From Symptom to Cause: Localizing Errors in Counterexample Traces

Thomas Ball
Microsoft Research
tball@microsoft.com

Mayur Naik*
Purdue University
mnaik@cs.purdue.edu

Sriram K. Rajamani
Microsoft Research
sriram@microsoft.com

ABSTRACT

There is significant room for improving users' experiences with model checking tools. An error trace produced by a model checker can be lengthy and is indicative of a *symptom* of an error. As a result, users can spend considerable time examining an error trace in order to understand the *cause* of the error. Moreover, even state-of-the-art model checkers provide an experience akin to that provided by parsers before syntactic error recovery was invented: they report a single error trace per run. The user has to fix the error and run the model checker again to find more error traces.

We present an algorithm that exploits the existence of correct traces in order to localize the error cause in an error trace, report a single error trace per error cause, and generate multiple error traces having independent causes. We have implemented this algorithm in the context of SLAM, a software model checker that automatically verifies temporal safety properties of C programs, and report on our experience using it to find and localize errors in device drivers. The algorithm typically narrows the location of a cause down to a few lines, even in traces consisting of hundreds of statements.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*model checking*; D.2.5 [**Software Engineering**]: Testing and Debugging; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

General Terms

Algorithms, Verification

Keywords

software model checking, debugging

1. INTRODUCTION

In recent years, model checking has gained wider use in checking properties of software [13, 7, 3]. Model checking is attractive for two main reasons. First, it does not require the user to provide annotations such as pre-conditions or loop invariants. Second, when a property violation is detected, a witness to the violation is produced in the form of an error trace (a counterexample) at the source level.

Despite these advantages, we believe there is significant room for improving users' experiences with model checkers. An error trace can be very lengthy and only indicates the *symptom* of the error. Users may have to spend considerable time inspecting an error trace to understand the *cause* of the error. Moreover, even state-of-the-art model checkers report a single error trace per run. The user has to fix the error and run the model checker again to find more error traces.

In this paper, we present a technique that localizes the error cause in an error trace, reports a single error trace per error cause, and generates multiple error traces having independent causes. Our technique requires no changes to model checking machinery—it simply uses the model checker as a subroutine.

Model checkers function by exhaustively exploring the reachable state space of a model of a program. Upon detecting a violation of a property, a model checker's internal data structures contain, in addition to the information needed to produce an error trace, information about correct traces — paths in which the property of interest is *not* violated. Our insight is to use correct traces to localize the likely cause of the error in an error trace. In particular, our algorithm identifies the transitions of an error trace that are not in any correct trace of the program. The program statements that induce these transitions are likely to contain the causes of the error. Next, the algorithm introduces **halt** statements in the program at the location of each cause and re-runs the model checker to produce additional error traces.

Example. Consider the example program in Figure 1. We wish to check that the program uses `AcquireLock` and `ReleaseLock` in strict alternation along all paths. We assume that the lock is not held on entering `main` and require that the lock not be held on exiting `main`.

When we input the above program and property to SLAM [3], it produces the error trace $t_1 = [1,\ 2,\ 4,\ 5]$. Our algorithm uses information about correct traces in the program to localize the cause of the error. This program has only one correct trace $t_2 = [1,\ 2,\ 3,\ 5,\ 6,\ 7,\ 9]$. The only portion of t_1 that does not intersect with t_2 is line 4 and our algorithm highlights it as the likely cause. Indeed, the

*This author performed the work reported here during a summer internship at Microsoft Research.

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright 2003 ACM 1-58113-628-5/03/0001 ...$5.00.

```
   main() {
1    AcquireLock();
2    if (...)
3      ReleaseLock();
     else
4      ...;
5    AcquireLock();
6    if (...)
7      ReleaseLock();
     else
8      ...;
9    return;
   }
```

Figure 1: Example program with improper lock usage.

program is missing a call to `ReleaseLock` on line 4.

The algorithm then introduces a **halt** statement at line 4 and invokes the model checker again. The **halt** statement instructs the model checker to stop exploring paths through the statement at line 4. As a result, the model checker reports a different error trace t_3 = [1, 2, 3, 5, 6, 8, 9]. Again, by comparing t_3 with the correct trace t_2, line 8 is highlighted as the potential cause of the error, another **halt** statement is introduced at line 8, and the model checker is invoked for the third time. At this point, it reports that there are no more error traces.

In summary, our algorithm automatically produces two error traces, namely, t_1 = [1, 2, 4, 5] with line 4 as the identified cause, and t_3 = [1, 2, 3, 5, 6, 8, 9] with line 8 as the identified cause. □

We present the following results in this paper:

- A technique for using a model checker as a subroutine to localize the error cause in an error trace, report one error trace per error cause, and generate multiple error traces having independent causes; the technique exploits the existence of "correct" transitions (transitions along correct traces) in the state space computed by the model checker.
- Efficient algorithms for computing the complete set of correct transitions in the intraprocedural and interprocedural cases; the time complexity of the algorithms is linear in the size of the state space computed by the model checker.
- Experimental results in the context of the SLAM toolkit showing that these algorithms are effective at localizing the causes of errors in real-world programs.

Outline. The remainder of the paper is organized as follows. Section 2 presents background material. Section 3 describes our general framework for localizing causes in error traces and generating multiple traces using a generic model checker. Section 4 presents the algorithm to identify correct transitions for the intraprocedural case and Section 5 extends it to the interprocedural case. Section 6 demonstrates the performance of our technique in the context of analyzing safety properties of Windows device drivers. Section 7 reviews related work, and Section 8 concludes with a note on future work.

