotation, (1) ESC/Java will assume, when checking the body of `main`, that `args` and all its elements are non-null, and (2) ESC/Java will check that any explicit calls to `main` supply a non-null argument with only non-null elements.

Fine points

Since an application of `\nonnullelements` does not explicitly include a quantifier, it may be used as an argument to `? :` and may occur inside an argument of `\old`.

Note that if *A* is of type *T*`[][]`, then `\nonnullelements(`*A*`)` implies that *A*`[`*i*`]` is non-null (if *i* is in bounds), but not necessarily that any *A*`[`*i*`][`*j*`]` is non-null.

3.2.13 `\fresh`

An expression of the form `\fresh(`*E*`)` where *E* is a specification expression of a reference type is a specification expression of type `boolean`. In a postcondition, it denotes that *E* is non-null and was not allocated in the

pre-state of the routine call.
(See also section 2.3.2.)

3.2.14 `\result`

Within a normal postcondition or a modification target of a non-`void` method, the special identifier `\result` is a specification expression whose type is the return type of the method. It denotes the value returned by the method. `\result` is allowed only within an `ensures`, `also_ensures`, `modifies`, or `also_modifies` pragma that modifies the declaration of a non-`void` method.

Fine points

Note that although `\result` may occur within a modifies pragma, it is not itself a specification designator (see section 2.3.1). Thus

```
modifies \result
```

is never a legal pragma. However pragmas such as, for example,

```
modifies \result.f
```

or

```
modifies \result[i]
```

may be legal, depending on the type of `\result`.

3.2.15 `\old`

In a postcondition, an expression of the form `\old(`*`E`*`)` where *`E`* is a specification expression is a specification expression of the same type as *`E`*. It denotes the same thing as *`E`* except that (1) any occurrences in *`X`* of a target field (see section 2.3.1.0) of the routine is interpreted according to the pre-state value of the field, and (2) if any modification target of the routine has the form *`X`*`[`*`i`*`]` or *`X`*`[*]`, then *all* array accesses within *`E`* are interpreted according to the pre-state contents of arrays. An expression of the form `\old(`*`E`*`)` may occur only in an `ensures`, `exsures`, `also_ensures`, or `also_exsures` pragma. The argument *`E`* may not itself include any uses of `\old` or `\fresh`.

Fine points

In a normal postcondition of a non-`void` method, the special identifier `\result` (section 3.2.14) may occur within an argument to `\old`. In this context, `\result` denotes, as usual, the value returned by the method, despite the fact that the returned value may not even be allocated in the pre-state (in which case the meaning of a field access `\result.`*`f`* or an array access `\result[`*`i`*`]` is unspecified). Similarly any occurrence of `this` (explicit or implicit) in a normal postcondition of a constructor denotes the constructed object, even within an argument of `\old`.

The following (correctly annotated) code example illustrates the semantics of `\old`.

```
class C {
```

```
  static C x, oldx, y;
  int f;
  static int oldxf;
  static int[] a, olda, b;
  static int oldai;
  static int i;

  //@ requires x != null & y != null;
  //@ requires a != null && 0 <= i & i < a.length;
  //@ modifies oldx, oldxf, x, x.f, olda, oldai, a, a[i], i;
  //@ ensures oldx == \old(x)
  //@ ensures oldxf == \old(x.f);
  //@ ensures \old(x).f == \old(x.f) + 1;
  //@ ensures (\exists C z; z == x & \old(z.f) == \old(y.f));
  //@ ensures olda == \old(a) & oldai == \old(a[i]);
  //@ ensures \old(a)[\old(i)] == \old(a[i]) + 1;
  static void m() {
    oldx = x;
    oldxf = x.f;
    x = y;
    oldx.f++;
    olda = a;
    oldai = a[i];
    a = b;
    olda[i]++;
    i++;
  }
}
```

Note the distinctions between `\old(x.f)` and `\old(x).f` and between `\old(a)[\old(i)]` and `\old(a[i])`. Note also that while ESC/Java does not allow the notation `x.\old(f)` to mean ``the original `f` field of the current value of `x`'' in the pragma, the same effect is achieved by the expression `\old(z.f)` within the pragma

```
//@ ensures (\exists C z; z == x & \old(z.f) == \old(y.f))
```

For further discussion of `\old`, including the interaction of `\old` and `modifies`, see sections 2.3.2 and 2.3.3.

3.2.16 `\lblneg` and `\lblpos`

[*This section may be skipped on first (and second) reading. It describes, incompletely, a feature of included mainly for use by the implementers.*]

An expression of the form `(\lblneg` *n E*`)` or `(\lblpos` *n E*`)` where *E* is a specification expression of type `boolean` and *n* is an identifier (called an expression label) is a specification expression of type `boolean`. Logically, the *labeled expression* `(\lblneg` *n E*`)` [or`(\lblpos` *n E*`)`] denotes the same thing as *E*, but when ESC/Java issues a warning, the warning message will mention the label *n* if, in the execution path associated with the warning, the expression *E* would evaluate to `false` [resp., `true`] at a point where the containing pragma is ``relevant'' and in circumstances where the value of the expression *E* is ``relevant'' to the pragma as a whole. The

details of the (heuristic) definition of ``relevant'' are beyond the scope of this manual and are subject to change.

Example

Suppose file `C.java` contains the following code:

```
class C {

  //@ requires (\lblpos fee i < 5) || (\lblpos fie i > 10);
  //@ ensures (\lblneg foe \result != 5) && (\lblneg fum \result > 0);
  int m(int i) {
      return i+1;
  }

}
```

Then output from the command `escjava C.java` includes the following warning:

```
C.java:7: Warning: Postcondition possibly not established (Post)
  }
  ^
Associated declaration is "C.java", line 4, col 6:
  //@ ensures (\lblneg foe \result != 5) && (\lblneg fum \result > 0 ...
      ^
Execution trace information:
    Executed return in "C.java", line 6, col 6.

Counterexample labels:
    fum    fee
```

The label `fee` comes from the positively labeled expression `(\lblpos fee i < 5)` in the `requires` pragma for method `m`, and tells us that in the execution path associated with the warning, `m` is called with an argument `i` such that `(i < 5) == true`. The label `fum` comes from the negatively labeled expression `(\lblneg fum \result > 0)` in the `ensures` pragma for `m`, and tells us that in the execution path associated with the warning, `m` returns a result `\result` such that `(\result > 0) == false`.

Fine points

Expression labels are in their own name space, so they never conflict with or hide any other kinds of identifiers.

Labeled expressions are allowed only in those places where quantified expressions are allowed (see section 3.2.10).

Limitation: There are many situations in which labels are ambiguous. For example, suppose file `D.java` contains the code:

```
class D {

  int a, b;
```

```
  //@ invariant (\lblpos foo a >= 0 & b >= 0) || (\lblpos bar a < 0 & b
<0);

  //@ requires x != null && y != null;
  D(D x, D y) {
    a = x.a;
    b = y.b;
  }

}
```

Then the command `escjava D.java` yields the warning:

```
D.java:10: Warning: Possible violation of object invariant (Invariant)
  }
  ^
...
Counterexample labels:
    bar    foo
```

There is no way to tell from this message whether ESC/Java is warning about the case where `a` has nonnegative `x` and `y` fields and `b` has negative `x` and `y` fields, or about the case where `a`'s fields are negative and `b`'s are nonnegative. (However, for this particular example, the `-counterexample` switch, described in appendix A, would resolve the ambiguity.)

3.2.17 `owner`

The standard `.spec` file (see section 5.1.2) for the class `java.lang.Object` gives every object `O` a ghost field `O.owner` of type `Object`:

ESC/Java input from file .../java/lang/Object.spec:

```
...
//@ ghost Object owner;
...
```

The intended use of this field is for situations is which the programmer wishes to specify that some field `f` is *unshared*. For example. if a class `T` with an instance field `f` declares the instance invariant

ESC/Java input from file .../T.java:

```
...
//@ invariant f.owner == this
...
```

then it follows that, at any point where the invariant holds, we cannot have `x.f == y.f` for two distinct objects `x` and `y` of type `T` (since the conditions `x.f.owner == x`, `y.f.owner == y`, together with `x.f == y.f` would imply `x == y`).

Example

Here is an example of a situation in which it is useful to specify that a field is unshared. [**Note:** Understanding this example may require more than a cursory reading. While we usually relegate such material to sections marked ``**Fine points**", the scenario described here is of a sort that most ESC/Java users are likely to encounter as soon as they try to check code with invariants of any complexity, specifically invariants involving both (1) quantification (including the quantification implicit in all instance invariants) and (2) indirect references like `x.f.g` or `x.a[i]` (including cases where an expression like `f.g` or `a[i]` implicitly means `this.f.g` or `this.a[i]`). A few minutes working through the details with pen or pencil in hand will lead to clear understanding, and the time will be well spent.]

Consider the following class, whose instances represent stacks of objects (to keep the example simple, we put a fixed limit of 10 on the size of a stack):

Input from file ObjStack.java:

```
 1: class ObjStack {
 2:   /*@ non_null */ Object [] a;
 3:   //@ invariant a.length == 10;
 4:   //@ invariant \elemtype(\typeof(a)) == \type(Object);
 5:   int n;  //@ invariant 0 <= n & n <= 10;
 6:   //@ invariant (\forall int i; n <= i & i < 10 ==> a[i] == null);
 7:
 8:   ObjStack() {
 9:     n = 0;
10:     a = new Object[10];
11:   }
12:
13:   //@ requires n < 10;
14:   void Push(Object o) {
15:     a[n++] = o;
16:   }
17:
18:   //@ requires n > 0;
19:   Object Pop() {
20:     Object o = a[--n];
21:     a[n] = null;
22:     return o;
23:   }
24:
25: }
```

The elements of a stack `x` are the first `x.n` elements of the array `x.a`. To avoid retaining pointers to garbage, we insure that the remaining elements of `x.a` are `null`, as specified by the `invariant` pragma on line 6.

