December 18, 1998

SRC Research Report

159

Extended Static Checking

David L. Detlefs
K. Rustan M. Leino
Greg Nelson
James B. Saxe

COMPAQ

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Systems Research Center

The charter of SRC is to advance both the state of knowledge and the state of the art in computer systems. From our establishment in 1984 by Digital Equipment Corporation (now Compaq), we have performed basic and applied research to support the company's business objectives. Our interests span scaleable systems (including hardware, networks, distributed systems, and programming languages and technology), the Internet (including the web, and internet appliances), and human/computer interaction.

Our strategy is to test the technical and practical value of our ideas by building hardware and software prototypes and using them as daily tools. Interesting systems are too complex to be evaluated solely in the abstract; extended use allows us to investigate their properties in depth. This experience is useful in the short term in refining our designs, and invaluable in the long term in advancing our knowledge. Most of the major advances in information systems have come through this strategy, including personal computing, distributed systems, and the Internet.

We also perform complementary work of a more mathematical flavor. Some of it is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. Other work explores new ground motivated by problems that arise in our systems research.

We have a strong commitment to communicating our results; exposing and testing our ideas in the research and development communities leads to improved understanding. Our research report series supplements publication in professional journals and conferences while our technical note series complements research reports and journal/conference publication by allowing timely dissemination of recent research findings. We seek users for our prototype systems among those with whom we have common interests, and we encourage collaboration with university researchers.

Extended Static Checking

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

December 18, 1998

Author Affiliations David L. Detlefs is a staff engineer at Sun Microsystems Laboratories. He can be reached at david.detlefs@sun.com. This work was completed by him and the other authors before he left SRC in 1996.

Abstract

The paper describes a mechanical checker for software that catches many common programming errors, in particular array index bounds errors, nil dereference errors, and synchronization errors in multi-threaded programs. The checking is performed at compile-time. The checker uses an automatic theorem-prover to reason about the semantics of conditional statements, loops, procedure and method calls, and exceptions. The checker has been implemented for Modula-3. It has been applied to thousands of lines of code, including mature systems code as well as fresh untested code, and it has found a number of errors.

0 Introduction

The authors of this paper were still children when the leading lights of computing declared that the world faced a "software crisis". While hardware improved every year, software was mired in complexity, and programming was expensive, error-prone, and grueling. But these gloomy observations didn't stop the software industry, which proceeded to grow its revenues and profits dramatically, and which has sustained that growth with continuous innovation. Was the software crisis a false alarm?

Not really. Although profitable, writing software is still expensive, error-prone, and grueling. Innovation in the software industry has mostly been confined to finding new things to do with software, and has not found many new ways to produce software. It is still common to see software disasters in which millions of dollars are spent writing a program that is abandoned before release, because the implementors simply can't get it to work. The tragic failure to establish software as a reliable engineering discipline is painfully clear from studies such as Leveson and Turner's *Investigation of the Therac-25 accidents* [31].

Although revenues and profits have grown, figures from economic consulting firm DRI/McGraw-Hill indicate that during a recent period of generally rising productivity, productivity in the software industry has actually fallen [10]. These computations may not be entirely accurate—since it is difficult to correct for inflation—but it seems safe to say that software productivity has not kept pace with productivity in other areas of the computing industry such as hardware. In fact, we would claim that the growth of the software industry has defied the stagnation of programming technology only because of the tailwind created by the million-fold improvement in price/performance of CPUs over the last thirty years.

As the information revolution moves out of its infancy, and we in the engineering community assume the responsibility of delivering on its vast promises, from robots to knowbots, it is insufficiently considered that our progress would be enormously accelerated if programming technology advanced on anything like the same curve as microprocessors.

Many silver bullets have been heralded as the solution to the software crisis. If you have a software problem, Structured Programming with PL/I is the answer! The Object-Oriented Revolution will bring mass production methods to software and make program fragments into inexpensive reusable reliable commodities! Program verification will eliminate all errors! Programming in natural language will eliminate the need for arcane languages; end users will express their requirements directly to the computer, and programming as a profession will wither away! None of the dreams behind these slogans has been fulfilled.

We believe that programming and its difficulties are here to stay. Instead of silver bullets, we advocate the strategy of studying the engineering practices of today's best programmers and developing practical tools that improve the process, without expecting to change its essential nature.

This paper describes a particular research project based on this strategy: the Extended Static Checker (called ESC), a programming tool that catches errors at compile time that ordinarily are not caught until runtime, and sometimes not even then. Examples are array index bounds errors, nil dereferences, and deadlocks and race conditions in multi-threaded programs. The tool is useful because the cost of an error is greatly reduced if it is detected early in the development process. The tool is intended to be like a type-checker or like the C tool *lint* [23]: its warnings are intended to be interpreted by the author of the program being checked.

The refinement of techniques for the prompt discovery of error serves as well as any other as a hallmark of what we mean by science.
— J. Robert Oppenheimer.

The checker is implemented using the technology of program verification. The program is annotated with specifications; the annotated program is presented to a "verification condition generator", which produces logical formulas that are provable if and only if the program is free of the particular class of errors under consideration, and these formulas are presented to an automatic theorem-prover.

This sounds like program verification, but it is not: firstly because we don't try to prove that a program does what it is supposed to do, only to check for certain specific types of errors; secondly because we are interested in failed proofs only, not in successful ones. A crucial point is that failed proofs are more useful than successful ones, since they warn the programmer of possible errors. In addition to being more useful, ESC is more feasible than full-scale program verification. For example, an unsound full-scale program verifier is an oxymoron, but the amount of unsoundness to tolerate in a static checker is a matter of engineering judgment: nobody expects a type-checker or a lint tool to find *all* errors; its utility is determined by the number of errors it finds weighed against the cost of running the tool. Similarly, since we are not promising that ESC will find all errors, we are free to declare that some kinds of errors are out of the tool's range.

This idea of extended static checking is not new. The first Ph.D. thesis that we know of that addressed the idea was by Dick Sites a quarter of a century ago [44], and the problem has held its own as a Ph.D. thesis topic ever since. But the research prototype checkers that have been implemented over the decades have made too many simplifying assumptions. They may not handle dynamically allocated data or object-oriented programming; they may not handle concurrency; they may require the source for the entire program in order to check any part of it; they may require the user to guide the theorem-prover or to provide complicated loop invariants. Such simplifying assumptions are a way of separating concerns, which can help to focus on particular aspects of the problem, thus gaining depth of insight. Indeed, our work builds on the deep insights produced by many earlier researchers, but our ESC project has followed a complementary mode of research, in which every effort is made to produce and test a realistic artifact. This complementary mode of research can often reveal surprises.

Most previous checking tools were tested only on small programs written by the tool implementors themselves. In contrast, our plan was to run our checker on significant pieces of the Modula-3 runtime system. By tackling realistic programs written by others (or in some cases, by ESC project members before the ESC project was launched), we hoped to learn the answers to the following questions:

- Could we generate verification conditions for such "systems programs"?
- Would we be able to turn failed verifications into specific error messages?
- How much of a burden would it be to write the necessary annotations?
- To what extent could we automate the theorem-proving task?

In summary, we were determined to stress-test an idea that had long been in gestation.

Our checker handles multi-threaded multi-module object-oriented programs. Our checker works on Modula-3 programs [40], but the techniques would work for any language in which address arithmetic is restricted, including Java, Ada, Oberon, and FORTRAN. Indeed, as this paper goes to press, a follow-on project has replicated the technology in a tool for Java.

Our checker performs modular checking: you can use it to check selected modules of a program without checking the entire program. Since modern programming is inconceivable without libraries, we consider modular checking to be essential.

The checker also allows you to check for selected classes of errors; for example, it is often useful to check for deadlocks and race conditions without checking for array index bounds errors.

When the checker produces spurious warnings, there are a variety of ways to suppress them, that is, to get the checker to ignore the spurious warnings and continue to report real errors.

Although our checker is a research prototype, with plenty of rough edges, we feel that it demonstrates the promise of the technology more clearly than previous checkers: ESC catches errors that no type-checker could possibly catch, yet it feels to the programmer more like a type-checker than a program verifier. The specifications required are statements of straightforward facts like inequalities, the error messages are specific and accurate with respect to source position and error type, and the theorem-proving is carried out behind the scenes automatically.

A type-checker is a Neanderthal program verifier: it isn't very smart, but it's hardy and it's friendly.
— Jim Morris.

```
SPEC ⟨procedure or method name⟩ ( ⟨formal parameter names⟩ )
  MODIFIES ⟨list of variables⟩
  REQUIRES ⟨precondition⟩
  ENSURES ⟨postcondition⟩
```

Figure 0: Procedure specification syntax.

Our main goal for this paper is to convey to the reader what it feels like to use a checker that operates in this intermediate level between type-checking and verification. Therefore, the paper consists largely of two examples. After the examples, we describe at a rather general level some of the novel aspects of our checker, but we have relegated many details to other papers.

1 The specification language

In this section, we lay the groundwork for the examples that form the heart of the paper by briefly introducing ESC's specification language. The design of the specification language reflects the structure of well-designed programs.

One pillar of program structure is *procedural abstraction*, in which a compound operation is named and parameterized and used by the rest of the program as though it were an elementary operation.

Procedural abstraction makes static analysis difficult, so difficult that many static analysis methods described in the literature apply only to programs without procedures, and many practical compilers perform no inter-procedural analysis at all, even if they perform extensive intra-procedural analysis. The reason for this is that the authors of compilers and other tools have not wanted to rely on specifications, but inter-procedural analysis fundamentally requires dealing with specifications, whether supplied by the programmer or inferred by the tool.

To perform inter-procedural analysis, ESC relies on programmer-supplied specifications of the form illustrated in Figure 0. Pre- and postconditions are formulas in a first-order theory that includes the Modula-3 built-in operators.