2. PRELIMINARIES

2.1 Control Flow Graphs

We represent programs abstractly via control flow graphs, following [18].

Each procedure p_i in program $P = \{p_1, \cdots, p_n\}$ is represented by a directed graph $G_i = (V_i, E_i, e_i, x_i)$ with vertices V_i, edges E_i, entry vertex e_i and exit vertex x_i. Vertices in V_i corresponding to procedures calls are denoted by $\mathit{Call}_i \subseteq V_i$. Each vertex v_c in Call_i is paired with a unique return-site vertex v_r in $\mathit{RetPt}_i \subseteq V_i$. Given a vertex v_r in RetPt_i, we denote the corresponding call vertex by $\mathit{toCall}(v_r)$.

Program P is represented by a control flow supergraph $G^* = (V^*, E^*, e_{main})$ that is the union of the (vertex- and edge-disjoint) control flow graphs of procedures, with additional edges representing the flow of control via procedure calls and returns. Specifically,

- $V^* = \bigcup_{1 \le i \le n} V_i$, and
- $E^* = E^0 \cup E^1$

where $E^0 = \bigcup_{1 \le i \le n} E_i$ and E^1 represents the flow of control from caller to callee and vice versa. Edge $(v, w) \in E^1$ iff there is a procedure p_i such that either:

(i) $v \in \mathit{Call}_i$, the procedure called at v is p_j, and $w = e_j$, the entry vertex of G_j, or

(ii) $w \in \mathit{RetPt}_i$, the procedure called at $\mathit{toCall}(w)$ is p_j, and $v = x_j$, the exit vertex of G_j.

Finally, we define the following subsets of V^*:

- $\mathit{Entry} = \{ e_i \mid 1 \le i \le n \}$,
- $\mathit{Exit} = \{ x_i \mid 1 \le i \le n \}$,
- $\mathit{Call} = \bigcup_{1 \le i \le n} \mathit{Call}_i$, and
- $\mathit{RetPt} = \bigcup_{1 \le i \le n} \mathit{RetPt}_i$.

2.2 Transitions and Traces

We assume that programs have global variables and procedures have local variables and formal parameters. A *state* of the program at a vertex is a valuation to the variables in scope at the vertex (before the execution of the statement associated with the vertex). Let Θ denote the set of all states.

Each vertex $v \in V^*$ has an associated statement and a transfer function $\delta(v)$ that maps a state to a set of states. Intuitively, if the statement at v is executed in state Ω then the resultant set of states is $\delta(v)(\Omega)$.

A *transition* of G^* is a pair $\langle (v_1, \Omega_1), (v_2, \Omega_2) \rangle$ such that $(v_1, v_2) \in E^*$ and $\Omega_2 \in \delta(v_1)(\Omega_1)$. A *projection* of a transition, **project** $(\langle (v_1, \Omega_1), (v_2, \Omega_2) \rangle)$, is the edge (v_1, v_2). Projections can be generalized to sets of transitions in the usual way. We write $(v_1, \Omega_1) \leadsto (v_2, \Omega_2)$ if $\langle (v_1, \Omega_1), (v_2, \Omega_2) \rangle$ is a transition.

A sequence $(v_1, \Omega_1), (v_2, \Omega_2), \ldots, (v_k, \Omega_k)$ is a *trace* of G^* if (1) $v_1 = e_{main}$, (2) $(v_i, \Omega_i) \leadsto (v_{i+1}, \Omega_{i+1})$ for $0 < i < k$, and (3) the return vertices in the sequence are properly matched with call vertices. (A formal definition of the third condition requires associating a distinguished open and closed parenthesis for each call and return, defining a context-free grammar for the language of balanced parentheses [18, 2].)

```
procedure Localize($G^*$, $\delta$, $v$, $f$)
begin
  while true do
    switch ModelCheck($G^*$, $\delta$, $v$, $f$) of
    case SUCCESS:
      output “success”; break
    case FAILURE($T$):
      let $C$ = GetCorrectTransitions($G^*$, $\delta$, $v$, $f$) and
      let $Causes$ = project($T$) $\setminus$ project($C$) in
        output $T$ as error trace with causes $Causes$
        if $Causes = \emptyset$ then
          break
        for each $(v_i, v_j) \in Causes$ do
          let $v_k$ = halt in
            $V^* := V^* \cup \{ v_k \}$
            $E^* := E^* - \{ (v_i, v_j) \} \cup \{ (v_i, v_k), (v_k, v_j) \}$
end
```

Figure 2: Error cause localization algorithm.

3. ERROR CAUSE LOCALIZATION ALGORITHM

We assume we are given a model checking function **ModelCheck** that takes as input:

- a control flow supergraph G^*,
- a transfer function $\delta : V^* \to \Theta \to 2^{\Theta}$,
- a specified vertex $v \in V^*$, and
- a “correctness” function $f : \Theta \to \mathbf{bool}$.

This function can be used to check safety properties, since safety checking can be transformed by a suitable product construction to reachability. The **ModelCheck** function determines if there is a trace T ending with (v, Ω) such that $f(\Omega) =$ false. If there is no such trace, then it returns SUCCESS. Otherwise, it returns FAILURE(T).

We assume that, as a side-effect, **ModelCheck** annotates each vertex v with States(v), the set of reachable states of v (*i.e.*, $\Omega \in$ States(v) iff there is a trace that ends with (v, Ω)).[1]

Of course, the existence of such a model checking function presumes that the set of reachable states is computable in finite time. If not, one must first construct a suitable abstraction of the program, such as a boolean program [2]. We will discuss the impact of abstraction on our algorithm in Section 6.