If we run ESC/Java on the source above, it produces the following complaint:

```
ObjStack: Push(java.lang.Object) ...
ObjStack.java:16: Warning: Possible violation of object invariant
(Invariant)
  }
```

```
  ^
Associated declaration is "ObjStack.java", line 6, col 6:
  //@ invariant (\forall int i; n <= i & i < 10 ==> a[i] == null)
      ^
Possibly relevant items from the counterexample context:
  …
  brokenObj<2> != this
  …
(brokenObj* refers to the object for which the invariant is broken.)
```

(For a basic explanation of counterexample contexts, see appendix A.) What is going on here is that ESC/Java has found a scenario in which the `Push` method fails to maintain the instance invariant on line 6. The line

```
brokenObj<1> != this
```

tells us that the object for which the invariant is broken is not the object (`this`) whose `Push` method is called. The problem is that, so far as ESC/Java can tell, there may be some object, call it `brokenObj`, such that the following conditions hold at the start of the execution of `Push`:

- `brokenObj != this`
- `brokenObj.a == this.a`
- `brokenObj.n == this.n`

Now consider the effect of the line

```
15:      a[n++] = o;
```

The evaluation of `n++` increases `this.n`, but leaves `brokenObj.n` unchanged. On the other hand, the array store does set `brokenObj.a[n]` (where `n` denotes the value before the increment) to `o`. Thus, after line 16, we have

- `brokenObj.a[brokenObj.n] == o`

so that if `o != null` (as may very well be the case), the instance invariant on line 6 is violated for `brokenObj`.

In fact, as illustrated by the example above, the correctness of the `Push` method of `ObjStack` depends on an implicit design decision: No two distinct `ObjStack`'s ever share the same `a` field. (The correctness of `Pop` also depends on this design decision, though not in a way that is checked by ESC/Java given the annotations in the example above.) The user can take advantage of the `owner` field to communicate this design decision to ESC/Java by adding the pragma:

```
7:   //@ invariant a.owner == this
```

(Given this invariant, it is no longer possible to have a scenario in which the conditions

- `brokenObj != this`
- `brokenObj.a == this.a`

both hold.) Having added this invariant, the user must also supply a `set` pragma in order to guarantee that the

constructor for `ObjStack` establishes the invariant:

```
 9:   ObjStack() {
10:     n = 0;
11:     a = new Object[10];
12:     //@ set a.owner = this;
13:   }
```

Without the `set` pragma, ESC/Java will generate a warning that the constructor fails to establish the invariant `a.owner == this`.

Fine points

Just as all constructors have the implicit postcondition `\fresh(\result)` (see section 2.3.2), all constructors (including the implicit constructors for arrays) have the implicit postcondition `this.owner == null`. In the example above, this implicit postcondition guarantees that `this.a.owner == null` after line 11. If `this.a.owner` could have had some non-null value (call it `brokenObj`) after line 11, then the `set` pragma might cause the invariant on line 7 to be violated for `brokenObj`.

Another way to guarantee that no two `ObjStack`'s share the same `a` field would be to include in the declaration of `ObjStack` the static invariant

```
(\forall ObjStack x, y; x != y ==> x.a != y.a)
```

The instance invariant `a.owner == this` is not only more succinct than the static invariant above, but also stronger. To see why, suppose we had another class `Foo` which declared a field `b` and the instance invariant `b.owner == this`. Then the invariants `a.owner == this` (for `ObjStack`) and `b.owner == this` (for `Foo`) together would imply not only

```
(\forall ObjStack x, y; x != y ==> x.a != y.a)
```

and

```
(\forall Foo x, y; x != y ==> x.a != y.b)
```

but also

```
(\forall ObjStack x; (\forall Foo y: (Object)x != y ==> x.a != y.b))
```

If, for example, ESC/Java were checking code that modified a component of the `b` field of some `Foo foo`, and if that code occurred in a scope where the invariant on line 6 of `ObjStack.java` were considered heuristically relevant (see the fine points in section 2.4.1), then ESC/Java would need this last consequence in order to eliminate from consideration scenarios in which the invariant on line 6 might become violated for some `ObjStack brokenObj` with `brokenObj.a == foo.b`.

See [LS99] for a different approach to specifying that certain fields are unshared.

3.3 Scoping, name resolution, and access control in specification expressions

The rules for scoping, accessibility, and resolution of names in specification expressions differ in several ways from those for Java expressions. We have described most of these differences elsewhere in this manual, but repeat them here to have them collected in one place.

Applications of the ESC/Java operators `\typeof`, `\elemtype`, `\max`, `\nonnullelements`,`\fresh`, and `\old` are parsed like method invocations, with the operator taking the role of the method name. Of course, applications of these operators are allowed in specification expressions even though ordinary method invocations are forbidden, as stated in section 3.1.

Bound variables introduced by `\forall` and `\exists` are scoped like Java local variables, as are the variables declared to name exceptions in `exsures` and `also_exsures` pragmas. For example, in a quantified expression `(\forall int k; a[k] == 0)`, the declaration `int k;` introduces the bound variable `k`, whose scope is the *SpecExpr* `a[k] == 0`. Furthermore, just as Java forbids declaration of an identifier as a local variable within the scope of a parameter or local variable of the same name [*JLS*, 14.3.2], so ESC/Java forbids declaration of an identifier as a bound variable within the scope of a parameter, local variable, or bound variable of the same name. Consequently, the quantified expression `(\forall int k;  a[k] == 0)` cannot occur in a scope where there is already a parameter, local variable, or bound variable named `k`.

The scoping rules of routine parameters, fields, `this`, and `super` are slightly different from those in Java. In particular:

- When a *SpecExpr* occurs in a modifier pragma (section 2.0.1) applied to a field, names in the *SpecExpr* are resolved as if the *SpecExpr* were part of the initializer of the field. Consequently, if the field is an instance variable, then `this` and `super` can be mentioned and any unqualified field name `f` is a synonym for `this.f`. While Java forbids field initializers from using other fields that are declared later in textual order (even though they are in scope), this restriction does not apply to ESC/Java pragmas (compare the examples in sections 2.5.1 and 2.7.0 to those in [*JLS*, 8.3.2.1, 8.3.2.2]).
- When a *SpecExpr* occurs in a modifier pragma applied to a routine declaration, names in the *SpecExpr* are resolved as if the *SpecExpr* were placed at the beginning of the routine body. Consequently, the names of the routine's parameters are in scope. Furthermore, if the routine is an instance method, then `this` and `super` can be mentioned, and if the routine is a constructor, then `this` and `super` can be mentioned in `ensures` pragmas. As usual, wherever `this` can be mentioned, an unqualified field name `f` is a synonym for `this.f`.
- When a *SpecExpr* occurs in a modifier pragma applied to an abstract method declaration, names in the *SpecExpr* are resolved as if the method could have a body and the *SpecExpr* were placed there. That is, the names of the method's parameters are in scope, `this` can be mentioned, any unqualified field name `f` is a synonym for `this.f`, and if the abstract method declaration occurs in a class (rather than in an interface) then `super` can be mentioned.
- In any other *SpecExpr*, the same parameters, local variables, and fields are in scope as in the Java context where the annotation containing the *SpecExpr* occurs; `this` can be mentioned (and any unqualified field name `f` is a synonym for `this.f`) if the *SpecExpr* occurs in a Java context where `this` can be mentioned; and `super` can be mentioned if the *SpecExpr* occurs in a Java context where `super` can be mentioned. Moreover, in a *SpecExpr* that occurs in a declaration pragma (see section 2.0.1) in a class, `this` and `super` can be mentioned and any unqualified field name `f` is a synonym for `this.f`.

A `ghost` field *`f`* (see section 2.6.0) is in scope in pragmas wherever an actual Java member of the class containing the `ghost` pragma declaring *`f`* would be in scope. The rules for name resolution are different for ghost variables than for ordinary variables. Java variables (if in scope) hide conflicting ghost variables of the same name

regardless of their points of declaration, and regardless of whether the Java variable is accessible. Ghost variables, on the other hand, hide neither Java variables nor other ghost variables. For example, given the declarations

Input form file I.java:

```
interface I {
  //@ ghost public int i;
  //@ ghost public int j;
  …
}
```

Input from file C.java:

```
class C {
 private int i;
 //@ ghost public int j;
 …
}
```

Input from file D.java:

```
class D extends C implements I {…}
```

a reference to `i` in a pragma in `D.java` would resolve to the real field of `C` rather than the ghost field of `I` (and would then generate an error because that field is not accessible in `D`), while a reference to `j` in a pragma in `D` would be ambiguous (since `C` and `I` both declare ghost fields named `j` and neither hides the other).

Note: The preceding rules about ghost variables, and the restriction against reusing a name in scope as a ghost field name (section 2.6.0), are intended to reduce the number of situations where a name unambiguously means one thing in Java code and unambiguously means a different thing in an adjacent pragma. Unfortunately, some such situations still remain. For example, if a ghost field and a class have the same name `N`, then the expression `N.f` might refer to a static field of class `N` in Java code, while meaning `this.N.f` in a nearby annotation.

The rules that make names accessible in specification expressions (*spec-accessible*) are in some cases more liberal that Java's access control rules [*JLS*, 6.6]. In particular:

- If a variable's declaration is annotated with a `spec_public` (section 2.5.0) pragma, then the variable is spec-accessible wherever it would have been spec-accessible if it had been declared `public`.
- A variable declared as `protected` is spec-accessible in the package where it is declared and in any subclass of the class where it is declared (without the additional restriction in [*JLS*, 6.6.2]).

Of course there are also some cases where variables--for example, ghost variables (section 2.6.0)--are spec-accessible but not Java accessible simply because they are not in scope in Java.

Finally, the label *Identifier* in a `\lblpos` or `\lblneg` expression is part of a separate name space. The label does not become available for use inside the *SpecExpr*. A label is permitted to have the same name as an identifier already in scope or as a label in an enclosing labeled expression. If a label has the same name as an identifier already in scope, it does not hide that identifier.

---

4 Warnings

ESC/Java issues warnings for conditions that it regards as run-time errors, and that, so far is it can tell, might actually occur at run-time.

The potential conditions that give rise to some ESC/Java warning types (specifically `Null`, `IndexNegative`, `IndexTooBig`, `Cast`, `ArrayStore`, `ZeroDiv`, and `NegSize`) are conditions that would be detected by the Java run-time system and give rise to exceptions (specifically, `NullPointerException`, `IndexOutOfBoundsException`, `ClassCastException`, `ArrayStoreException`, `ArithmeticException`, and `NegativeArraySizeException`). The current ESC/Java regards these conditions as run-time errors, and generates warnings for them even if the program actually catches the resulting exceptions.

> **Fine point:** In some cases multiple warnings may arise from the same cause. For example, if a variable is dereferenced in multiple arms of a `switch` statement but is not dereferenced before the `switch` statement, and ESC/Java cannot confirm that the variable is non-null, then it will issue a warning for the first dereference of the variable in each arm of the `switch` statement. In order to reduce the likelihood of flooding the user with redundant warnings, ESC/Java will issue at most 10 warnings for any method or routine before moving on to the next routine. Users can change the maximum number of warnings per routine by setting the `PROVER_CC_LIMIT` environment variable (see the `escjava(1)` man page).

