Generating error traces from verification-condition counterexamples

K. Rustan M. Leino[a], Todd Millstein[b], and James B. Saxe[c]

[a]Microsoft Research, Redmond, WA, USA

[b]UCLA Computer Science Department, Los Angeles, CA, USA

[c]HP Labs, Palo Alto, CA, USA

A technique for finding errors in computer programs is to translate a given program and its correctness criteria into a logical formula in mathematics and then let an automatic theorem prover check the validity of the formula. This approach gives the tool designer much flexibility in which conditions are to be checked, and the technique can reason about as many aspects of the given program as the underlying theorem prover allows. This paper describes a method for reconstructing, from the theorem prover's mathematical output, error traces that lead to the program errors that the theorem prover discovers.

0. Introduction

Mechanical tools that analyze software in search of errors can help find software defects earlier in the development process, which reduces the overall cost of software development. Some software analyses focus on a particular kind of program error, for example index bounds errors in programs with arrays [4] or car-of-atom errors in functional programs [7]. A more general approach is to encode the program's correctness as a *verification condition*, a logical formula that is valid if and only if the program is free of the classes of errors that are analyzed. A theorem prover can then be used to determine the validity of the verification condition.

While this translation of a program into mathematics provides a flexible approach for encoding a variety of program properties—as many as the theories of the underlying theorem prover support—it may not be obvious how to translate the theorem prover's output back into something that a programmer can make sense of. In this paper, we show a method for constructing the verification condition in such a way that it is easy to reconstruct, from the theorem prover's output, a trace in the program that leads to the error that the theorem prover has discovered. The method has been implemented in the Extended Static Checker for Java (ESC/Java) [8]. It enables ESC/Java to produce warning messages like the one shown in Figure 0.

Our method can be used with a variety of refutation-based theorem provers. Our implementation of the method in ESC/Java makes use of a special *labeling* feature, which ESC/Java's theorem prover Simplify [11] supports. The labeling feature lends itself to an efficient implementation of the method.

We start by showing the generation of verification conditions for a toy language. We

```
Cup.java:13: Warning: Possible null dereference (Null)
        int y = t.f;
                 ^
Execution trace information:
    Executed else branch in "Cup.java", line 5, col 1.
    Reached top of loop after 0 iterations in "Cup.java", line 8, col 1.
    Executed then branch in "Cup.java", line 9, col 12.
    Executed break in "Cup.java", line 10, col 2.
```

Figure 0. Example output from ESC/Java, calling attention to a possible error and showing an execution trace leading to the program point of the reported error.

then describe our view of the theorem prover and labels. In Section 2, we review how the labeling mechanism was used in the Extended Static Checker for Modula-3 (ESC/Modula-3) [10], and also in ESC/Java, to pinpoint the *location* of an error, that is, the point in the program where the error is manifested. In Section 3, we show our method, which allows the tool to output not just the location of the error, but also an execution trace leading to the error. Section 4 considers some programming constructs beyond those in the toy language and Section 5 considers some implementation issues. The paper ends with some related work and a conclusion.

1. Preliminaries

To explain our method, we introduce a simple imperative programming language. Programs written in this *source* language can contain a single kind of error, array index out-of-bounds errors. Section 4 discusses the extension of our method to a richer language.

To find errors, we translate the source language into a verification condition in two steps [17]: we first desugar the source language into a more primitive *intermediate* language and then compute verification conditions from the intermediate language according to the semantics of *weakest preconditions* [6,22].

1.0. Source language

The source language operates on (implicitly declared) variables whose type is either integer or array of integer. The grammar of the source language is shown in Figure 1. To keep the language as simple as possible, an array A can be used only in the special assignment statement $x := A[i]$, where x denotes an integer variable and i denotes an (integer-valued) expression. For brevity, we have omitted array updates, so the arrays are effectively read-only. Also, for simplicity, we assume that all arrays have length 100. Expressions used to index into arrays must therefore be non-negative integers less than 100. The verification conditions we will produce are valid if and only if the source program is free of array index out-of-bounds errors.

$$\begin{array}{rcl}
\mathit{Stmt} & ::= & \mathit{Id} := \mathit{Expr} \\
 & | & \mathit{Id} := \mathit{Id}\ [\ \mathit{Expr}\ ] \\
 & | & \mathit{Stmt}\ ;\ \mathit{Stmt} \\
 & | & \mathbf{if}\ \mathit{Cond}\ \mathbf{then}\ \mathit{Stmt}\ \mathbf{else}\ \mathit{Stmt}\ \mathbf{end} \\
\mathit{Cond} & ::= & \mathit{Cond} \wedge \mathit{Cond} \mid \mathit{Cond} \vee \mathit{Cond} \mid \neg \mathit{Cond} \\
 & | & (\mathit{Cond}) \\
 & | & \mathit{Expr} = \mathit{Expr} \mid \mathit{Expr} < \mathit{Expr} \\
 & | & \mathbf{false} \mid \mathbf{true} \\
\mathit{Expr} & ::= & \mathit{Expr} + \mathit{Expr} \mid \mathit{Expr} - \mathit{Expr} \\
 & | & (\mathit{Expr}) \\
 & | & \mathit{Id} \\
 & | & 0 \mid 1 \mid 2 \mid \cdots
\end{array}$$

Figure 1. The grammar of the source language.

1.1. Intermediate language

We translate ("*desugar*") source programs into programs in a more primitive intermediate language whose grammar is shown in Figure 2. In the intermediate language, there are maps instead of arrays; the difference is that maps can be applied (using the function *select*) to any integer, whereas arrays can be indexed only at integers between 0 and 99 inclusive.

The commands in our simple intermediate language always terminate, but there are three kinds of termination: normal, erroneous, and miraculous. We use normal termination to model state changes in the source program. We use erroneous termination to model errors in the source program. For our source language, these errors are array index out-of-bounds errors. We use miraculous termination to model the programmer being "off the hook" (that is, absolved of responsibility for any subsequent error; see, *e.g.*, [0,20,22,21,17]).

