Exploiting Traces in Program Analysis

Alex Groce and Rajeev Joshi

Laboratory for Reliable Software**,
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA 91109, USA

{Alex.D.Groce,Rajeev.Joshi}@jpl.nasa.gov
http://eis.jpl.nasa.gov/lars

Abstract. From operating systems and web browsers to spacecraft, many software systems maintain a log of events that provides a partial history of execution, supporting post-mortem (or post-reboot) analysis. Unfortunately, bandwidth, storage limitations, and privacy concerns limit the information content of logs, making it difficult to fully reconstruct execution from these traces. This paper presents a technique for modifying a program such that it can produce exactly those executions consistent with a given (partial) trace of events, enabling efficient analysis of the reduced program. Our method requires no additional history variables to track log events, and it can slice away code that does not execute in a given trace. We describe initial experiences with implementing our ideas by extending the CBMC bounded model checker for C programs. Applying our technique to a small, 400-line file system written in C, we get more than three orders of magnitude improvement in running time over a naïve approach based on adding history variables, along with fifty- to eighty-fold reductions in the sizes of the SAT problems solved.

1 Introduction

Analysis of systems that have failed after deployment is a fact of life in all engineering fields. When a bridge collapses or an engine explodes — or a computer program crashes — it is important to understand why in order to avoid future failures arising from the same causes. In the case of software, a patch may be able to correct the flaw and restore a system to working order, making tools for analyzing failure even more valuable.

The motivation for trace-based analysis of programs is straightforward: critical software systems, including file systems, web servers, and even robots exploring the surface of Mars, often produce traces of system activity that humans use to diagnose faulty behavior. Reconstructing the full state or history of a

** The work described in this paper was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

program from these traces or logs is difficult: the traces contain limited information, due to the overhead of instrumentation, privacy concerns, and (in the case of space missions) limited storage space and communication bandwidth. Almost all programmers are familiar with the difficulty of this detective work: after all, "`printf`-debugging" is a particularly common case of trace-based analysis.

The goal of our work is to exploit failure traces in order to increase the scalability of precise program analyses. In particular, we show how restricting program behaviors given a trace can dramatically decrease the size of the SAT formulas in bounded model checking. Given the program source and a trace log, it should be possible to use bounded model checking to find detailed, concrete program executions compatible with the trace — even in cases where the full program is too large to be model checked.

Because the ultimate goal is to provide tool support for programmers dealing with anomalies in remote spacecraft, we refer to trace elements (or `printfs`) as `EVR`s, after the JPL shorthand for Event Reporting. An `EVR` is a command which appends information to a running log. An `EVR` may print a constant string and serve simply to indicate the control flow of the program, or it may contain the current values of critical variables.

A secondary benefit of our work is that program traces are useful as *specifications*. `EVR`s and `printf`s are useful for debugging *because* they provide a high-level description of program behavior. In many cases, a bug is discovered by a programmer reading a trace and noticing an event sequence that should not be possible. The techniques that allow reconstruction of concrete executions given a trace also make it possible to check properties such as: "the system must not produce trace σ" or "the system must be able to produce trace σ". We extend the language of traces to include hidden and wildcard events, producing a restrictive but convenient property language.

This paper contributes two novel techniques. First, we introduce a general method for adding `assume` statements to a deterministic program to restrict its behavior to exactly those executions compatible with a given trace — without introducing history variables or state. Second, we make use of the information gathered in the `assume` statement-generation to *slice* [20] the program, removing portions of the source code based on the information in the program trace.

The first technique is best understood by noting that $\texttt{EVR}(a)$ can be seen as an operation that appends the string a to a history variable, `log`. Adding $\texttt{assume}(\texttt{log} = \sigma)$ at the end of the program will restrict it to behaviors matching the trace σ. For deterministic programs, our analysis computes assumptions that are logically equivalent but do not mention `log`. This direct encoding in terms of control flow and data values aids the SAT solver in propagating constraints — and reduces the size of the state space. The value of slicing may be observed in a more concrete example: consider a program containing complex fault-handling routines. If execution of these routines always produces `EVR`s, and those `EVR`s do not appear in the trace, the fault handling component(s) can be completely eliminated during analysis, with a potential for a drastic reduction in the size of SAT instances used in model checking. Our approach addresses

common variations of the basic problem, including the case where only a suffix of the full trace is known, as well as the presence of multiple, unsynchronized traces.