Section 4.0 describes the parts of ESC/Java warning messages. The remaining subsections of this section describe all the types of warnings issued by the current ESC/Java. [*These descriptions may be skipped on first reading, or until the reader is confronted with an ESC/Java warning whose meaning or cause is unclear.*]

A recommended discipline for using ESC/Java is to annotate your program sufficiently so that ESC/Java produces no warnings, and in this process to resort to the use of `nowarn` or `assume` pragmas only in cases where other alternatives are impractical. Below we include occasional tips about pragmas that might be added in response to particular warnings. The `-suggest` command-line option (see section 5.1.1 or the `escjava(1)` man page) causes ESC/Java to offer suggestions (of varying quality) for pragmas that might be added in response to some warnings.

4.0 Parts of ESC/Java warning messages

The *primary part* of each ESC/Java warning message gives a brief description of the kind of condition being warned of, including a parenthesized ``warning type" name, and indicates a source code location--the ``dynamic location" of the warning--for the control point at which the condition potentially occurs.

For some warning types, the message additionally indicates the source code location of (the first character of the initial keyword of) the pragma--the warning's *associated pragma*--that causes ESC/Java to regard the condition as a run-time error.

If the warning is an `Invariant` warning, the message will usually include a *partial counterexample context*, which may be of help to the user to tell which object that might--so far has ESC/Java can determine--have its one of its invariants violated and why.

An ESC/Java warning message may include an *execution trace* listing interesting (see the fine points below) control decisions on some execution path that--so far as ESC/Java can determine--may plausibly lead to the run-time error mentioned in the warning.

Finally, an ESC/Java might list some labels that occur in pragmas an are relevant to the scenario being warned about. (See section 3.2.16 for some examples.)

Example

Suppose the file `C.java` contains the following class declaration.

```
 1: class C {
 2:
 3:  int i;
 4:  int[] x;
 5:   /*@ invariant i > 0
 6:       ==> x != null
 7:   */
 8:
 9:   void m (int[] p, int[] q) {
10:     i = 10;
11:     int[] t;
12:     if (p != null) {
13:       t = p;
14:     } else {
15:       t = q;
16:     }
17:     x = t;
18:   }
19: }
```

Then the output from the command

```
escjava C.java
```

includes one warning message.

The primary part of the warning message indicates that the some invariant might not hold when control reaches the end of the method `m`:

```
CC.java:18: Warning: Possible violation of object invariant (Invariant)
  }
  ^
```

The next part of the warning message gives the associated pragma, an `invariant` pragma starting on line 5:

```
Associated declaration is "C.java", line 5, col 6:
  /*@ invariant i > 0 ...
      ^
```

The ellipsis indicates that the associated pragma may continue beyond the part that is shown in the message. (In this case it extends onto the next line of the program.)

Since this is an `Invariant` warning, it includes a partial counterexample context:

```
Possibly relevant items from the counterexample context:
  brokenObj == this
(brokenObj* refers to the object for which the invariant is broken.)
```

Here we see that the object whose invariant might be violated is `this`, that is, the object whose `m` method is being executed.

The final part of the warning message is an execution trace, indicating a scenario in which the right hand side `c` of the assignment on line 12 might in fact evaluate to `null`:

```
Execution trace information:
    Executed else branch in "C.java", line 14, col 11.
```

Fine points

The command-line option `-plainwarning` suppresses output of partial counterexample contexts in `Invariant` warnings.

The command-line option `-counterexample` causes ESC/Java to supply counterexample contexts with all warnings.

The execution trace in a warning mentions the following events on the execution path associated with the warning:

- execution of a `return` statement (provided evaluation of the expression to be returned completes normally)
- execution of a `throw` statement (provided evaluation of the exception to be thrown completes normally)
- execution of a `break` statement
- execution of a `continue` statement
- commencement of any branch of an `if` statement (including the implicit empty `else` branch)
- commencement of any branch of a `switch` statement (including the implicit `default`)
- commencement of evaluation of either arm of a conditional (`? :`) expression
- short-circuit completion of evaluation of a conditional (`&&` and `||`) expression
- exceptional completion of a routine call
- commencement of any iteration of a loop (see the fine points of section 2.4.3 for information about ESC/Java's treatment of loops)
- entrance to a `finally` block, when the associated `try` block terminates with a `throw`, `return`, `break`, or `continue`
- exit from a `finally` block, when the block is entered after a `throw`, `return`, `break`, or `continue` and the body of the `finally` block completes normally (so that the `throw`, `return`, `break`, or `continue` is resumed after the `finally` block)

ESC/Java actually associates some execution path with each warning, but if the execution path associated with a warning includes no events of the kinds listed above, then ESC/Java omits the line

```
Execution trace information:
```

from the warning message.

The command-line option `-notrace` suppresses output of execution traces.

The command-line option `-suggest` causes ESC/Java to accompany certain of its warning messages with suggestions for pragmas that may eliminate those warnings. See section 5.0 for some examples.

4.1 `ArrayStore` warning

An `ArrayStore` warning warns that the control may reach an assignment `A[I] = E` when the value of `E` is not assignment compatible with the actual element type of `A`. (This condition would result in an `ArrayStoreException` being thrown at run time [*JLS*, 10.10, 15.25.1].)

Tip: See section 3.2.4 for discussion of a common cause of spurious `ArrayStore` warnings and examples of annotations to avoid them.

4.2 `Assert` warning

An `Assert` warning warns that control may reach a pragma `assert E` when the value of `E` is `false`.

4.3 `Cast` warning

A `Cast` warning warns that control may reach a cast `(T)E` when the value of `E` cannot be cast to the type `E`. (This condition would result in a `ClassCastException` being thrown at run time [*JLS*, 5.4, 15.15].)

Tip: `Cast` warnings often arise in connection with the use ``container" classes, when the programmer intends a particular container to be used exclusively to hold elements of some particular type `T` (a proper subtype of `Object`) but the methods for extracting elements are declared to return results of type `Object`. The second example in section 2.6.2 shows how the programmer can use a `ghost` variable to express the design decision that the container will hold only instances of `T` (so that objects extracted from the container can always safely be cast to type `T`).

4.4 `Deadlock` warning

A `Deadlock` warning warns that control may reach a `synchronized` statement or a call to a `synchronized` method that would acquire a lock in violation of the locking order. That is, it warns of the possibility that a thread might attempt to acquire a lock `L` when

```
\lockset[L] | \max(\lockset) < L
```

is false.

In the current ESC/Java, `Deadlock` warnings are disabled by default, but can be enabled by use of the command line option `-warn Deadlock`.

The only way that ESC/Java can ever show that the execution of a `synchronized` statement or a `synchronized` method body will not result in a potential locking order violation (and thus in a potential deadlock) is by using information supplied in pragmas. The usual way to supply the necessary information is to

use an axiom pragma to supply information about the locking order, and to use a `requires` pragma to supply information about the locks held on entry to any routine whose body includes a `synchronized` statement or a call to a `synchronized` method.

A `Deadlock` warning has no associated pragma. Typically the warning results from the absence of some pragma that would supply the information needed to show that the locking order is obeyed. While a user might sometimes blame a `Deadlock` warning on a bug in some specific pragma, there is no general mathematical rule for uniquely ascribing such blame.

4.5 `Exception` warning

An `Exception` warning warns that a routine may terminate abruptly by throwing an exception that is not an instance of any type listed in the routine's throws clause.

Note that ESC/Java's treatment of unchecked exception classes [JLS, 11.1] is different from Java's. Java's compile-time checking never requires a throws clause to mention an unchecked exception class. By contrast, when ESC/Java checks the body of a routine *`R`*, it considers the possibility of exceptions being thrown by any `throw` statement in the body of *`R`* and by any call from *`R`* to a routine with a nonempty throws clause, and it issues a warning if (so far as it can determine) it is possible that *`R`* may terminate abruptly with such an exception other than an instance of a type mentioned in *`R`*'s throws clause. (However, ESC/Java does not consider the possibility of exceptions being thrown other than by `throw` statements or in accordance with the throws clauses of called routines.)

Tips: The control point associated with an `Exception` warning is the end of the routine body, rather than the point at which the exception is first thrown. This is technically correct because the potential error is not throwing the exception, but letting the exception escape the routine body without being caught. However, in trying to understand why the warning is being issued and what to do about it, you are more likely to be interested in knowing where the problematic exception might be thrown, as indicated by the execution trace (see section 4.0) in the warning message.

The Java language requires that routines declare all checked exceptions that they might throw [*JLS*, 11.2]. Consequently, ESC/Java's `Exception` warning is of interest only in connection with unchecked exceptions. Even if your own code makes no mention of unchecked exceptions it may call library routines whose throws clauses mention unchecked exceptions.

It may tempting to take the view that unchecked exceptions are unchecked precisely because it is not worthwhile to check for their presence at compile time, and therefore always to run ESC/Java with the `-nowarn Exception` command-line option (see the `escjava(1)` man page for descriptions of ESC/Java command-line options). However, the whole purpose of ESC/Java is to do more sophisticated static checking than compilers do, and you may make better use of its capabilities by employing a less extreme and more fine-grained treatment of unchecked exceptions. Suppose, for example, that your program calls a library method with the declaration

```
/**  Returns the element-wise sum of a and b.  Throws
     a NullPointerException if either a or b is null.
     Throws an IndexOutOfBoundsException if a and b are
     not of the same length.
**/
public static int[] add(int[] a, int[] b)
```

```
  throws NullPointerException, IndexOutOfBoundsException
{ …
}
```

and that it is your intention never to supply arguments that give rise to exceptions (and therefore not to bother with code to detect and handle the exceptions at run time). In this case, you might get some useful checking from ESC/Java by creating a `.spec` file (see section 5.1) containing a declaration for `add` with the `throws` clause removed and a `requires` pragma supplied in its place:

```
/** …
<esc> requires a != null & b != null && a.length == b.length;
</esc>
**/
public static int[] add(int[] a, int[] b);
```

Alternatively, you might supply `exsures` pragmas specifying the conditions under which the exceptions may be thrown:

```
/** ...
<esc> exsures (NullPointerException)
        a == null | b == null;
      exsures (IndexOutOfBoundsException)
        a != null & b != null && a.length != b.length;
</esc>
**/
public static int[] add(int[] a, int[] b);
```

With this last specification, ESC/Java will consider execution paths in which a call to `add` terminates exceptionally, but only if it cannot verify the arguments are non-null and of equal length.