Figure 2 presents the high-level structure of our algorithm in the procedure **Localize**. If **ModelCheck** returns FAILURE(T), then the algorithm invokes the function **GetCorrectTransitions** which uses the results of **ModelCheck** to find the transitions that belong to correct traces (with respect to vertex v and correctness function f). Formally, transition $t \in$ **GetCorrectTransitions**(G^*,δ,v,f) iff there is a trace T' containing t such that T' ends with (v, Ω) and $f(\Omega) =$ true. As we shall see later, the complexity of **GetCorrectTransitions** is linear in the size of the control flow graph G^* and the number of states (that is, it is of the same complexity as **ModelCheck**).

```
function GetCorrectTransitions($G^*$, $\delta$, $v$, $f$)
begin
  var worklist: set of $(V^*, \Theta)$ :=
    $\{(v, \Omega) \mid \Omega \in \text{States}(v) \wedge f(\Omega) = \mathbf{true}\}$
  var visited: set of $(V^*, \Theta) := \emptyset$
  var $M$: set of transition $:= \emptyset$
  while worklist $\neq \emptyset$ do
    remove $(v_j, \Omega_j)$ from worklist
    if $(v_j, \Omega_j) \notin$ visited then
      visited := visited $\cup \{(v_j, \Omega_j)\}$
      for each $(v_i, v_j) \in E^*$ do
        for each $\Omega_i \in \text{States}(v_i)$ do
          if $\Omega_j \in \delta(v_i)(\Omega_i)$ then
            worklist := worklist $\cup \{(v_i, \Omega_i)\}$
            $M := M \cup \{\langle (v_i, \Omega_i), (v_j, \Omega_j) \rangle\}$
  return $M$
end
```

Figure 3: Intraprocedural algorithm for computing correct transitions.

Next, the cause of the error is computed as $Causes =$ **project**(T) $\setminus$ **project**(C).[2] That is, the cause lies along edges that belong to the error trace T but do not belong to any correct trace. If $Causes$ is non-empty, the algorithm introduces a **halt** statement along each edge in $Causes$, and repeat the entire procedure. Since G^* is finite and each iteration introduces at least one **halt** statement, the algorithm is guaranteed to terminate.

4. COMPUTING INTRAPROCEDURAL CORRECT TRANSITIONS

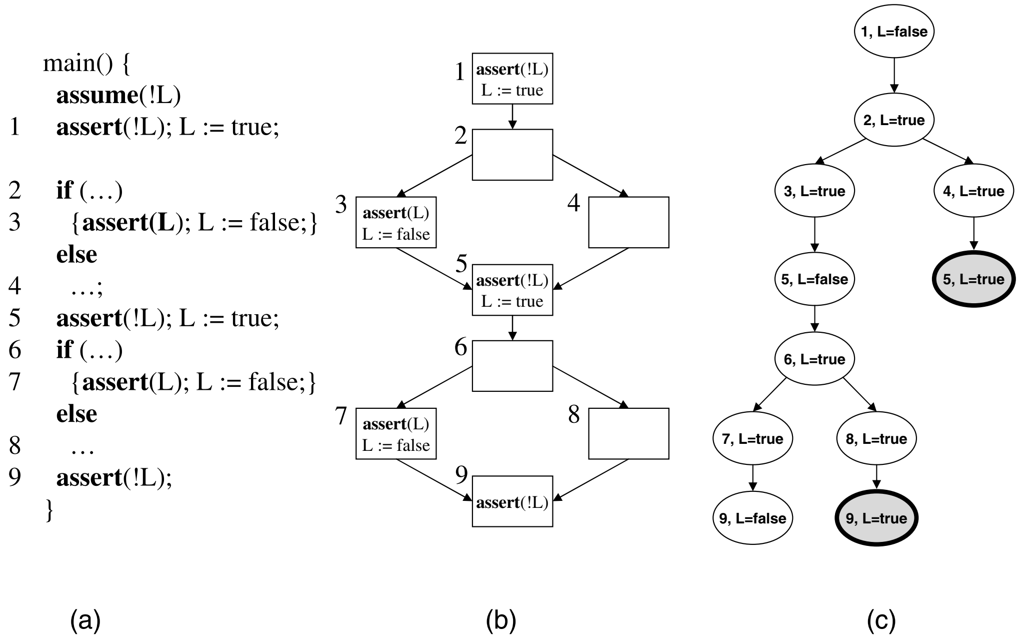
Figure 3 presents the algorithm **GetCorrectTransitions** for the intraprocedural case (a single-procedure program without calls). The simple idea behind our algorithm is to work backwards in the state space computed by **ModelCheck** from the states at vertex v that satisfy the correctness function f (see the initialization of *worklist*). While the worklist is not empty, a pair (v_j, Ω_j) is removed. If this pair has not been visited before, each pair (v_i, Ω_i) such that $(v_i, v_j) \in E^*$ and $\Omega_j \in \delta(v_i)(\Omega_i)$ is added to the worklist. The transition $(v_i, \Omega_i) \rightsquigarrow (v_j, \Omega_j)$ is a correct transition and is added to set M, which is returned from the function.

We now revisit the example of Figure 1 to illustrate the operation of **GetCorrectTransitions**.