We implemented our approach as an extension to CBMC [13], a bounded model checker for ANSI-C programs. Analyzing a trace with known length allows us to avoid considering loops and non-terminating execution, making the problem a natural fit for bounded model checking. BMC also determinizes C programs by making all external inputs explicit. We analyzed a model of a small file system and a resource arbiter. As expected, using a trace to guide exploration improved the performance of model checking over a naïve approach based on adding history variables, providing more than three orders of magnitude improvement in running times as well as a fifty- to eighty-fold reduction in the sizes of the SAT problems produced.

2 Reducing a Program with Respect to a Trace

We now formalize the notion of reducing a statement S with respect to a trace σ. The motivation for reduction is improving the scalability of tool-based program analysis. Ideally, we would like to construct a new statement T such that T has *exactly* those executions of S matching σ — i.e., (i) all executions of S that produce σ are executions of T, (ii) all executions of T are executions of S, and (iii) all executions of T produce σ. Here, (i) ensures that we miss no executions that produce σ, (ii) ensures that the verifier produces no "false alarms", and (iii) ensures that we ignore executions that do not produce σ. Of these, (i) is critical: soundness is essential to further analysis; (ii) and (iii) are desirable but not necessary. Constructing a reduced statement T satisfying all three conditions is difficult in general, but is possible given restrictions on S. In this section, we describe these restrictions, and show how a reduced statement T may be constructed given S satisfying these restrictions.

2.1 Notation

To simplify the exposition, we describe our approach in the context of a simple `do-od` language with `assume` and `EVR` statements. A program is a tuple $(\mathcal{V}, \Sigma, S)$ where $\mathcal{V}$ is a set of typed program variables that contains a special variable `log` of type Σ^*, Σ is a finite alphabet of symbols, and S is a statement according to the syntax shown in Figure 1. In this figure, the nonterminal v denotes a variable name in $\mathcal{V}$, the nonterminal E denotes an expression (whose syntax we do not elaborate in this paper), and a denotes a symbol in Σ. A statement is said to be "well-formed" when it does not mention the variable `log`.

The meaning of a program is given in terms of pre- and post-condition semantics in the usual way. We expect that readers are familiar with all but the last construct of this language, and thus omit a full semantics of the language. The semantics of the remaining construct, the `EVR` statement, is given as follows: for any symbol a in Σ, `EVR`(a) is equivalent to "`log` $:=$ `log` $\bullet\, a$". That is, `EVR`(a) appends the symbol a to the variable `log`.

$$\texttt{<}S\texttt{>} \;::=\; v \;\texttt{:=}\; E \mid \texttt{IF}\; E \;\texttt{THEN}\; S \;[\;\texttt{ELSE}\; S\;]\;\texttt{FI} \mid \texttt{WHILE}\; E \;\texttt{DO}\; S \;\texttt{END} \mid S\;;\;S \mid \texttt{SKIP} \mid \texttt{ASSUME}\; E \mid \texttt{ASSERT}\; E \mid \texttt{EVR}(a)$$

Fig. 1: Language syntax

2.2 A Simple Construction

Suppose that we are given a program $(\mathcal{V}, \Sigma, S)$ and a string σ over Σ. As described above, we want to construct a reduced program $(\mathcal{V}, \Sigma, T)$ satisfying conditions (i), (ii) and (iii) above. It is not hard to show that the desired statement T satisfies the following statement equality:

$$T \;=\; \texttt{assume}\;(\texttt{log} = \langle\rangle)\;;\; S\;;\; \texttt{assume}\;(\texttt{log} = \sigma) \tag{1}$$

That is, T consists of exactly those executions of S that, started in a state in which the log is empty, either terminate in a state in which the log is σ, or do not terminate at all[1]. This equation suggests a simple construction: replace occurrences of $\texttt{EVR}(a)$ in S with code for appending a to `log`, and add the two `assume` statements shown above.