4.6 `IndexNegative` warning

An `IndexNegative` warning warns that control may reach an array access *`A`*`[`*`I`*`]` when the value of the index *`I`* is negative. (This condition would result in an `IndexOutOfBoundsException` being thrown at run time [*JLS*, 11.5.1.1, 15.12.1].)

4.7 `IndexTooBig` warning

An `IndexTooBig` warning warns that control may reach an array access *`A`*`[`*`I`*`]` when *`I`* `>= `*`A`*`.length`. (This condition would result in an `IndexOutOfBoundsException` being thrown at run time [*JLS*, 11.5.1.1, 15.12.1].)

4.8 `Invariant` warning

An `Invariant` warning warns that some object invariant may not hold when control reaches a routine call, or that some object invariant may not hold on exit from the current body. The warning is associated with the `invariant` pragma that gives the potentially violated object invariant.

Tip: An `invariant` warning is normally accompanied by a partial counterexample context describing conditions under which, so far as ESC/Java can determine, the indicated invariant might be violated, for example:

```
Possibly relevant items from the counterexample context:
  ...
  \typeof(brokenObj<3>) == \typeof(this)
  brokenObj<3> != this
  brokenObj<3> != null
  ...
```

In this display, some inflected form of the identifier `BrokenObj` is used to name the object for which the invariant is broken, known as the *broken object*. If, as in the example above, the displayed formulas tell what the broken object is *not* equal to, but don't tell what it *is* equal to, then the likely cause of the warning is that the program modifies some field (call it `f`) of some object (call it `t`) and that ESC/Java hypothesizes that this modification might break the invariant for some other object, for example a hypothetical object `u` such that `u != t` but `u.f == t.f`. If the programmer's intention is that no such sharing of `f` fields can occur, the programmer can communicate this intention to ESC/Java by supplying an appropriate invariant near the declaration of the field, for example:

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

For a more detailed example of the scenario outlined above, see the discussion of the `owner` field and its use in section 3.2.17.

Fine points

The command-line option `-plainwarning` suppresses output of partial counterexample contexts in `Invariant` warnings. This may be useful in cases where ESC/Java's output is read by another program.

In the partial counterexample context in an `Invariant` warning for a constructor, the object being constructed may be named by an inflected form of the identifier `RES`. For example, in the `ObjStack` example from section 3.2.17, we wrote:

> Without the `set` pragma, ESC/Java will generate a warning that the constructor fails to establish the invariant `a.owner == this`.

The warning might be something like:

```
ObjStack.java:12: Warning: Possible violation of object invariant
(Invariant)
  }
  ^
Associated declaration is "ObjStack.java", line 7, col 6:
  //@ invariant a.owner == this;
      ^
Possibly relevant items from the counterexample context:
  brokenObj<1> == RES:9.13
(brokenObj* refers to the object for which the invariant is broken.)
```

Here, the equality `brokenObj<1> == RES:9.13` indicates that the object whose invariant may be violated is the object being constructed.

4.9 `LoopInv` warning

A `LoopInv` warning warns that some loop invariant may not hold at the start of an iteration of a loop (including an iteration in which the The pragma is associated with the `loop_invariant` pragma that gives the potentially violated loop invariant. (For more details of ESC/Java's treatment of loop invariants, see sections 2.4.3 and C.0.1.)

4.10 `NegSize` warning

A `NegSize` warning warns of a possible attempt to allocate an array of negative length. (This condition would result in a `NegativeArraySizeException` being thrown at run time [*JLS*, 15.9].)

4.11 `NonNull` warning

A `NonNull` warning warns of a possible attempt to assign the value `null` to a variable whose declaration is modified by a `non_null` pragma, or to call a routine with an actual parameter value of `null` when the declaration of the corresponding formal parameter is modified by (or inherits) a `non_null` pragma. The warning is associated with the `non_null` pragma that is potentially violated.

Tips: If the right hand side of the assignment indeed never evaluates to `null`, you must somehow communicate the reason to ESC/Java. For some ideas about this, see the tips given in connection with `Null` warnings in section 4.13. Alternatively, it may be that the `non_null` pragma associated with the warning is too strong and should be replaced by an annotation that only requires the affected field or variable to be non-null under certain conditions. See also the comments in section 2.4.0 about `non_null` vs. `invariant` and `requires` pragmas.

4.12 `NonNullInit` warning

A `NonNullInit` warning warns that a constructor may fail to establish a non-null value for an instance field of the constructed object when the declaration of that instance field is modified by a `non_null` pragma. The warning is associated with the `non_null` pragma that is potentially violated.

4.13 `Null` warning

A `Null` warning warns of a possible attempt to dereference `null`, for example, by field access *`O`*`.`*`f`*, an array access *`O`*`[`*`i`*`]`, a method call *`O`*`.`*`m`*`(...)`, a `synchronized` statement `synchronized (O) ...`, or a `throw` statement `throw` *`O`*, where *`O`* evaluates to `null`. (Any of these would result in a `NullPointerException` being thrown at run time [*JLS*, 11.5.1.1, 14.17, 15.10, 15.11, 15.12].)

> **Remark:** *JLS* doesn't say that throwing `null` results in a `NullPointerException`, but experimentation with `javac(5)` and `java(5)` reveals that it does.

Tips: If the expression *`O`* is a formal parameter, consider adding a `non_null` pragma to the parameter's declaration or supplying a `requires` pragma stating that the parameter must be non-null under certain conditions. If *`O`* is a field access *`P`*`.`*`g`*, consider adding a `non_null` pragma to *`g`*'s declaration, supplying an `invariant` pragma stating that *`g`* is non-null under certain conditions, or (if *`P`* involves parameters of the current routine) supplying an appropriate `requires` pragma. If *`O`* is an array access, consider supplying a `requires` or `invariant` pragma using `\nonnullelements` (see section 3.2.12). If *`O`* is a method call, consider annotating

the called method with `ensures \result != null` or `ensures Q ==> \result != null` for some appropriate condition `Q`. See the second example in section 2.6.2 for an example of pragmas guaranteeing that an element extracted from a container will be non-null.

4.14 `OwnerNull` warning

As described in section 3.2.17, every constructor has the implicit postcondition `this.owner == null`. An `OwnerNull` warning warns that a constructor may return an object whose `owner` field is non-null.

4.15 `Post` warning

A `Post` warning warns that a routine body may fail to establish some normal postcondition (on terminating normally) or some exceptional postcondition (when terminating by throwing an exception of a relevant type). The warning is associated with the `ensures`, `exsures`, `also_ensures`, or `also_exsures` pragma that gives the potentially violated postcondition.

Tips: If a `Post` warning seems mysterious, the problem might be that the programmer intended to refer to the post-state value of some field, but forgot to include a `modifies` or `also_modifies` pragma naming that field as a target. See section 2.3.3 for further discussion of this point. It can also be useful to examine the execution trace that accompanies the warning. For example, you might see that the trace reported be ESC/Java involves execution of a `return` statement in the middle of the method that you had overlooked.

4.16 `Pre` warning

A `Pre` warning warns that control may reach a routine call when some precondition of the routine does not hold. The warning is associated with the `requires` pragma that gives the potentially violated precondition.

4.17 `Race` warning

A `Race` warning warns of a possible attempt to access a monitored field while not holding the requisite lock(s). The warning is associated with the `monitored` or `monitored_by` pragma giving the lock(s) that should be held.

Bug/limitation: If there are multiple `monitored` and/or `monitored_by` pragmas for the same field `f`, a `Race` warning for an access to `f` will mention only one of these pragmas, and perhaps not the most relevant one.

4.18 `Reachable` warning

A `Reachable` warning warns that control may reach an `unreachable` pragma.

4.19 `Unreadable` warning

An `Unreadable` warning warns that control may reach a read access to a variable when the expression in a `readable_if` pragma modifying the variable's declaration is false. The warning is associated with the `readable_if` pragma.

4.20 `Uninit` warning

An `Uninit` warning warns that control may reach a read access to a local variable before execution of any assignment to the variable other than an initializer in a declaration modified by an `uninitialized` pragma. The warning is associated with the `uninitialized` pragma.

4.21 `ZeroDiv` warning

A `ZeroDiv` warning warns of a possible attempt to apply the integer division (`/`) or remainder (`%`) operator with zero as the divisor. (This condition would result in an `ArithmeticException` being thrown at run time [*JLS*, 15.16.2, 15.16.3].)

---

5 Command-line options and environment variables

The operation of ESC/Java is controlled by a variety of command-line options and environment variables. The primary source of information on these is the `escjava(1)` man page included with the ESC/Java release (see appendix B). In this section, we describe a small number of options that are of particular importance, or that seem to merit more extensive descriptions than those on the man page.

5.0 `-suggest`

The `-suggest` command-line option causes ESC/Java to accompany certain of its warning messages with suggestions for pragmas that may eliminate those warnings.

Examples

Running ESC/Java with the command line

```
escjava -suggest Bag.java
```

on the version of `Bag.java` in section 0.0 will produce such suggestions as

```
Bag.java:6: Warning: Possible null dereference (Null)
    n = input.length;
              ^
Suggestion [6,13]: perhaps declare parameter 'input' at 5,12 in Bag.java
with 'non_null'
```

Running ESC/Java with the `-suggest` option on the file `T.java` from section 3.2.4 will produce the following warning and suggestion:

```
T.java:5: Warning: Type of right-hand side possibly not a subtype of array
element type (ArrayStore)
    a[i] = x;
         ^
Suggestion [5,9]: perhaps declare method 'storeObject' at 4,14 in T.java
with 'requires \elemtype(\typeof(a)) == \type(T);'
```

Running ESC/Java with the `-suggest` option on the file `ObjStack.java` from section 3.2.17 will produce the following warning and suggestion:

```
ObjStack.java:16: Warning: Possible violation of object invariant
(Invariant)
  }
  ^
Associated declaration is "ObjStack.java", line 6, col 6:
  //@ invariant (\forall int i; n <= i & i < 10 ==> a[i] == null);
      ^
Possibly relevant items from the counterexample context:
  (0 <= (brokenObj<2>).(n:15.6))
  …
(brokenObj* refers to the object for which the invariant is broken.)

Suggestion [16,2]: perhaps declare instance invariant 'invariant
this.a.owner == this;' in class ObjStack (near associated declaration at
"ObjStack.java", line 6, col 6)
```

Fine points

ESC/Java does not supply suggestions for all warnings, and the suggestions that it does supply are heuristically chosen and may be incorrect. For example, if the contents of file `C.java` are

```
 1: class C {
 2:
 3:   int n;
 4:
 5:   static int m(C a, C b) {
 6:     if (a != null) {
 7:       return a.n;
 8:     } else {
 9:       return b.n;
10:     }
11:   }
12: }
```

then the command `escjava -suggest C.java` will give a warning about and possible null dereference on line 9, accompanied by the suggestion:

```
Suggestion [9,14]: perhaps declare parameter 'b' at 5,22 in C.java with
'non_null'
```

It might actually be better to declare the method `m` with the precondition

```
//@ requires a != null | b != null;
```

since it might be the intention of the programmer to support callers that meet only this precondition and not necessarily the more stringent condition that `b` always be non-null.

