Extended Static Checking for Java

Cormac Flanagan K. Rustan M. Leino* Mark Lillibridge

Greg Nelson James B. Saxe Raymie Stata

Compaq Systems Research Center 130 Lytton Ave. Palo Alto, CA 94301, USA

ABSTRACT

Software development and maintenance are costly endeavors. The cost can be reduced if more software defects are detected earlier in the development cycle. This paper introduces the Extended Static Checker for Java (ESC/Java), an experimental compile-time program checker that finds common programming errors. The checker is powered by verification-condition generation and automatic theorem-proving techniques. It provides programmers with a simple annotation language with which programmer design decisions can be expressed formally. ESC/Java examines the annotated software and warns of inconsistencies between the design decisions recorded in the annotations and the actual code, and also warns of potential runtime errors in the code. This paper gives an overview of the checker architecture and annotation language and describes our experience applying the checker to tens of thousands of lines of Java programs.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Program Verification

General Terms

Design, Documentation, Verification

Keywords

Compile-time program checking

1. INTRODUCTION

Over the last decade, our group at the Systems Research Center has built and experimented with two realizations of a new program checking technology that we call *extended static checking* (ESC): "static" because the checking is performed without running the program, and "extended" because ESC catches more errors than are caught by conventional static checkers such as type checkers. ESC uses an automatic theorem-prover to reason about the semantics of programs, which allows ESC to give static warnings about many errors that are caught at runtime by modern programming languages (null dereferences, array bounds errors, type cast errors, *etc.*). It also warns about synchronization errors in concurrent programs (race conditions, deadlocks). Finally, ESC allows the programmer to record design decisions in an annotation language, and issues warnings if the program violates these design decisions. Our first extended static checker, ESC/Modula-3, has been described elsewhere [8]. This paper provides an overview of our second checker, ESC/Java. It is not our goal in this paper to give a complete description of ESC/Java, but rather to give an overview that includes citations to more complete descriptions of particular aspects of the checker.

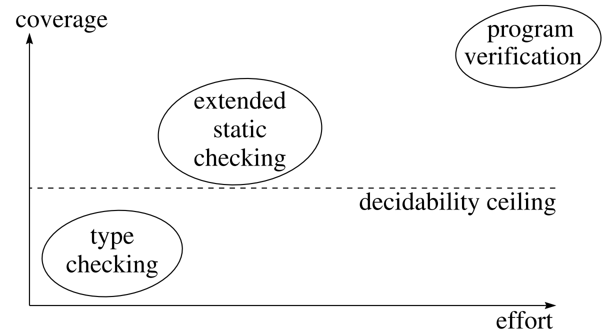

Figure 1: Static checkers plotted along the two dimensions coverage and effort (not to scale).

Static checking can improve software productivity because the cost of correcting an error is reduced if it is detected early. Figure 1 compares ESC with other static checkers on two important dimensions: the degree of error coverage obtained by running the tool and the cost of running the tool. In the upper right corner is full functional program verification, which theoretically catches all errors, but is extremely expensive. In the lower left corner are static checking techniques that are widely used, which require only modest effort, but catch only a limited class of errors: conventional type checkers and type-checker-like tools such as `lint` [23]. These two corners of Figure 1 exercise a magnetic fascination on programming researchers, but we suggest that the

*Current address: Microsoft Research, One Microsoft Way, Redmond, WA 98052.

PLDI'02, June 17-19, 2002, Berlin, Germany.

middle of the diagram is promising, and it is there that we position ESC: we hope to produce a cost-effective tool by catching more errors than a type system or lint-like tool at a cost much less than full functional program verification.

The horizontal line in Figure 1 labeled the "decidability ceiling" reflects the well-known fact that the static detection of many errors of engineering importance (including array bounds errors, null dereferences, *etc.*) is undecidable. Nevertheless, we aim to catch these errors, since in our engineering experience, they are targets of choice after type errors have been corrected, and the kinds of programs that occur in undecidability proofs rarely occur in practice. To be of value, all a checker needs to do is handle enough simple cases and call attention to the remaining hard cases, which can then be the focus of a manual code review.

A distinguishing feature of our work is that ESC/Modula-3 and ESC/Java both perform *modular* checking: that is, they operate on one piece of a program at a time—it is not necessary to have the source of the whole program in order to run the checker. In our case, a "piece" is a single routine (method or constructor). Whether an automatic checker or manual checking like code reviews is used, modular checking is the only checking that scales. Consequently we consider modular checking to be an essential requirement.

The cost of modular checking is that annotations are needed to provide specifications of the routines that are called by the routine being checked. We argue that, in the absence of an automatic checker, manual checking depends on these same annotations, typically in the form of English comments (which, not being machine checkable, easily get out of synch with the source code over the life of a program). Unlike the complicated predicate logic specifications that seem to be required for full functional verification, ESC annotations are straightforward statements of programmer design decisions. Indeed, we are excited about the prospect that the use of ESC in the classroom may help in the notoriously difficult job of teaching students to write good comments, since ESC is a practical tool that gives error messages of the form "missing comment" and "inaccurate comment".

Two attributes of an ideal static checker are (1) if the program has any errors then the checker will report some error (called "soundness" by mathematical logicians); and (2) every reported error is a genuine error rather than a false alarm (called "completeness" by mathematical logicians). In extended static checking, we do not take either of these attributes to be a requirement. After all, the competing technologies (manual code reviews and testing) are neither sound nor complete. Certainly false alarms are undesirable, since winnowing through the warnings to find the real errors is an added cost of running the tool, and certainly soundness is desirable, since every missed error is a lost opportunity for the checker to be useful, but insisting that the checker meet either ideal is mistaken on engineering grounds: if the checker finds enough errors to repay the cost of running it and studying its output, then the checker will be cost-effective, and a success. To achieve a cost-effective tool requires making good engineering trade-offs between a variety of factors, including: missed errors (unsoundness), spurious warnings (incompleteness), annotation overhead, and performance.

The major novelty of ESC/Java compared to ESC/Modula-3 is that ESC/Java has a simpler annotation language. An important innovation contributing to this simplicity is

```
 1: class Bag {
 2:     int size;
 3:     int[] elements;   // valid:  elements[0..size-1]
 4:
 5:     Bag(int[] input) {
 6:         size = input.length;
 7:         elements = new int[size];
 8:         System.arraycopy(input, 0, elements, 0, size);
 9:     }
10:
11:     int extractMin() {
12:         int min = Integer.MAX_VALUE;
13:         int minIndex = 0;
14:         for (int i = 1; i <= size; i++) {
15:             if (elements[i] < min) {
16:                 min = elements[i];
17:                 minIndex = i;
18:             }
19:         }
20:         size--;
21:         elements[minIndex] = elements[size];
22:         return min;
23:     }
24: }
```

Figure 2: Original version of `Bag.java`.

the *object invariant*, an annotation construct that will be described later. The simpler annotation language, together with the fact that ESC/Java targets a more popular programming language, has allowed us to get more user experience with ESC/Java than we did with ESC/Modula-3. This in turn has led us to engineer a number of improvements in the usability of the checker, for example execution-trace information in warning messages.