A procedure specification is a contract between the implementer and the client: the client contracts to call the procedure in a state where the precondition is true, and the implementer contracts (a) to modify no variable except those in the `MODIFIES` list, and (b) to return only in a state satisfying the postcondition. Thus when checking the body of the procedure, ESC assumes that the precondition is true initially, and checks that when the procedure returns the postcondition is true and only those variables in the `MODIFIES` clause have been modified. Conversely,

```
SPEC VAR v : ⟨type⟩
SPEC REP v = ⟨concrete representation⟩
```

Figure 1: Data abstraction syntax.

when checking a client, ESC checks that the precondition is true at the point of call, and assumes that the call respects both the postcondition and the `MODIFIES` clause.

For example, here are a Modula-3 declaration and ESC specification of a simple random number generator:

```
VAR seed: INTEGER;
PROCEDURE Rand(n: INTEGER): INTEGER;
<* SPEC Rand(n)
     MODIFIES seed
     REQUIRES 0 < n AND seed # 0
     ENSURES 0 <= RES AND RES < n AND seed' # 0 *>
```

The example also illustrates several other points. First, for a variable listed in the `MODIFIES` clause (such as `seed`), occurrences in the postcondition can be unprimed or primed (`seed'`). A primed occurrence denotes the value of the variable in the post-state, while an unprimed occurrence denotes the value of the variable in the pre-state. Second, the reserved name `RES` is used to denote the result of the procedure, if any. Third, Modula-3 pragma brackets, `<* *>`, surround the ESC annotations to distinguish them from ordinary Modula-3. Fourth, `x # y` is Modula-3 syntax to assert that the values of `x` and `y` are different.

Another pillar of program structure is *data abstraction*, in which a collection of program variables (or *concrete variables*) are considered conceptually to represent a single *abstract variable*. The abstract variable is used by clients for reasoning about the semantic effect of operations on the abstraction, and the concrete variables are used by the implementation to operate efficiently on the state.

For example, the abstract value of a complex number z could be represented concretely in terms of two floating point Cartesian coordinates $z.x$ and $z.y$; alternatively the same abstraction might be represented in terms of the polar coordinates $z.r$ and $z.theta$. Generally, the module structure of a program is arranged so that the concrete representation of an abstraction is invisible (that is, out of scope) within the modules that use the abstraction (its *clients*) [42].

Figure 1 shows the ESC syntax for declaring an abstract variable and specifying its *representation* (also known as its *abstraction function*). For example,

```
<* SPEC VAR a: INTEGER *>
VAR c: INTEGER;
<* SPEC REP a = c*c *>
```

declares an abstract integer variable `a`, a concrete integer variable `c`, and specifies that the square of `c` represents `a`.

We haven't described the whole specification language, but we have described enough to give the first example.

2 An elementary application of Modula-3 writers

Both of the two extended examples in this paper use Modula-3's standard I/O library, which is designed around the key abstraction of a monitored object-oriented buffered stream. In particular, a *reader* is an input stream and a *writer* is an output stream. Each stream object contains a buffer and methods for managing the buffer. Different stream subclasses override the methods in different ways, so that, for example, a file reader refills the buffer from the disk and a network reader refills the buffer from the network. Class-independent code can be common to all subclasses; for example, `Wr.PutInt` writes the ASCII representation of an integer into a writer's buffer. The Modula-3 design is described in Chapter 6 of Nelson's Modula-3 book [40], the original design is described by Stoy and Strachey [46]. We describe various aspects of the relevant interfaces as we need them for our examples.

For each type of stream, the Modula-3 I/O library provides two interfaces, a basic interface for simple clients, and an advanced interface that offers additional functionality (in particular, access to the buffer structure) at the price of additional complexity.

The basic writer interface. Our first example uses only the basic writer interface `Wr`, which is presented in Figure 2. Here is a translation of Figure 2 from Modula-3 into English. The writer class is given the global name `Wr.T`, where `Wr` is the name of the interface and `T` is by convention the principal type declared in the interface. The class is declared as an opaque object type, publicly known only to be a subtype of the built-in class `ROOT`. (In Modula-3, classes correspond to object types.) The actual declaration of the representation type is hidden in the advanced interface, and is invisible to clients of the basic interface. The procedures `PutChar`, `PutText`, and `Close` have the given signatures and specifications (`TEXT` is Modula-3's predeclared string type). (The actual interface has eleven procedures instead of three, but these are representative. The interested reader can

```
INTERFACE Wr;

TYPE T <: ROOT;

<* SPEC VAR valid: MAP T TO BOOLEAN *>
<* SPEC VAR state: MAP T TO ANY *>

PROCEDURE PutChar(wr: T; ch: CHAR);

<* SPEC PutChar(wr, ch)
     MODIFIES state[wr]
     REQUIRES valid[wr] *>

PROCEDURE PutText(wr: T; txt: TEXT);

<* SPEC PutText(wr, txt)
     MODIFIES state[wr]
     REQUIRES valid[wr] AND txt # NIL *>

PROCEDURE Close(wr: T);

<* SPEC Close(wr)
     MODIFIES valid[wr] *>

END Wr.
```

Figure 2: The (simplified) writer interface.

find the ESC-annotated version of the full interface referenced from the Extended Static Checking homepage on the Web [9].)

A `U`-valued field `f` declared in a class `T` corresponds semantically to a map from `T` to `U`. Thus the declarations of `valid` and `state` can be thought of as `BOOLEAN`-valued and `ANY`-valued fields of writers. Although Modula-3 uses the syntax `x.f` to denote the `f` field of an object `x`, the annotation language uses the syntax `f[x]` when `f` is abstract.

The specifications for the writer interface fall into a common pattern that we call the *state/validity paradigm*. In this paradigm, there are two abstract variables, `valid` and `state`. The idea is that `valid[wr]` represents the condition that `wr` is a properly-initialized valid writer, and `state[wr]` represents all other state of `wr` (for example, its contents and position). If we were doing full-scale program verification, there would be pages of specifications about `state`; but since we are doing extended static checking only, there is almost nothing to say about `state`, except to specify that some procedures may modify it. (Not even the type of `state` is relevant, so we use the special ESC type `ANY`.) In the state/validity paradigm, specifications become very stylized. A typical procedure or method operating on a writer `wr` (like `PutChar`) has the specification

```
MODIFIES state[wr]
REQUIRES valid[wr]
```

Because `valid` does not appear in the `MODIFIES` list, this specification implies the preservation of validity.

Often a few procedures will have some additional annotations—for example, `PutText` requires that its text argument be non-`NIL`—but for simple interfaces the stylized specifications of the state/validity paradigm are the lion's share of what has to be written for an ESC verification.

The text writer interface. Figure 3 illustrates the *text writer* interface, a particular writer subclass that occurs in our example. A text writer is a writer that doesn't do any output; it just stores everything that has been written in an internal buffer and provides a procedure `GetText` that returns the contents of the buffer as a `TEXT`.

Because `TextWr.T` is a subclass of `Wr.T`, the specification variables `valid` and `state` apply to text writers, and are used in the specification of the text writer interface. The method `init` initializes a text writer (establishes `valid'[twt]`) and returns it (establishes `RES=twt`). The specification is entirely typical for initialization methods in the state/validity paradigm. Similarly, the specification of `GetText` is typical of the state/validity paradigm, with one additional postcondition conjunct `RES # NIL`.

```
INTERFACE TextWr;
IMPORT Wr;

TYPE T <: Wr.T OBJECT
  METHODS
    init(): T
  END;

<* SPEC T.init(twr)
     MODIFIES Wr.valid[twr], Wr.state[twr]
     ENSURES Wr.valid'[twr] AND RES=twr *>

PROCEDURE GetText(twr: T): TEXT;

<* SPEC GetText(twr)
     MODIFIES Wr.state[twr]
     REQUIRES Wr.valid[twr]
     ENSURES RES # NIL *>

END TextWr.
```

Figure 3: The text writer interface.

```
1 PROCEDURE ArrayCat(a: ARRAY OF TEXT): TEXT =
2   VAR twr := NEW(TextWr.T); BEGIN
3     FOR i := 1 TO NUMBER(a) DO
4       Wr.PutText(twr, a[i])
5     END;
6     RETURN TextWr.GetText(twr)
7   END ArrayCat;
```

Figure 4: The first (erroneous) version of the example program uses a text writer `twr` to accumulate the growing concatenation of the elements of `a`.

Example. Our first example program is the kind of elementary programming exercise that might be assigned to students first learning to program with output streams, and we have seeded our program with elementary errors. The problem is to program a procedure that takes an array of texts as an argument and returns a single text containing the concatenation of all the texts in the array. This could be done rather straightforwardly by repeatedly calling the Modula-3 binary concatenation operation `Text.Cat`, but doing so leads to a performance trap, since with most implementations of text concatenation, the total cost in time of this simple approach can be proportional to the square of the length of the final result. A good way to avoid this quadratic cost is to use a text writer, which leads us to the procedure `ArrayCat` shown in Figure 4. The procedure allocates a text writer, writes the elements of the array to the writer in order, and finally retrieves and returns a text containing everything that was written. (This approach avoids the quadratic cost if text writers are well-implemented.)