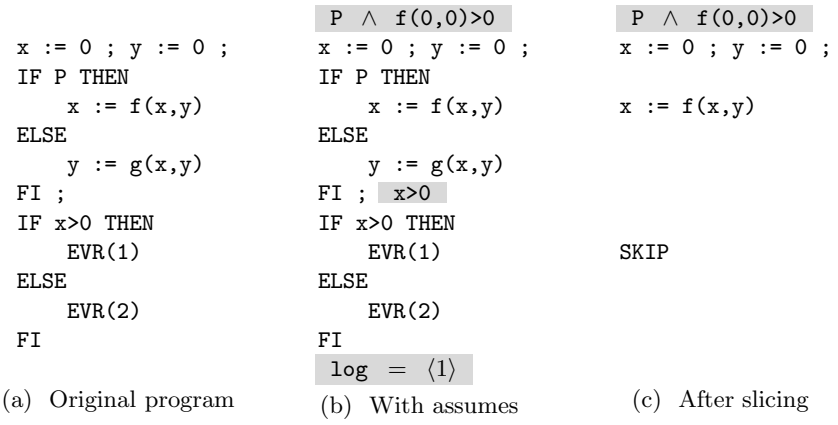
As discussed in Section 4, experience with this simple construction for model checking C programs shows that the addition of such `assume` statements sometimes reduces analysis time significantly (in one instance, time to find an error improves from 17,608 seconds to 105 seconds). Unfortunately, this construction does not suffice to analyze large programs (see Table 2 in Section 4). The limitations of this construction are twofold: (a) knowledge of σ is not exploited in order to simplify the program, and (b) the introduction of `log` as a new program variable adds additional state, which increases the size of the state space to be explored. We now discuss how we avoid these limitations.

2.3 Pushing `assume` Statements Through a Program

Consider the program shown in Figure 2a, where `f` and `g` denote complex computations involving `x` and `y`. Suppose that we want to analyze this program given the singleton trace $\langle 1 \rangle$. We see that this trace is produced only if `x` is assigned a positive value; since the second branch of the first `IF` statement does not modify `x`, knowledge of the trace should allow us to discard the (complex) details of the computation of `g` in our analysis.

One way to achieve this is by pushing `assume` statements through a program. As illustrated in Figure 2b, we can push the final `assume` statement with the predicate $(\texttt{log} = \langle 1 \rangle)$ backwards through the program. This allows us to add an `assume` statement with the predicate `(x > 0)` between the two `IF` statements; in turn, this allows us to introduce an `assume`(P) at the beginning of the program and thus remove the first `ELSE` branch.

[1] Alternatively, we could require that T only have terminating executions. Since CBMC produces unrolled (hence terminating) programs, we do not explore this alternative in this paper.



(a) Original program (b) With assumes (c) After slicing

Fig. 2: Example program for trace reduction. Shaded expressions are assumptions.

We are therefore interested in conditions under which we can push `assume`s through a program. To this end, we consider the following equation: for given statement S and predicate Q, solve for P in

$$\text{solve } P \ : \qquad S \ ; \ \texttt{assume}(Q) \ \subseteq \ \texttt{assume}(P) \ ; \ S \tag{2}$$

where we write $S \subseteq T$ to mean that all executions of S are executions of T. Note that this equation has many solutions in general — e.g., $P = \mathit{true}$. This is related to the observation that one can always push weak assumptions through a program. However, because we want T to include as few unnecessary executions as possible, we are usually interested in the *strongest* solution in P to this equation. It is not hard to show that the strongest solution to this equation exists, and can be expressed in terms of Dijkstra's weakest-precondition transformer as $\neg \mathit{wp}(S, \neg Q)$. Recall that $\mathit{wp}(S, Q)$ denotes the set of states from which all executions of S terminate in states satisfying Q, whereas $\mathit{wlp}(S, Q)$ denotes states from which all *terminating* executions of S end in states satisfying Q. Therefore, the dual expression $\neg \mathit{wp}(S, \neg Q)$ denotes the set of states from which either there is an execution of S that terminates in Q, or an execution of S that fails to terminate.