An assignment command $x := E$ updates the state by setting the variable x to the value of the expression E and then terminates normally. The commands **assert** P and **assume** P terminate normally, without changing the state, if P evaluates to **true**; otherwise, the **assert** statement terminates erroneously and the **assume** statement terminates miraculously. That is, the command **assert** P in effect says that the programmer is responsible for making sure P holds, whereas **assume** P says that the programmer is off the hook if P does not hold.

The command $C\ ;\ D$ executes C and then, if C terminates normally, executes D. The command $C \,\square\, D$ arbitrarily ("demonically") chooses to execute either C or D. We let ; bind more strongly than $\square$ and use curly braces to indicate a different grouping.

The expressions in the intermediate language include the *select* function, which is used to dereference maps. The **lblneg** and **lblpos** expressions are explained later. As suggested by the grammar, some formulas are *atomic*, meaning that they have no proper sub-formulas. A *literal* is an atomic formula or its negation. According to the grammar,

$$
\begin{array}{rcl}
\mathit{Cmd} & ::= & \mathit{Id} := \mathit{Term} \\
 & | & \mathbf{assert}\ \mathit{Formula} \\
 & | & \mathbf{assume}\ \mathit{Formula} \\
 & | & \{\mathit{Cmd}\} \\
 & | & \mathit{Cmd}\ ;\ \mathit{Cmd} \\
 & | & \mathit{Cmd}\ \Box\ \mathit{Cmd} \\
\mathit{Formula} & ::= & \mathit{Formula} \Rightarrow \mathit{Formula} \\
 & | & \mathit{Formula} \land \mathit{Formula} \mid \mathit{Formula} \lor \mathit{Formula} \mid \neg \mathit{Formula} \\
 & | & (\mathbf{lblneg}\ \mathit{Id}\colon\ \mathit{Formula}) \mid (\mathbf{lblpos}\ \mathit{Id}\colon\ \mathit{Formula}) \\
 & | & (\mathit{Formula}) \\
 & | & \mathit{Atom} \\
\mathit{Atom} & ::= & \mathit{Term} = \mathit{Term} \mid \mathit{Term} < \mathit{Term} \\
 & | & \mathit{Id} \\
 & | & \mathbf{false} \mid \mathbf{true} \\
\mathit{Term} & ::= & \mathit{Term} + \mathit{Term} \mid \mathit{Term} - \mathit{Term} \\
 & | & \mathit{select}(\mathit{Term}, \mathit{Term}) \\
 & | & (\mathit{Term}) \\
 & | & \mathit{Id} \\
 & | & 0 \mid 1 \mid 2 \mid \cdots
\end{array}
$$

Figure 2. *Cmd* gives the grammar of the intermediate-language commands. *Formula* also gives the grammar for the formulas of verification conditions.

$\mathit{Desugar}(x := E) \ =$
$\quad x := E$
$\mathit{Desugar}(x := A[E]) \ =$
$\quad \textbf{assert}\ \neg(E < 0) \wedge E < 100\ ;\ x := \mathit{select}(A, E)$
$\mathit{Desugar}(S\ ;\ T) \ =$
$\quad \mathit{Desugar}(S)\ ;\ \mathit{Desugar}(T)$
$\mathit{Desugar}(\textbf{if}\ P\ \textbf{then}\ S\ \textbf{else}\ T\ \textbf{end}) \ =$
$\quad \{\textbf{assume}\ P\ ;\ \mathit{Desugar}(S)\ \Box\ \textbf{assume}\ \neg P\ ;\ \mathit{Desugar}(T)\}$

Figure 3. The desugaring of source-language statements into intermediate-language commands.

both atomic formulas and terms can be identifiers. The term identifiers correspond to program variables; the formula identifiers are uninterpreted predicate symbols, which we use later in the paper.

Here and throughout, $\neg$ binds more strongly than $\wedge$ and $\vee$, which in turn bind more strongly than $\Rightarrow$.

The translation from the source language into the intermediate language is performed by the function *Desugar*, defined in Figure 3. The **if** statement is desugared into an arbitrary choice between two commands, each one guarded by an **assume** statement, which has the effect of the programmer getting off the hook if the inappropriate branch is chosen.

1.2. Verification conditions

Verification-condition formulas follow the same grammar as the formulas in the intermediate language shown in Figure 2. To translate from the intermediate language into formulas, we use weakest preconditions [6,22]. For any command C and formula R on the post-state of C, the weakest precondition of C with respect to R, written $wp(C, R)$, characterizes the set of pre-states of C from which execution terminates normally in a state satisfying R or terminates miraculously. Formally, we define wp over the structure of commands, as shown in Figure 4, where $R[x := E]$ denotes the formula identical to R but with each occurrence of x replaced with E.

We have now defined a translation from the source language to verification conditions. In particular, for any source program S, the formula:

$$wp(\mathit{Desugar}(S),\ \textbf{true}) \qquad (0)$$

is valid if and only if S is free of errors (that is, if and only if erroneous termination is not a possible outcome of $\mathit{Desugar}(S)$).

1.3. Proof procedure

We postulate an automatic proof procedure (theorem prover) that takes a logical formula as input and produces as output an indication of whether or not the formula is

$$\begin{array}{rcl}
wp(x := E,\ R) & = & R[x := E] \\
wp(\mathbf{assert}\ P,\ R) & = & P \wedge R \\
wp(\mathbf{assume}\ P,\ R) & = & P \Rightarrow R \\
wp(\{C\},\ R) & = & wp(C, R) \\
wp(C\ ;\ D,\ R) & = & wp(C,\ wp(D, R)) \\
wp(C\ \Box\ D,\ R) & = & wp(C, R) \wedge wp(D, R)
\end{array}$$

Figure 4. The weakest preconditions of commands with respect to any formula R on the post-state.

valid.[0] An indication of "invalid" is accompanied with a *counterexample context*, as we describe next.