Running ESC on the `ArrayCat` procedure of Figure 4 produces a warning about an array index bounds error:

array index bounds error, line 4:

```
    Wr.PutText(twr, a[i])
```

(The exact format of an ESC error message is identical to that of a compiler error message; in this paper, we use italics and underlining to convey the same information.) The error message also includes a so-called "error context" which is a long list of atomic formulas that characterize the situation in which the error can occur. Because it is long, we won't show the error context here, but we remark that a study of the context reveals that it implies the formula `i=NUMBER(a)`, which is in fact the condition in which the array bounds error can occur: in Modula-3, open arrays are indexed from 0, but the `FOR` loop was written as though they were indexed from 1. Correcting the error in one natural way produces the following improved program:

```
1 PROCEDURE ArrayCat(a: ARRAY OF TEXT): TEXT =
2   VAR twr := NEW(TextWr.T); BEGIN
3     FOR i := 0 TO NUMBER(a)-1 DO
4       Wr.PutText(twr, a[i])
5     END;
6     RETURN TextWr.GetText(twr)
7   END ArrayCat;
```

This version of the loop eliminates the warning about the array bounds error. But now ESC complains about this program as follows:

precondition failed, line 4:

```
Wr.PutText(twr, a[i])
```

A study of the error context reveals that it implies the formula `NOT valid[twr]`. That is, ESC has detected and warned about the failure to initialize `twr` (the program allocated the text writer, but failed to initialize it). To correct this error, we add a call to the `init` method, which requires inserting the seven characters "`.init()`":

```
1 PROCEDURE ArrayCat(a: ARRAY OF TEXT): TEXT =
2   VAR twr := NEW(TextWr.T).init(); BEGIN
3     FOR i := 0 TO NUMBER(a)-1 DO
4       Wr.PutText(twr, a[i])
5     END;
6     RETURN TextWr.GetText(twr)
7   END ArrayCat;
```

This correction eliminates both of the previous warnings, but ESC gives one more warning:

precondition failed, line 4:

```
Wr.PutText(twr, a[i])
```

A study of the error context shows that the `TEXT` argument to `PutText` is equal to `NIL`, which is forbidden by the precondition of `PutText` but not ensured by `ArrayCat`, which blindly passes `a[i]`, whether or not it is `NIL`. This error forces a rethinking of the design for `ArrayCat`: what should we do about `NIL` entries in the `TEXT` array? Two designs come immediately to mind: to ignore `NIL`s or to forbid `NIL`s. Either design is easy to get through ESC. In the design where `NIL`s are ignored, the procedure is recoded as follows:

```
PROCEDURE ArrayCat(a: ARRAY OF TEXT): TEXT =
  VAR twr := NEW(TextWr.T).init(); BEGIN
    FOR i := 0 TO NUMBER(a)-1 DO
      IF a[i] # NIL THEN Wr.PutText(twr, a[i]) END
    END;
    RETURN TextWr.GetText(twr)
  END ArrayCat;
```

and of course ESC, which understands `IF` statements, is perfectly happy with this version. In the design where `NIL`s are forbidden, the specification for `ArrayCat` is strengthened with a quantified precondition:

```
<* SPEC ArrayCat(a)
     REQUIRES (ALL [i: INTEGER]
                 0 <= i AND i < NUMBER(a)
                 IMPLIES a[i] # NIL ) *>
```

ESC is perfectly happy with this design, too: the stronger precondition suppresses the error message. Furthermore, ESC will enforce the stronger precondition wherever `ArrayCat` is called.

We would like to make several comments about this example.

First, although careful specifications were required for the writer and text writer interfaces, the beginning programmer was able to make use of ESC without writing any specifications for his program at all. No preconditions or loop invariants were required in `ArrayCat`. We think that this is as it should be: anybody qualified to design the interfaces of a stream library understands preconditions and postconditions and abstractions at some level, and will find an explicit notation for their design decisions to be a useful tool rather than a burden. On the other hand, many simple errors in programs can and should be identified by reading the unannotated erroneous program; to require a loop invariant in order to check `ArrayCat` seems pedagogical and heavy-handed.

Second, the reader should be aware that, although we have concentrated in this example on the checking of a client of the I/O system, we have in fact also used ESC to check the implementation of text writers. In the implementation, a representation declaration is made to give the concrete meaning of `valid[twr]` in terms of the concrete fields of `twr`. And this representation is used by ESC when checking the body of procedures like `PutText` and `GetText`, that require validity as a precondition and whose implementations depend on the concrete meaning of validity.

A third remark that this example allows us to make is that it is up to the user to choose a point on the continuum between full functional-correctness verification and minimal extended static checking. For example, it is in fact true that initializing a text writer leaves its contents empty, but our specifications did not reflect this truth. If we wanted to, we could reflect this by strengthening the postcondition of `init` along the following lines:

```
<* SPEC T.init(twr)
     MODIFIES Wr.valid[twr], Wr.state[twr]
     ENSURES Wr.valid'[twr] AND RES=twr
             AND Wr.state'[twr] = "" *>
```

(This would also require changing the type of the state of a writer from `ANY` to `SEQ[CHAR]`. Also, the notation `""` is not actually correct for the empty character sequence.) It would be easy to concoct an artificial example in which this stronger specification would be essential for some ESC verification. For example, the absence of array bounds errors in some client might depend on the fact that a newly initialized text writer is empty. But this is a slippery slope. If `init`'s effect on the state is specified fully, why not `PutChar`'s as well? Without discipline, you can quickly slide into the black hole of full correctness verification. Luckily, our experience has been that many ESC verifications can be successfully completed with almost no specifications at all about the contents and meanings of abstract types, other than the specification of validity. You can go a long way just relying on the state/validity paradigm: that is, the specifications for each procedure record accurately how the procedure affects and requires validity, but all other side effects are swept under the ample rug of `MODIFIES state[wr]`. We believe this is a key reason why ESC verifications can be more cost effective than full correctness verifications.

3 An advanced application of readers

In this section, we will describe the use of ESC on a more sophisticated program, `WhiteSpace.Skip`. The example is also a short client program of the Modula-3 I/O system, but differs in several ways from the example in the previous section. A rather minor difference is that this program is a client of input streams rather than output streams. A more important difference is that in this example, we will pay attention to the synchronization protocol that is designed into both readers and writers. In the previous example, we omitted synchronization in order to simplify the exposition. Another important difference is that in this section we will see a program that uses the advanced interface to deal with the buffer structure of the stream, instead of exclusively using the procedures in the basic interface.

`PROTECT v BY mu`	shared variable `v` is not to be accessed without holding the lock `mu`
`PROTECT f BY SELF`	for every object `t`, shared field `t.f` is not to be accessed without holding the lock `t`
`LL`	set of locks held by the current thread
`sup`	supremum (maximum) in the programmer-declared locking order

Figure 5: Locking-level syntax. The second form of `PROTECT` can be used only when `f` is a field declared in a subclass of `MUTEX`.

How ESC checks for synchronization errors. Our experience has been that many synchronization errors are failures to acquire locks (causing race conditions) or acquiring locks out of order (causing deadlocks). Therefore, we have designed the ESC annotation language to catch these simple errors; Figure 5 shows the syntax. The programmer declares which locks protect which shared variables and which locks can or must be held on entry to various procedures. The programmer also declares a partial order in which threads are allowed to acquire locks. ESC checks that shared variables are never accessed without holding the lock that protects them, and also checks that threads acquire locks in strictly increasing order (Modula-3 features non-reentrant locks). This doesn't prove correctness—more expensive techniques like monitor invariants would be required for that—but it does catch many common errors.

The locking order on mutexes is denoted by "<", and the programmer specifies it using a general facility for adding axioms to an ESC verification: `SPEC AXIOM`. For example, the Modula-3 window system is based on an object called a `VBT`. `VBT`s are arranged in trees, and can be locked only from a leaf of the tree toward the root, not vice versa. This rule is declared in the `VBT` interface as follows:

```
<* SPEC AXIOM (ALL [v: VBT.T] v < v.parent) *>
```

Axioms about the locking order arise only in subtle situations. In particular, the example we are about to present acquires only one reader lock at a time, so we don't need to declare axioms about the locking order.

The basic reader interface. Figure 6 shows the basic Modula-3 interface `Rd`, including its ESC specifications. The only new features of Figure 6 relate to concurrency. The synchronization protocol designed into readers is highly stylized, we call it the *monitored-object paradigm*: An object is treated like a monitor in that mutual exclusion is provided for threads operating on the object via procedure

```
INTERFACE Rd;
IMPORT Thread;
EXCEPTION EndOfFile; Failure(TEXT);

TYPE T <: MUTEX;

<* SPEC VAR valid: MAP T TO BOOLEAN *>
<* SPEC VAR state: MAP T TO ANY *>

PROCEDURE GetChar(rd: T): CHAR
    RAISES {EndOfFile, Failure, Thread.Alerted};

<* SPEC GetChar(rd)
     MODIFIES state[rd]
     REQUIRES valid[rd] AND sup(LL) < rd *>

PROCEDURE EOF(rd: T): BOOLEAN
    RAISES {Failure, Thread.Alerted};

<* SPEC EOF(rd)
     MODIFIES state[rd]
     REQUIRES valid[rd] AND sup(LL) < rd *>

PROCEDURE UnGetChar(rd: T);

<* SPEC UnGetChar(rd)
     MODIFIES state[rd]
     REQUIRES valid[rd] AND sup(LL) < rd *>

PROCEDURE Seek(rd: T; n: CARDINAL)
    RAISES {Failure, Thread.Alerted};

<* SPEC Seek(rd, n)
     MODIFIES state[rd]
     REQUIRES valid[rd] AND sup(LL) < rd *>

END Rd.
```

Figure 6: The (simplified) basic reader interface.

calls and method calls. The mutual exclusion is achieved by locking the object itself, whose type is a subtype of `MUTEX`, Modula-3's predeclared mutual exclusion semaphore type. Acquiring the lock is equivalent to entering the monitor.

The text of the interface in Figure 6 reflects the monitored-object paradigm in two ways. First, `Rd.T` is declared to be an opaque subtype not of `ROOT` but of `MUTEX`. Second, the monitor entry procedures have the extra precondition `sup(LL) < rd`, which reflects the requirement that they be called from a state in which it is legal to acquire the lock `rd`.

The advanced reader interface. The basic `Rd` interface is the one used by most simple clients, but it is insufficient for more sophisticated clients. For example, since it hides the buffer and the method for refilling the buffer, it is insufficient for clients that implement new classes of readers. Figure 7 shows the `RdrRep` interface, which provides the specifications needed for more sophisticated clients. With the `RdrRep` interface we get beyond the boiler-plate ESC specification paradigms, and start to put the specification language through its paces.

The interface begins by revealing the representation of the type `Rd.T`, which is opaque in the basic interface. The representation is an object type containing a `buff` field, which is a reference to an array of characters. The Modula-3 keyword `BRANDED` substitutes name equivalence for Modula-3's default structural equivalence for types. In addition to the `buff` field, `Rd.T` also contains several integer and boolean fields. The integer fields determine the active portion of the buffer, according to a convention illustrated in Figure 8. The boolean fields are irrelevant for this example. The full interface contains several methods, but we show only the `seek` method, since the others are irrelevant for our example.