Unfortunately, although the strongest solution to equation (2) satisfies conditions (i) and (ii) above, it does not guarantee (iii), because there may be executions of the RHS that are not in the LHS. To derive assumptions guaranteeing (iii), we need to solve for P in the following equation:

$$\text{solve } P \ : \qquad S \ ; \ \texttt{assume}(Q) \ = \ \texttt{assume}(P) \ ; \ S \tag{3}$$

This equation is a strict equality. Thus, for any solution P, the right-hand side denotes *exactly* those computations of S that end in states satisfying Q.

The problem with this strict condition is that solutions do not exist in general. The difficulty is illustrated by the following simple example. With $\square$ denoting

nondeterministic choice, consider the statement S given by

$$\texttt{(x := x+1)} \ \square \ \texttt{(x := x+2)}$$

and let Q be the predicate `(x=2)`. Clearly, this equation has no solution for P.

It is not hard to show that for programs that are total[2](in the sense that they can be executed from any state), equation (3) has at most one solution. The more interesting question is when the equation has at least one solution in P. This is addressed by the following result.

Lemma 1 *Let S be a total, deterministic statement. For any predicate Q, equation (3) has a unique solution in P, given by $wlp(S, Q)$, the weakest liberal precondition of Q with respect to S.*

This lemma states that for total, deterministic programs, pushing `assume`s through the program is equivalent to computing *wlp*.

We can also ask when it is possible to push `assume`s forward through a program. In this case, we are interested in solutions for Q in

$$\text{solve } Q \ : \qquad \texttt{assume}(P) \ ; \ S \ \subseteq \ S \ ; \ \texttt{assume}(Q) \tag{4}$$

It is not hard to show that the strongest solution for Q in this equation is $sp(S, P)$, the strongest postcondition of P with respect to S. On the other hand, the strict equation (3) has a solution in Q for arbitrary P only if S is invertible[3]. In general, while determinism is not too strict a requirement (for instance, all sequential C programs are deterministic), invertibility is typically too restrictive. For instance, constant initializations, such as `x := 1`, are not invertible. (To see this, try solving for Q in equation (4) with S being `x:=1` and P being `x=0`.)

However, there are situations in which forward propagation is useful. For instance, *passive* programs which consist only of `assume` statements are trivially invertible. Such programs are often encountered in verification [7, 14]. Because CBMC generates passive programs, we use forward propagation in our implementation.

Once `assume`s have been pushed through the program (either forward or backward), they can be used to remove branches whose guards are refuted by the assumptions. Note that this requires a check to determine which guards are refuted by each assumption. In our implementation, we achieve this with a simple heuristic: for any `assume`(p) appearing before a conditional `IF` q `THEN` S_1 `ELSE` S_2 `FI`, if $p \Rightarrow q$ then we may replace the conditional with S_1 without altering the semantics of the passive program. The amount of slicing obtained depends on the amount of computational effort given to these implications. Our experience so far is that even simple syntactic tests produce effective slicing.

[2] Such programs are sometimes called "non-miraculous" since they satisfy Dijkstra's Law of the Excluded Miracle [4]

[3] To see this, replace S with its relational converse $\sim S$, and solve for Q instead of P in equation (3). The equation is then identical to (3) but with S replaced by $\sim S$. The condition above then states that $\sim S$ should be deterministic, which is the same as saying that S is invertible.

2.4 Removing Trace Variables

By pushing assumptions through a program, we can determine that certain guards are always false, and thus remove certain branches from the code, thereby reducing the size of the program being analyzed. However, since the desired postcondition is $(\texttt{log} = \sigma)$, a naive application of this method requires explicit introduction of the variable `log`. In general, if the alphabet Σ has k symbols, and the given trace σ has length n, addition of `log` adds roughly $n \cdot log_2(k)$ bits to the state space. Since this is linear in n, the length of the trace, the overhead can be considerable when σ is long. In this subsection, we discuss a technique that allows us to work with predicates that do not mention the variable `log`, thus avoiding any overhead.