A counterexample context Q for a given formula P is a conjunction of literals (that is, possibly negated atomic formulas) that satisfies the following two conditions:

$(Q \Rightarrow \neg P)$ valid

and

Q satisfiable

The first condition states that Q entails $\neg P$; the second condition states that Q is satisfiable.[1] Equivalently to thinking of Q as a counterexample context for P, we may think of Q as a *satisfying context* for $\neg P$.

To check a source program S for errors, we construct the verification condition (0) and pass it to the proof procedure. If the proof procedure responds with "valid", the program is correct; if the proof procedure responds with a counterexample context, the program contains an error.

For example, consider the following contrived source program:

```
if $\neg(k < 10) \wedge k < 20$ then
   $j := k$
else
   $j := k$                                  (1)
end;
$x := A[j]$
```

[0] For the purpose of this paper, it does not matter if the proof procedure is mathematically complete, but an incompleteness in the proof procedure can result in spurious warnings being produced by the program checker.

[1] Or rather, satisfiable insofar as the proof procedure can tell, since the proof procedure may in general be incomplete. From now on, we won't continue to belabor the point about completeness.

The desugared version of this program is:

```
{    **assume** $\neg(k < 10) \wedge k < 20$ ;
     $j := k$
$\Box$    **assume** $\neg(\neg(k < 10) \wedge k < 20)$ ;
     $j := k$} ;
**assert** $\neg(j < 0) \wedge j < 100$ ;
$x := \mathit{select}(A, j)$
```

The verification condition for the program is:

$$\begin{array}{l} (\neg(k < 10) \wedge k < 20 \Rightarrow \\ \qquad \neg(k < 0) \wedge k < 100 \wedge \textbf{true}) \wedge \\ (\neg(\neg(k < 10) \wedge k < 20) \Rightarrow \\ \qquad \neg(k < 0) \wedge k < 100 \wedge \textbf{true}) \end{array}$$

A counterexample context for this program is:

$$k < 0$$

which reveals the existence of an error in the program.

In the next section, we describe how to determine the source-program location of an error from a counterexample context. Before we do that, we need to explain labels.

1.4. Labels

The time has come to explain the **lblneg** and **lblpos** formulas, which are used to label sub-formulas. Intuitively, (**lblneg** L: P) and (**lblpos** L: P) are logically equivalent to P but cause the proof procedure to emit the label L when producing a counterexample in which P has the indicated sense (negative or positive, respectively; that is, $\neg P$ or P), and in which the labeled occurrence of P is "relevant". Formally, we define the labeled formulas as follows:

$$\begin{array}{lcl} (\textbf{lblneg}\ L\colon\ P) & = & P \vee L \\ (\textbf{lblpos}\ L\colon\ P) & = & P \wedge \neg L \end{array} \qquad (2)$$

where, on the right-hand side, we use label L as an uninterpreted predicate symbol.

If the identifiers used as labels are distinct and distinct from other identifiers, **lblneg** is used only in positive contexts, and **lblpos** is used only in negative contexts, then the labeling of sub-formulas in a formula P yields a formula P' that is valid if and only if P is. Worded differently, $\neg P$ and $\neg P'$ are *equisatisfiable*. Here's a proof sketch. Let's write P as $\mathcal{P}(A, B, C)$, where $\mathcal{P}$ is monotonic in A and antimonotonic in B. That is, $\mathcal{P}$ uses A only in positive contexts and B only in negative contexts. Let L and M be any identifiers that do not occur in $\mathcal{P}(A, B, C)$ (but we allow L and M to be the same identifier). We can now think of P' as:

$$\mathcal{P}((\textbf{lblneg}\ L\colon\ A),\ (\textbf{lblpos}\ M\colon\ B),\ C)$$

We calculate:

$$
\begin{array}{ll}
 & \mathcal{P}(A, B, C) \text{ valid} \\
\Rightarrow & \quad \{ \text{ weaken positive contexts and strengthen negative contexts } \} \\
 & \mathcal{P}(A \lor L,\ B \land \neg M,\ C) \text{ valid} \\
= & \quad \{ \text{ (2): definition of } \mathbf{lblneg} \text{ and } \mathbf{lblpos} \ \} \\
 & \mathcal{P}((\mathbf{lblneg}\ L\colon A),\ (\mathbf{lblpos}\ M\colon B),\ C) \text{ valid} \\
= & \quad \{ \text{ (2): definition of } \mathbf{lblneg} \text{ and } \mathbf{lblpos} \ \} \\
 & \mathcal{P}(A \lor L,\ B \land \neg M,\ C) \text{ valid} \\
\Rightarrow & \quad \{ \text{ a formula remains valid under instantiation, so instantiate} \\
 & \quad\quad \text{both } L \text{ and } M \text{ with } \mathbf{false} \ \} \\
 & \mathcal{P}(A \lor \mathbf{false},\ B \land \neg\mathbf{false},\ C) \text{ valid} \\
= & \quad \{ \text{ logic } \} \\
 & \mathcal{P}(A, B, C) \text{ valid}
\end{array}
$$

which shows, by mutual implication, that P and P' are equally valid.

The *label set* of a counterexample context Q is the set of label symbols that appear negated in Q. We assume that the proof procedure does not gratuitously negate label symbols, but that it only gives values to label symbols when such values are needed to produce a counterexample context.

Next, we look at how labels can be used in the generation of verification conditions so that their appearance in a counterexample context tells us something about the errors in a source program.

2. Error locations

The translation from the source language to the intermediate language introduces an **assert** statement for each check to be performed on the program. For example, in our toy language, an assertion is generated for every array access in a program, to ensure that the access is within the appropriate bounds. If the proof procedure produces a counterexample context, it is because at least one of these **assert** statements cannot be proven always to succeed. We would like to extract from a counterexample context enough information to tell which of the many assertions is potentially erroneous.