The `SPEC PROTECT` annotation specifies that the fields of a reader are protected by the reader itself; that is, a thread is not allowed to read or write any of the reader's fields unless it has acquired the reader lock. This annotation is typical of the monitored-object paradigm.

Next we come to the specification of `seek`. This method is responsible for performing the class-specific computation involved in repositioning the buffer: the call `rd.seek(n)` repositions the buffer so that byte number `n` of the source of the reader is present in the buffer. In particular, `rd.seek(rd.hi)` will advance to the next buffer of data (since `rd.hi` is the index of the first byte that is beyond the current buffer of data). The method returns `SeekResult.Ready` if the repositioning is successful; if `n` is beyond the end of the reader, it returns `SeekResult.Eof`. (If `dontBlock` is set and the `seek` method can't do its job without risking blocking, it is allowed to return `SeekResult.WouldBlock`. But that isn't relevant for this example.)

```
INTERFACE RdrRep;
IMPORT Rd, Thread;

TYPE
  SeekResult = {Ready, WouldBlock, Eof};
  CharRefArray = BRANDED REF ARRAY OF CHAR;

REVEAL Rd.T = MUTEX BRANDED OBJECT
    buff: CharRefArray;
    st, lo, cur, hi: CARDINAL;
    seekable, intermittent: BOOLEAN
  METHODS
    seek(n: CARDINAL; dontBlock := FALSE): SeekResult
        RAISES {Rd.Failure, Thread.Alerted}
  END;

<* SPEC PROTECT
     Rd.T.buff, Rd.T.st, Rd.T.lo, Rd.T.cur, Rd.T.hi,
     Rd.T.seekable, Rd.T.intermittent
   BY SELF *>

<* SPEC Rd.T.seek(rd, n, dontBlock)
     MODIFIES Rd.state[rd]
     REQUIRES Rd.valid[rd] AND sup(LL) = rd *>

<* SPEC DEPENDS Rd.valid[rd: Rd.T] ON
     rd.st, rd.lo, rd.cur, rd.hi, rd.buff *>

<* SPEC REP Rd.valid[rd: Rd.T] IFF
     {NonNil: rd # NIL} AND
     {BuffNonNil: rd.buff # NIL} AND
     {LoBeforeCur: rd.lo <= rd.cur} AND
     {CurBeforeHi: rd.cur <= rd.hi} AND
     {BuffAmple: rd.st+rd.hi-rd.lo <= NUMBER(rd.buff^)} *>

<* SPEC DEPENDS Rd.state[rd: Rd.T] ON
     rd.st, rd.lo, rd.cur, rd.hi, rd.buff, rd.buff^,
     rd.seekable, rd.intermittent *>

END RdrRep.
```

Figure 7: The (simplified) advanced reader interface.

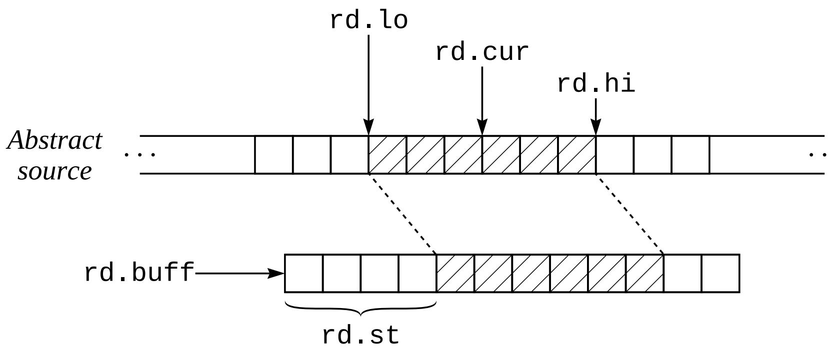

Figure 8: Reader representation.

The ESC specification of the `seek` method is much simpler than the informal functional specification. The specification is the standard one for a monitored object in the state/validity paradigm. The only new point to notice is that the locking-level precondition is `sup(LL)=rd` instead of `sup(LL)<rd`. This reflects the design decision that `seek` is a so-called internal monitor method (to be called from within the monitor) rather than a monitor entry method (to be called from outside the monitor).

The `SPEC DEPENDS` annotations can be ignored for now; they will be explained in Section 5.

The `SPEC REP` declaration specifies the concrete representation of the abstract variable `valid[rd]`. Clients of the basic `Rd` interface care whether readers are valid, but don't care what validity means in concrete terms; clients of `RdrRep` do care, since they have access to the reader's concrete representation. Thus this interface is an appropriate place to declare this representation. The `SPEC REP` declaration for `Rd.valid` declares that a reader is valid if it is non-nil, its buffer is non-nil, its `lo`, `cur`, and `hi` fields are in ascending order, and its buffer's size is ample. The conjuncts of the formula are labeled; these labels are optional, but including them allows ESC to make its error messages more useful. Figure 8 illustrates these conditions.

The `RdrRep` interface is fairly subtle. Instead of the simple idiom of `valid` and `state`, it exercises many of the features of ESC's specification language. This is because the interface must carefully balance the requirements of simple clients, subclass implementations, and the class-independent implementation. Designing such a critical interface is inherently difficult. We believe that any programmer who is skilled enough to do a good job designing the `RdrRep` interface will not be intimidated by the specification language.

Example. Behind our next example there is a story. A student working in our laboratory as a research intern had written a parser, and when his parser was slow, he complained to one of the authors (Nelson) that the Modula-3 input library was inefficient. Nelson asked the intern to measure the performance more carefully and figure out where the time was going.

The intern reported back the next day that “you wouldn’t believe it, but almost all the time is going into skipping white space in the lexer!”. Nelson said he did believe this report, and asked how the lexer skipped white space. The answer was:

```
MODULE WhiteSpace;
IMPORT Rd, Thread;

CONST WhiteChars =
        SET OF CHAR {' ', '\t', '\n', '\r'};

PROCEDURE Skip(rd: Rd.T) RAISES {Rd.EndOfFile,
    Rd.Failure, Thread.Alerted} =
  VAR ch: CHAR; BEGIN
    REPEAT
      ch := Rd.GetChar(rd)
    UNTIL NOT ch IN WhiteChars;
    Rd.UnGetChar(rd)
  END Skip;

BEGIN
END WhiteSpace.
```

Nelson suggested to the intern that instead of making a monitor entry call per character, it would be more efficient to import `RdrRep` and skip the white space directly in the buffer itself. The intern resisted, protesting “Isn’t that a violation of abstraction? I can’t believe that *you*, Greg Nelson, of all people, would violate abstraction by pawing over the grotty buffer!”. Nelson made an appropriate response, and the intern implemented a new version. Later, he reported back cheerfully, “I tried your idea and it worked! And you’re right, it’s much faster!”. But the intern complained that the `RdrRep` interface was confusing and requested that Nelson check over his code.

Nelson has a vivid memory of the intern’s program: it is listed in Figure 9. The procedure consists of a single loop. The loop begins by testing `rd.cur <`

```
MODULE WhiteSpace;
IMPORT Rd, RdrRep, Thread;

CONST WhiteChars =
        SET OF CHAR {' ', '\t', '\n', '\r'};

PROCEDURE Skip(rd: Rd.T)
    RAISES {Rd.Failure, Thread.Alerted} =
  BEGIN
    LOOP
      IF rd.cur < rd.hi THEN
        IF rd.buff[rd.cur] IN WhiteChars
          THEN INC(rd.cur)
          ELSE RETURN
        END
      ELSIF Rd.EOF(rd)
        THEN RETURN
        ELSE Rd.Seek(rd, rd.cur)
      END
    END
  END Skip;

BEGIN
END WhiteSpace.
```

Figure 9: The intern’s attempt at writing an efficient procedure that skips white space.

`rd.hi`. If the test succeeds, the current character is present in the buffer, and it can be tested for whiteness and skipped if necessary. If the test `rd.cur < rd.hi` fails, the procedure has exhausted the current buffer without finding a non-white character. In this case, the loop uses the call `Rd.Seek(rd, rd.cur)` to advance to the next buffer, after testing that there is another buffer via the call `Rd.EOF(rd)`.

This all occurred before ESC was written, so the errors in the intern's code were found and removed by other means. But today we can run ESC on the code. Running it produces, first, the following error message:

nil dereference error, line 11:

```
IF rd.cur < rd.hi THEN
```

This warning is boring; it is just ESC's way of saying that it can't do much useful checking unless you provide it with a specification. Of course the client is not supposed to call `WhiteSpace.Skip(rd)` if `rd` is `NIL`, or for that matter if `rd` is not valid. So to get rid of this spurious error, we add to the interface the typical specification

```
<* SPEC Skip(rd)
     MODIFIES Rd.state[rd]
     REQUIRES Rd.valid[rd] *>
```

And now running ESC produces a more interesting error:

race condition reading shared field, line 11:

```
IF rd.cur < rd.hi THEN
```

This warning is useful: the program erroneously reads (and in fact also writes) shared data fields without holding the lock that protects them.

It is not surprising that the intern made this error. Programmers who are not experienced at concurrent programming have a regrettable tendency to ignore all the comments about concurrency in an interface they are using, and since casual testing rarely reveals these errors, they do not surface until much later.

To fix the race condition, we must either require that the reader be locked on entry to the procedure, or else we must lock the reader inside the procedure. The later fix is preferable, since it makes `WhiteSpace.Skip` a monitor entry procedure, just like `GetChar` or `EOF`. Therefore, let us fix the error by bracketing the body of the procedure with `LOCK rd DO ... END`. At the same time, we change the precondition to be as follows:

```
REQUIRES Rd.valid[rd] AND sup(LL) < rd
```

(If we had forgotten to add `sup(LL) < rd` to the precondition, ESC would warn of a possible deadlock at the `LOCK rd`, since the verifier can't prove that it is legal to lock `rd` on entry to the procedure.)