Figure 4(a) contains the example rewritten to replace each call to `AcquireLock` and `ReleaseLock` with statements to check the property that locks are alternately acquired and released. The variable L is true at a vertex iff the last action to execute was a lock acquisition. Figure 4(b) shows the control flow graph of this program and Figure 4(c) shows its reachable state-space graph (as computed by **ModelCheck**). Let us focus on vertex 5 in the control flow graph and determine whether or not the **assert** statement at that vertex can fail (*i.e.*, the correctness function f returns true iff the expression $!L$ is true). Given this query,

[1] For the interprocedural case, the **ModelCheck** function will have to compute more auxiliary information, as discussed in Section 5.

[2] In Section 6, we will discuss a variation of the algorithm that computes $Causes$ as **project**($T \setminus C$), and its effect on error cause localization.



```
    main() {
      assume(!L)
1     assert(!L); L := true;

2     if (...)
3       {assert(L); L := false;}
      else
4       ...;
5     assert(!L); L := true;
6     if (...)
7       {assert(L); L := false;}
      else
8       ...
9     assert(!L);
    }
```

Figure 4: (a) The program of Figure 1 rewritten with assertions. (b) The control flow graph of this program. (c) The reachable state-space graph of this program. The grey bold nodes represent error states (states in which an assertion will fail).

ModelCheck produces the error trace $T = (1, L = \text{false}) \rightsquigarrow (2, L = \text{true}) \rightsquigarrow (4, L = \text{true}) \rightsquigarrow (5, L = \text{true})$, which shows how the assertion can fail.

Note that, in the reachable state-space graph, it is also possible to reach vertex 5 in a state in which the assertion does not fail. So, **GetCorrectTransitions** initializes its worklist to the pair $(5, L = \text{false})$ and proceeds backwards in the state space of the program, generating as its result the set of correct transitions

$$\begin{array}{c}(1, L = \text{false}) \rightsquigarrow (2, L = \text{true})\\ (2, L = \text{true}) \rightsquigarrow (3, L = \text{true})\\ (3, L = \text{true}) \rightsquigarrow (5, L = \text{false})\end{array}$$

Since the projection of the set of correct transitions is $\{(1, 2), (2, 3), (3, 5)\}$ and the projection of the set of transitions in error trace T is $\{(1, 2), (2, 4), (4, 5)\}$, the value of *Causes* in the **Localize** procedure of Figure 2 is $\{(2, 4), (4, 5)\}$, which localizes the cause of the error to line 4 of the program.

Since there are only three feasible paths in this program (see the program's state space in Figure 4(c)), both error traces could have been generated by a naive approach in which the model checker generates all error traces. However, this approach will report multiple error traces having the *same* cause. Consider the program from Figure 1 with the extra conditional statement "**if** (...) A **else** B" inserted before the first `AcquireLock` call. Neither branch of the conditional alters the state of the lock. There are six feasible paths in this program: two correct traces and four error traces. The naive approach would report all four error traces but our algorithm still reports two error traces, one per cause. In practice, many branches in real-world programs are irrelevant to the property being checked. If we do not identify error traces by causes, the model checker will report huge numbers of error traces for the same cause.

5. COMPUTING INTERPROCEDURAL CORRECT TRANSITIONS

Precise model checking and computation of correct transitions in the interprocedural case must take into account the calling context of each called procedure. In particular, the algorithms should only analyze paths in which each call and return are properly matched.

5.1 Additional Model Checker Assumptions

The **ModelCheck** function must record at each vertex $v \in V^*$ more detailed information than States(v). In particular, we assume it stores a set of *path edges* at each vertex and a set of *summary edges* at each call vertex [18, 2].

Intuitively, a path edge incident on vertex $v \in V^*$ in procedure p_g is a pair of states (Ω_g, Ω) such that there is a trace from the entry vertex of `main` to the entry vertex of p_g at state Ω_g and there is a continuation of the trace that reaches (v, Ω) without exiting p_g. Formally, a *path edge* incident on vertex $v \in V^*$ in procedure p_g is a pair of states (Ω_g, Ω) such that there is a trace $T = T_0.T_1$ where $T_0 = (v_0, \Omega_1) \rightsquigarrow ... \rightsquigarrow (v_i, \Omega_i)$, and $T_1 = (v_i, \Omega_i) \rightsquigarrow ... \rightsquigarrow (v_j, \Omega_j)$, and $v_0 = e_{main}$, and $(v_i, \Omega_i) = (e_g, \Omega_g)$ where e_g is the entry vertex of the procedure p_g, and $(v_j, \Omega_j) = (v, \Omega)$, and for all $i \le k < j$, $v_k \neq x_g$ where x_g is the exit vertex of p_g. Let PE(v) denote the set of path edges incident on v.

Let v_c be a call to a procedure p_g. The set of summary edges at v_c represents the effect of p_g as a transfer function. Formally, the set of *summary edges* at v_c is

$$\begin{array}{ll}\text{Summary}(v_c) = \{ & (\Omega_1, \Omega_2) \mid \exists\ (\Omega_g, \Omega_i) \in \text{PE}(x_g) \text{ and}\\ & \Omega_g \in \delta(v_c)(\Omega_1) \text{ and } \Omega_2 \in \delta(x_g)(\Omega_i)\ \}\end{array}$$