Despite its limitations, the `-suggest` option can be of considerable help to users in paring down the initial batch of mostly-spurious warnings that ESC/Java typically produces when it is first run on a body of unannotated code.

(A project currently under way at Compaq SRC is exploring automated techniques for inferring ESC/Java annotations [FL00, FJLxx].)

5.1 Specification (`.spec`) files and the ESC/Java's class path

This section discusses the algorithm that ESC/Java uses to find declarations of classes, and ways in which the user can control that algorithm.

In order to check the routine bodies of a class *C*, ESC/Java may need various information about some other type (class or interface) *T*. This information, which we call the *specification* of *T*, may include both information introduced by pragmas (such as `invariant` and `requires`) and information from the Java language (such as routine signatures and types of fields), but does not include routine bodies.

If ESC/Java needs the specification for a type *T* and can find only a binary (`.class`) file and no source file for *T*, then it can produce a simple specification based on the signature and type information included in the binary file. ESC/Java can also obtain specifications from specification (`.spec`) files (see section 5.1.2).

5.1.0 File reading modes

ESC/Java has two *modes* for reading files: *full mode* and *spec-only mode*. In order for ESC/Java to check the routine bodies in a file, it must read the file in full mode. When ESC/Java reads a file in spec-only mode, it only obtains specifications that can be used to check routine bodies in other files. ESC/Java can read `.java` files in either mode. It can read `.class` files in spec-only mode, but not in full mode.

When ESC/Java reads a source file in spec-only mode, it performs very limited processing--in particular, very liberal syntactic error-checking--on the bodies of routines. We do not specify here the exact degree to which the error-checking is liberalized, except to state that where a routine body would normally be expected, ESC/Java will accept (at least) any of the following when reading a source file in spec-only mode:

> a semicolon `;` (as in the Java syntax for an `abstract` method declaration).
>
> an empty body `{ }` (note that no `return` statement is required, even for a non-`void` method).
>
> a method body consisting of legal Java code legally annotated by ESC/Java pragmas (that is, a body that ESC/Java would accept if it were reading the file in full mode).

5.1.1 The ESC/Java class path (`-classpath`, `CLASSPATH`, `-bootclasspath`)

Like the Java compiler (`javac(5)`), ESC/Java uses a class path to look for declarations of types (classes and interfaces) that are not declared in files named on the command line. For a description of how ESC/Java uses the class path, see section 5.1.3 below.

The (full) class path is the concatenation of two parts: `classpath` and `bootclasspath`, where `classpath` is

- the argument of the `-classpath` command-line option, if any, on the command line, or else
- the value of the `CLASSPATH` environment variable if one has been set, or else
- a default value

and `bootclasspath` is

- the argument of the `-bootclasspath` command-line option, if any, on the command line, or else
- a default value.

The default values of `classpath` and `bootclasspath` are subject to change. At the time of writing, the default `classpath` is "." and the default `bootclasspath` includes a directory of selected `.spec` files (see section 5.1.2) for selected library classes and interaces together with directories for the normal Java system libraries (the same default versions used by `srcjava(1)`).

Fine point

If a command line contains multiple occurrences of the `-classpath` option, as in

```
escjava -classpath P1 -classpath P2 sourcefiles
```

only the last one (*P2* in the example) is used. The same applies for multiple occurrences of `-bootclasspath`.

Tip: The `-v` (verbose) command-line option makes ESC/Java output various information including the full classpath. Thus, you can learn the current default value of either `classpath` or `bootclasspath` at your site by setting the other to a known value and looking for the other in the output produced with `-v`. For example, you can learn the value of `bootclasspath` by typing

```
escjava -classpath xxx -v | grep classpath
```

on Unix systems or

```
escjava -classpath xxx -v | find "classpath"
```

on Windows systems.

5.1.2 Specification (`.spec`) files

There are times when the specification that ESC/Java can derive automatically from a `.class` file is inadequate, but when it is inconvenient or impossible for the user to add pragmas to the `.java` source file. (For example, the user's file system may not contain a copy of the `.java` file.) In such situations, the user can supply the needed pragmas in ESC/Java specification (`.spec`) files, which are similar to `.java` source files except that (1) ESC/Java always uses spec-only mode when reading a specification file (so routine bodies may always be omitted from specification files) and (2) a specification file *T*`.spec` may contain a declaration of only the single type *T*.

Since Java compilers do not look for files with extension `.spec`, one can use the same class path for the Java compiler as for ESC/Java with no danger of inadvertently pointing the compiler at a crippled source file.

Caveat: When ESC/Java reads a `.spec` file, it does not check that the contents of that file are in any way consistent with those of a `.java` or `.class` file that a compiler might find on the same class path.

5.1.3 How ESC/Java decides which files to read and in which modes

In this section we describe how ESC/Java decides which files to read and which modes to read them in. The short version of the story is that ESC/Java uses its class path about the same way that a Java compiler does, except that ESC/Java prefers `.spec` files over `.java` files. See the fine points below for a more complete (but not completely complete) story.

Fine points

[Note: Some of the behaviors in described in the next few paragraphs change when ESC/Java is run with the `-depend` command-line option. See section 5.1.4 for details.]

For each filename *F* on the command line, ESC/Java looks for the exact file *F* (with relative path names evaluated from the current working directory). If ESC/Java can't find file *F*, it issues an error message and goes on to look for the next file, if any. If ESC/Java finds file *F*, then it reads file *F* in full mode (as a Java source file, regardless of the filename extension), storing the type declarations it reads in an internal cache.

Once ESC/Java has read and cached the type declarations from files named on the command line, it may then need the declarations of additional types used (directly or indirectly) by the types already read. When ESC/Java needs the declaration of a type *P*`.`*T*, it can find it in any of the following places:

(0) already in ESC/Java's internal cache.

(1) in a file named *C*`/`*P'*`/`*T*`.spec`, where *P'* is the relative path name whose directory-path components are the simple-name components of *P* taken in order (e.g., on Unix, *P'* would be the result of replacing dots in *P* with slashes), and *C* is the first directory on the class path (see section 5.1.1) such that file *C*`/`*P'*`/`*T*`.spec` exists.

(2) in a file named *C*`/`*P'*`/`*T*`.java`, where *P'* is as in (1) above and *C* is the first directory on the class path such that file *C*`/`*P'*`/`*T*`.java` exists.

(3) in a file named *C*`/`*P'*`/`*T*`.class`, where *P'* is as in (1) above and *C* is the first directory on the class path such that file *C*`/`*P'*`/`*T*`.class` exists.

(4) in a `.java` file found by finding a file *C*`/`*P'*`/`*T*`.class` as in (3) above and reading the internal field that names the source file from which it was compiled.

(If names of `.jar` or `.zip` files occur as class path components, ESC/Java acts is if the class path included directories holding the expanded contents of those files.)

ESC/Java's ranking of these alternatives is, from most favored to least favored: 0, 1, 2, 4, 3. Note that this means that, for example *C1*`/`*P'*`/`*T*`.spec` is favored over *C2*`/`*P'*`/`*T*`.java` even if *C2* precedes *C1* in the class path. If ESC/Java doesn't have the declaration already in its cache and needs to read it from a file, it will read the file in spec-only mode.

Whenever ESC/Java reads a `.java` file in order to get the declaration of a type *T*, it will also read and cache any other type declarations in that file.

> **Remark:** In accordance with [*JLS*, 7.6], implementations may forbid a file *T*`.java` from declaring a type *U* other than *T*, when (1) the type *U* is referred to by code in other compilation units of the package in which the type *U* is declared, or (2) the type *U* is declared `public`. ESC/Java does not enforce this restriction. Note, however, that

when the restriction is obeyed, alternative (4) above will never arise.

During the process just described, various errors may occur. For example, a file may not contain a declaration of the class suggested by the file name. We do not attempt here a complete enumeration of these conditions. Also, it is beyond the scope of this manual to describe exactly which type declarations ESC/Java looks for in order to check a given type.

5.1.4 `-depend`

[Note: If you are not an ESC/Java wizard, and don't aspire to be, you should probably skip this section and never use `-depend`.]

The `-depend` command-line option causes ESC/Java to change its behavior from that described above in the following ways:

- ESC/Java will prefer to read `.java` files rather than `.spec` files. That is, the ranking given in section 5.1.3 switches from 0, 1, 2, 4, 3 to 0, 2, 4, 1, 3.
- When reading a `.java` file (alternative 2 or 4), then ESC/Java will read the file in full mode, rather than spec-only mode. This is in contrast to the normal behavior, where only `.java` files named on the command line are read in full mode.
- Instead of checking only classes declared in files named on the command line, ESC/Java will also check the classes that those classes depend on (including indirectly) provided that finds their declarations in `.java` files.

Caveat: As currently implemented the `-depend` option is likely to give unsatisfactory results in cases where both a `.spec` file and a `.java` file declaring the same class can be found on the class path. When run with `-depend` ESC/Java will prefer to read the class declaration from the `.java` file so that it can read in full mode and check the class's routine bodies. On the other hand, successful checking of clients of the class will likely require use of pragmas found only in the (ignored) `.spec` file and not in the `.java` file.

---

6 Java language support and limitations

The *Java*™ *Language Specification* [*JLS*] defines the Java 1.0 language. The ``Inner Classes Specification'' [*ICS*] specifies the additional language features supported in Java 1.1 and Java 1.2. The current version of ESC/Java accepts all Java language features described in [*JLS*]; it also accepts all Java language features described in [*ICS*] with the following two exceptions:

- When ESC/Java reads a class `C` from file `C.class`, and one of `C`'s members is a class `C.D`, ESC/Java will look for the file `C$D.class` only in the directory where it found `C.class`.
- ESC/Java will not check any routine body that mentions an anonymous class.