Now the checker gets one line further before it complains:

array index bounds error, line 12:

```
    IF rd.buff[rd.cur] IN WhiteChars
```

The intern has confused stream indexes with buffer indexes. The index `rd.cur` is a stream index and could be enormous. Looking at Figure 8, we see that the correction is to substitute

```
rd.buff[rd.st+rd.cur-rd.lo]
```

for `rd.buff[rd.cur]`. It is interesting to note that the array index bounds error in `ArrayCat` that was detected in Section 2 was an off-by-one error, which could have been detected by many ad-hoc techniques. But the bounds error in `WhiteSpace.Skip` is not an off-by-one error; it is caused by a confusion over data structure invariants. We believe that the theorem-proving and verification methods used by ESC are necessary to catch such errors.

It may be a bit surprising that such a blatant error was not revealed by the intern's testing. Presumably, he tested his code only on standard disk files (where `st` is 0) and only on files that were smaller than the 8 KB reader buffers of the standard library. Within the first buffer, stream indexes and buffer indexes agree.

The next error from the checker is

precondition failed, line 16:

```
    ELSIF Rd.EOF(rd)
```

It is unfortunately common to introduce a deadlock when correcting a race condition, and this is just what we did above when we locked the reader on entry to `WhiteSpace.Skip`. The checker is warning us that the attempt to call `Rd.EOF(rd)` from within the procedure would self-deadlock. The warning message is "precondition failure", since the locking-level requirement for `Rd.EOF` is specified in its precondition.

To fix this deadlock, we observe that the procedure is somewhat inconsistent: the top of the loop is coded in the style of a client of the advanced interface `RdrRep`, for example by directly accessing the fields of the reader, but the bottom half is coded in the style of a client of the basic interface `Rd`, for example by calling the operations `Rd.EOF` and `Rd.Seek`. The correction is to code the bottom of the loop in the same style as the top, by calling the `seek` method directly. Instead of the `EOF` and `Seek` procedure calls in

```
ELSIF Rd.EOF(rd)
  THEN RETURN
  ELSE Rd.Seek(rd, rd.cur)
END
```

we invoke the `seek` method on `rd`:

```
ELSIF rd.seek(rd.cur) = RdrRep.SeekResult.Eof THEN
  RETURN
END
```

After this correction, the checker finds no more errors.

This concludes our second example. In the next few sections of the paper, we describe at a high level some of the crucial aspects of the design of our checker.

4 Tool architecture: A bird's eye view

Figure 10 shows a diagram of the major modules of our checker. The verification condition generator parses and type-checks an annotated program and produces a logical formula called the verification condition. This condition is valid if the program is consistent with its annotations and free of the errors in Figure 11. The condition is submitted to an automatic theorem-prover, just like in program verification, but unlike in program verification, we have no interest in the case where the theorem prover succeeds. Instead, the tool post-processes theorem-prover failures into meaningful error messages.

The checker is programmed in Modula-3. To parse and type-check Modula-3, it uses the Olivetti Modula-3 front-end toolkit, designed and implemented by Mick Jordan [24]. Not counting the toolkit, the verification condition generator is 34000 lines of code and the theorem-prover is 26000 lines of code. The system is available from the ESC home page on the Web [9].

5 Generalized data abstraction

An important property of our checker is that it works on individual modules; you don't need to provide it with a complete program. The checker reasons about procedure calls and method calls using specifications, not implementations. The basic idea of reasoning about procedure calls using preconditions, postconditions, and `MODIFIES` clauses has been understood for several decades, but we found that the basic idea that works so well on the examples in the program verification literature did not work on the standard Modula-3 libraries.

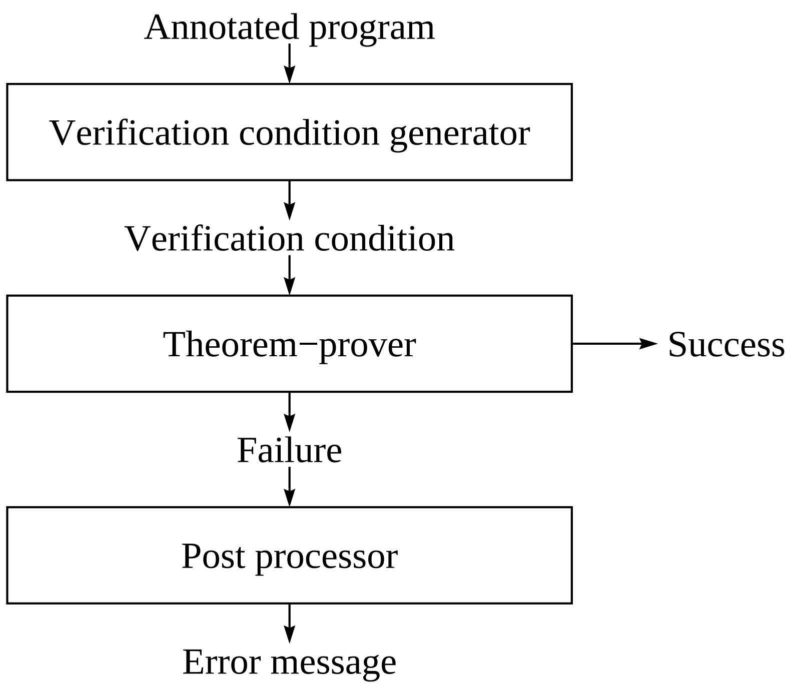

Figure 10: Block diagram of the ESC tool.

array bounds error
`NIL` dereference
subrange error
narrow fault (type-cast error)
functional procedure fails to return a value
exception not in `RAISES` clause
`CASE` value handled by no arm
`TYPECASE` value handled by no arm
divide or `MOD` by zero

accessing protected variable
    without a lock
acquiring locks out of order
precondition violation
postcondition violation
program invariant violation
`MODIFIES` clause violation

Figure 11: Errors reported by ESC.

The problem is not that the libraries use unsafe code or low-level tricks; the problem is that they use patterns of data abstraction that are richer than those treated in the literature. It turned out to be a major problem to design a checker that allows modular checking and supports the patterns of data abstraction that are used in modern object-oriented designs. In this overview paper, we have space only to sketch the kinds of difficulties and hint at our solutions; for a fuller treatment, we refer the reader to our companion paper *Abstraction and specification revisited* [30].

A basic dilemma. In writing specifications for a multi-module program, we face a fundamental dilemma. Specifications require that procedure declarations include a list of what variables can be modified by a call to the procedure. But in a properly modularized program, the variables modified by a procedure are usually private to the implementation, and are not in scope at the point of declaration of the procedure.

Solution: abstraction. The solution to the dilemma is data abstraction. The specification describes the side effects of the procedure in terms that are of use to its clients, that is, in terms of abstract variables. The concrete variables that are used to represent the abstract variables can be confined to the private scope of the implementation. Generally the representation function is private to the same implementation scope.

Using data abstraction to solve the basic dilemma has several important consequences.

Downward closure. The first consequence of using data abstraction is that abstract variables can appear in `MODIFIES` lists; and that the meaning of an abstract variable in a `MODIFIES` list is that the license to modify the abstract variable implies the license to modify the concrete variables that represent it. For example, consider the implementation of `Rd.GetChar`:

```
PROCEDURE GetChar(rd: Rd.T): CHAR
    RAISES {EndOfFile, Failure, Thread.Alerted} =
  VAR res: CHAR; BEGIN
    LOCK rd DO
      IF rd.cur = rd.hi THEN
        IF rd.seek(rd.cur) = RdrRep.SeekResult.Eof
          THEN RAISE EndOfFile
        END
      END;
```

```
      res := rd.buff[rd.st + rd.cur - rd.lo];
      INC(rd.cur);
      RETURN res
    END
  END GetChar;
```

Obviously, this modifies `rd.cur`, but `rd.cur` does not occur in the `MODIFIES` list of `GetChar` (see the listing of the `Rd` interface in Figure 6). Why doesn't the checker complain? Because `rd.cur` is part of the representation of the abstract variable `state[rd]`, which `GetChar` is allowed to modify.

Thus the `MODIFIES` list

```
MODIFIES state[rd]
```

of `GetChar` is "downward closed" to

```
MODIFIES state[rd], cur[rd], ...
```

where the ellipses stand for the other concrete variables representing `state[rd]`. The checker does not complain about the updates to `rd.cur` in `GetChar` because the `MODIFIES` list is closed before the verification condition is generated. (The reader may wonder how the checker knows which variables are part of the representation of `state[rd]`; this is explained below.)

Protecting related abstractions. The second consequence of using data abstraction is that postconditions must be strengthened to "protect related abstractions". To see this, consider the question: what prevents `GetChar` from destroying the validity of its reader? Since `valid[rd]` does not occur in the `MODIFIES` list of `GetChar`, callers (and our checker when reasoning about a caller) will trust that `GetChar` preserves validity. But `GetChar` is allowed to modify the state of the reader, and thereby, through downward closure, is also allowed to modify the representation of the state, which includes concrete variables that are part of the representation of `valid`. Thus the license to modify the state threatens to modify validity. Evidently, the omission of `valid` from the `MODIFIES` list must impose on the implementor the obligation of proving that the changes to the state are such that `valid[rd]` is unchanged. Indeed, our checker strengthens the postcondition of `GetChar` with the conjunct `valid'[rd] = valid[rd]`, so that if validity is destroyed, the checker will complain.

Here is another (more contrived) example. If two abstract variables `a` and `b` both are represented in terms of two concrete variables `c` and `d`, then in a scope where `a`, `b`, `c`, and `d` are all visible,

```
MODIFIES a
```

is desugared into

```
MODIFIES a, c, d
ENSURES b' = b
```

That is, `c` and `d` are included because of downward closure, and the related abstraction `b` is protected from change by adding it to the postcondition.

In a scope where `a` and `b` are visible but `c` and `d` are not visible, the original `MODIFIES` list

```
MODIFIES a
```

remains unchanged after downward closure and protection of related abstractions.