For each $v_c \in \mathit{Call}$, we define $\text{SE}(v_c)(\Omega) = \{\ \Omega' \mid (\Omega, \Omega') \in \text{Summary}(v_c)\ \}$.

```
function GetCorrectTransitions($G^*$, $\delta$, $v$, $f$): set of $transition$
begin
    var $M$: set of $transition$ := $\emptyset$
    var $visited$: set of $(V^*, \Theta, \Theta)$ := $\emptyset$
    var $WL_1$, $WL_2$: set of $(V^*, \Theta, \Theta)$

    $WL_1 := \{(v, \Omega_g, \Omega) \mid (\Omega_g, \Omega) \in \mathrm{PE}(v) \wedge f(\Omega) = \mathrm{true}\}$
    $WL_2 := \emptyset$

    // Phase 1: ascend to callers via call edges, don't descend to callee via return edges
    while $WL_1 \neq \emptyset$ do
        remove $(v_j, \Omega_g, \Omega_j)$ from $WL_1$
        if $(v_j, \Omega_g, \Omega_j) \notin visited$ then
            $visited := visited \cup \{(v_j, \Omega_g, \Omega_j)\}$
            if $v_j \in RetPt$ then
                let $(T, W_c, W_x)$ = PropagateToCallAndExit$(v_j, \Omega_g, \Omega_j)$ in
                    $M := M \cup T$ ; $WL_1 := WL_1 \cup W_c$; $WL_2 := WL_2 \cup W_x$
            else
                let $(T, W)$ = Propagate$(v_j, \Omega_g, \Omega_j)$ in
                    $M := M \cup T$; $WL_1 := WL_1 \cup W$

    // Phase 2: descend to callee via return edges, don't ascend to callers via call edges
    while $WL_2 \neq \emptyset$ do
        remove $(v_j, \Omega_g, \Omega_j)$ from $WL_2$
        if $(v_j, \Omega_g, \Omega_j) \notin visited$ then
            $visited := visited \cup \{(v_j, \Omega_g, \Omega_j)\}$
            if $v_j \in RetPt$ then
                let $(T, W_c, W_x)$ =PropagateToCallAndExit$(v_j, \Omega_g, \Omega_j)$ in
                    $M := M \cup T$ ; $WL_2 := WL_2 \cup W_c \cup W_x$
            else if $v_j \notin Entry$ then
                let $(T, W)$ = Propagate$(v_j, \Omega_g, \Omega_j)$ in
                    $M := M \cup T$ ; $WL_2 := WL_2 \cup W$
    return $M$
end
```

Figure 5: Interprocedural algorithm for computing correct transitions.

5.2 Interprocedural Algorithm

Figures 5 and 6 presents the algorithm **GetCorrectTransitions** for the interprocedural case. The basic idea is the same as that for the intraprocedural case: to work backwards in the state space computed by **ModelCheck** from the states at vertex v that satisfy the correctness function f. However, the algorithm must be context-sensitive, *i.e.*, it must work backwards only along paths in which each call and return are properly matched.

Our algorithm is similar to the interprocedural slicing algorithm of Horowitz, Reps, and Binkley [14], having two phases: **Phase1** can "ascend" from the entry vertex of a procedure p_g to the vertices that call p_g, but it cannot "descend" from a return point in *RetPt* to the corresponding callees' procedure exit; **Phase2**, on the other hand, can descend to procedure exits but cannot ascend to call vertices. (Both phases use the summary edges to move across a call, while properly accounting for its effect). The phases operate on worklists WL_1 and WL_2 respectively. Each worklist consists of triples $(v_j, \Omega_g, \Omega_j)$ where $v_j \in V^*$ and $(\Omega_g, \Omega_j) \in \mathrm{PE}(v_j)$. That is, the worklist consists of path edges incident on vertices that are awaiting processing.

While WL_1 is not empty, the first phase removes a triple $(v_j, \Omega_g, \Omega_j)$ and processes it as follows:

- If $v_j \in RetPt$, the **PropagateToCallAndExit** function (see Figure 6) is called. This function uses the summary edges for the corresponding call vertex $toCall(v_j)$ to propagate across the call, resulting in new correct transitions in T and new triples W_c at the call vertex. The function also computes new triples W_x at the exit of the procedure called at $toCall(v_j)$. The sets T and W_c are accumulated in M and WL_1, respectively, while W_x is accumulated in WL_2 to await processing in **Phase 2** (recall that the first phase does not descend to analyze callee procedures).

- If $v_j \notin RetPt$, the **Propagate** function is called, which simply pushes triples backwards through all predecessors of v_j, identifying new correct transitions and new triples. That is, for each predecessor v_i of v_j, if $(\Omega_g, \Omega_i) \in \mathrm{PE}(v_i)$ and $\Omega_j \in \delta(v_i)(\Omega_i)$, then the transition $(v_i, \Omega_i) \rightsquigarrow (v_j, \Omega_j)$ is added to T and a new triple $(v_i, \Omega_g, \Omega_i)$ is added to W. T and W are accumulated in M and WL_1, respectively.