To this end, in ESC/Modula-3 [10], and subsequently also in ESC/Java [8,17], assertions in the intermediate language are adorned with negative labels. The labels encode the kind of error and the location in the source program of the statement that desugars into the assertion. We change the desugaring of array accesses from that given in Figure 3 to:

$$
\begin{array}{l}
\mathit{Desugar}(x := A[E]) \ = \\
\quad \mathbf{assert}\ (\mathbf{lblneg}\ \mathit{ArrayAccess@l.c}\colon \neg(E < 0) \land E < 100)\ ; \\
\quad x := \mathit{select}(A, E)
\end{array}
$$

where l and c denote the line and column in the source program where the assignment occurs, and where '@' and '.' are special characters that can appear in intermediate-language identifiers.

Consider again our example (1) from the previous section. Assuming the array access is at line 218, column 23 of the source program, the desugared version becomes:

```
{   assume $\neg(k < 10) \wedge k < 20$ ;
    $j := k$
□   assume $\neg(\neg(k < 10) \wedge k < 20)$ ;
    $j := k$} ;
assert (lblneg $\mathit{ArrayAccess}$@218.23: $\neg(j < 0) \wedge j < 100$) ;
$x := \mathit{select}(A, j)$
```

The verification condition then becomes:

$$\begin{array}{l} (\neg(k < 10) \wedge k < 20 \Rightarrow \\ \quad (\textbf{lblneg}\ \mathit{ArrayAccess}@218.23\text{: } \neg(k < 0) \wedge k < 100) \wedge \textbf{true}) \wedge \\ (\neg(\neg(k < 10) \wedge k < 20) \Rightarrow \\ \quad (\textbf{lblneg}\ \mathit{ArrayAccess}@218.23\text{: } \neg(k < 0) \wedge k < 100) \wedge \textbf{true}) \end{array} \tag{3}$$

Expanding the **lblneg** expressions, we get:

$$\begin{array}{l} (\neg(k < 10) \wedge k < 20 \Rightarrow \\ \quad ((\neg(k < 0) \wedge k < 100) \vee \mathit{ArrayAccess}@218.23) \wedge \textbf{true}) \wedge \\ (\neg(\neg(k < 10) \wedge k < 20) \Rightarrow \\ \quad ((\neg(k < 0) \wedge k < 100) \vee \mathit{ArrayAccess}@218.23) \wedge \textbf{true}) \end{array}$$

Finally, the counterexample context produced is now:

$$k < 0 \ \wedge \ \neg \mathit{ArrayAccess}@218.23$$

The appearance of the negated label in the counterexample context reveals which array access in the source program is potentially erroneous.

We remark that there are two occurrences in (3) of the sub-formula labeled with *ArrayAccess*@218.23. This duplication comes about from the definition of $wp(C \ \square \ D,\ R)$ in Figure 4. The duplication suggests a way to generate labels that encode not just the location of an error but also the entire execution path leading to the error: simply define

$$wp(C \ \square \ D,\ R) \ = \ wp(C, R) \wedge wp(D, R') \tag{4}$$

where R' is R in which the labels bear additional qualifications to encode branching through the second command (D). However, in practice, it is important for performance reasons to avoid this duplication altogether, so it is not possible to use the idea (4). ESC/Java follows a scheme [9,16] that sometimes gives verification conditions in a form significantly smaller and/or easier-to-prove than those generated according to Figure 4. For the present example, this scheme produces only one occurrence of the formula labeled with *ArrayAccess*@218.23. For simplicity in this paper, we continue to use the straightforward encoding of $wp(C \ \square \ D,\ R)$ as shown in Figure 4.

3. Error traces

The error location itself sometimes does not contain enough information for a programmer to figure out how the error may arise. In our running example (1), we learn that the array access at the end is potentially erroneous, but we don't know whether the problem arises by passing through the **then** or the **else** branch of the preceding **if** statement. In this section, we describe how to use labels to determine a complete *error trace* that represents a scenario under which the erroneous statement may fail.

We extend the labeling method of the previous section. In addition to labeling assertions, which represent the various program checks, we now label each **assume** statement, which represents a particular branch of an **if** statement. The assumed conditions end up in antecedents (negative positions) in the verification condition, so we use **lblpos** formulas. We change the desugaring of **if** statements from that given in Figure 3 to:

$$\begin{aligned} &\mathit{Desugar}(\textbf{if } P \textbf{ then } S \textbf{ else } T \textbf{ end}) \;= \\ &\qquad \{\quad \textbf{assume } (\textbf{lblpos } \mathit{Then}@\mathit{lt}.\mathit{ct}\colon\ P)\ ;\ \mathit{Desugar}(S) \\ &\qquad \square\quad \textbf{assume } (\textbf{lblpos } \mathit{Else}@\mathit{le}.\mathit{ce}\colon\ \neg P)\ ;\ \mathit{Desugar}(T)\} \end{aligned} \tag{5}$$

where lt and ct are the line and column of the **then** branch and le and ce are the line and column of the **else** branch.