Soundness lost. Both the downward closure and the protection of related abstractions are reflected in the checker's semantics of data abstraction as rules for desugaring specifications. The desugaring depends on which variables represent which other variables. The knowledge of this representation information is different in different scopes, and therefore the desugaring is different in different scopes. This is frightening, since it raises the possibility that after desugaring, the specification used in reasoning about the calls to a procedure can be different from the specification used in checking its implementation. Thus, it is no longer clear that checking the modules individually ensures that the composite program is free of errors.

In fact, it is easy to see that without imposing additional conditions, we have no hope of sound modular checking. For example, suppose that `c` is part of the representation of an abstract variable `a`, that both `a` and `c` are visible in some scope, and that no declaration in the scope gives any clue of the connection between the variables. Then the checker has no chance of reasoning correctly about the program, since modifications of `c` may affect `a`, and modifications of `a` (via procedure calls) may affect `c`, and neither side effect can be expected by the checker.

The rep-visibility requirement. A simple way to restore soundness is to impose the following requirement, which we might call the *rep-visibility requirement*:

> Whenever `c` is part of the representation of an abstract variable `a`, and both `a` and `c` are visible in some scope, then the representation of `a` must also be visible in that scope.

The classic treatment of data abstraction by C.A.R. Hoare implicitly imposes this requirement, since it requires that the representation and all its concrete variables be declared together [18].

Unfortunately, we found many examples in the Modula-3 libraries where the rep-visibility requirement is too strong. As one simple example, in the scope of `RdrRep` (and of `WhiteSpace.Skip`) the concrete variables that represent `Rd.state` are visible, but the representation itself is not. Furthermore, it would be very awkward to place a representation declaration for `state[rd]` in this scope, for two reasons. First, since we are doing ESC verification only, we don't want to get bogged down in the complexities of the state. We would prefer never to declare the representation of `state[rd]` at all. Second, even if we were doing full-scale verification, the representation of the state of a reader is subclass-specific, so not all variables that are part of the representation are in scope, but the scope `RdrRep` is class-independent. (The infeasibility of the rep-visibility requirement is also manifest from the more detailed example in our companion paper *Abstraction and specification revisited* [30].)

Explicit dependencies. We therefore introduce a new specification language construct, `DEPENDS`, which is a way of specifying that one variable is part of the representation of another, without giving the actual representation. In the case of readers, we write

```
DEPENDS Rd.state[rd: Rd.T] ON
  rd.st, rd.lo, rd.cur, rd.hi, rd.buff, rd.buff^
```

in interface `RdrRep`. Thus we commit the design decision that these concrete variables are part of the representation of `state[rd]`, while deferring the decision of what the representation is.

The depends-visibility requirement. Armed with `DEPENDS`, we weaken the rep-visibility requirement to the *depends-visibility requirement*:

> Whenever `c` is part of the representation of an abstract variable `a`, and both `a` and `c` are visible in some scope, then the dependency of `a` on `c` must also be visible in that scope.

Sketch of the rest of the story. We hope this section has given the reader a flavor of the issues we have wrestled with in trying to produce a sound modular checker. Summarizing briefly, practical systems programs use information hiding in ways that make it problematical to generate verification conditions in a sound and modular fashion. A key ingredient of our solution to the problem is the explicit declaration of dependencies. In the full story, there are several kinds of dependency declarations, and different requirements are imposed on different kinds of dependencies. For the details, we refer the reader to our companion paper *Abstraction and specification revisited* [30].

One problem in this area that stumped us is a form of rep exposure that we call abstract aliasing. We have been unable to design a statically enforceable programming discipline for avoiding this problem. Our best efforts are described in our companion paper *Wrestling with rep exposure* [4]. In the meantime, we take the view that it is the programmer's responsibility to avoid abstract aliasing.

6 Verification condition generation

More than half of the code in our checker is devoted to the task of translating the annotated Modula-3 into the verification condition (VC) to be presented to the theorem-prover. This task is governed by the classic laws of Hoare logic [17], but the demands of our checker require some novel approaches. As usually expounded, Hoare logic provides a framework for checking a hand-constructed proof of some program property. Such a proof typically requires invariants at many control points, but we want VC generation to be totally automatic. Therefore, we translate the annotated Modula-3 into a version of Dijkstra's guarded commands [7, 39], and then use their weakest-precondition equations to generate the VC. This approach provides a better foundation for an automatic tool, since weakest preconditions have more of a calculational flavor than Hoare logic.

Our theorem-prover has a novel feature that allows our checker to report specific error messages: any subformula of the theorem-prover input can be *labeled*. If the prover finds a counterexample, it emits the set of labels of relevant subformulas that are false in the counterexample. The implementation of this feature will be described in our companion paper *An automatic theorem-prover for program checking* [5]. The VC generator uses this feature by labeling the proof obligations in the VC corresponding to each possible error. The name of the label encodes the source position and error type. This makes it straightforward to translate failed proofs into specific error messages.

The translation of Modula-3 control structures into guarded commands is quite straightforward, but the effect of the type system on the translation is more interesting. Like pre- and postconditions, type declarations contain declarative information about the program. Our checker uses this information. Thus, the effective precondition of a procedure is its `REQUIRES` clause conjoined with the precondition implicit in the procedure declaration. For example, the effective precondition assumed on entry to the body of

```
PROCEDURE P(x: CARDINAL);
```

must imply $0 \leq \texttt{x}$, since the type system guarantees this. Slightly more subtly, consider the procedure

```
PROCEDURE Q(t: T);
```

where the type `T` contains a `CARDINAL`:

```
TYPE T = OBJECT val: CARDINAL; link: T END;
```

The effective precondition assumed on entry to the body of `Q` must imply $0 \leq \texttt{t.val}$, $0 \leq \texttt{t.link.val}$, $0 \leq \texttt{t.link.link.val}$, *etc.* (as far as these are defined). Since ESC's theorem-prover is for the untyped predicate calculus, we took the expedient, if inefficient, approach of encoding Modula-3's type system in untyped first-order logic. For example, the extra precondition assumed for the body of `Q(t)` is `IsT(t)`, where `IsT` is axiomatized in the first-order language of the theorem-prover. We spare the reader the full complexity of the axiomatization, but here's a simplified version:

$$(\forall t :: \texttt{IsT}(t) \Rightarrow t = \texttt{NIL} \vee (\texttt{IsCARDINAL}(t.\texttt{val}) \wedge \texttt{IsT}(t.\texttt{link})))$$
$$(\forall x :: \texttt{IsCARDINAL}(x) \Rightarrow 0 \leq x)$$

Similarly, the extra precondition assumed for the body of `P(x)` above is `IsCARDINAL(x)`. The language-enforced condition that variables have values of their declared types is important not only for assumed preconditions but also for other parts of the program, including assumed postconditions, `TYPECASE` and `NARROW`, and loops. For a full account of this subject for a language smaller than Modula-3, see Leino's *Ecstatic: An object-oriented language with an axiomatic semantics* [29].

Semantic correctness is only half the battle: Logically equivalent forms of the verification condition can cause the heuristic search done by the theorem-prover to perform different patterns of case analyses, which can have dramatic performance consequences. Although there seems to be no way to guarantee that the theorem-prover will not choose a disastrously slow pattern of case analyses, luckily we have found a few straightforward heuristics that seem to prevent this in practice. This often neglected issue is in fact a crucial aspect of VC generation. For the details of some of the heuristics, see our companion paper *An automatic theorem-prover for program checking* [5].

7 The theorem-prover

We did our first experiments using the Larch prover [12]. Since this prover requires human guidance to find a proof, the programmer had to guide the prover through a proof, and an error would be revealed by the failure of the process. Damien Doligez found a locking error in auto-flush writers using this prover, but for mere mortals it is too laborious to be practical.

Therefore, we wrote our own theorem-prover, which is designed to be totally automatic, and which is powered by automatic decision procedures for the functions and predicates that are important in programming (in particular, equality and arithmetic). The overall structure of the theorem-prover follows the design in the Ph.D. thesis of one of the authors (Nelson) [37]. Another important requirement that shaped the design of the theorem-prover is that failed proofs lead to comprehensible error messages. In particular, the prover is refutation-based: to prove a verification condition VC, the prover attempts to satisfy $\neg VC$. In full-scale program verification, the failure to satisfy $\neg VC$ implies that VC is valid and the program meets its specification; in an ESC verification, the satisfaction of $\neg VC$ gives an error context for the original program. The systematic exhaustive search techniques are identical, although the purpose is different.

By far the most time-consuming part of running the checker is the backtracking search in the theorem-prover. We find that the checker is usually between five and fifty times slower than the compiler. This is too slow to use routinely with every compilation, but it is fast enough to be useful. (Hardware designers have learned that it is worthwhile to run simulations and design checkers, even if they are so slow they have to be run overnight. We think that ESC can offer the same benefits, but for software instead of hardware. As another point of comparison, many good programming teams make use of code reviews, in which a committee studies a program line by line. Overnight ESC runs are cheap by comparison.) We have sometimes found it irritating that the time taken by the prover is unpredictable.

The input to our prover is a formula of the untyped first-order predicate calculus, with equality and function symbols, quantifiers, arithmetic, and McCarthy's *store* and *select* functions [35]. Quantifiers are handled by a heuristic matcher that exploits equalities and can be guided by user-supplied "trigger" terms. The details of the theorem-prover design will be described in a companion paper [5].

8 Soundness considered harmful

As we have mentioned several times, failed proofs are turned into error messages. But what if the proof of the verification condition succeeds? In this case the tool has nothing useful to report, and its output is "Sorry, can't find any more errors". We have found in demonstrating the tool that people laugh at this message, but we want to be absolutely clear that in this case we do not claim to have rigorously proved the absence of all errors, since our tool's verification engine has some sources of unsoundness that are included by design. Two of these are:

- There are some kinds of errors that we do not try to find: it is the programmer's responsibility to avoid them. These include arithmetic overflow and abstract aliasing (see [30, 4]).

- Although it is possible to use the checker with loop invariants, we generally use neither programmer-supplied nor inferred loop invariants (see Section 9), and in this mode we generate a precondition for the loop that is weaker than the true infinite limiting precondition. That is, the verification condition generation is unsound.