2. AN EXAMPLE OF USING ESC/JAVA

Perhaps the simplest way to impart a feeling for what it's like to use ESC/Java is to present an example in some detail. Figure 2 shows a small skeleton of a class of integer bags (aka multisets). The class provides only two operations: a bag may be constructed from an array of integers, and the smallest element of a bag may be extracted.

To invoke our checker, the user invokes it just as she would the Java compiler, but with "`escjava`" replacing the name of the compiler on the command line: `escjava Bag.java`. In response, over about the next ten seconds on a 200 MHz Pentium Pro PC, ESC/Java produces 5 warnings:

```
Bag.java:6: Warning: Possible null dereference (Null)
    size = input.length;
                ^
Bag.java:15: Warning: Possible null dereference (Null)
      if (elements[i] < min) {
                  ^
Bag.java:15: Warning: Array index possibly too large (...
      if (elements[i] < min) {
                  ^
Bag.java:21: Warning: Possible null dereference (Null)
    elements[minIndex] = elements[size];
                                 ^
Bag.java:21: Warning: Possible negative array index (...
    elements[minIndex] = elements[size];
                                 ^
```

The first of the warnings is a complaint that the *Bag* constructor may dereference null (if it is called with a null argument). There are two reasonable responses to this: either bulletproof the constructor so it can be called with null (producing an empty bag), or forbid calling the constructor with null. For this example, we assume the user chooses the second response. Traditionally, this would involve adding an English comment "This constructor may not be called with null" and hoping that programmers writing code that uses *Bag* obey this requirement.

Instead, with ESC/Java the user inserts (after line 4) a checker-readable comment (called an *annotation*) expressing the same thing:

$4a$: //@ **requires** *input* != **null**

The @-sign at the start of this Java comment tells ESC/Java that it is an ESC/Java annotation.

This annotation tells the checker that the constructor has a *precondition* of *input* != **null**. When ESC/Java checks a routine, it assumes that the routine's preconditions hold on entry; at a call site, ESC/Java issues a warning if it cannot verify the preconditions of the called routine. For *Bag* users, the annotation both provides documentation and lets them use ESC/Java to check that they are using *Bag* correctly.

The second and fourth warnings complain (for different execution paths) that method *extractMin* may dereference null (if called when the field *elements* is null). These warnings may seem spurious: the constructor sets *elements* to a non-null value initially and *extractMin* does not assign to *elements*. Note, however, that *elements* is not a private field so that client code and (future) subclasses may modify it.

These warnings would be arguably spurious if *elements* was declared private: no use of *Bag* could cause the warned about errors. Unfortunately, detecting this requires examining all the other code of *Bag* to make sure that there is no assignment of null to *elements*, which ESC/Java cannot do because it checks methods in isolation. Although annoying in this case, in more realistic cases where determining if any code assigns null to *elements* can be difficult for a human reader, these warnings serve the useful purpose of complaining about missing useful documentation.

To specify the design decision that *elements* is always non-null, the user annotates the declaration of *elements* (line 3):

$3'$: /*@**non_null***/ **int**[ ] *elements* ; // ...

ESC/Java generates a warning whenever it appears that code may assign null to a field declared non-null; it also checks that constructors initialize such fields to non-null values. Parameters may also be declared non-null; for example, instead of adding line 4a, the user could have changed line 5:

$5'$: *Bag*(/*@**non_null***/ **int**[ ] *input*) {

Indeed, we recommend that users use **non_null** where possible, both because it is easier to type, and because, being a specialized form, it is easier to verify and produce precise warning messages for.

The remaining two warnings complain of possible subscript errors. The checker is worried that future code might set *size* to a bad value. Here we need an *object invariant*:

$2a$: //@ **invariant** 0 <= *size* && *size* <= *elements.length*

An object invariant is a property that the programmer intends to hold at every routine boundary for every initialized instance of a class. The checker will now attempt to prove that the *size* field is correct after a *Bag* is initially constructed and that calling the *extractMin* method preserves its correctness. The checker will now also be able to use the invariant to reason that subscript errors cannot occur because the *size* field was incorrect on entry to *extractMin*.

Having made these changes, the user reruns the checker to check for more possible errors. Surprise! The checker again complains about possible subscript errors—the same two warnings in fact (which in Figure 2 are lines 15 and 21).

Looking more closely at the warning for line 15, the user recalls that Java arrays are indexed from 0, and changes:

14: **for** (**int** *i* = 1 ; *i* <= *size* ; *i*++) {

to:

$14'$: **for** (**int** *i* = 0 ; *i* < *size* ; *i*++) {

What about the warning that *size* may be negative at line 21? We know that *size* is at least 0 when *extractMin* is called. But what if *extractMin* is called when the bag is empty? Then *size* will be −1 by that line because of the decrement statement on the previous line. Oops! The user inserts a guard for the assignment to fix things:

$20a$: **if** (*size* >= 0) {
21: *elements*[*minIndex*] = *elements*[*size*] ;
$21a$: }

Running the checker yields a new warning, complaining that *extractMin* fails to reestablish the object invariant:

```
Bag.java:26: Warning: Possible violation of object in-
variant
  }
  ^
Associated declaration is "Bag.java", line 3, col 6:
  //@ invariant 0 <= size  &&  size <= elements.length
      ^
Possibly relevant items from the counterexample context:
  brokenObj == this
(brokenObj* refers to the object for which the invariant
is broken.)
```

This warning has three parts. The first says that an invariant may be broken at the end of method *extractMin*. The second says which invariant is involved. The third says that the object whose invariant may be broken is **this** rather than some other bag. The programmer acted too hastily; the if statement just inserted also needs to protect the decrement of *size*:

$19a$: **if** (*size* > 0) {
20: *size*−− ;
21: *elements*[*minIndex*] = *elements*[*size*] ;
$21a$: }

Now, rerunning the checker yields no warnings. This means that the checker is unable to find more potential errors, not necessarily that the program is bug free.

3. ARCHITECTURE

ESC/Java is the second extended static checker developed at the Systems Research Center. Its architecture is similar to that of the earlier checker [8], which targeted the Modula-3 language. Like that of traditional compilers, ESC/Java's architecture is best thought of as a pipeline of data processing stages (see Figure 3). We describe each stage in turn.

Front End. ESC/Java's front end acts similarly to that of a normal (Java) compiler, but parses and type checks ESC/Java annotations as well as Java source code. The

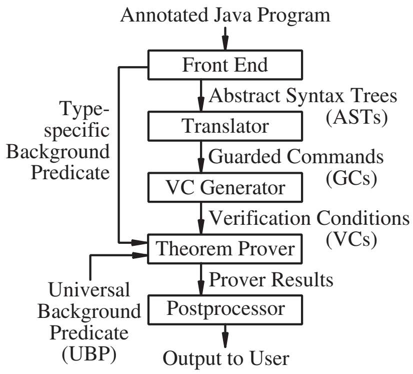

Figure 3: The basic steps in ESC/Java's operation.

front end produces *abstract syntax trees* (ASTs) as well as a *type-specific background predicate* for each class whose routines are to be checked. The type-specific background predicate is a formula in first-order logic encoding information about the types and fields that routines in that class use. For example, the type-specific background predicate for a final class T or any client of a final class T will include the conjunct $(\forall S :: S <: T \Rightarrow S = T)$.