The desugared version of our running example (1) now looks as follows:

$$\begin{aligned} &\{\quad \textbf{assume } (\textbf{lblpos } \mathit{Then}@214.5\colon\ \neg(k < 10) \wedge k < 20)\ ; \\ &\qquad j := k \\ &\square\quad \textbf{assume } (\textbf{lblpos } \mathit{Else}@216.5\colon\ \neg(\neg(k < 10) \wedge k < 20))\ ; \\ &\qquad j := k \\ &\}\ ; \\ &\textbf{assert } (\textbf{lblneg } \mathit{ArrayAccess}@218.23\colon\ \neg(j < 0) \wedge j < 100)\ ; \\ &x := \mathit{select}(A, j) \end{aligned}$$

The verification condition, with labels expanded, becomes:

$$\begin{aligned} &((\neg(k < 10) \wedge k < 20 \wedge \neg \mathit{Then}@214.5) \;\Rightarrow \\ &\qquad ((\neg(k < 0) \wedge k < 100) \vee \mathit{ArrayAccess}@218.23) \wedge \textbf{true}) \;\wedge \\ &((\neg(\neg(k < 10) \wedge k < 20) \wedge \neg \mathit{Else}@216.5) \;\Rightarrow \\ &\qquad ((\neg(k < 0) \wedge k < 100) \vee \mathit{ArrayAccess}@218.23) \wedge \textbf{true}) \end{aligned}$$

Finally, the counterexample context produced is now:

$$k < 0 \;\;\wedge\;\; \neg \mathit{ArrayAccess}@218.23 \;\;\wedge\;\; \neg \mathit{Else}@216.5$$

The negated labels in the counterexample context now reveal both a particular array access in the source program and an execution trace to that access that is potentially erroneous.

4. Other programming constructs

In this section, we consider how to deal with a larger source language, one that contains short-circuit boolean operators, exceptions, loops, and procedures.

4.0. Short-circuit boolean operators

Short-circuit boolean operators in expressions also give rise to branches in executions. To describe our method for short-circuit boolean operators, we consider a small extension of our source language. We modify the grammar for the **if** statement as follows:

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \ldots \\ & | & \textbf{if } \mathit{CCond} \textbf{ then } \mathit{Stmt} \textbf{ else } \mathit{Stmt} \textbf{ end} \\ \mathit{CCond} & ::= & \mathit{Cond} \\ & | & \mathit{Cond}\,\textbf{cand}\,\mathit{CCond} \end{array}$$

where **cand** is the short-circuit operator *conditional-and*. For simplicity, we allow the new operator only as a top-level operator in the guard of **if** statements.

The new desugaring of **if** is different from the previous desugaring (5) in two ways. First, we introduce a fresh variable κ to hold the value of the evaluated guard expression. Using $\mathit{Eval}(\mathit{CP})$ to denote the desugaring that causes conditional expression CP to be evaluated and its result to be assigned to variable κ, we change the desugaring of **if** statements from (5) to:

$$\begin{array}{l} \mathit{Desugar}(\textbf{if } \mathit{CP} \textbf{ then } S \textbf{ else } T \textbf{ end}) \ = \\ \quad \mathit{Eval}(\mathit{CP}) \ ; \\ \quad \{ \quad \textbf{assume } (\textbf{lblpos } \mathit{Then@lt.ct}\colon\ \kappa) \ ; \mathit{Desugar}(S) \\ \quad \Box \quad \textbf{assume } (\textbf{lblpos } \mathit{Else@le.ce}\colon\ \neg\kappa) \ ; \mathit{Desugar}(T)\} \end{array}$$

Second, to track the branching introduced by the short-circuit boolean operator, we define *Eval* as follows, for any non-conditional expression P and any expression Q:

$$\begin{array}{l} \mathit{Eval}(P) \ = \\ \quad \kappa := P \\ \mathit{Eval}(P\,\textbf{cand}\,Q) \ = \\ \quad \kappa := P \ ; \\ \quad \{ \quad \textbf{assume } \kappa \ ; \mathit{Eval}(Q) \\ \quad \Box \quad \textbf{assume } (\textbf{lblpos } \mathit{Cand@l.c}\colon\ \neg\kappa)\} \end{array}$$

where l and c denote the line and column in the source program of the operator **cand**.[2]

Note that we introduce a label only for the case when evaluation of **cand** is actually short-circuited. We found from experience with ESC/Java that too many error-trace labels in the tool's output (and hence in the resulting error trace presented to users) can be more distracting than helpful. Therefore, we use a label only in the "unexpected" case where the short-circuit nature of the conditional operator is used.

4.1. Exits and exceptions

Programming languages may offer several mechanisms to avert normal control flow, for example to exit loops prematurely or to signal a special condition that is handled in some enclosing context. To describe our method for such language features, we extend our source language with named blocks and exits (which can be generalized into a fuller exception mechanism):

$$\begin{array}{rcl} \mathit{Stmt} & ::= & \ldots \\ & | & \textbf{name } \mathit{Id} \ \{\mathit{Stmt}\} \\ & | & \textbf{exit } \mathit{Id} \end{array}$$

[2] Some readers may have noticed the conflation of formulas and terms: the desugaring shown here assigns formulas to the variable κ. ESC/Java actually employs a slightly more complex desugaring (beyond the scope of this paper) involving a reflection of the formulas **false** and **true** into the term domain.

A block statement **name** N $\{S\}$ introduces a name N that can be used in **exit** statements inside S. The execution of such a statement **exit** N causes the entire block statement to terminate; that is, **exit** N transfers control to the point immediately following the block statement.

To desugar named blocks, we extend the intermediate language with the ability to raise and handle *exceptions*:

```
Cmd ::= ...
     |  raise
     |  try Cmd catch Cmd end
```

In the presence of these features, intermediate-language commands have a new possible outcome: they can terminate *exceptionally* in some state. The **raise** statement always terminates exceptionally, without changing the program state. The statement

try C **catch** D **end**

executes C and then, if C terminates exceptionally, executes D. The command D is often called an *exception handler*. We omit from this paper a weakest-precondition formalization of commands with exceptional outcomes, but see, *e.g.*, [17,5,15].