The idea is to consider predicates in a "log-canonical" form. Let σ be a given trace of length n over Σ, and let $\sigma \uparrow i$ ("σ upto i") denote the first[4] i characters of the string σ. We say that a predicate R is in log-canonical form provided there is a vector t of predicates, such that R can be expressed as

$$(\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; log = \sigma \uparrow i) \tag{5}$$

where none of the predicates t_i mention the variable `log`. Because σ is fixed, this predicate is compactly represented by storing *only* the vector t (which does not mention `log`). For any such vector t, we write $\widehat{t}$ to denote the predicate shown in (5). The usefulness of this notion is due to the following result.

Lemma 2 *Let S be a well-formed deterministic program as defined above, and let P be a predicate in log-canonical form. Then $wp(S, P)$ is also in log-canonical form.*

The proof of Lemma (2) is by induction over the grammar shown in Figure 1. Since S is deterministic, $wp(S, \_)$ distributes over the existential quantification in P. For the first five constructs, the proof is straightforward, using the assumption that none of the guards or expressions in the program mention `log`, since S is well-formed. For the remaining case, $\texttt{EVR}(a)$, we calculate

$$\begin{array}{ll}
 & wp(\texttt{EVR}(a), \widehat{t}) \\
\equiv & \quad \{\ \text{definition of } \widehat{t}\ \} \\
 & wp(\texttt{EVR}(a), (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; \texttt{log} = \sigma \uparrow i)) \\
\equiv & \quad \{\ \text{semantics of } \texttt{EVR}(a);\ \text{the } t_i \text{ don't mention } \texttt{log}\ \} \\
 & (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; wp(\texttt{EVR}(a), \texttt{log} = \sigma \uparrow i)) \\
\equiv & \quad \{\ \text{meaning of } \texttt{EVR}(a) \text{ as appending to } \texttt{log}\ \} \\
 & (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; \texttt{log} \bullet a = \sigma \uparrow i) \\
\equiv & \quad \{\ \text{properties of } \bullet, \text{ and using } \sigma[i-1] \text{ to mean the } i^{th} \text{ character in } \sigma\ \} \\
 & (\exists\, i \,:\, 0 < i \leq n \;\wedge\; t_i \;\wedge\; \sigma[i-1] = a \;\wedge\; \texttt{log} = \sigma \uparrow (i-1)) \\
\equiv & \quad \{\ \text{introducing } u \text{ (see below) and replacing } i \text{ with } j+1\ \} \\
 & (\exists\, j \,:\, 0 \leq j \leq n \;\wedge\; u_j \;\wedge\; \texttt{log} = \sigma \uparrow j) \\
\equiv & \quad \{\ \text{definition of } \widehat{u}\ \} \\
 & \widehat{u}
\end{array}$$

[4] Thus, $\sigma \uparrow 0$ denotes the empty string.

where we have introduced the vector of predicates u, defined as

$$u_j \equiv (t_{j+1} \wedge \sigma[j] = a) \quad \text{for} \quad 0 \leq j < n \quad \text{and} \quad u_n \equiv \mathit{false}$$

Since σ is a fixed string, the predicate $\sigma[j] = a$ is a constant predicate (either *true* or *false*). Furthermore, by assumption, no t_j mentions `log`. Thus the u_j don't mention `log` either, and hence $\widehat{u}$ is also in log-canonical form.

Finally, recall that we are interested in constructing a statement T satisfying equation (1). Note that both the initial predicate ($\texttt{log} = \langle\rangle$) and the final predicate ($\texttt{log} = \sigma$) can be written in log-canonical form using appropriate vectors of predicates; for instance, ($\texttt{log} = \langle\rangle$) corresponds to the vector $[\mathit{true}, \mathit{false}, ...\mathit{false}]$. As shown in this section, we can push these predicates through the program (either backwards or forwards as appropriate). In doing so, we keep track of only vectors of predicates t_i that do not mention the variable `log`. Thus the `assume`s added to the reduced statement T do not mention `log`.