Translator. The next stage translates each routine body to be checked into a simple language based on Dijkstra's guarded commands (GCs) [10]. ESC/Java's guarded command language includes commands of the form **assert** E, where E is a boolean expression. An execution of a guarded command is said to "go wrong" if control reaches a subcommand of the form **assert** E when E is false. Ideally the body of a routine R should translate into a guarded command G such that (1) G has at least one potential execution that starts in a state satisfying the background predicate of R's class and goes wrong, if and only if (2) there is at least one way that R can be invoked from a state satisfying its specified preconditions and then behave erroneously by, for example, dereferencing null or terminating in a state that violates its specified postconditions.

In practice, the translation is incomplete and unsound, so neither the "if" nor the "only if" above always holds. We mention a few sources of inaccuracy in translation here. For a more extensive discussion of incompleteness and unsoundness in ESC/Java, see appendix C of the ESC/Java user's manual [33].

Modular checking. In accordance with the principle of modular checking, when ESC/Java produces the guarded command for a routine R, it translates each routine call in R according to the specification, rather than the implementation, of the called routine. Consequently, the resulting (nondeterministic) guarded command G may be able to go wrong in ways involving behaviors of called routines that are permitted by their specification, but can never occur with the actual implementations. Of course, modular checking has the advantage that if R is correct with respect to the specifications of the routines it calls, it will continue to behave correctly after the implementations are replaced or overridden, so long as the new implementations continue to meet the specifications.

Overflow. We do not model arithmetic overflow because allowing the checker to consider cases such as adding positive integers and getting a negative sum leads to many spurious warnings.

Loops. A precise semantics for loops can be defined using weakest fixpoints of predicate transformers [10]; unfortunately, fixpoints are not merely uncomputable but difficult to compute in many practical cases. Therefore, ESC/Java approximates the semantics of loops by unrolling them a fixed number of times and replacing the remaining iterations by code that terminates without ever producing an error. This misses errors that occur only in or after later iterations of a loop.

Command-line options let the user control the amount of loop unrolling or substitute a sound alternative translation for loops that relies on the user to supply explicit *loop invariants*. By default, we unroll loops one and a half times (the half refers to an additional execution of the loop guard): Using two unrollings on ESC/Java's front end, Javafe (see section 6.3), produced only one plausibly interesting new warning but took 20% longer; five unrollings doubled the time but produced no new non-spurious warnings. We have found that even expert users have difficulty providing correct and sufficiently strong loop invariants.

VC Generator. The next stage generates *verification conditions* (VCs) for each guarded command. A VC for a guarded command G is a predicate in first-order logic that holds for precisely those program states from which no execution of the command G can go wrong. The computation of a VC is similar to the computation of a weakest precondition [10], but ESC/Java's VC-generation includes optimizations [19] to avoid the exponential blow-up inherent in a naive weakest-precondition computation.

ESC/Modula-3 also used guarded commands as an intermediate stage rather than deriving VCs directly from ASTs. ESC/Java goes a step further in factoring the software complexity of VC generation by using a "sugared" form of the GC language (not shown) as an intermediate stage between ASTs and the basic GC language that is input to the VC generator. The initial step of translating ASTs to sugared GCs is bulky and tedious, incorporating many Java-specific details, but need only be written once. The desugaring step and the final VC-generation step are much simpler and easily rewritten as we explore different soundness/incompleteness tradeoffs and possible performance improvements. For further discussion see [35].

Theorem Prover. For each routine R, the next stage invokes our automatic theorem prover, Simplify [9], on the conjecture

$$\mathrm{UBP} \wedge \mathrm{BP}_T \Rightarrow \mathrm{VC}_R \tag{1}$$

where VC_R is the VC for R, BP_T is the type-specific background predicate for the class T in which R is defined, and UBP is the *universal background predicate*, which encodes some general facts about the semantics of Java—for example, that the subtype relation is reflexive, antisymmetric, and transitive; and that all array types are subtypes of `java.lang.Object`. Subject to the translation limitations already discussed, the conjecture (1) will be valid iff the routine R has no errors. For a complete background predicate for a simple object-oriented language, see [27].

Postprocessor. The final stage postprocesses the theorem prover's output, producing warnings when the prover is unable to prove verification conditions. Simplify, originally designed for use by ESC/Modula-3 and later evolved for use by ESC/Java, has several properties that aid the postprocessor in constructing user-sensible warning messages, rather than just marking a routine as possibly being erroneous.

Counterexample contexts and labels. When it fails to find a proof for a conjecture, Simplify normally finds and reports one or more *counterexample contexts*, each counterexample context being a conjunction of conditions that (1) collectively imply the negation of the conjecture and (2) have not been shown by the prover to be mutually inconsistent. The input to Simplify can include positive and negative *labels* on some of its subformulas. A label has no effect on its subformula's logical value, but each counterexample includes positive labels of true subformulas and negative labels of false subformulas deemed heuristically relevant to that counterexample.

By carrying information about source code locations through the various stages of processing, ESC/Java is able to label each runtime check with sufficient information to produce a detailed warning including the type of error and its location. For example, the postprocessor generates the first warning in the example of Section 2 from the label in:

```
(LBLNEG |Null@0.6.16| (NEQ |input:0.5.12| null))
```

within the VC for the *Bag* constructor.

Unlike ESC/Modula-3, ESC/Java attaches labels not only to pieces of the VC that correspond to error conditions, but also to pieces of the VC that correspond to the execution of particular fragments of the source code; by using these labels, it can construct an execution trace indicating a possible dynamic path to the potential error [38].

Multiple counterexample contexts. Although a given routine may be able to fail in multiple interesting ways, a different counterexample may be required to demonstrate how each failure can occur. We have modified Simplify to generate multiple counterexample contexts for a given conjecture when possible. This allows us to generate multiple warnings per routine. Simplify keeps track of the labels reported with counterexample contexts and uses this information to keep from generating multiple counterexamples that would turn into too-similar warning messages.

Time and counterexample limits. We limit the time Simplify spends on each routine (5 minutes by default) as well as the number of counterexample contexts Simplify may produce (10 by default). If either limit is exceeded, ESC/Java issues a *caution* to the user indicating that the routine might not have been fully checked. The counterexample limit safeguards against the possibility that ESC/Java might issue a flood of apparently distinct warnings all arising from a single underlying problem with the program.

Incompleteness. ESC/Java VCs are formulas in a theory that includes first-order predicate calculus, which is only semi-decidable: any procedure that proves all valid formulas loops forever on some invalid ones. By allowing Simplify to sometimes report a "counterexample" that might, with more effort, have been shown to be inconsistent, it is able to produce more counterexamples within the time allotted. Such spurious counterexamples lead to spurious warnings. While Simplify is not guaranteed to be bug-free, it incorporates no intentional unsoundnesses, which would lead to missed warnings.