Java 1.2 includes the same language features as Java 1.1, but differs from 1.0 and 1.1 in the versions of the standard libraries that it includes. The current release of ESC/Java includes `.spec` files for a subset of the standard libraries. This subset is far from complete, but the `.spec` files that are included in the release are intended to correspond to classes and interfaces that are standard for Java 1.2.

Fine point

While ESC/Java accepts almost all of the language constructs described in [*JLS*] and [*ICS*], the semantics ESC/Java ascribes to those constructs differs in numerous details--including, but not limited to, those mentioned in appendix C and other parts of this manual--from the semantics specified in [*JLS*] and [*ICS*]. Likewise the annotations in the `.spec` files available in the ESC/Java release may fail for various reasons to capture the semantics of the actual JDK libraries (see section C.0.10).

Appendix A: Overview of how ESC/Java works

This appendix gives a very rough sketch of ESC/Java's internal operation.

The operation of ESC/Java consists of the following steps:

First, ESC/Java loads, parses, and type checks the files named on the command line, as well as any other files needed because their contents are directly or indirectly used by files name on the command line. (Section 5.1 describes where ESC/Java looks for files not named on the command line.)

Next, for each class whose routine bodies are to be checked, ESC/Java generates a type-specific *background predicate* encoding such information as subtype relations, types of fields, etc. in the class to be checked and the classes and interfaces it uses.

Next, ESC/Java translates each routine to be checked into a logical formula called a *verification condition* (*VC*). As an intermediate step in this translation, ESC/Java produces a command in an intermediate language [LSS99] based on Dijkstra's guarded commands. The intermediate language includes commands of the form `assert` *E*, where *E* is a boolean expression of the language. An execution of a command is said to ``go wrong'' if control reaches subcommand of the form `assert` *E* when *E* is false. Ideally, when a routine *R* is translated into a command *C* and thence to a verification condition *V*, the following three conditions should be equivalent:

> (1) There is no way that *R* can be invoked from a state satisfying its specified preconditions and then behave erroneously by, for example, dereferencing `null`, violating an `assert` pragma, terminating in a state that violates its specified postconditions, etc.
>
> (2) There is no execution of *C* that starts in a state satisfying the background predicate of *R*'s class and then goes wrong.
>
> (3) *V* is a logical consequence of the background predicate.

In practice, the translation is incomplete and unsound, so there may be semantic discrepancies between *R*, *C*, and *V*.

Finally, ESC/Java invokes the Simplify (`Simplify(1)`) theorem prover, asking it to prove each body's verification given the appropriate background predicate. If an attempted proof succeeds (or if Simplify exceeds specified resource limits in attempting the proof, or if ESC/Java exceeds specified resource limits generating the verification condition), then ESC/Java reports no warnings for the body. If the proof fails (other than by exceeding resource limits), Simplify produces a potential *counterexample context*, from which ESC/Java derives a warning message.

Fine points

The command-line option `-counterexample` makes ESC/Java print selected parts of each counterexample, sugared into a somewhat Java-like syntax. For example, in section 3.2.17, we gave an example file `D.java` and said of ESC/Java's output: ``There is no way to tell from this message whether ESC/Java is warning about the case where `a` has nonnegative `x` and `y` fields and `b` has negative `x` and `y` fields, or about the case where `a`'s fields are negative and `b`'s are nonnegative." If we ran ESC/Java on `D.java` with `-counterexample` option, the output would include something like:

```
Counterexample context:
...
    ((y:7.11).(a@pre:3.6) < 0)
    (0 <= (x:7.6).(b:3.9))
    ((y:7.11).(b:3.9) < 0)
    (0 <= (x:7.6).(a:3.6<1>))
...
```

from which one can infer that ESC/Java happens to be reporting the former case. As you may also infer from the excerpt above, the `-counterexample` option is intended mainly for expert users; to give further details about deciphering counterexample contexts (for example, the meanings of the inflections `@pre:3.6` and `:3.6<1>` on the field name `a`) is beyond the scope of this manual.

Appendix B: Installing and using ESC/Java at your site

The ESC/Java group maintains a web site at `http://research.compaq.com/SRC/esc/`. The tool may be downloaded from this site for educational and research use.

After you download the ESC/Java release according to the instructions on the web site, all files in the release will be in a single directory (chosen by you) at your site. We will write ``*escjavaRoot*'' to denote this directory.

Among the contents of *escjavaRoot* are the following files and subdirectories:

- *escjavaRoot*`/bin/escjava` (on Unix) or *escjavaRoot*`\bin\escjava.bat` (on Windows) contains the execution script for ESC/Java. The web site includes instructions for setting things up so that a command of the form ``escjava` [*options*] *sourcefiles*" will run this script.
- *escjavaRoot*`/doc/man1/escjava.1` contains the Unix `man(1)` page for ESC/Java.
- *escjavaRoot*`/doc/escjava.html` contains the man page for ESC/Java in HTML format.
- *escjavaRoot*`/examples/` contains some examples of source code on which to run ESC/Java, including the `Bag` example from section 0. Subdirectories named after sections of this manual contain examples taken from or pertinent to those sections.
- *escjavaRoot*`/lib/specs/` contains `.spec` files corresponding to selected JDK library files. (**Note:** The specifications in these `.spec` files may not always match the actual semantics of the corresponding library files; see section C.0.10.)

If you have a question, comment, or bug report concerning ESC/Java or this manual, you should start by checking the FAQ on the ESC/Java web site. If the FAQ does not address the issue adequately, you can email the ESC/Java group at `<escjava@research.compaq.com>`. You can also use this address to let us know if

you have produced `.spec` files for additional JDK library files and would like to share them. (**Note:** Restrictions may apply in distributing annotated or modified versions of Sun's JDK files. See the Sun Community Source License agreement at http://www.sun.com/software/java2/license.html).

Appendix C: Sources of unsoundness and incompleteness ESC/Java

C.0 Known sources of unsoundness

An unsoundness is a circumstance that causes ESC/Java to miss an error that is actually present in the program it is analyzing. Because ESC/Java is an extended static checker rather than a program verifier, some unsoundnesses are incorporated into the checker by design, based on intentional trade-offs of unsoundness with other properties of the checker, such as frequency of false alarms (incompleteness), efficiency, etc. Continuing experience, and new ideas, may lead to reevaluation of these trade-offs, with some sources of unsoundness possibly being eliminated and others possibly being added in future versions of ESC/Java.

In this section, we have attempted to describe, or at least allude to, all known causes of unsoundness in the current ESC/Java. If you become aware of any that we have missed, please bring them to our attention (see appendix B).

C.0.0 Trusting pragmas

The `assume`, `axiom`, and `nowarn` pragmas allow the user to introduce assumptions into the checking process. ESC/Java trusts them. If the assumptions are invalid, the checking can miss errors. Besides the possibility of an `assume`, `axiom`, or `nowarn` pragma being outright ``wrong,'' there are the following subtleties:

- As mentioned in the description of the `axiom` pragma (section 2.4.2) and illustrated in an example in section 2.7.2, axioms can mention mutable state. ESC/Java assumes that all (heuristically relevant) axioms hold at the start of any routine body being checked, but does not check that they still hold before each routine call or at the end of the routine body being checked.
- Certain ESC/Java warnings (for example `Null` warnings) correspond to conditions that would result in Java exceptions (for example `NullPointerException`). If a program is written intentionally to raise and then handle such an exception, the user might put a `nowarn` pragma on the line where the exception would be raised. In such a case the current ESC/Java will not check that there actually is a handler, nor will it check for any errors that might occur in the handler, or dynamically after execution of the handler.

C.0.1 Loops

The current ESC/Java does not consider all possible execution paths through a loop. It considers only those that execute at most one complete iteration (plus the test for being finished before the second iteration), as explained in section 2.4.3. This is simple, and avoids the need for loop invariants, but it is unsound.

The user can modify ESC/Java's treatment of loops by using the `-loop` command-line option. The `-loop` option takes an argument of the form *n*, *n*`.0`, or *n*`.5`, where *n* is a non-negative integer literal. The argument specifies

the number of loop iterations ESC/Java should consider. Suppose, for example, that the program being checked includes the fragment

```
//@ loop_invariant E;
while (B) {
  S
}
```

If you run ESC/Java with the `-loop n` (or, equivalently, `-loop n.0`) on the command line, it will consider execution paths that include up to *n* executions of

```
//@ assert E;      // but giving a LoopInv warning
if (!(B)) break;
S
```

plus one additional execution of `assert E`. If you run ESC/Java with `-loop n.5` on the command line, it will consider execution paths that include up to *n* executions of

```
//@ assert E;      // but giving a LoopInv warning
if (!(B)) break;
S
```

plus one additional execution of

```
assert E;          // but giving a LoopInv warning
if (!(B)) break;
```

In either case, code following the loop is checked only for execution paths in which the sequences described above terminate by a `break` out of the loop (including the implicit `if (!(B)) break`), the throwing of an exception, or the execution of a `return` statement. ESC/Java will not consider code following the loop for execution paths that ``fall through'' to the end of the unrollings (for example, by executing the final `if (!(B)) break` in the unrolling for `-loop 1.5` when *B* evaluates to `true`).

The default behavior of ESC/Java is the same as that given by `-loop 1.5`. Larger values of the parameter make ESC/Java's checking more nearly sound, but not perfectly sound. Larger values of the parameter can result in significantly slower checking and increased memory usage, especially when checking routines that include large loops bodies and/or nested loops. Loops that manifestly require many iterations to terminate normally pose a particular difficulty. Suppose, for example, that the program being checked contains a fragment of the form

```
for (int i = 0; i < 1000; i++) {S1}
S2
```

where *S1* never exits abruptly. Then the ESC/Java will never consider executions that reach *S2* unless it is run with a `-loop` option with an argument greater than 1000, which would almost certainly result in impractically large verification conditions or impractically slow checking.

Fine point

In the unrollings described above, execution of a `break` causes normal completion of the entire loop, and

execution of `continue` causes normal completion of the current unrolled copy of *S*.

C.0.2 Object invariants

When checking the implementation of a method, ESC/Java assumes initially that all allocated objects satisfy their invariants. But when checking a call to a method, ESC/Java imposes a weaker condition on the caller: all actual parameters of the call and all static fields that are in scope are shown to satisfy their invariants, but not every object in existence. Since more is assumed than is proved, this is unsound. It seems difficult to design a sound discipline that is not impractically strict; the current rule is a compromise that seems useful. See [LS97] for a more detailed discussion of the interaction of object invariants with scoping.