We don't view these unsoundnesses as problems, since there are plenty of errors that the checker can find. We think it important to use engineering judgment to decide which kinds of errors are worth checking for, based on the different costs and benefits of each kind of check. To categorically require that the tool be sound means that it must catch all kinds of errors, which avoids a difficult cost-benefit tradeoff by retreating to a mathematical idealization. This is nothing more than a breach of engineering responsibility. (Interestingly enough, our theorem-prover is sound, as far as we know; it has been in VC generation that we have found it valuable to leave some kinds of errors to the programmer.)

9 Loops

An important point to notice about the two extended examples we showed in previous sections is that the programmer is not required to supply loop invariants. ESC implements three techniques that greatly reduce, or completely eliminate, the need for programmer-supplied loop invariants.

The first technique infers a loop invariant by static analysis of the loop body, using a version of the abstract interpretation method of Cousot and Cousot [3]. We thank François Bourdoncle for help with the design and implementation of this part of ESC.

The second technique ("loop modification inference", or LMI) guesses a loop invariant by strengthening the part of the enclosing procedure's postcondition that comes from the `MODIFIES` clause.

The third technique ("even weaker precondition", or ewp) eliminates the need for loop invariants by considering only those computations in which the loop is executed a bounded number of times (in particular, 0 or 1 times). This apparently crude technique is remarkably effective in practice. Of course, it is not sound. In fact, it is a good example of the wonderful liberation we get by dropping the shackles of soundness.

The user can activate these techniques using a command-line switch. (The abstract interpretation switch is no longer supported.) The accounts of the two examples in this paper assumed the ewp technique. The checking of the readers/writers package described in Section 10 was performed using each of the three techniques, but when using LMI, the checking resulted in three spurious warnings.

10 Experience

In this section, we report on our experience using the checker. Looking over our experiments, we find that we used the checker to perform three different levels of verification: ESC verifications (which check the absence of the errors listed in Figure 11), locking-level verifications (which check the absence of deadlocks and race conditions only), and functional-correctness verifications (which are like ESC verifications but also check functional correctness). These levels of verification are denoted by ESC, LL, and F in Figure 12. Each of these levels of verification also checks the program to be consistent with its annotations. (We say "check" instead of "prove" since, as explained in Section 8, the VC generator leaves certain errors to the programmer to avoid.) Averaged over the 20000 source lines, the annotation overhead was a 13.6% increase in the number of lines. We find this to be a reasonable price to pay for the additional checking.

In the standard Modula-3 I/O library, we have done an ESC verification of the class-independent readers code as well as all the standard reader subclasses. We have done a locking-level verification of most of the class-independent writers code, and an ESC verification of several writer subclasses. The annotated code is available from the ESC home page [9]. This exercise did not uncover any errors in the I/O library, but it taught us many things about data abstraction that are described in Section 5 and our companion papers [30, 4].

We have done two experiments in which we turned the checker on its own source code. One of the more complicated modules in our checker is `Simplex`, the part of the automatic theorem-prover that reasons about linear inequalities. A straightforward but very detailed module in our checker is `ParseSpec`, the recursive-descent parser for the annotation language. We have done ESC verifications of both `Simplex` and `ParseSpec`. The specifications we wrote for `ParseSpec` ensured not only the absence of errors but also the proper shape of the parse trees constructed. This is more than ESC verification but less than functional correctness, so we list it as ESC+ in Figure 12. We did not find errors in the well-exercised `Simplex` module, but we found several in `ParseSpec`.

Most of our verifications have been of mature code. To test ESC on undebugged code, two of the authors (Leino and Nelson) teamed up with Rajit Manohar to write a new writer class, `PrettyWr`, a writer that formats its output stream by judicious insertion of line breaks and indentation, and forwards the result to another writer. This is a short but tricky program; it took the three of us two days to design and code. ESC found four errors (a violation of the validity invariant, a failure to declare an exception that needed to be propagated, a self-deadlock, and the access to a shared field without holding the protecting lock). After fixing the errors and proceeding to test the program, we found two more errors: an infinite recursion (which is an error within the range of the ESC techniques, but not handled by our current checker), and a failure to format correctly caused by missing an assignment to a boolean (which was beyond the scope of our experiment, since we didn't try to annotate for functional correctness).

In another experiment to run ESC on fresh code, Leino teamed up with Cormac Flanagan to write a program that generates and prints random mazes. ESC found no errors in the first version, which also performed without error when it was tested. Shortly thereafter, Leino introduced an optimization, and with it an initialization error, which ESC reported.

We conclude from these experiments that in fresh code, ESC can catch a substantial fraction of the errors that are ordinarily detected by debugging.

The Modula-3 windowing library, Trestle, is highly concurrent and requires careful synchronization. Allan Heydon has done a locking-level verification of the Trestle Tutorial [34], in which he discovered a latent bug. This latent bug would have been difficult to find by testing, since it would strike only in Trestle implementations in which selection values were communicated lazily between address spaces. While the Trestle specification was designed to allow lazy communication of selection values, all Trestle implementations to date communicate selection values eagerly.

One of the authors (Leino) had been working on supporting free-hand annotations in an on-line document viewing system, and in the course of this work he extended the Trestle library with a module (called `CubicPath`) that converts polygonal paths into smooth cubic splines. He applied ESC to this module, but found no errors.

In addition to the I/O library, we have done ESC verifications of other parts of the standard Modula-3 libraries [20]. Most of this code is mature and well exercised and we found only one error. In a recent addition to the library, the generic `Sequence` module, we found the following glaring error: instead of `i := i MOD n`, the code read

Package	files	l.o.c.	l.o.a.	l.o.a./l.o.c.	checking level
Readers/writers	14	2495	470	0.188	ESC/LL
OS	3	674	75	0.111	
`Simplex`	3	2157	184	0.085	ESC
`ParseSpec`	2	2559	793	0.309	ESC+
`List`	1	110	23	0.209	
`PrettyWr`	3	411	115	0.279	ESC
`Maze`	5	403	93	0.230	ESC
`Rand`	1	24	12	0.500	
Trestle Tutorial	28	2201	169	0.076	LL
Trestle	27	6736	346	0.051	
`CubicPath`	2	633	110	0.173	ESC
`Path`	1	179	22	0.122	
`Sequence`	3	587	185	0.315	F
`Text`	3	381	103	0.270	F
`Fmt`	1	296	17	0.057	
TOTAL	97	19846	2717	0.136	

Figure 12: Packages checked by ESC, showing for each package the number of files, lines of code (l.o.c.), lines of annotation (l.o.a.), proportion of annotation lines to code lines, and the level of verification performed. Indented rows show interfaces outside the package that were annotated in order to check the package.

```
IF n <= i THEN i := i - n END
```

where there was a possibility of `i` being as large as `2*n`. This error had not been exposed by testing. Later, we extended the annotations in `Sequence` to perform a functional-correctness verification, but this did not reveal any more errors. We also did a functional-correctness verification of the `Text` module. Figure 12 presents some statistics about the verifications mentioned in this section. Not surprisingly, it shows that the ratio of annotation lines to code lines is noticeably higher for functional-correctness verification.

In several ESC verifications described above, it was necessary to annotate some of the interfaces used. For example, in checking the `CubicPath` module, it was necessary to specify the `Path` interface. In checking the readers/writers package and the Trestle Tutorial, it was necessary to specify a number of interfaces, but we don't list their names individually. Figure 12 includes statistics on the annotations of these imported interfaces.

In a rather different sort of experiment, one of the authors (Detlefs) and George

Necula have used ESC to reason about dynamic reachability in linked structures. Their hope was to replace garbage collection with explicit deallocation statements, and to check by ESC that the explicit deallocations are all safe, thus combining the safety of garbage collection with the efficiency of explicitly managed storage. They succeeded with several modest-sized programs involving linear lists, but the reasoning required about reachability was very expensive. Perhaps this approach can become practical, but not without some more work.

11 The tarpit of creeping aspirations

In this section, we record a cautionary note suggested by our experience.

Two important features of our annotation language are `NOWARN` and `ASSUME`. Adding the annotation `NOWARN` *L* to a line suppresses any errors of type *L* associated with that line. The annotation `ASSUME` *P* lets the programmer take responsibility that *P* holds at the point of the annotation. These are useful when ESC emits a spurious warning. Using `NOWARN` and `ASSUME` can feel like defeat, and there is a temptation to work harder, adding specifications to convince the theorem-prover that the warning is indeed spurious. The problem is that you may spend hours or days persuading the checker of the validity of a piece of code that was never really in any doubt to begin with. We call this the problem of creeping aspirations. Well-trained scientists are particularly susceptible to the problem.

As an example of the danger of creeping aspirations, the ESC verification of the maze-generating program depends on the fact that it is an invariant of the union-find data structure that the number of equivalence classes is positive. No doubt ESC could be dragged through a proof of this fact, but the reader who looks at this example on the Web [9] will find the line

```
<* SPEC ASSUME 0 < t.numClasses *>
```

which illustrates the more pragmatic approach that we advocate.

Of course the `NOWARN` and `ASSUME` features are unsound, but that doesn't bother us. On the contrary, we are convinced that these features are essential to make a checker useful.

12 Related work

We don't know of any system as semantically thorough and automatic as ours, but many systems have solved pieces of the puzzle.

Full-scale, but not automatic, program verifiers include the early systems of James King [26, 25] and Peter Deutsch [6], the Stanford Pascal Verifier [32], the

Gypsy Verification Environment [14] for developing programs by iterative refinement of specifications, the Penelope [15] verification system for a subset of Ada, and the coalgebra-based Java verifier LOOP [22].

Automatic static checkers that are based on conventional compiler flow analysis rather than program verification are not as semantically thorough as our checker, because, for example, they ignore the semantics of conditional statements. Checkers of this kind include LCLint [8], which checks C programs annotated with a version of Larch/C [16]; Daniel Jackson's Aspect, a novel system that warns of CLU procedures that fail to update the (representation of the abstract) variables they are specified to modify [21]; Nicholas Sterling's static race analysis tool Warlock [45]; and Joseph Korty's Sema, a Lint-like tool for detecting deadlocks in a semaphore-based Unix kernel [27].