4. ANNOTATION LANGUAGE

The largest difference between ESC/Java and ESC/Modula-3 lies in the annotation language. In this section, we describe the main features of the ESC/Java annotation language. The full annotation language is described in the ESC/Java user's manual [33].

4.1 General design considerations

An important design decision for the annotation language has been to make it as Java-like as possible. This has two major advantages: it makes ESC/Java easier to learn, encouraging first-time use; and it makes the annotations more readable to non-ESC/Java users, increasing their value as primary documentation.

To a first approximation, annotations appear like other Java declarations, modifiers, or statements, but enclosed in Java comments that begin with an @-sign. Expressions contained in annotations are side-effect free Java expressions, with a few additional keywords and functions.

Beyond the syntactic issues lie deeper design problems of which annotations to include and what they should mean. Ideally, the annotations capture significant programmer design decisions, and do so succinctly. Equally important, and of more difficulty in the design of the annotation language, is that users not be required to document properties that are tedious to specify and don't significantly enable the detection of important software errors.

Our annotation language has also been shaped by a collaborative effort with Gary Leavens *et al.* to make the Java Modeling Language (JML [25, 26]) and the ESC/Java annotation language as similar as feasible. The goals of ESC/Java and JML are different: JML is intended to allow full specification of programs, whereas ESC/Java is intended only for light-weight specification. Therefore, some differences in the two languages remain, both syntactic and semantic. However, many programs annotated with ESC/Java annotations are amenable to processing with tools targeting JML and sometimes vice versa, and programmers who learn one language should have little trouble picking up the other.

4.2 Data abstraction vs. object invariants

The specification language for ESC/Modula-3 included general data abstraction [32]. "Abstract variables" could be declared (including abstract object fields), which were unknown to the compiler and used only for the purposes of the specification language. The exact meaning of an abstract variable is given by a "representation declaration", which specifies the value of the abstract variable as a function of other variables (abstract or concrete) [22].

General data abstraction is very powerful. In ESC/Modula-3 verifications, we found that it was generally used in a very stylized way that we call the state/validity paradigm. In this paradigm, two abstract variables are declared as "fields" of each object type: *valid* and *state*. The idea is that $x.valid$ means that the object x satisfies the internal validity invariant of x's type, and $x.state$ represents the abstract state of the object x.

In a full functional correctness verification, there would be many, many specifications to write about *state*, but in a typical ESC/Modula-3 verification, very little is said about it: it appears in the modifies list of those operations that can change it, and the concrete variables that are part of its representation are declared to so be (by means of an *abstraction dependency*).

In contrast to *state*, ESC/Modula-3 checking depends heavily on *valid*: almost all operations on an object x have $x.valid$ as a precondition; initialization operations on x have $x.valid$ as a postcondition; and any operations that destroy the validity of x (*e.g.*, an operation to close file x) have $x.valid$ in their modifies list, indicating that they are not guaranteed to preserve validity. These uses of *valid* enforce a protocol on a client using the type: the client must call a proper initialization operation before using the object, and may continue to use the object up until the object's validity is compromised.

The representation declaration for *valid* defines the meaning of validity in concrete terms. This declaration is typically placed in the implementation where it is required. In verifying the implementation of an operation on the type, the precondition $x.valid$ is translated into concrete terms by the usual process of data abstraction, since the representation is in scope. The occurrences of *valid* in client-visible specifications enforce proper behavior on the part of the client, even though the representation is invisible to the client, because the representation matters only to the implementation, not the client.

General data abstraction is sweetly reasonable, but it is more complicated than we would like (by about a hundred pages [32]). Mindful of the principle that perfection is achieved not when there is nothing left to add but when there is nothing left to remove, we decided to leave data abstraction out of ESC/Java. In ESC/Modula-3, data abstraction was used almost exclusively in the state/validity paradigm. ESC/Java object invariants provide much of the checking that is provided by the abstract variable *valid* in the valid/state paradigm. And the checking provided by the abstract variable *state* did not find many real errors.

ESC/Java does support *ghost fields* (see section 4.5), which are known only to the checker, not to the compiler. Ghost fields can be used as a substitute for abstract fields, but while abstract fields change automatically when their representation changes, ghost fields must be updated by an explicit assignment: Consider the case of concrete c, abstract a, and **rep** $a \equiv c{*}c$. We could mimic this situation with ghost field g and an explicit ghost assignment **set** $g = c{*}c$ whenever c changes. Adding the object invariant $g = c{*}c$ will provide protection against the error that the update to g is inadvertently omitted. But this requires considerably more annotation than the abstract variable approach.

The practical manifestation of these remarks is that the abstract variable state/validity paradigm has an advantage over object invariants in its ability to accurately specify methods that destroy validity. Close methods are a source of false alarms in ESC/Java, and while they could in principle be avoided soundly by introducing a ghost field *valid* and updating it in close and init methods, it is more common simply to suppress these warnings, which provides no checking against the error that the invalid object is later passed to a method that requires validity.

In spite of the remarks in the previous paragraphs, our judgment in retrospect is that the decision to remove abstract variables and introduce object invariants and ghost variables was on the whole a successful step towards the goal of producing a cost-effective engineering tool.

4.3 Routine specifications

Routine specifications can contain any or all of the following parts: **requires** P, **modifies** M, **ensures** Q, and **exsures** $(T\ x)$ R, where the precondition P, normal postcondition Q, and exceptional postcondition R for exception type T are boolean specification expressions, and the modifies list M is a list of lvalues (like $o.f$). In Q, the keyword **\result** refers to the value returned, if any. In R, the variable x refers to the exception thrown. In both Q and R, an expression of the form **\old**(E) refers to the value of the expression E in the pre-state.

The modifies list specifies which fields the routine may modify. Implicit in this list is that every routine is allowed to modify the fields of any objects allocated since the start of the routine invocation. At a call site, ESC/Java assumes that only those variables indicated by the modifies list are changed by the call. However, ESC/Java does not check that an implementation obeys its modifies list. This unsoundness is motivated as follows: writing down all the variables that a routine may modify is impossible, since many of these variables may be out of scope and some of them may even reside in subclasses yet to be written.

This fundamental and underappreciated problem can be solved with data abstraction [32] as we did in ESC/Modula-3. One of the costs of omitting data abstraction in ESC/Java is that we are unable, in general, to check modifies lists. There are a variety of approaches that a checking tool can take with respect to modifies lists: ESC/Modula-3 took a theoretically sound approach at the cost of complexity in the annotation language. ESC/Java takes the other extreme: modifies lists are assumed to be given correctly by the programmer—the tool uses but does not check them. Recent work suggests that theoretically sound solutions may still be feasible in a practical checker, by imposing some restrictions on the programming model [39, 34].