While WL_2 is not empty, the second phase removes a triple $(v_j, \Omega_g, \Omega_j)$ and processes it as follows:

- If $v_j \in Entry$, then nothing is done (recall that the second phase does not ascend to callers).

```
function Propagate($v_j$: vertex, $\Omega_g$: $\Theta$, $\Omega_j$: $\Theta$): (set of $transition$, set of $(V^*, \Theta, \Theta)$)
begin
    var $T$: set of $transition$ := $\emptyset$
    var $W$: set of $(V^*, \Theta, \Theta)$ := $\emptyset$
    for each $(v_i, v_j) \in E^*$ do
        for each $(\Omega'_g, \Omega_i) \in \mathrm{PE}(v_i)$ do
            if $(\Omega_g = \Omega'_g)$ and $(\Omega_j \in \delta(v_i)(\Omega_i))$ then
                $T := T \cup \{\langle(v_i, \Omega_i), (v_j, \Omega_j)\rangle\}$; $W := W \cup \{(v_i, \Omega_g, \Omega_i)\}$
    return $(T, W)$
end

function PropagateToCallAndExit($v_j$: vertex, $\Omega_g$: $\Theta$, $\Omega_j$: $\Theta$):
    (set of $transition$, set of $(V^*, \Theta, \Theta)$, set of $(V^*, \Theta, \Theta)$)
begin
    var $T$ : set of $transition$ := $\emptyset$
    var $W_c$: set of $(V^*, \Theta, \Theta)$ := $\emptyset$
    var $W_x$: set of $(V^*, \Theta, \Theta)$ := $\emptyset$
    let $v_c = toCall(v_j)$ and $x_h$ = exit vertex of procedure $p_h$ called at $v_c$ in
        for each $(\Omega'_g, \Omega_c) \in \mathrm{PE}(v_c)$ do
            if $(\Omega_g = \Omega'_g)$ and $(\Omega_j \in \mathrm{SE}(v_c)(\Omega_c))$ then
                $T := T \cup \{\langle(v_c, \Omega_c), (v_j, \Omega_j)\rangle\}$; $W_c := W_c \cup \{(v_c, \Omega_g, \Omega_c)\}$
                for each $(\Omega_h, \Omega_i) \in \mathrm{PE}(x_h)$ do
                    if $\Omega_h \in \delta(v_c)(\Omega_c)$ and $\Omega_j \in \delta(x_h)(\Omega_i)$ then
                        $W_x := W_x \cup \{(x_h, \Omega_h, \Omega_i)\}$
    return $(T, W_c, W_x)$
end
```

Figure 6: Propagation functions used by interprocedural algorithm for computing correct transitions.

- If $v_j \in RetPt$, the **PropagateToCallAndExit** function is called. T, W_c, and W_x are computed as before. T is accumulated in M while W_c and W_x are accumulated in WL_2.
- Otherwise, the **Propagate** function is called. T and W are computed as before and accumulated in M and WL_2, respectively.

6. EXPERIMENTAL ASSESSMENT

We have implemented our algorithm in the SLAM toolkit [3], which checks temporal safety properties of sequential C programs. We checked 30 Windows device drivers for 2 properties: (i) `SpinLock` which expresses that locks should be acquired and released in strict alternation; it consists of 2 assertions and 2 states, and (ii) `IrpCompletion` which specifies how I/O request packets should be processed; it consists of 18 assertions and more than 50 states.

Figure 7 presents the results of our experiments performed on a 2.2 GHz Pentium PC with 1.5 GB RAM. In all, 15 error traces were reported for 8 of the drivers. Error trace (15) was a violation of `SpinLock`; the rest were violations of `IrpCompletion`. The cause was localized precisely in 11 of the error traces. All error traces had single causes; we did not find any error trace with multiple causes. The cause was not localized in each of error traces (1), (4), and (11) because of *coincidental correctness*: a situation in which every control-flow edge in the error trace is contained in a correct trace. In this case, the variable *Causes* in Figure 2 is the empty set. We illustrate this problem by means of two examples drawn from our experiments, and then show how computing *Causes* as **project**$(T \setminus C)$ instead of **project**$(T) \setminus$ **project**(C) solves this problem.

Consider the program in Figure 8. We wish to check if it returns the same value as the function `foo`, which is called exactly once along every path. The program has one error trace t_1 = [1, 2, 3, 7, 9, 4, 6]. The error cause lies on lines 3 and 4: the value returned by `foo` is ignored at line 3 and the value SUCCESS is unconditionally assigned to the variable `status` at line 4. The algorithm fails to localize this cause since it belongs to a portion of t_1 that intersects with correct trace t_2 = [1, 2, 3, 7, 8, 4, 6].[3] Note, however, that the transition

(4, g=FAILURE) $\rightsquigarrow$ (6, g=FAILURE, status=SUCCESS)

in the error trace is not in any correct trace. In particular, in the correct trace t_2 the transition corresponding to the control edge $(4, 6)$ is

(4, g=SUCCESS) $\rightsquigarrow$ (6, g=SUCCESS, status=SUCCESS).

in which both variabes g and *status* have the value SUCCESS at line 6.

Consider the program in Figure 9. We wish to check whether the function `bar` is called at most once along every path. The program has one error trace t_1 = [1, 2, 3, 9, 10, 4, 5, 6, 7, 8] in which `bar` is called twice. The error cause is on line 3: `baz` is called assuming that the return value of `foo` is always SUCCESS. The algorithm fails to localize this cause since it belongs to a portion of t_1 that intersects with correct trace t_2 = [1, 2, 3, 9, 4, 5, 6, 7, 8].