2.5 Extension to Suffixes

Because a trace may have a bounded length, discarding old events after a buffer fills, it is important to handle the case where σ is a *suffix* of the program's execution history. A useful benefit of handling suffixes is the potential to produce a shorter trace matching the suffix; this may be critical when the actual execution extended over a long period of time – both for reasons of analysis scalability and human understanding. In this case, the problem definition is: given a program $(\mathcal{V}, \Sigma, S)$ and a finite string σ of length n over Σ, construct a statement T such that,

$$T \quad = \quad \texttt{assume}(\texttt{log} = \langle\rangle) \; ; \; S \; ; \; \texttt{assume}(\texttt{log} \downarrow n = \sigma) \tag{6}$$

where we write $\texttt{log} \downarrow i$ to mean the last i characters of `log`. In this case, we define $\widehat{t}$ to mean the following:

$$(\exists\, i \;:\; 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; \texttt{log} \downarrow i = \sigma \uparrow i)$$

We leave it to the reader to check that this canonical form is preserved by *wp* computations as discussed above.

3 Implementation

Our analysis is implemented as an extension to CBMC [13], a bounded model checker [3] for ANSI-C programs. Given a program and a set of *unwinding depths* U (the maximum number of times each loop may be executed), CBMC produces constraints encoding all executions of the program not exceeding loop bounds. CBMC converts constraints into CNF and calls a Boolean satisfiability solver, such as zChaff [18] or LIMMAT [2]. A satisfying solution is a counterexample showing a property violation, whereas a proof of unsatisfiability indicates that the code cannot, within the given loop bounds, violate any properties. CBMC

handles all ANSI C types and pointer operations, and checks safety properties such as assertion violations, null pointer dereferences, and array bound errors. CBMC supports `assume` statements in C source, with the expected semantics.

In order to support analysis of traces, we extended CBMC to recognize two *event reporting* functions in C source: `EVR` takes as argument a constant string (an identifier for the event, e.g., `EVR(‘‘timeout’’)`) and `EVR_value` takes an event identifier and an expression (typically an event-relevant program variable, e.g., `EVR(‘‘timeout’’,thread_id)`). A trace, for CBMC, is a sequence of event identifiers, where each identifier produced by an `EVR_value` call includes a value. Our trace language also allows event alphabet restrictions and the use of sets of events in the sequence.

3.1 Analyzing a Simple Program

Consider the program in Figure 3. The program is atypical in that a trace allows near-total reconstruction of the program inputs (though `p` and `q` cannot be precisely determined). For example, if the trace is $\sigma = \langle$`foo 2, foo 1`$\rangle$, we know the value of `input` and constraints on the values of `p` and `q`. It is this knowledge that our analysis will exploit in analyzing the program.

```
void foo () {                                  void bar() {
  x--;                                           x++;
  EVR_value("foo",x);                            EVR("bar");
}                                              }

int main (int input, bool p, bool q) {
    x = input;                                 x#1 == input#0
 1  if (p)                                     x#2 == x#1 - 1;
      foo(); 2                                 x#3 == (p#0 ? x#2 : x#1)
 3  if (q)                                     x#4 == x#3 - 1;
      foo(); 4                                 x#5 == (q#0 ? x#4 : x#3)
 5  if (p && q)
      bar(); 6                                 x#6 == x#5 + 1;
    else                                       x#7 == x#5 - 1;
      foo(); 7                                 x#8 == (p#0 ∧ q#0 ? x#6 : x#7)
 8  assert ((x+1) == input);                   assert ((x#8 + 1) == input#0)
}
```

Fig. 3: `example.c`

As discussed in Section 2.3, our implementation uses a forward analysis to compute assumptions and slices as CBMC generates the equational form of the program. This avoids a second pass over the transformed source code. The right side of Figure 3 shows the passive equational form of `example.c`. In the remainder, we will omit the renamings of `p` and `q`, as these inputs are never assigned.

CBMC produces predicate vectors (as described in Section 2.4) as it converts the program equations into SAT. If we restrict behavior to match σ, the vector has three elements, corresponding to the conditions under which 0, 1, or all elements of the trace have been consumed. As shown in eq. (5), the interpretation of $[t_0, t_1, t_2]$ is $(t_0 \wedge log = \langle\rangle) \vee (t_1 \wedge log = \langle \texttt{foo 1} \rangle) \vee (t_2 \wedge log = \langle \texttt{foo 2, foo 1} \rangle)$.