Overriding methods inherit specifications from the methods they override. Users may also strengthen the postconditions of overrides (using **also_ensures** and **also_exsures**), which is sound. Strengthening the precondition or extending the modifies list would not be sound [28]. Nevertheless, because ESC/Java does not provide any data abstraction, it seemed more prudent to allow the unsound **also_modifies** than not: this lets users express design decisions about what variables a method override may modify, even though the corresponding checking would not be sound. We were much less compelled to add **also_requires**, because we deemed its use a probable programmer error (some programmers design their programs as if it would be sound to strengthen preconditions in subclasses). However, again due to the lack of abstraction features, we do allow **also_requires** in one special case, as described in the user's manual [33].

In addition to single-inheritance subclassing, Java features multiple inheritance from interfaces. A sound way to combine multiple inherited specifications for a method is to take their mathematical join in the lattice of conjunctive predicate transformers [47, 31], as is done in JML [25]. However, this approach leads to problems like how to explain pre- and postcondition violations in warning messages. ESC/Java uses the simpler, but unsound, approach of taking the union of all specification parts inherited from overridden methods.

4.4 Object invariants

Designing a good, mechanically-checkable object invariant system is a complicated task. For space reasons, we discuss only briefly some of the more important issues involved.

Operation boundaries. Conceptually, the object invariants for an object o may be broken only during an "operation" on o. The question arises of what to consider an operation. The simplest approach, which we have largely adopted, is to consider each routine by itself to be a separate operation. In particular, we do not consider the subroutines called by a routine to be part of the same operation; this means that the routine must restore any invariants it has broken before it may call any subroutines.

One exception is that users may mark methods with the modifier **helper**; such methods are considered part of the operation of the routine calling them. This is achieved by inlining calls to helpers, which goes beyond modular checking. This annotation is useful in some cases, but is expensive if used excessively.

Invariant enforcement. In practice, enforcing invariants on all operation boundaries is too strict: a method could not call even so harmless a function as square root while an invariant is broken. Although the called code will expect all invariants to hold, it is usually safe to allow subroutine calls so long as the code being called cannot even indirectly reach the object(s) whose invariants are broken. Because this property is not locally checkable without an unwieldy amount of annotation, ESC/Java checks invariants at call sites only for arguments and static fields. This heuristic disallows calls likely to be an error (*e.g.*, passing an invalid object as an argument) and allows the calls like square root that are almost always okay.

Invariant placement. Because the entire program may not be available, it is not possible to get the effect of checking all invariants without adopting restrictions about where invariants are declared [32, 36]. Roughly, an invariant mentioning a field f must be visible whenever f is. This usually means that the invariant must be declared in the class that declares f. We have some understanding of the restrictions needed, but ESC/Java does not enforce them.

Constructors. We do not require **this** to satisfy its invariants in the case where a constructor terminates exceptionally; this is unsound in the rare case where a copy of **this** survives the constructor termination.

4.5 Ghost fields

ESC/Java's lack of data abstraction simplifies the annotation language and checker, but reduces the expressiveness of the annotation language. Nevertheless, it is sometimes useful to describe the behavior of a class in terms of some additional state that the Java program leaves implicit. For this purpose, ESC/Java provides *ghost fields*, which are like ordinary fields in Java, except that the compiler does not see them.

Ghost fields can also be used to specify behavior dependent on the abstract state of an object, for which abstract variables would be less suitable. For example, the usage protocol for the *hasMoreElements* and *getNextElement* methods of the *Enumeration* class can be expressed in terms of a boolean ghost field denoting the condition that more elements are available.

A common use of ghost fields in ESC/Java is to make up for the absence of generic types (parametric polymorphism) in Java. For example, a class *Vector* can introduce a ghost field *elementType*:

//@ **ghost public \TYPE** *elementType*

The special type **\TYPE** in ESC/Java denotes the type of Java types. Using ghost field *elementType*, the methods of class *Vector* can be specified. For example, the methods for adding and retrieving an element from a vector can be specified as follows:

//@ **requires \typeof**(*obj*) <: *elementType*
public void *addElement*(*Object obj*) ;

//@ **ensures \typeof**(**\result**) <: *elementType*
public *Object elementAt*(**int** *index*) ;

A client of *Vector* would set the *elementType* ghost field using ESC/Java's **set** annotation before using a vector. For example,

Vector v = **new** *Vector*() ;
//@ **set** *v.elementType* = **\type**(*String*) ;

creates a vector v intended to hold strings.

4.6 Escape hatches

We have designed ESC/Java's annotation language and checking so that common programmer design decisions can easily be expressed. However, situations arise where convincing the checker about the correctness of the design requires more powerful annotation features and checking than are provided. For these situations, one needs *escape hatches* to get around the strictness of the static checking system.

ESC/Java's **nowarn** annotation suppresses selected warnings about the source line where it appears. ESC/Java's `-nowarn` command-line switch turns off selected warnings for the entire program, allowing users to customize the degree of checking. One use of this is to turn off all warnings except those for violations of pre-conditions, post-conditions, and object invariants in order to check if a program is consistent with a user-specified protocol; see [29] for an example. A more precise escape hatch is the statement annotation

//@ **assume** P

which causes ESC/Java to blindly assume that condition P holds at this program point without checking it. Uses of **assume** and **nowarn** make good focused targets for manual code reviews.

Sometimes simple properties of a routine (*e.g.*, not dereferencing null) depend on more complex properties of the routines it calls. In such cases, judicious use of escape hatches can save users from *specification creep* that otherwise tends inexorably toward the upper right corner of Figure 1.

5. PERFORMANCE

ESC/Java checks each routine by invoking the automatic theorem prover Simplify. Since theorem proving is often expensive, and undecidable in the worst case, a potential problem with ESC/Java is that it could be too slow for interactive use.

We therefore put considerable effort (some of which was spent already in the ESC/Modula-3 project) into improving the performance of the theorem prover, into encoding background predicates in a form that plays to the strengths

Routine size	# of routines	Percentage checked within time limit				
		0.1s	1s	10s	1min	5mins
0–10	1720	27	90	100	100	100
10–20	525	1	74	99	100	100
20–50	162	0	33	94	99	100
50–100	35	0	0	74	94	100
100–200	17	0	0	53	82	94
200–500	5	0	0	0	80	100
500–1000	1	0	0	0	0	100
total	2331	20	80	98	> 99	> 99

Figure 4: Percentage of routines of various sizes in the Java front end benchmark that can be checked within a given per-routine time limit.

Annotation type	Annotations per KLOC	
	Javafe	Mercator
`non_null`	8	6
`invariant`	14	10
`requires`	28	16
`ensures`	26	2
`modifies`	4	0
`assume`	1	11
`nowarn`	6	0
other	4	1
total	94	48

Figure 5: Number and kinds of annotations required in the benchmarks.

of the theorem prover, and into generating VCs in a form that is more amenable to efficient proving. The combined result of these various optimizations is that the performance of ESC/Java is sufficient for the majority of cases.