For each of the above cases, localization can be successfully achieved by computing *Causes* as (1) **project**$(T \setminus C)$

[3] There are also two other correct traces that cover the control edges of t_1 not covered by t_2.

name of device driver	LOC	error trace	number of transitions in:		cumulative running time (sec.) for:	
		ID	error cause	error trace	model checking	cause localization
mouse packet filter	984	1	0	73	44.6	7.8
		2	4	110		
serial mouse port driver	7441	3	1	56	62.0	185.9
keyboard packet filter	1067	4	0	73	44.1	7.8
		5	4	107		
IEEE 1394 bus driver	5818	6	7	45	205.8	114.1
		7	7	44		
		8	1	60		
		9	3	81		
		10	0	85		
keyboard class driver	13161	11	0	158	365.4	97.8
i8042 port driver	22168	12	1	127	41.6	10.6
		13	5	124		
packet-based DMA driver	24971	14	1	75	40.0	5.4
serial port driver	30905	15	3	248	19.9	6.4

Figure 7: Results of analyzing Windows device drivers using SLAM. There is one row per error trace discovered. Column two is the number of lines of code in the driver. Column five is the number of transitions in the error trace, while column four is the number of transitions in the cause, as identified by our algorithm. For each identified cause, we verified manually that the transitions reported in column four were indeed the real cause of the error. Columns six and seven report, for each device driver, the total running time of the model checker and the error cause localization algorithm respectively over all error traces reported for that driver.

```
1 int g;
2 if (...) {
3    foo();
4    status := SUCCESS;  // error: return value
  } else                 // of foo is ignored
5    status := foo();
6 assert(status = g);
  return status;

  int foo() {
7    if (...) {
8       g := SUCCESS; return g;
     } else {
9       g := FAILURE; return g;
     }
  }
```

Figure 8: Example of variable-value error.

```
1 int cnt := 0;
2 status := foo();
3 baz();        // error: baz should have been called
4 ...           // only if status = SUCCESS
5 if (status = FAILURE)
6    bar();
7 ...
8 assert(cnt ≤ 1);

  void baz() {
9    if (...)
10      bar();
  }

  void bar() { cnt++; }
```

Figure 9: Example of control flow error. Function `foo` is defined in Figure 8.

instead of (2) $\textbf{project}(T) \setminus \textbf{project}(C)$ in Figure 2. In general, approach (1) is more precise because it takes into account differences in the state components before projecting the state away, while approach (2) projects away the state components and then computes the difference.

However, a drawback of this approach is that it localizes the error cause to an entire suffix of the error trace whose first element is the actual cause. For instance, in each of the above programs, it localizes the error cause to the suffix of the error trace t_1 beginning at line 3. It would be interesting to explore ways of combining the two approaches to precisely localize error causes.

The cause was not localized in error trace (10) because the current implementation of our algorithm in the Slam toolkit computes correct transitions using abstract states as opposed to concrete states, namely, it computes correct transitions in a boolean abstraction constructed from the C program [1]. As a result, some of these transitions might be infeasible in the C program and thereby misguide cause localization. For instance, for the program in Figure 10, Slam abstracts away the predicate on line 4 in the boolean program. As a result, our algorithm finds the correct trace $t_1 = [1,\ 2,\ 8,\ 10,\ 11,\ 3,\ 4,\ 5,\ 6,\ 7]$ which, while is infeasible in the C program, is feasible in the boolean program. The model-checking phase of Slam detects the error trace $t_2 = [1,\ 2,\ 8,\ 10,\ 11,\ 3,\ 4,\ 6,\ 7]$ in the boolean program, which is confirmed to be a feasible trace in the C program. The error cause lies on line 11. However, our algorithm fails to localize it since line 11 is in correct trace t_1. Such imprecision can be addressed by refining the boolean abstraction

```
1  int cnt := 0;
2  status := taz();
3  ...
4  if (status = SUCCESS)
5     bar();
6  ...
7  assert(cnt = 1);

   void taz() {
8     if (...) {
9        return SUCCESS;
      else {
10       ... // error: bar is not called
11       return FAILURE;
      }
   }
   void bar() { cnt++; }
```

Figure 10: Example of model imprecision.

to include more predicates to rule out the infeasible correct paths. In the above example, this would require adding the predicate (`status` = SUCCESS) to the boolean abstraction. Automating this idea is a topic for future work.

7. RELATED WORK

7.1 Multiple Counterexamples

There is little work on generating multiple counterexamples during model checking. The most relevant work is embodied in the Verisim testing tool for network protocols [5]. Verisim consists of a network simulator that generates traces simulating a network and a trace checker that determines whether a trace satisfies an extended LTL formula ϕ. If a violation is detected, a technique called *tuning* is used to replace the formula ϕ by a formula ψ that ignores the violation. Tuning is not fully automatic and does not attempt to localize the cause of the violation.

In the context of detecting security violations in network models, techniques have been proposed to generate a set of counterexamples from a model checker as a graph [21]. Our work differs from this effort in that we localize causes in counterexamples and generate one counterexample per cause.

7.2 Error Cause Localization

There are several techniques for error cause localization that are complementary to the approach presented in this paper.

The work most closely related to ours is that of Groce and Visser [10]. Given error trace T, their technique computes a set of *negatives*, error traces (including T) that lead to the same assertion violation as T, and a set of *positives*, correct traces that lead to the assertion but do not violate it. Analysis of the common features and the differences between positives and negatives provides succinct and useful information about the error trace to the user.

Jin, Ravi, and Somenzi [15] present a game-theoretic technique that partitions an error trace into *fated* segments, controlled by the environment attempting to force the system into an error, and *free* segments, controlled by the system attempting to avoid the error. Fated segments manifest unavoidable progress towards the error while free segments contain choices that, if avoided, might have prevented the error.

There is a large body of work on locating the sources of type errors in implicity typed, higher-order languages with let-polymorphism like Haskell, Miranda, O'Caml, and Standard ML [23, 16, 4, 8, 22, 6]. These techniques usually employ the underlying type inference algorithm to identify the source of a type error as a program point or the program subtree rooted at that point or, more recently, as a set of program points (a *slice*) [22].

