TOOL DEMONSTRATION

An Extended Static Checker for Modula-3

K. Rustan M. Leino and Greg Nelson
Systems Research Center, Digital Equipment Corporation
{rustan,gnelson}@pa.dec.com

This paper briefly introduces the Extended Static Checker for Modula-3 (called ESC), a programming tool that will catch 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 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 that it is free of certain specific types of errors; secondly because we are interested in failed proofs only, not in successful ones. Failed proofs are more useful than successful ones, since they warn the programmer of possible errors. Also, failed proofs are found more quickly than successful ones.

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 thirty years ago, 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 years have made too many simplifying assumptions. They may handle only sequential control structures; they may handle no data structures except integers and integer arrays; they may require that the entire program consist of a single module; they may require the user to guide the theorem-prover or to provide complicated loop invariants. These assumptions facilitate the implementation of prototype checkers, but they also destroy the engineering utility of the checker. We argue that these simplifying assumptions can be dropped; that the time has come for extended static checking to be deployed instead of studied.

Our checker handles multi-threaded multi-module object-oriented programs. The theorem-proving is completely automatic. Our checker reports errors by line number and error type. Our checker works on Modula-3 programs, but the techniques would work for any language in which address arithmetic is restricted, including Oberon, Ada, Java, and FORTRAN.

To deal with multi-threaded programs, we introduce a technique that we call *locking-level verification*. Basically this means that the programmer declares which locks (semaphores) protect which shared variables, and which locks can or must be held at entry to various procedures. The programmer also declares a partial order in which locks are allowed to be acquired. The system checks that shared variables are never accessed without holding the appropriate locks, and that locks are never acquired out of

order. This technique doesn't prove correctness—more expensive methods like monitor invariants would be required for that—but it does catch many common errors.

Our checker is modular: 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 strategy 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 range errors.

Using the checker in its most picky mode, where it checks for all runtime errors and also for race conditions and deadlocks, we have checked essentially all of the standard Modula-3 input/output library (which is based on readers and writers, which are object-oriented buffered streams), and also the standard generic sequence implementation, and also several modules from the checker itself. Using the checker in a more forgiving mode, in which it checks only for deadlocks and race conditions, we have checked the Trestle Tutorial, a suite of about a dozen programs that exercise the Trestle window system. The checker discovered a locking error in the tutorial.

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 effective solutions to the biggest outstanding problems that have confined extended static checking to the realm of the Ph.D. thesis.

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 understandable, and the theorem-proving is carried out behind the scenes automatically.

Example We have applied ESC to reasonably large libraries, but here we have space only for a very simple example: we describe how ESC might find errors in a small procedure on the scale of an exercise in an introductory programming course.

The exercise is to program a procedure that accepts an array of integers as an argument and returns a `TEXT` (Modula-3's predeclared string type) that contains the concatenation of the decimal representations of the elements of the array. To avoid the quadratic cost of repeated text concatenations, the procedure allocates a *text writer*, which is a form of buffered output stream whose output can be retrieved as a `TEXT`, writes the elements of the array to the text writer in order, and finally retrieves and returns a text containing everything that was written. Before presenting the procedure, we present the annotated text writer interface:

```
INTERFACE TextWr;
TYPE T <: ROOT;
<* SPEC VAR valid : MAP T TO BOOLEAN *>
<* SPEC VAR state : MAP T TO ANY *>
PROCEDURE Init(t : T);    (* Initialize the text writer t. *)
  <* SPEC Init(t) MODIFIES valid[t], state[t] ENSURES valid'[t] *>
PROCEDURE PutInt(t : T; i : INTEGER);    (* Write the ASCII version of i to t. *)
  <* SPEC PutInt(t, i) MODIFIES state[t] REQUIRES valid[t] *>
```

```
PROCEDURE GetText(t : T) : TEXT; (* Return the text that has been written to t. *)
  <* SPEC GetText(t) MODIFIES state[t] REQUIRES valid[t] *>
PROCEDURE Close(t : T);    (* Destroy t, reclaiming its internal buffers. *)
  <* SPEC Close(t) MODIFIES valid[t], state[t] *>
ENDTextWr.
```

This interface declares an *opaque type* `TextWr.T`. The client of the interface knows the name of the type and the fact that it is a subtype of `ROOT` (that is, that it is an object type) and knows the signatures of the procedures `Init`, `PutInt`, `GetText`, and `Close`, but the client knows nothing else about the type. The representation of the type is declared in another, more private module, which will not be shown here.