We have applied ESC/Java to a variety of programs, and we report on the performance of ESC/Java on the largest of these programs, Javafe, ESC/Java's Java front end. This program contain 41 thousand lines of code (KLOC) and 2331 routines. Figure 4 illustrates the performance of ESC/Java on this program using a 667 MHz Alpha processor. The routines in Javafe are categorized according to their size (in lines of code). For each category, the figure shows the number of routines in that category, together with the percentage of those routines that can be checked within a particular per-routine time limit. The results show that most of the routines are fairly small (under 50 lines), and that ESC/Java can check the great majority of these routines in less than 10 seconds. Thus, the performance of ESC/Java is satisfactory, except for a small number of particularly complex routines. There is only one routine (of the 2331 routines in this program) that ESC/Java is unable to verify within the default five-minute time limit.

6. EXPERIENCE

This section describes some of our experience applying ESC/Java to a variety of programs. These programs include the Java front end Javafe, portions of the web crawler Mercator [21], and an assortment of smaller programs.

6.1 Annotation overhead

ESC/Java is an annotation-based checker, which relies on the programmer to provide annotations giving lightweight specifications for each routine. Thus, one of the costs of using ESC/Java is the overhead of writing the necessary annotations. In most cases, these annotations are straightforward and they document basic design decisions, for example, that a method argument should never be null, or that an integer variable should be a valid index to a particular array.

Our experience in annotating Javafe and Mercator indicates that roughly 40–100 annotations are required per thousand lines of code. Figure 5 illustrates the number and kinds of annotations required for these programs.

For both of these programs, the annotations were inserted after the program was written using an iterative process: The program was first annotated based on an inspection of the program code and a rough understanding of its behavior; this initial set of annotations was subsequently refined based on feedback produced by ESC/Java when checking the annotated program. Typically, we found that a programmer could annotate 300 to 600 lines of code per hour of an existing, unannotated program. This overhead is expensive for larger programs and is an obstacle to using ESC/Java to catch defects in large, unannotated programs. While it is possible to use ESC/Java only on selected modules, it is still necessary to annotate all the routines called in other modules. We are investigating annotation inference techniques [17, 16] that help reduce the annotation burden on legacy code.

Instead of writing annotations after the program has been developed, a better strategy for using ESC/Java is to write appropriate annotations and run the checker as early as possible in the development cycle, perhaps after writing each method or class. Used in this manner, ESC/Java has the potential to catch errors much earlier than testing, which can only catch errors after the entire program or an appropriate test harness has been written. Since catching errors earlier makes them cheaper to fix, we suggest that using ESC/Java in this manner may reduce software development costs in addition to increasing program reliability.

The following subsections illustrate ESC/Java's ability to find software defects that have proven difficult to catch using testing.

6.2 Mercator

The authors of the web crawler Mercator, Allan Heydon and Marc Najork, used ESC/Java to check portions of Mercator. Since Heydon and Najork were not involved with the ESC/Java project, their experience may be typical of average ESC/Java users.

Heydon and Najork annotated and checked 4 packages from Mercator containing 7 KLOC in roughly 6 hours. Toward the end, ESC/Java caught a previously-undetected bug in a hash table implementation. This hash table is implemented as an array indexed by hash code with each entry in this array pointing to a secondary array of values with that hash code. It is possible for an entry in the main array to be null. However, the checkpointing code, which writes the hash table to disk, failed to check for a null pointer in the main array.

This defect did not show up during testing since the hash table is checkpointed only after it is heavily loaded, and thus all entries in the main array are likely to be non-null. This defect may also be missed during a code review, especially if the design decision that entries in the main array may be null is not documented. Because ESC/Java requires explicating design decisions such as these, it can detect defects where these decisions are not respected.

6.3 Javafe

One of us (Lillibridge) spent about 3 weeks annotating ESC/Java's front end, which at that time measured about 30,000 lines. This process found about half a dozen previously undetected errors. Lillibridge assessed these errors as not having been worth 3 weeks to discover, but the benefit of the annotations had just started. Since that time, we have run ESC/Java on Javafe before checking in any changes, revealing an additional half dozen errors, each of which was detected shortly after it was introduced. Next, we'll describe one particularly interesting experience.

One of us (Leino) performed a major piece of surgery on the complicated class in Javafe that correlates source code locations with positions in an input stream. After finishing the edits, Leino ran the compiler, launched ESC/Java on the Javafe sources, and logged off for the day. The next morning, the Javafe regression suite had passed, but running ESC/Java with the new front end on even simple programs caused it to crash.

After spending 2+ hours pinpointing the error, Leino wondered why ESC/Java had not detected it. He had forgotten to check ESC/Java's output before starting to debug. Indeed, ESC/Java had found the error—a failure to establish a precondition that a stream be "marked"—and ESC/Java's output also revealed the same error at a different call site.

In this case, ESC/Java's output could have saved 2+ hours in tracking down the problem. The entire run of ESC/Java on Javafe had taken 73 minutes, but would have used less than 3 minutes had it been applied only to the file containing the errors. Correcting the errors was also tricky, and Leino was able to insert annotations and run ESC/Java to check his understanding of the various state changes in the program.

We draw two conclusions from our experience with Javafe.

First, keeping machine-checkable design decisions in a program provides a payoff during code maintenance, which is valuable because the cost of maintenance can easily outweigh the cost of the initial code development.

Second, several of the errors that ESC/Java detected in Javafe were violations of design decisions that are not easily extracted from the code. These include protocol designs like "call this routine only on streams that have been marked". ESC/Java provides a flexible and powerful framework for checking such protocol errors. In fact, some may consider using ESC/Java in a mode where it checks only for violations of user-supplied annotations, not for crashes that the language will catch at runtime [29].

6.4 Other user experience

ESC/Java is available for download at `research.compaq.com/SRC/esc`. In the last year, we have received more than 100 emails from users, and the stream of questions seems to increase rather than subside. We have heard some success stories where ESC/Java has found errors in code, several times surprising the authors of the code.

Some other users, pulled by the magnetic fascination in the upper right corner of Figure 1, have attempted to use ESC/Java to perform full functional correctness verification. In a way we are flattered, but we are not surprised that most of them have run into difficulties caused by incompletenesses in Simplify, which was engineered to be automatic rather than complete. We believe that learning when to give up and put in an **assume** annotation will lead to more cost-effective use of ESC.

7. RELATED WORK

The closest work related to ESC/Java is our previous work on extended static checking, ESC/Modula-3 [8]. Whereas the research on ESC/Modula-3 can be summarized as answering (affirmatively) the question, "can an extended static checker find bugs and be made automatic?", ESC/Java has focused on the questions "how simple can the annotation language be?" and "is the checker cost-effective?". Besides targeting a different and more popular language, ESC/Java incorporates many innovations over ESC/Modula-3, including: greater similarity between source and annotation languages, use of object invariants instead of data abstraction, execution traces [38], multiple warnings per method, timeouts, generation of suggestions for how to respond to warnings (not described in this paper), different treatments of loops, a multi-stage translation to guarded commands [35], a different VC generation from guarded commands [19], differently chosen engineering trade-offs, a comprehensive user's manual [33], a variety of new annotations including **helper**, and a variety of internal improvements. Overall, we believe we have produced a tool that is simpler to use, at the price of missing more errors than ESC/Modula-3 did. For a perspective on the building of ESC/Modula-3 and ESC/Java, see [30].