Named blocks and exits are desugared as follows. First, we introduce a special variable ξ that records the label in the most recent **exit** statement. Without concern about labels, we would then define *Desugar* on the new statements as follows:

```
Desugar(exit N) =
    $\xi := N$ ; raise
Desugar(name N {S}) =
    try
        Desugar(S)
    catch
            assume $\xi = N$
        $\Box$  assume $\neg(\xi = N)$ ; raise
    end
```

where the block names are treated as symbolic constants with distinct values. The desugaring of **exit** records the name of the specified exit and then raises an exception. The block statement desugars into a **try** command that behaves like the desugaring of S, except that it turns any exceptional termination where $\xi = N$ into normal termination. With labels, we use the following desugaring:

```
Desugar(exit N) =
    assume (lblpos Exit@l.c: true) ; $\xi := N$ ; raise
Desugar(name N {S}) =
    try
        Desugar(S)
    catch
            assume (lblpos EndBlock@lb.cb: $\xi = N$)
        $\Box$  assume $\neg(\xi = N)$ ; raise
    end
```

where l and c denote the line and column of the **exit** statement and lb and cb denote the line and column of the end of the named block. We record the point of the **exit** by a positive label on **true**, which will appear in any counterexample context that involves the **exit**. This label allows resulting error traces to highlight the origin of a thrown exception. By also including a label on entry to the exception handler, resulting error traces highlight where the thrown exception is caught, which can be many source lines apart from the **exit** statement. We have chosen not to include a label for the case where the exception is re-raised, to be handled by an enclosing block, but this could easily be incorporated.

4.2. Loops

We consider the addition of an iterative construct to the source language:

$$\begin{array}{lcl} \mathit{Stmt} & ::= & \ldots \\ & | & \textbf{loop } \{\textbf{inv } \mathit{Cond}\} \ \mathit{Stmt} \ \textbf{end} \end{array}$$

where the condition gives programmers the ability to specify a loop invariant (which may be the trival invariant **true**). For simplicity, the loop has no explicit loop guard; instead, the loop is an infinite loop that can be exited prematurely by enclosing the loop inside a named block and using **exit** statements inside the loop (see the previous subsection).

ESC/Java provides two different desugarings of loops. We describe each one.

The full ("safe", "sound") translation of a loop considers an arbitrary iteration of the loop body. For a loop at line l and column c, this desugaring is:

$$\begin{array}{l} \mathit{Desugar}(\textbf{loop } \{\textbf{inv } J\} \ S \ \textbf{end}) \ = \\ \qquad \textbf{assert } (\textbf{lblneg } \mathit{LoopInvInit@l.c}\colon \ J) \ ; \\ \qquad \ldots \ ; \\ \qquad \textbf{assume } J \ ; \\ \qquad \textbf{assume } (\textbf{lblpos } \mathit{LoopHead@l.c}\colon \ \textbf{true}) \ ; \\ \qquad \mathit{Desugar}(S) \ ; \\ \qquad \textbf{assert } (\textbf{lblneg } \mathit{LoopInvMaintained@l.c}\colon \ J) \ ; \\ \qquad \textbf{assume false} \end{array}$$

where the "..." assigns arbitrary values to the assignment targets of the loop. By following this arbitrary assignment with the command **assume** J, the intermediate program effectively models the execution of an arbitrary number of loop iterations (*cf.* [17]). The command **assume false** at the end of the desugaring has the effect of ending the checking after this arbitrary iteration (but remember that the loop body may use **exit** statements to continue execution beyond the loop).

This desugaring uses different labels for the two loop-invariant checks, one for checking the loop invariant on entry to the loop and one for checking the maintenance of the loop invariant by the loop body [17]. These labels allow error traces to give programmers precise information about why J is not a valid loop invariant. For the benefit of error traces, we also include a label at the beginning of the arbitrary iteration.

ESC/Java also provides a limited desugaring of loops, in which it unrolls loops a specified number of times. This limited desugaring does not capture all possible executions of the loop, and so may cause the checker to miss errors; in return, the checker is faster and can do some useful checking without the need for non-trivial loop invariants [10,8]. For a loop at line l and column c, this limited desugaring for k unrollings is:

$$
\begin{aligned}
&\mathit{Desugar}(\textbf{loop}\ \{\textbf{inv}\ J\}\ S\ \textbf{end})\ =\\
&\qquad \textbf{assert}\ (\textbf{lblneg}\ \mathit{LoopInvInit}@l.c\text{: }\ J)\ ;\\
&\qquad \mathit{Unroll}(\textbf{loop}\ \{\textbf{inv}\ J\}\ S\ \textbf{end},\ 0,\ k)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Unroll}(\textbf{loop}\ \{\textbf{inv}\ J\}\ S\ \textbf{end},\ j,\ k)\ =\\
&\qquad \begin{cases}
\left.\begin{array}{l}
\textbf{assume}\ (\textbf{lblpos}\ \mathit{LoopHead}.j@l.c\text{: }\ \textbf{true})\ ;\\
\mathit{Desugar}(S)\ ;\\
\textbf{assert}\ (\textbf{lblneg}\ \mathit{LoopInvMaintained}@l.c\text{: }\ J)\ ;\\
\mathit{Unroll}(\textbf{loop}\ \{\textbf{inv}\ J\}\ S\ \textbf{end},\ j+1,\ k)
\end{array}\right] & \text{if } j < k\\
\textbf{assume false} & \text{otherwise}
\end{cases}
\end{aligned}
$$

For example, for $k = 1$, this desugaring yields:

$$
\begin{aligned}
&\textbf{assert}\ (\textbf{lblneg}\ \mathit{LoopInvInit}@l.c\text{: }\ J)\ ;\\
&\textbf{assume}\ (\textbf{lblpos}\ \mathit{LoopHead}.0@l.c\text{: }\ \textbf{true})\ ;\\
&\mathit{Desugar}(S)\ ;\\
&\textbf{assert}\ (\textbf{lblneg}\ \mathit{LoopInvMaintained}@l.c\text{: }\ J)\ ;\\
&\textbf{assume false}
\end{aligned}
$$

We have simplified this explanation a bit too much. As stated, the resulting labels would not all be distinct, and thus would not allow the tool to determine which loop-body branches were considered in which unrolling of the loop. For example, suppose the loop body contains an **if** statement and suppose the number of loop unrollings (k above) is set to 2. Then, if a counterexample context arises because of a program error that is reached by taking the **then** branch in the first iteration and the **else** branch in the second iteration, then both of the labels *Then*@*lt*.*ct* and *Else*@*le*.*ce* would appear negated in the counterexample context, but there would be no information as to which branch is taken during which iteration.