Program slicing [24] (especially dynamic slicing [17]) can be used to find the set of statements in an error trace that may be relevant to the cause of the error. Program slicing uses data and control dependences to slice away statements that do not directly affect a statement of interest. However, such slicing can be misleading when an error is one of omission. For example, in Figure 4(a), program slicing with respect to the **assert** statements will slice away the portion of the code in which the omission may lie.

Algorithmic debugging [20] involves isolating an erroneous procedure by starting from an external point of failure (*e.g.*, an incorrect output value) and asking the user a series of questions related to the behavior of procedures in the program (*e.g.*, "should `concat`($[a\,b]$, $[c\,d\,e]$) return $[a\,b\,d\,e]$"?). Algorithmic debugging is interactive and it does not localize the cause of the error within a procedure it declares erroneous.

Delta debugging is a combinatorial testing algorithm that narrows the difference in the program state between a passing run and a failing run to isolate the statements constituting the cause of the error [25]. Given a correct program P and a set of changes $C = \{c_1, \cdots, c_n\}$ to P that yield an incorrect program P', delta debugging applies subsets of C to P and runs the resulting programs in order to determine whether the same error is manifested as in P'. In this manner, delta debugging uses non-erroneous runs to determine the set of changes responsible for the error.

7.3 Anomaly Detection

Static analysis techniques like Meta-Level Compilation [11] and dynamic analysis tools like Daikon [9], DIDUCE [12], and Eraser [19] all use the idea that consistent behavior in a program is correct behavior while inconsistent behavior is a likely error. For example, if `AcquireLock` and `ReleaseLock` are called in strict alternation 99 out of 100 times, then the single anomalous behavior is flagged as a possible error. Our algorithm can be viewed as a static technique that exploits consistent behavior to determine the cause of a given error as opposed to the error itself.

8. CONCLUSIONS

Techniques such as model checking and dataflow analysis have the capability to find subtle errors in programs. Nonetheless, the problem of finding the cause of an error is relegated to the user. We have shown how to localize the cause in error traces generated by model checkers. The key idea is to find transitions in the error trace that appear in no correct trace. Our techniques are quite general and should be applicable to error detection tools based on dataflow analysis as well.

A number of interesting research questions remain open.

First, is it possible (in some cases) to suggest a fix to an erroneous program? For example, in the case of the program in Figure 1, it would be fairly straightforward to enhance our algorithm to output a suggested fix of introducing a call to `ReleaseLock` at each of lines 4 and 8.

Second, what other kind of information can be used to help localize the cause of an error? Program slicing and algorithmic debugging provide information in the form of dynamic data dependences and user input. Dynamic data dependences from an error trace track the flow of values between statements and could be very helpful in tracing back from an assertion failure to the variable definitions that caused it. Likewise, user input about which functions in a program can be "trusted" (*e.g.*, library functions) could be used to guide the search for a cause.

9. REFERENCES

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. Automatic predicate abstraction of C programs. In *PLDI 01: Programming Language Design and Implementation*, pages 203–213. ACM, 2001.

[2] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for Boolean programs. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 113–130. Springer-Verlag, 2000.

[3] T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *POPL 02: Principles of Programming Languages*, pages 1–3. ACM, 2002.

[4] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2(1-4):17–30, 1993.

[5] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, Feb. 2002.

[6] O. Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *ICFP 01: International Conference on Functional Programming*, pages 193–204. ACM, 2001.

[7] J. Corbett, M. Dwyer, J. Hatcliff, C. Pasareanu, Robby, S. Laubach, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000: International Conference on Software Engineering*, pages 439–448. ACM, 2000.

[8] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[9] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions in Software Engineering*, 27(2):1–25, February 2001.

[10] A. Groce and W. Visser. What went wrong: Explaining counterexamples. Technical Report 02-08, RIACS, USRA, 2002.

[11] S. Hallem, B. Chelf, Y. Xie, and D. Engler. A system and language for building system-specific, static analyses. In *PLDI 02: Programming Language Design and Implementation*, pages 69–82. ACM, 2002.

[12] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE 2000: International Conference on Software Engineering*, pages 291–301. ACM, 2002.

[13] G. Holzmann. Logic verification of ANSI-C code with Spin. In *SPIN 00: SPIN Workshop*, LNCS 1885, pages 131–147. Springer-Verlag, 2000.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, Jan. 1990.

[15] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *TACAS 02: Tools and Algorithms for Construction and Analysis of Systems*, LNCS 2031, pages 445–459. Springer-Verlag, 2002.

[16] G. F. Johnson and J. A. Walz. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL 86: Principles of Programming Languages*, pages 44–57. ACM, 1986.

[17] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*, 29(10):155–163, October 1988.

[18] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL 95: Principles of Programming Languages*, pages 49–61. ACM, 1995.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, Nov. 1997.

[20] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1982. ACM Distinguished Dissertation.

[21] O. Sheyner, S. Jha, and J. M. Wing. Automated generation and analysis of attack graphs. In *IEEE Symposium on Security and Privacy*, pages 273–284. IEEE, 2002.

[22] F. Tip and T. B. Dinesh. A slicing-based approach for locating type errors. *ACM Transactions on Software Engineering and Methodology*, 10(1):5–55, Jan. 2001.

[23] M. Wand. Finding the source of type errors. In *POPL 86: Principles of Programming Languages*, pages 38–43. ACM, 1986.

[24] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[25] A. Zeller. Yesterday, my program worked. today, it does not. why? In *FSE 99: Foundations of Software Engineering*, pages 253 – 267. ACM, 1999.