A goal that is several decades old is providing in a programming language features to write down more information than is strictly needed for compilation. The earliest serious attempt we know of is Euclid [24], which included constructs to express, for example, pre- and postconditions of procedures. The motivation was to enable programs to be verified, but the formal language semantics did not reach machine-checkable maturity. Instead, pre- and postconditions, and other assertions, were checked at runtime. A newer and more widely used language in this spirit is the object-oriented language Eiffel [37], whose pre- and postconditions and object invariants are also checked at runtime. A language in progress is Vault [7], whose promising ideas include using type-like features to take a next step in preventing certain kinds of resource-management programming errors.

Another research area related to extended static checking is *refinement types*. A refinement type is essentially a restricted form of object invariant. Refinement types have been investigated mostly for functional programming languages (*e.g.*, Xi and Pfenning [50]), but some work has been done recently for imperative languages [49].

There are other compile-time techniques for finding errors in programs. Unlike the ESC approach which uses programmer-supplied annotations for documenting design decisions, most other techniques have gone in the direction of completely eliminating annotation overhead, often to the extent that annotations cannot be supplied. Another difference between ESC and these other techniques is that the others do not build in modular checking from the start; rather, they are typically applied to the entire program. We mention three such techniques:

Symbolic execution is the underlying technique of the successful bug-finding tool PREfix for C and C++ programs [3]. For each procedure, PREfix synthesizes a set of execution paths, called a *model*. Models are used to reason about calls, which makes the process somewhat modular, except that fixpoints of models are approximated iteratively for recursive and mutually recursive calls.

PREfix gets by with an ad hoc constraint solver, rather than a theorem prover and an underlying logic. There is no annotation language, but due to a scheme for associating a weight with each warning (and surely also due to the large number of possible errors in C and C++), users can sort warnings by weight, thereby easily ignoring less relevant warnings.

Abstract interpretation [6] is a more established technique; it uses heuristics to iteratively build up an abstract model of a program. The absence of errors in the abstract program implies the absence of errors in the given program. Abstract interpretation has been applied successfully in many applications, including space-rocket controllers [44].

Symbolic model checking [2] is a technique whose success in finding hardware design errors has rubbed off on the software checking community. A popular idea is to use *predicate abstraction* [20] to reason about a given (infinite-state) program as a finite-state system that is model checked. An intriguing system based on this idea, including an automatic engine for incrementally inferring the predicates used in the abstraction, is SLAM [1]. Other tools that verify properties of software systems using finite-state models are Bandera [5] and Java PathFinder 2 [46].

In addition to these three techniques, there are other tools that have been useful in program development. For example, the LCLint tool [15] has become a part of the environment for building Linux. Some tools have focused on checking for particular kinds of errors, like concurrency errors: Warlock [43] to mention one, but see our ESC/Modula-3 report [8] which reports on our experience with extended static checking for finding concurrency errors and mentions several other pieces of related work in this field. Recent work by Engler *et al.* shows that a surprisingly effective technique for finding errors is a heuristic scan for irregularities in program source code [12].

Going beyond the bug-finding tools are tools geared toward the full verification of programs. While most such systems have remained in academic labs where they have been applied to small textbook programs or the verification of specific algorithms, some systems have been applied to actual safety-critical programs. A successful example is the B system [48], which was used to construct a part of the Metro subway system in Paris. Another example of a full-verification tool is Perfect Developer for the object-oriented language Escher [14]. Both the B and Escher systems restrict the programming language to various extents and require some manual guidance of their underlying theorem provers.

A "big brother" of the ESC/Java annotation language, the Java Modeling Language (JML) [25] allows for fuller specifications of Java programs. As mentioned in Section 4.1, through a collaboration with the designers of JML, we have tried to smooth out any gratuitous differences between the two specification languages, more or less making ESC/Java a subset of JML. The LOOP tool [45] translates JML-annotated Java into verification conditions that can be used as input to the theorem prover PVS [41]. The relation between JML, LOOP, and ESC/Java is described in some more detail in a short paper [26]. The similarities of the tools was an asset in specifying and checking the JavaCard interface [42] and an electronic purse application [4]. The similarities have also allowed JML and ESC/Java to be used in concert in teaching a software engineering course at Kansas State University [11].

Finally, we mention some intriguing work built on top of ESC/Java. Michael Ernst *et al.* have investigated the dynamic inference of likely program invariants [13]. Recently, this inference has been used to produce ESC/Java annotations [40]. Another annotation inference system for ESC/Java is Houdini [17, 16], which uses ESC/Java as a subroutine in inferring annotations.

8. CONCLUSIONS

Over the past two years, ESC/Java has been used to check a variety of programs, including moderately large systems such as Javafe and Mercator. The experience of both ESC/Java developers and other users supports the thesis that ESC/Java can detect real and significant software defects. In addition, the performance of ESC/Java is sufficient for interactive use on all but the most complex of methods.

ESC/Java's design incorporates a trade-off between soundness and usefulness, and in some cases sacrifices soundness to reduce the annotation cost or to improve performance. Examples of unsound features include loop unrolling and the partial enforcement of object invariants. By and large, our experience supports this limited introduction of unsoundness as a technique that clearly reduces the cost of using the checker, and we believe the number of bugs missed due to these features is small. Our ongoing research continues to tackle some of the more significant sources of unsoundness, including loops (by loop invariant inference [18]) and modifies lists [34].

ESC/Java's annotation language is Java-like and uses object invariants (as opposed to ESC/Modula-3's use of data abstraction). Both of these design decisions helped keep the annotation language intuitive. Object invariants are somewhat less expressive than data abstraction, but this limitation did not appear to cause problems in practice.

Despite ESC/Java's success at finding real errors, feedback from our users suggests that the tool has not reached the desired level of cost effectiveness. In particular, users complain about an annotation burden that is perceived to be heavy, and about excessive warnings about non-bugs, particularly on unannotated or partially-annotated programs. However, these users retroactively annotated and checked existing programs, rather than using ESC/Java to support development throughout a project's life-cycle.

At this point, it is uncertain if, over the lifetime of a software project, ESC/Java is a cost-effective tool for use by mainstream programmers. Certainly, our experience over the past two years in using ESC/Java to support development of Javafe has been encouraging, and it is possible that ESC/Java would be useful to highly-disciplined programming teams.

We are hopeful that additional research on reducing spurious warnings and lowering the perceived annotation burden (for example, by annotation inference, both statically as in Houdini [17] and dynamically as in Daikon [13]) may yield an extended static checking tool that could add significant value to the process of engineering software. In the meantime, we believe ESC/Java *is* suitable for use in a classroom setting as a resource for reinforcing lessons on modularity, good design, and verification.