To avoid this problem, ESC/Java adds a sequence number to each label generated during verification-condition generation, with the property that the sequence numbers reached in any execution of the intermediate-language program appear in ascending order. ESC/Java then sorts the trace labels returned by the theorem prover according to their sequence numbers, and prints the error trace information in that order.

4.3. Procedures

Without going into the details of procedure desugaring (see, *e.g.*, [17,15]), we briefly make a couple of notes about labels and procedure calls.

We have found that it is sometimes useful to point out calls in an error trace. This is especially true if a procedure call is inlined. To include such a label, our desugaring uses a command like:

$$\textbf{assume}\ (\textbf{lblpos}\ \mathit{Call}@l.c\text{: }\ \textbf{true})$$

at the beginning of the procedure-call desugaring.

If a call is inlined, it may also be useful to provide a label at the point where execution is resumed in the caller, since the caller and the return in the callee may be many source lines apart. Moreover, even if calls are not inlined, we have found it useful to include a label for the case that corresponds to an exceptional return from the procedure. To avoid getting multiple occurrences of the same label when a procedure is inlined in several places, ESC/Java adds sequence numbers to labels, as we described above for unrolled loop iterations.

5. Implementation considerations

Our definition of labeled formulas (2) allows our method to be used with any proof procedure that does not gratuitously negate label symbols in its counterexample contexts. However, if labeled expressions are actually expanded into disjunctions or conjunctions according to (2), there is sometimes an adverse impact on prover performance.

Consider, for example, a source program containing the fragment:

$x := A[j]$;
$y := A[j] + 1$

In the absence of labels, the two array accesses give rise to the same assertion in the intermediate language, namely:

assert $\neg(j < 0) \wedge j < 100$

Since there is no assignment to j between the two instances of these assertions, they will give rise to two instances of the same subformula in the verification condition. With labels, expanded according to (2), the two assignments may give rise to different assertions:

assert $(\neg(j < 0) \wedge j < 100) \vee \mathit{ArrayAccess}@104.11$;
. . .
assert $(\neg(j < 0) \wedge j < 100) \vee \mathit{ArrayAccess}@105.11$

and thus to different subformulas in the verification condition, making it more likely that work done by the prover in refuting the possibility of a bounds error on the first access will have to be repeated in order to refute the possibility of a bounds error on the second access.

In the course of the ESC/Modula-3 project, some verification conditions were found to suffer from performance problems of the sort described above. Consequently, the Simplify theorem prover was changed to accept labeled formulas in its input and treat them specially [11,10]. The result is that Simplify's performance on ESC/Modula-3 and ESC/Java verification conditions is almost identical to its performance on the same verification conditions with all instances of (**lblpos** L: P) and (**lblneg** L: P) replaced by P. That is, there is virtually no extra cost for reporting error locations and traces, compared to merely reporting the presence of a potential error. By contrast, the naive treatment (2) has, on average (although not always) a deleterious effect on performance [11].

Simplify also uses labels to control its generation of *multiple counterexample contexts* [11]. Briefly, certain labels are designated as *major* labels. After reporting a counterexample context that includes a major label, Simplify continues to search for additional counterexamples, but ignores portions of the search space that would give rise to counterexample contexts containing any already-reported major label. ESC/Java uses this facility by generating major labels for error locations and minor labels for execution traces. In this way, it can report possible program errors at many locations in a single routine, but will report only one execution trace (rather than a potentially exponential number) leading to each possible error.

The combination of error locations and error traces has turned out to be sufficiently informative that this information is the only information that ESC/Java reports in most of its warning messages. In a few cases (notably for broken object invariants [18]), however, it has been found necessary and useful to extract and report additional information from counterexample contexts.

6. Related work

We know of one other program checker that uses an automatic theorem prover to analyze programs and reports execution traces leading to the possible errors it discovers, namely JACK [2]. JACK's approach is quite different from ours: whereas ESC/Java generates one verification condition per method implementation, JACK generates one verification condition (called a lemma) for each execution path through a method implementation. This makes it trivial to report a particular execution trace for each unproved lemma, but has the considerable drawback of generating a possibly exponential number of lemmas (which in practice ESC/Java usually avoids [9]).

Dynamic program analyzers can offer more than the source location of a manifested error. While it may be too expensive to record a complete history of branches and data values that lead to an error, the additional information of a stack trace at the time the error manifests itself can be useful. Some dynamic checking tools, like Eraser [24], record a collection of stack traces during a program's execution, and display in the tool's error reporting a selection of these stack traces that contribute to the manifestation of the error.

In model checking [3,23], one computes a set of states reachable forward from the initial states or backward from the error states. In this setting, the application of next-state or previous-state relations naturally lends itself to keeping track of how a state is reached. This information can then be used to report error traces (see, *e.g.*, SPIN [13] or SMV [19]). Going beyond static text and tabular output, some tools offer graphical replay of error traces (see, *e.g.*, [14]).

Recent work in model checking has considered the generation of multiple correct and erroneous execution traces, from which the checking tool heuristically arrives at possible *sources* of a manifested error [1,12].

7. Conclusions

We have presented a method for instrumenting verification conditions with information that makes it easy to produce, from a theorem prover's output, execution traces leading to the errors discovered. Our method builds on ideas used to instrument verification conditions with information about the location of errors [10]. We have implemented our method in ESC/Java.