The annotation language of ESC is basically very conventional: each procedure is annotated with a pre- and post-condition (introduced by the keywords `REQUIRES` and `ENSURES`, respectively) and a modifies list, which is the list of variables that the procedure is allowed to modify. For a variable `x` in the modifies list, the postcondition uses the notation x' to denote the post-value of `x` and the unadorned identifier `x` to denote its pre-value.

The annotation language also allows the declaration of *abstract variables*, also called *specification variables*. Two abstract variables are used in the annotations of the text writer interface, as in many others: `valid` and `state`. The concrete representations of these variables are revealed in the implementation of text writers and are not visible in the interface, which is intended for clients of the abstraction. At an abstract level, `valid[t]` holds iff the text writer `t` has been properly initialized and `state[t]` represents all the rest of the client-visible state of the text writer (that is, its contents). If we were doing full-scale program verification, the interface would specify a great deal about the state, but for our purposes the interface specifies only that the state exists and specifies which of the procedures change it.

Here is our hypothesized erroneous program written for converting an array of integers into a `TEXT`:

```
PROCEDURE ArrayToText(a : ARRAY OF INTEGER) : TEXT =
  VAR twr := NEW(TextWr.T); BEGIN
    FOR i := 1 TO NUMBER(a) DO TextWr.PutInt(twr, a[i]) END;
    RETURN TextWr.GetText(twr)
  END ArrayToText;
```

On this example, ESC reports two errors: the first error is an array bounds violation in `ArrayToText`. Here is the essence of the error message that ESC produces: "warning: possible array bounds error: `TextWr.PutInt(twr, a[i])` ".

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 careful study of the context will reveal that it implies the formula $i = NUMBER(a)$, which is in fact the condition in which the bounds error can occur: in Modula-3, open arrays are addressed from `0`, but the `FOR` loop was written as though they were addressed from `1`. Correcting

the error in one natural way produces the following improved FOR loop: FOR i := 0 TO LAST(a) DO ...

But ESC complains about this program too, as follows: "warning: precondition failed: TextWr.PutInt(twr, a[i]) ". A study of the error context reveals that it contains the formula NOT valid[twr]. That is, ESC has detected and warned about the failure to initialize twr. Correcting this error changes the beginning of the procedure implementation to VAR twr := NEW(TextWr.T); BEGIN TextWr.Init(twr);. And with this program ESC is unable to find fault.

We would like to make several comments about this example.

First, although careful specifications were required for the text writer interface, 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 ArrayToText. We think that this is as it should be: anybody qualified to design interfaces understands preconditions and postconditions and abstractions at some level, and will find an explicit notation for their design decisions to be a tool rather than a burden; on the other hand, many simple errors in programs that use an interface can and should be identified by reading the unannotated erroneous program.

Second, in the actual Modula-3 I/O system, the type TextWr is declared as a subtype of a more general writer (Wr.T). Operations like PutInt and Close apply to any writer. GetText applies to text writers only. We have ignored this aspect of the example to save words, but the actual ESC checker handles objects and subtyping gracefully.

Third, the reader should be aware that this is only half an example. The other half is the checking of the implementation of writers and text writers. In these implementations, *representation* declarations are made to give the meaning of valid[twr] in terms of the concrete fields of twr (including both generic and subtype-specific conjuncts). These representations are used by ESC when checking the body of procedures like PutInt(wr) and GetText(twr) that depend on the concrete meaning of validity.

Fourth, it is in fact true that initializing a text writer leaves its contents empty. If we wanted to, we could reflect this in the postcondition of Init as follows:

```
<* SPEC Init(t)
   MODIFIES valid[t], state[t] ENSURES valid'[t] AND state'[t] = "" *>
```

It would be easy to concoct an artificial example in which this stronger specification would be essential, if, say, the absence of array bounds errors in some client depended on the fact that a newly initialized text writer was empty. But this is a slippery slope. If Init's effect on the state is specified fully, why not PutInt'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 valid/state 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 great rug of MODIFIES state[t]. We believe this is a key reason why ESC verifications are more cost effective than full correctness verifications.

More information and references can be found on the Web at www.research.digital.com/SRC/esc/Esc.html.