9. ACKNOWLEDGMENTS

Todd Millstein, as a research intern at Compaq SRC, implemented the reporting of execution traces in ESC/Java's output. Caroline Tice and Rajeev Joshi lent helping hands in programming and porting. Additionally, we're grateful to Allan Heydon, Marc Najork, and other ESC/Java users who have shared their experiences of using the tool and given us feedback.

10. REFERENCES

[1] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In M. B. Dwyer, editor, *Proc. 8th SPIN Workshop*, volume 2057 of *LNCS*, pages 103–122. Springer, May 2001.

[2] J. R. Burch *et al.* Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *SP&E*, 30(7):775–802, June 2000.

[4] N. Cataño and M. Huisman. Formal specification of Gemplus' electronic purse case study. In *Proc. of Formal Methods Europe (FME 2002)*. Springer-Verlag, 2002. To Appear.

[5] J. Corbett *et al.* Bandera: Extracting finite-state models from Java source code. In *Proc. 22nd ICSE*, June 2000.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, pages 238–252. ACM, 1977.

[7] R. DeLine and M. Fähndrich. Enforcing high-level protocols in low-level software. In *Proc. PLDI 2001*, pages 59–69, 2001.

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq SRC, Dec. 1998.

[9] D. L. Detlefs, G. Nelson, and J. B. Saxe. A theorem prover for program checking. Research Report 178, Compaq SRC, 2002. In preparation.

[10] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Englewood Cliffs, NJ, 1976.

[11] M. Dwyer, J. Hatcliff, and R. Howell. CIS 771: Software specification. Kansas State Univ., Dept. of Comp. and Inf. Sciences, Spring 2001.

[12] D. Engler *et al.* Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. 18th SOSP*, pages 57–72. ACM, 2001.

[13] M. D. Ernst *et al.* Dynamically discovering likely program invariants to support program evolution. In *Proc. ICSE 1999*, pages 213–224. ACM, 1999.

[14] Escher Technologies, Inc. Getting started with Perfect. Available from `www.eschertech.com`, 2001.

[15] D. Evans, J. V. Guttag, J. J. Horning, and Y. M. Tan. LCLint: A tool for using specifications to check code. In D. S. Wile, editor, *Proc. 2nd SIGSOFT FSE*, pages 87–96. ACM, 1994.

[16] C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Inf. Process. Lett.*, 77(2–4):97–108, Feb. 2001.

[17] C. Flanagan and K. R. M. Leino. Houdini, an annotation assistant for ESC/Java. In J. N. Oliveira and P. Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *LNCS*, pages 500–517. Springer, Mar. 2001.

[18] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proc. 29th POPL*, page to appear. ACM, Jan. 2002.

[19] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Proc. 28th POPL*, pages 193–205. ACM, 2001.

[20] S. Graf and H. Saïdi. Construction of abstract state graphs via PVS. In O. Grumberg, editor, *Proc. 9th CAV*, volume 1254 of *LNCS*, pages 72–83. Springer, 1997.

[21] A. Heydon and M. A. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, Dec. 1999.

[22] C. A. R. Hoare. Proof of correctness of data representations. *Acta Inf.*, 1(4):271–81, 1972.

[23] S. C. Johnson. Lint, a C program checker. Comp. Sci. Tech. Rep. 65, Bell Laboratories, 1978.

[24] B. W. Lampson, J. J. Horning, R. L. London, J. G. Mitchell, and G. J. Popek. Report on the programming language Euclid. Technical Report CSL-81-12, Xerox PARC, Oct. 1981.

[25] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06f, Dept. of Comp. Sci., Iowa State Univ., July 1999.

[26] G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, pages 105–106. ACM, 2000.

[27] K. R. M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *FOOL 4*, 1997.

[28] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *Proc. OOPSLA '98*, pages 144–153. ACM, 1998.

[29] K. R. M. Leino. Applications of extended static checking. In P. Cousot, editor, *8th Intl. Static Analysis Symp.*, volume 2126 of *LNCS*, pages 185–193. Springer, July 2001.

[30] K. R. M. Leino. Extended static checking: A ten-year perspective. In R. Wilhelm, editor, *Informatics—10 Years Back, 10 Years Ahead*, volume 2000 of *LNCS*, pages 157–175. Springer, Jan. 2001.

[31] K. R. M. Leino and R. Manohar. Joining specification statements. *Theoretical Comp. Sci.*, 216(1–2):375–394, Mar. 1999.

[32] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq SRC, Nov. 2000.

[33] K. R. M. Leino, G. Nelson, and J. B. Saxe. ESC/Java user's manual. Tech. Note 2000-002, Compaq SRC,

Oct. 2000.

[34] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Proc. PLDI 2002*, 2002.

[35] K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs *et al.*, editor, *Formal Techniques for Java Programs*, Tech. Report 251. Fernuniversität Hagen, May 1999.

[36] K. R. M. Leino and R. Stata. Checking object invariants. Tech. Note 1997-007, DEC SRC, Jan. 1997.

[37] B. Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall Intl., 1988.

[38] T. Millstein. Toward more informative ESC/Java warning messages. In J. Mason, editor, *Selected 1999 SRC summer intern reports*, Tech. Note 1999-003. Compaq SRC, 1999.

[39] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. Technical Report 02-02, Dept. of Comp. Sci., Iowa State Univ., Feb. 2002. To appear in Concurrency, Practice and Experience.

[40] J. W. Nimmer and M. D. Ernst. Automatic generation and checking of program specifications. Technical Report 823, MIT Lab for Computer Science, Aug. 2001.

[41] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. K. Srivas. PVS: Combining specification, proof checking, and model checking. In R. Alur and T. A. Henzinger, editors, *Proc. 8th CAV*, volume 1102 of *LNCS*, pages 411–414. Springer, 1996.

[42] E. Poll, J. van den Berg, and B. Jacobs. Specification of the JavaCard API in JML. In J. Domingo-Ferrer, D. Chan, and A. Watson, editors, *Fourth Smart Card Research and Advanced Application Conference (CARDIS'2000)*, pages 135–154. Kluwer Acad. Publ., 2000.

[43] N. Sterling. WARLOCK — a static data race analysis tool. In *Proc. Winter 1993 USENIX Conf.*, pages 97–106. USENIX Assoc., Jan. 1993.

[44] M. Turin, A. Deutsch, and G. Gonthier. La vérification des programmes d'ariane. *Pour la Science*, 243:21–22, Jan. 1998. (In French).

[45] J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In *Proc. TACAS*, volume 2031 of *LNCS*, pages 299–312. Springer, 2001.

[46] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *International Conference on Automated Software Engineering*, Sept. 2000.

[47] J. M. Wing. *A Two-Tiered Approach to Specifying Programs*. PhD thesis, MIT Laboratory for Computer Science, May 1983. (Available as MIT LCS tech. report no. 229).

[48] J. B. Wordsworth. *Software Engineering with B*. Addison-Wesley, 1996.

[49] H. Xi. Imperative programming with dependent types. In *Proc. 15th LICS*, pages 375–387, June 2000.

[50] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proc. 26th POPL*, pages 214–227. ACM, Jan. 1999.