Using the underlying theorem prover's built-in support for *labels*, we have found the overhead of our instrumentation to be negligible. Through experiments, we have found a set of program points that are often useful to include in error traces. We have also found a number of branch points that are commonly "expected", and that are therefore better suppressed in the error reporting to make error messages more focused.

One opportunity to suppress lines in the reported error traces that our implementation conspicuously does not act on is removing trace information that follows immediately from the error location. For example, if an error is manifested in the **then** branch of an **if** statement, then our implementation ends up mentioning the **then** branch in the error trace, despite the fact that this is the only way for an execution to reach the error location.

The error messages produced by ESC/Java draw almost exclusively from the labels reported by the theorem prover. That is, the rest of the counterexample context produced by the theorem prover is mostly ignored, suggesting we've been successful at extracting the most vital information from the theorem prover in a simple way.

On the other hand, for cases where the error traces are not enough to explain why an error may occur, we have not been able to usefully extract further information from the counterexample context, except as alluded to in Section 5. Such further harvesting of information from counterexample contexts would make for interesting future work.

Acknowledgments

The labeling mechanism in Simplify was developed by Dave Detlefs, Greg Nelson, and one of the authors (Saxe) as part of the ESC/Modula-3 project, which used the labels to report precise error locations. The work reported on in this paper was performed by the authors at the Compaq Systems Research Center.

REFERENCES

0. R.-J. Back, J. von Wright, Refinement Calculus: A Systematic Introduction, Graduate Texts in Computer Science, Springer-Verlag, 1998.
1. T. Ball, M. Naik, S. Rajamani, From symptom to cause: Localizing errors in counterexample traces, in: Conference Record of the 30th Annual ACM Symposium on Principles of Programming Languages, volume 38, number 1 in SIGPLAN Notices. ACM (Jan. 2003).
2. L. Burdy, A. Requet, J.-L. Lanet, Java applet correctness: a developer-oriented approach, in: K. Araki, S. Gnesi, D. Mandrioli (Eds.), FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Vol. 2805 of LNCS, Springer, 2003, pp. 422–439.
3. E. M. Clarke, E. A. Emerson, Design and synthesis of synchronization skeletons using branching-time temporal logic, in: D. Kozen (Ed.), Logic of Programs, Workshop, 1981, Vol. 131 of LNCS, Springer, 1982, pp. 52–71.
4. P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, ACM, 1978, pp. 84–96.
5. F. Cristian, Correct and robust programs, IEEE Trans. Softw. Eng. 10 (1984) 163–174.

6. E. W. Dijkstra, A Discipline of Programming, Prentice Hall, Englewood Cliffs, NJ, 1976.
7. C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, M. Felleisen, Catching bugs in the web of program invariants, in: Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI), volume 31, number 5 in SIGPLAN Notices. ACM (May 1996).
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata, Extended static checking for Java, in: Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), volume 37, number 5 in SIGPLAN Notices. ACM (May 2002).
9. C. Flanagan, J. B. Saxe, Avoiding exponential explosion: Generating compact verification conditions, in: Conference Record of the 28th Annual ACM Symposium on Principles of Programming Languages, ACM, 2001, pp. 193–205.
10. D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe, Extended static checking, Research Rep. 159, Compaq SRC (Dec. 1998).
11. D. Detlefs, G. Nelson, J. B. Saxe, Simplify: A theorem prover for program checking, Tech. Rep. HPL-2003-148, HP Labs (Jul. 2003).
12. A. Groce, W. Visser, What went wrong: Explaining counterexamples, in: T. Ball, S. K. Rajamani (Eds.), Model Checking Software, Vol. 2648 of LNCS, Springer, 2003, pp. 121–135.
13. G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5) (1997) 279–295.
14. W. Janssen, R. Mateescu, S. Mauw, P. Fennema, P. van der Stappen, Model checking for managers, in: D. Dams, R. Gerth, S. Leue, M. Massink (Eds.), Theoretical and Practical Aspects of SPIN Model Checking, 5th and 6th International SPIN Workshops, Vol. 1680 of LNCS, Springer, 1999, pp. 92–107.
15. K. R. M. Leino, Toward reliable modular programs, Ph.D. thesis, California Instit. Tech., tech. rep. Caltech-CS-TR-95-03. (1995).
16. K. R. M. Leino, Efficient weakest preconditions, Tech. Rep. MSR-TR-2004-34, Microsoft Research (Apr. 2004).
17. K. R. M. Leino, J. B. Saxe, R. Stata, Checking Java programs via guarded commands, in: B. Jacobs, G. T. Leavens, P. Müller, A. Poetzsch-Heffter (Eds.), Formal Techniques for Java Programs, Tech. Rep. 251, Fernuniversität Hagen, 1999, also available as Tech. Note 1999-002, Compaq SRC.
18. K. R. M. Leino, G. Nelson, J. B. Saxe, ESC/Java user's manual, Technical Note 2000-002, Compaq SRC (Oct. 2000).
19. K. L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, 1993.
20. C. Morgan, The specification statement, ACM Trans. Prog. Lang. Syst. 10 (3) (1988) 403–419.
21. J. M. Morris, A theoretical basis for stepwise refinement and the programming calculus, Sci. Comput. Programming 9 (3) (1987) 287–306.
22. G. Nelson, A generalization of Dijkstra's calculus, ACM Trans. Prog. Lang. Syst. 11 (4) (1989) 517–561.
23. J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems in CESAR, in: M. Dezani-Ciancaglini, U. Montanari (Eds.), International Symposium on

Programming, 5th Colloquium, Vol. 137 of LNCS, Springer, 1982, pp. 337–351.

24. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. E. Anderson, Eraser: A dynamic data race detector for multi-threaded programs, ACM Transactions on Computer Systems 15 (4) (1997) 391–411, also appears in *Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, pages 27–37, Operating System Review 31(5), 1997.