Another source of unsoundness in the checking of object invariants arises because, as we mentioned in section 2.4.1, when ESC/Java checks the body of any routine *R*, it does not consider all invariants but only a heuristically chosen ``relevant'' subset. If an invariant is deemed irrelevant during the checking of a routine that calls *R*, but deemed relevant during the checking of *R*, then the invariant will not be checked (even for parameters) at the call site, but will nonetheless be assumed to hold initially during the verification of *R*. (If ESC/Java cautioned the user against unenforceable object invariants of the sort mentioned in the fine points of section 2.4.1, the situation would be partially, but not entirely, ameliorated.) Conversely, ESC/Java might consider some invariant to be irrelevant to *R*, but relevant to a caller. In this case, ESC/Java will not check that the body of *R* preserves the invariant, but will nonetheless assume, while checking the caller, that the invariant is preserved by the call.

C.0.3 Modification targets

When reasoning about a call to a routine, ESC/Java assumes that the routine modifies only its specified modification targets (as given in `modifies` and/or `also_modifies` pragmas modifying the routine and any routines it overrides). But when checking the implementation of a method, the current ESC/Java doesn't check that the implementation modifies only the specified targets. Thus `modifies` and `also_modifies` pragmas are purely a way of describing the programmer's intent in the form that the checker can use as an assumption.

C.0.4 The `also_modifies` and `also_requires` pragmas

The `also_modifies` and `also_requires` pragma are unsound because they allow an overriding method to have a weaker specification than the method it overrides.

Suppose that a method `U.m` overrides a method `T.m`, and suppose that some method `r` contains a call of the form *E*`.m(...)` where *E* has static type `T` but might evaluate at run time to a value of type `U`. While checking the code containing the call ESC/Java will use the specification of `T.m`, but the actual call (even in the absence of other sources of unsoundness) might only meet the weaker semantics specified for `U.m`. In particular:

- If `U.m` is declared with an `also_modifies` pragma, the call might modify parts of the state that are specified as modification targets of `U.m` but not of `T.m`, but ESC/Java will check the implementation of `r` under the assumption that the call modifies only the targets specified for `T.m`. This would be a source of unsoundness even if ESC/Java guaranteed that `U.m` modifies only its specified modification targets (which it doesn't, as just mentioned in section C.0.3). Rustan Leino [Leino98] has designed a programming methodology that would avoid this unsoundness, but in the current ESC/Java we have decided to leave it to the programmer to use `also_modifies` with care so as to avoid introducing the unsoundnesses described in Leino's paper.
- If `U.m` is declared with an `also_requires` pragma, then ESC/Java will use the preconditions specified in

the `also_requires` pragma when it checks the implementation of `U.m`, but will not enforce those preconditions at the call *`E`*`.m(...)`.

C.0.5 Multiple inheritance

ESC/Java treats multiple inheritance of preconditions and modification targets unsoundly.

Suppose that a method `C.m` inherits from `A.m` and `B.m`, where `C` is a class, and either `A` and `B` are both be interfaces that `C` implements, or one is an interface that `C` implements and the other is a class that `C` extends; and suppose that some method `r` contains a call of the form *`E`*`.m(...)`, where *`E`* has static type `A` but might evaluate at run time to a value of type `C`.

When checking the body of `C.m`, ESC/Java will assume that all preconditions for `m` declared in (or inherited by) either `A` or `B`, hold initially. On the other hand, when checking the call *`E`*`.m(...)` in the body of `r`, where expression *`E`* has static type `A`, ESC/Java will only check the preconditions of `A.m`, and not those of `B.m`.

Similarly, when ESC/Java checks code after the call, it will assume that the call modifies at most the modification targets specified for `A.m`. This would be a source of unsoundness even if ESC/Java checked that the body of `C.m` modified only its declared modification targets.

C.0.6 Arithmetic overflow

ESC/Java reasons about integer arithmetic as though machine integers were of unlimited magnitude. This is both an unsoundness and an incompleteness, but it simplifies the checker and reduces the annotation burden for the user, while still allowing ESC/Java to catch many common errors.

The Simplify theorem prover used by ESC/Java (see appendix A) includes a decision procedure for linear rational arithmetic based on the simplex algorithm. If integer operations in Simplify's simplex module result in overflows, they will silently be converted to incorrect results. This is a potential source both of unsoundness and of incompleteness (see also section C.1.0).

C.0.7 Ignored exceptional conditions

ESC/Java checks for specific conditions that could give rise to a `NullPointerException`, `IndexOutOfBoundsException`, `ClassCastException`, `ArrayStoreException`,`ArithmeticException`, or `NegativeArraySizeException`, and warns of those conditions as potential errors. It ignores the all other cases where instances of unchecked exception classes (for example, `OutOfMemoryError`, `StackOverflowError`, `ThreadDeath`, `SecurityException`) [*JLS*, 11.2, 20.22] might be thrown either synchronously or asynchronously, except by explicit `throw` statements in a routine body being checked or in accordance with the throws clauses of routines called by a routine being checked.

C.0.8 Constructor leaking

There are a number of ways in which a constructor can make the new object under construction available in contexts where its instance invariants are assumed to hold, but without actually having established those instance invariants. For example:

- A constructor may terminate abruptly by throwing `this` without establishing object invariants for `this`.

- A supertype constructor may store `this` into a field of a global variable, and then return to the subtype constructor, which subsequently terminates abnormally.
- When a constructor of a class *`T`* is called as a supertype constructor from a constructor of a subtype *`S`*, the supertype constructor may establish the instance invariants of *`T`* for `this`, then perform a method call `this.`*`m`*`(...)` that dynamically dispatches to *`S`*`.m`. However, the correctness of the body of *`S`*`.m` might depend on instance invariants declared in *`S`* and not established at the call site.

For a more detailed examination of the constructor leaking problem, see [LS97].

In addition to the problems with invariants described above, constructor leaking can result in unsound checking of race conditions. When checking a constructor body ESC/Java does not require that any lock be held in order to accesses a field of `this`, even if the field is declared with at `monitored_by` or `monitored` (section 2.7.1) pragma. The reason is that ESC/Java assumes that no other thread yet has access to `this` and thus that no actual race can result. If this assumption is false, for example if the constructor stores `this` into a globally-accessible data structure from which another thread can read it, unsound checking--in the form of undetected race conditions--could result.)

C.0.9 Static initialization

The current ESC/Java does not perform extended static checking of static initializers [*JLS*, 8.5] and initializers for `static` fields. It neither checks for the possibility that they do not give rise to errors such as null dereferences, nor does it check that they establish or maintain static or instance invariants.

C.0.10 Class paths and `.spec files`

Java (`javac(5), java(5)`) and ESC/Java (`escjava(1)`), when run with the same class path, make different choices between `.spec`, `.java`, and `.class` files. The current ESC/Java doesn't check that the contents of the file it chooses are related to the contents of the file `javac(5)` would choose. Consider, for example, a scenario where a file `Foo.java` uses a class `Bar`, and where both a `Bar.java` and a `Bar.spec` file can be found on the class path. In response to the command `escjava Foo.java`, ESC/Java will check the routine bodies in `Foo.java` under the assumption that calls to methods of `Bar` have the semantics specified in `Bar.spec` (ignoring the file `Bar.java`). In response to the command `escjava Bar.java`, ESC/Java will check the method bodies in `Bar.java` against the specifications in `Bar.java` (ignoring the file `Bar.spec`). In neither case is there any checking that either the specifications or the bodies of routines in `Bar.java` have any connection to the specifications of corresponding routines in `Bar.spec`, or even that `Bar.java` and `Bar.spec` declare routines with the same names and signatures.

Note in particular that the specifications in the `.spec` files available for download via the ESC/Java web site (see appendix B) may disagree agree with the actual semantics of the corresponding JDK classes and interfaces, for any of a variety of reasons, including but not limited to the following:

- The annotations in the `.spec` files were added mostly in reaction to specific situations encountered by members of the ESC/Java team in our use of ESC/Java, rather than as part of any systematic effort to specify any set of routines completely (or even as completely as practical given the limitations of the ESC/Java annotation language).
- There may be ``version skew'' between the `.spec` files you download and the JDK files in use at your site.
- The `.spec` files may have intentional semantic differences from the corresponding JDK files for

methodological reasons (see, for example, the discussion of unchecked exceptions in section 4.5).

- The `.spec` files, the JDK files, or both may simply contain errors.

(See also the disclaimers near the beginning of this manual.)

C.0.11 Shared variables

ESC/Java depends on programmers to supply `monitored` and `monitored_by` pragmas telling which locks protect which shared variables. In the absence of such annotations, ESC/Java will not produce a warning when a routine might access a variable without holding the appropriate lock. Even when the user does specify which locks protect which variables, there is another potential source of unsoundness: ESC/Java assumes that the value of a shared variable stays unchanged if a routine releases and then reacquires the lock that protects it, ignoring the possibility that some other thread might have acquired the lock and modified the variable in the interim.

C.0.12 Initialization of fields declared `non_null`

There is an unsoundness in ESC/Java's checking that constructors assign non-null values to fields declared `non_null`. Consider the following program:

```
1:  class C {
2:   /*@ non_null */ Object f;
3:
4:    C() {
5:       m();
6:    }
7:
8:   //@ modifies this.f;
9:    void m() {
      …
      }
   }
```

When checking the implementation of the constructor for `C`, ESC/Java will assume (based solely on the pragmas on lines 2 and 8) that the method `m` returns with `this.f` set to a non-null value. While checking the body of `m`, ESC/Java will check that any assignments to `f` indeed assign non-null values. However, if the body of `m` can complete normally without assigning to `this.f`, then ESC/Java's assumption that `this.f` is always non-null after line 6 will be unsound.

C.0.13 String literals

Java's treatment of string concatenation (see [*JLS*, 3.10.5]) is not accurately modeled by ESC/Java. This is a source both of unsoundness and of incompleteness.

C.0.14 Search limits in Simplify

If Simplify cannot find a proof or a (potential) counterexample for the verification condition (see appendix A) for a routine within a set time limit, then ESC/Java issues no warnings for the method, even though it might have issued a warning if given a longer time limit. If Simplify reaches its time limit after reporting one or more

(potential) counterexamples, then ESC/Java will issue one or more warnings, but perhaps not so many warnings as it would have issued if the time limit were larger. You can set the time limit to *n* seconds, where *n* is a positive integer, by setting the environment variable `PROVER_KILL_TIME` to *n*. If `PROVER_KILL_TIME` is not set, ESC/Java sets it to 300 before invoking Simplify.