Our approach is perhaps closest to that of Steve German's Runcheck verifier [13]. German seems to have been the first to have given up full-scale correctness verification in order to achieve a more automatic tool. While German's work was mostly for integer and integer array programs, we have exercised our tool on realistic concurrent object-oriented multi-module programs. Another early tool influenced by German's work is the Ford Pascal-F Verifier [36].

The earliest and most forthright exposition known to us of the goal we are pursuing is the conclusion of Dick Sites's thesis [44].

We agree with the widespread view that if a tool is to be popular, it must somehow spare its users the burden of annotating every loop with an invariant. This view has stimulated a large body of work on automatic inference of program invariants, including the pioneering work of Wegbreit [48] and the systematic theory of abstract interpretation introduced by Cousot and Cousot [3]. Three interesting program checkers based on abstract interpretation are François Bourdoncle's Pascal checker Syntox [1], Alain Deutsch's Ada checker [47], and Cormac Flanagan's Scheme checker MrSpidey [11]. As explained in Section 9, we implemented and experimented with a version of abstract interpretation for finding loop invariants, but our experience led us to the conclusion that it is better to do without loop invariants altogether, rather than to synthesize them.

The goal of improving programming productivity is served also by better tools supporting traditional testing and runtime checking. It is plausible that a systematic, disciplined use of a dynamic checker like Eraser [43] would do as well as ESC at detecting race conditions and deadlocks. Also in the area of detecting race conditions is the Cilk tool Nondeterminator-2 [2], which is based on an intriguing combination of static and dynamic checking, but which works only for fork/join synchronization, not for locks.

13 Conclusion

The formal undecidability of most questions of static analysis have led most programmers to conclude that in reviewing code for errors, only a human programmer can take accurate account of the semantics of tests and updates to data structures: that type-checking and data flow analysis are the upper limit of semantic analysis compatible with automation. Our most general conclusion is that this widely-held pessimistic view is mistaken: by adopting the technology of program verification while leaving behind its most quixotic goals, it is possible to build a checker that achieves an unprecedented combination of automatic operation with semantically accurate analysis.

At a more specific level, we found positive answers to several of the specific questions that we were investigating:

- We are able to generate verification conditions for realistic systems programs, but doing so required us to introduce two new techniques: the locking-level annotations for handling concurrency and the `DEPENDS` annotation for reconciling data abstraction with information hiding.

- We are able to turn failed proofs into specific error messages. The major work required for this was in the theorem-prover.

- The annotation burden is minimal. Most of the annotations are straightforward inequalities or other conditions that an experienced programmer will record anyway, in English comments if not in the checker's annotation language. Programmer-supplied loop invariants are not required for useful checking.

- The theorem-proving can be carried out automatically, with no user guidance. Although ESC is not as easy to use as a type-checker, it feels more like a type-checker than a program verifier.

On the other side of the ledger, the theorem-prover is too slow to use the checker routinely with every build. Also, the unpredictability of the performance is annoying.

From a mathematical or methodological point of view, the most interesting outcome of our project is the theory of dependencies sketched in Section 5, because this theory seems to point the way to sound modular reasoning about object-oriented systems. Thus this theory increases our understanding of how to structure large programs, but it is more subtle and complicated than we would wish.

Today's best engineering organizations produce software by starting with design methods that (in principle) yield programs that are correct by construction, and then following up with a disciplined testing effort. We are optimistic that this engineering process could be improved by carefully including some amount of extended static checking.

Acknowledgments

Mark Manasse and Greg Nelson designed the details of the locking-level notation while programming the Trestle window system [33]. Although there was no checker available at the time, the hand-written and hand-checked specifications were very helpful in avoiding synchronization errors. The success of this notation helped to spur the development of the checker.

Damien Doligez performed the first successful experiment in which verification conditions were generated for locking-level specifications, and proved using the Larch prover. Steve Glassman first used ESC to perform a locking-level verification of the readers and writers package. Allan Heydon performed the locking-level verification of the Trestle Tutorial.

Shun-Tak Leung and Allan Heydon provided comments on drafts of this paper.

The whole project benefited from helpful suggestions from Jim Horning and Raymie Stata.

References

[1] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation (PLDI)*, pages 46–55. ACM SIGPLAN Notices 28(6), June 1993.

[2] Guang-Ien Cheng, Mingdong Feng, Charles E. Leiserson, Keith H. Randall, and Andrew F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 298–309, June 1998.

[3] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[4] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, July

1998. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[5] David L. Detlefs, Greg Nelson, and James B. Saxe. An automatic theorem-prover for program checking, 1999. To appear.

[6] L. Peter Deutsch. *An Interactive Program Verifier*. PhD thesis, University of California, Berkeley, Berkeley, CA 94720, 1973.

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

[8] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: A tool for using specifications to check code. In *SIGSOFT '94: Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Enginnering*, pages 87–96. Software Engineering Notes 19(5), December 1994.

[9] Extended Static Checking home page, Compaq Systems Research Center. On the Web at `www.research.digital.com/SRC/esc/Esc.html`.

[10] Christopher Farrell. Industry outlook. *Business Week*, pages 54+ (see especially page 76), 9 January 1995.

[11] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI)*, pages 23–32. ACM SIGPLAN Notices 31(5), May 1996.

[12] Stephen J. Garland and John V. Guttag. *LP, the Larch Prover: Version 3.1*, January 1995. Available as `www.sds.lcs.mit.edu/larch/LP/overview.html`.

[13] Steven M. German. Automating proofs of the absence of common runtime errors. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 105–118, 1978.

[14] Donald I. Good. Mechanical Proofs about Computer Programs. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*, pages 55–75. International Series in Computer Science. Prentice Hall, 1985.

[15] D. Guaspari, C. Marceau, and W. Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, September 1990.

[16] John V. Guttag and Jim Horning. Introduction to LCL: A Larch/C interface language. Research Report 74, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, July 1991. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969. Reprinted as Chapter 4 of [19].

[18] C. A. R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972. Reprinted as Chapter 8 of [19].

[19] C. A. R. Hoare and C. B. Jones, editors. *Essays in Computing Science*. International Series in Computer Science. Prentice Hall, 1989.

[20] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Research Report 113, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1993. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[21] Daniel Jackson. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology*, 4(2):109–145, April 1995.

[22] Bart Jacobs, Joachim van den Berg, Marieke Huisman, and Martijn van Berkum. Reasoning about Java classes (preliminary report). In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*, pages 329–340. ACM SIGPLAN Notices 33(10), October 1998.

[23] S. C. Johnson. Lint, a C program checker. Computer Science Technical Report 65, Bell Laboratories, Murray Hill, NJ 07974, 1978.

[24] Mick Jordan. An extensible programming environment for Modula-3. In *SIGSOFT '90: Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 66–76. Software Engineering Notes 15(6), December 1990.

[25] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[26] James Cornelius King. *A Program Verifier*. PhD thesis, Carnegie-Mellon University, Pittsburg, PA 15213, September 1969.

[27] Joseph A. Korty. Sema: A Lint-like tool for analyzing semaphore usage in a multithreaded UNIX kernel. In *Proceedings of the Winter 1989 USENIX Conference*, pages 113–123. USENIX Association, January–February 1989.

[28] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, Pasadena, CA 91125, January 1995. Technical Report Caltech-CS-TR-95-03.

[29] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `www.cs.williams.edu/~kim/FOOL/FOOL4.html`.

[30] K. Rustan M. Leino and Greg Nelson. Abstraction and specification revisited. Internal manuscript KRML 71, Digital Equipment Corporation Systems Research Center, 1998. To appear as a SRC Research Report 160. See also the first author's PhD thesis [28].

[31] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[32] D. C. Luckham, S. M. German, F. W. von Henke, R. A. Karp, P. W. Milne, D. C. Oppen, W. Polak, and W. L. Scherlis. Stanford Pascal Verifier user manual. Technical Report STAN-CS-79-731, Stanford University, 1979.

[33] Mark S. Manasse and Greg Nelson. Trestle reference manual. Research Report 68, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1991. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[34] Mark S. Manasse and Greg Nelson. Trestle tutorial. Research Report 69, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, May 1992. Available from `www.research.digital.com/SRC/publications/src-rr.html`.

[35] John McCarthy and James Painter. Correctness of a compiler for arithmetic expressions. In J.-T. Schwartz, editor, *Proceedings of Symposia in Applied Mathematics*. American Mathematical Society, 1967.

[36] John Nagle and Scott Johnson. Practical program verification: Automatic program proving for real-time embedded systems. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*, pages 48–58, January 1983.

[37] Charles Gregory Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, Stanford, CA 94305, 1980. See also [38, 41].

[38] Greg Nelson. Combining satisfiability procedures by equality-sharing. In W. W. Bledsoe and D. W. Loveland, editors, *Automated Theorem Proving: After 25 Years*, volume 29 of *Contemporary Mathematics*, pages 201–211. American Mathematical Society, Providence, RI, 1984.

[39] Greg Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.

[40] Greg Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice Hall, Englewood Cliffs, NJ, 1991.

[41] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[42] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972. Reprinted as `www.acm.org/classics/may96/`.

[43] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997. Also appears in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 27–37, Operating System Review 31(5), 1997.

[44] Richard L. Sites. *Proving that Computer Programs Terminate Cleanly*. PhD thesis, Stanford University, Stanford, CA 94305, May 1974. Technical Report STAN-CS-74-418.

[45] Nicholas Sterling. Warlock — a static data race analysis tool. In *Proceedings of the Winter 1993 USENIX Conference*, pages 97–106. USENIX Association, January 1993.

[46] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part II: Input/output and filing system. *The Computer Journal*, 15(3):195–203, 1972.

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

[48] Ben Wegbreit. Property extraction in well-founded property sets. *IEEE Transactions on Software Engineering*, 1(3):270–285, September 1975.