There is also a bound on the number of counterexamples that Simplify will report for any conjecture, and thus on the number of warnings that ESC/Java will issue for any routine. You can set the bound to a positive integer *n* setting the environment variable `PROVER_CC_LIMIT` to *n*. If `PROVER_CC_LIMIT` is not set, ESC/Java sets it to 10 before invoking Simplify.

C.0.15 Integer arithmetic bug in Simplify

Simplify includes a complete decision procedure for linear rational arithmetic and some heuristics for integer arithmetic. We have recently learned that one of the procedures implementing the integer arithmetic heuristics is buggy in a way that leads to unsoundness. This unsoundness in not one that we intended to design into the checker, and we are investigating the problem further. (For an unrelated source of unsoundness the discussion of arithmetic overflow in section C.0.6).

C.0.16 Quantifiers and allocation

When *T* is a reference type, specification expressions of the forms `(\forall T t; ...)` and `(\exists T t; ...)` (sections 3.2.10, 3.2.11) quantify over allocated instances of *T*. If a method allocates new objects but is not annotated with a postcondition mentioning containing an occurrence of `\fresh` (section 3.2.13) or `\old` (section 3.2.15), ESC/Java may infer (unsoundly) that some property holds for all allocated objects after completion of a call, when the property may in fact not hold for objects allocated during the call. This unsoundness results from a performance optimization and seems rarely to result in problems (missing warnings) in practice.

C.1 Some sources of incompleteness

An incompleteness is a circumstance that causes ESC/Java to warn of an potential error, when it is in fact impossible for that error to occur in any run of the program it is analyzing. Because ESC/Java attempts to check program properties that are, in general, undecidable, some degree of incompleteness in inevitable. In addition, ESC/Java's implementers have been willing to accept some evitable incompleteness in order to improve performance and keep the tool simple. We list here some principal sources of incompleteness in ESC/Java, but we do not attempt a complete enumeration of sources of incompleteness.

C.1.0 Incompleteness of the theorem-prover

The verification conditions that ESC/Java give to the Simplify theorem prover are in a language that includes first-order predicate calculus (with equality and uninterpreted function symbols) along with some (interpreted) function symbols of arithmetic.

Since the true theory of arithmetic is undecidable, Simplify is necessarily incomplete. In fact, the incompleteness of Simplify's treatment of arithmetic goes well beyond that necessitated by Gödel's Incompleteness Theorem. In particular:

- Simplify has no built-in semantics for multiplication, except by constants.
- Simplify doesn't support mathematical induction.

Also, first-order predicate calculus (FOPC) is only semidecidable--that is, all valid formulas of FOPC are provable, but any procedure that can prove all valid formulas must loop forever on some invalid ones. But it is not useful for Simplify to loop forever, since ESC/Java issues warnings only when Simplify reports (potential) counterexamples. Therefore Simplify will sometimes report a (potential) counterexample *C*, even when it is possible that more work could serve to refute *C*, and even to prove the entire verification condition. More particularly:

- The way Simplify makes use of a universally quantified formula, say `(\forall` *T* t_1, ..., t_n; *B*), is by selectively instantiating the body *B* with substitutions for t_1, ..., t_n determined by matching certain ``triggering patterns" against a set of terms already under consideration. In some cases, the triggering may be overly restrictive, preventing Simplify from finding instances that are actually needed for the proof.
- After Simplify instantiates the body of a universally-quantified formula, the terms in the instantiated body may match the triggering patterns of other universally quantified formulas, triggering instantiations of their bodies, and so on. To avoid infinite looping Simplify bounds the depth to which such sequences of matching may cascade. In some cases, Simplify may report a potential counterexample that could in fact have been refuted by deeper matching.

C.1.1 Incomplete modeling of Java semantics

Ideally, the verification condition for a routine *R* would be a formula that was valid if and only if *R* were free of the kinds of potential errors ESC/Java aims to detect. In fact, the verification conditions that ESC/Java generates fall short of modeling the full semantics of Java in many ways. For example:

- ESC/Java's built-in semantics for floating-point operations are extremely weak--not strong enough to prove `1.0 + 1.0 == 2.0` or even `1.0 != 2.0`.
- ESC/Java's built-in semantics for strings are quite weak--strong enough to prove `"Hello world" != null`, but not strong enough to prove the assertion `s == 'l'` after the assignment `c = "Hello world".charAt(3)`.
- ESC/Java treats exceptions thrown by the run-time system as errors, even in programs that include code to catch them.
- The ESC/Java release includes `.spec` files for only a few JDK libraries, and even the `.spec` files supplied do not fully capture the informal semantics of the specified routines (see also section C.0.10).
- According to rules of the Java type system, if neither of two distinct classes *S* and *T* is a subtype of the other, then *S* and *T* have no non-null instances in common. ESC/Java's modeling of the Java type system is good enough to enforce this disjointness for explicitly-named types, but not for all types (such as the dynamic element types of array variables).
- As mentioned in section C.0.6, the Simplify theorem prover may exhibit unsoundness due to integer overflow. In order to reduce the likelihood of overflow occurring in the prover, ESC/Java treats all integer literals of absolute magnitude greater than 1000000 as symbolic values whose relative ordering is known but whose exact values are unknown. Thus, ESC/Java can prove the assertions `2+2 == 4` and `2000000 < 4000000` but not `2000000+2000000 = 4000000`.
- While ESC/Java recognizes the Java 1.2 expressions of the form *T*`.class`, where *T* is a Java *Type* [*JLS*, 19.4], ESC/Java's semantics for such expressions is extremely limited. For example, ESC/Java can determine that `int.class` is a non-null instance of `java.lang.Class`, but not that it is distinct from `short.class`, or even that it is equal to `Integer.TYPE`. The implementers of ESC/Java currently have no plans to significantly strengthen its semantics for `java.lang.Class` in the absence of clear need. (In particular, we have no plans for creating any connection between Java's Reflection API [*Refection*] and

ESC/Java's specification type `\TYPE`.)

C.1.2 Modular checking

ESC/Java's use of modular checking modular checking causes it to miss some inferences that might be possible through whole program analysis.

- When translating a method call `E.m(...)`, ESC/Java uses the spec of `m` for the static type of `E`, even if it is provable that the dynamic type of `E` at the call site will always be a subtype that overrides `m` with a stronger spec.
- ESC/Java makes no attempt to infer method specifications. (However, see [FL00, FJLxx].)

References

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe, ``Extended Static Checking", Compaq SRC Research Report 159, December 1998. Available on the web at http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-159.html• . [Cited in the acknowledgments.]

[FJLxx] Cormac Flanagan, Rajeev Joshi, and K. Rustan M. Leino, ``Annotation inference for modular checkers", to appear in *Information Processing Letters*. [Cited in sections 5.0 and C.1.2.]

[FL00] Cormac Flanagan and K. Rustan M. Leino, ``Houdini, an annotation assistant for ESC/Java", Compaq SRC Technical Note 2000-003, September 2000. Available on the web at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-2000-003.html. [Cited in sections 5.0 and C.1.2.]

[*ICS*] ``Inner Classes Specification", Sun Microsystems, on the web at http://java.sun.com/products/jdk/1.1/docs/guide/innerclasses/spec/innerclasses.doc.html. [Cited in section 6.]

[*JLS*] James Gosling, Bill Joy, and Guy Steele, *The Java™ Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996. Also available on the web at http://java.sun.com/docs/books/jls/html/index.html. [Cited in places too numerous to mention.]

[LBR99] Gary T. Leavens, Albert L. Baker, and Clyde Ruby, ``JML: A notation for detailed design", in Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175-188, Kluwer Academic Publishers, Boston, 1999. [Cited in the preface.]

[LBR00] Gary T. Leavens, Albert L. Baker, and Clyde Ruby, ``Preliminary design of JML: A behavioral interface specification language for Java", Technical Report 98-06j, Iowa State University, Department of Computer Science, May 2000. Available on the web at www.cs.iastate.edu/~leavens/JML.html. [Cited in the preface.]

[Leino98] Rustan M. Leino, ``Data groups: Specifying the modification of extended state", in *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '98), volume 33(10) of *ACM SIGPLAN Notices*, pages 144-153, October 1998.

Available on the web (by permission of the ACM) in PostScript at ftp://ftp.digital.com/pub/DEC/SRC/publications/rustan/krml83-oopsla98.ps and in PDF at ftp://ftp.digital.com/pub/DEC/SRC/publications/rustan/krml83-oopsla98.pdf. [Cited in sections 2.3.8 and C.0.4.]

[LLPRJ00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs, ``JML: notations and tools supporting detailed design in Java", to appear in *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '00). Also available as Department of Computer Science, Iowa State University, TR #00-15, August 2000, on the web in PDF at ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.pdf and in PostScript at ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/TR.ps.gz. [Cited in the preface.]

[LSS99] K. Rustan M. Leino, James B. Saxe, and Raymie Stata, ``Checking Java programs via guarded commands", in *Formal Techniques for Java Programs*, workshop proceedings, Bart Jacobs, Gary T. Leavens, Peter Müller, and Arnd Poetzsch-Heffter, editors, Technical Report 251, Fernuniversität Hagen, 1999. Also available as Compaq SRC Technical Note 1999-002, on the web at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1999-002.html. [Cited in section 2.1.2.]

[LS97] K. Rustan M. Leino and Raymie Stata, ``Checking Object Invariants", Compaq SRC Technical Note 1997-007, January 1997. Available on the web at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-1997-007.html. [Cited in sections 2.3.4, 2.4.1, and C.0.2.]

[LS99] K. Rustan M. Leino and Raymie Stata, ``Virginity: A contribution to the specification of object-oriented software", *Information Processing Letters*, 70 (1999), pages 99-105. [Cited in (fine points of) section 3.2.17.]

[*Reflection*] ``Reflection" (section of JDK documentation), Sun Microsystems, on the web at http://java.sun.com/products/jdk/1.1/docs/guide/reflection/. [Cited in section C.1.1.]

[SLS00] Silvija Seres, with K. Rustan M. Leino and James B. Saxe, ``ESC/Java Quick Reference", Compaq SRC Technical Note 2000-004, October 2000. Available on the web at http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/abstracts/src-tn-2000-004.html. [Cited in the preface and acknowledgments.]