Loc	Events Consumed		
	A $\langle\rangle$	**B** $\langle$`foo 2`$\rangle$	**C** $\langle$`foo 2, foo 1`$\rangle$
1	*true*	*false*	*false*
2	*false*	`x#2 == 2`	*false*
3	`¬p`	`p ∧ x#2 == 2`	*false*
4	*false*	**3A** `∧ x#4 == 2`	**3B** `∧ x#2 == 2 ∧ x#4 == 1`
5	`¬q ∧ ¬p`	`(q ∧` **4B**`) ∨ (¬q ∧` **3B**`)`	`q ∧` **4C**
6	*false*	*false*	*false*
7	*false*	**5A** `∧ x#7 == 2`	**5B** `∧ x#7 == 1`
8	*false*	`¬(p ∧ q) ∧` **7B**	`¬(p ∧ q) ∧` **7C**

Table 1: Vectors as `example.c` is analyzed with σ. We refer to previous vector entries in a row-column format (i.e., `3B` is row 3, column B: `p ∧ x#2 == 2`).

Table 1 shows the elements of the vectors at 8 program locations (labeled as 1-8 in Figure 3. When pushing assumptions forward, we begin with a vector interpreted as constraining the log to be empty: $[true, false, false]$ (the first row of Table 1). At location 2 the modified vector requires that `x`'s value at the location of the `EVR_value` call match the value in σ.

The vector for location 6 is *false*: if this branch is taken, the sequence of events cannot possibly match σ. When the vector for a branch is false, that branch can be sliced away. We slice the program by changing the equational form and relying on the model checker's ability to prevent un-referenced variables from appearing in the SAT constraints. The final assumption will force the program to take the `ELSE`-branch, which makes it safe to simplify the conditional expression for `x#8` to (*false* `? x#6 : x#7`), which simplifies to `x#7`. The equation for `x#6` can then be discarded. The sliced version of the program produces a SAT problem with 696 variables and 2,312 clauses. Without slicing (leaving the irrelevant then-branch in place), the program requires 834 variables and 2,701 clauses.

3.2 Analyzing with Only a Suffix of a Trace

If we allow σ to be a suffix of the complete trace, the allowed program behaviors are the same (in this example, though not in general), but the analysis is altered. The first row of each vector is always *true*, as it is always possible to *begin* consuming events. The then-branch of the third conditional cannot be sliced away in the initial pass through the program — any events may appear before σ begins. The `bar`-branch can still be sliced away, as it is easy to note that the final condition (**8C**) implies `¬(p ∧ q)` — all allowed executions of the program will

have to take the else-branch. Our analysis does not attempt to extract *all* such implications, but slices based on those that are trivially implied by the assumption (appearing on both sides of a disjunction, or either side of a conjunction, recursively), which has provided near-optimal slicing in our experience.

3.3 Using Traces as Specifications

Traces can be also be used as specifications. In order to use a trace as a specification, CBMC performs the same analysis as above, but searches for *any* execution of the program, rather than searching for property violations. We allow for multiple traces, alphabet restriction, and *sets* of events. With multiple traces, the tool maintains vectors for each trace and assumes the conjunction of all final conditions. This feature can be useful for post-mortem analysis as well, e. g., in the case of traces over different events produced by independent threads without time-stamps. Restricting which EVRs are taken into account is useful for specification: many events may be irrelevant to the property in question, although they appear in the actual code and traces. The utility of sets of events for specification should be obvious — e.g., for specifying that a file should be written to disk when either a `close` or `sync` operation occurs (see below in the experimental results). Handling alphabet restriction and event sets requires only a small modification of the mechanism for checking whether the ith event of a trace matches a particular alphabet symbol in an EVR call.

4 Experimental Results

We applied the technique to a small file system model, consisting of about 400 lines of C code. The model allows basic operations such as opening, closing, reading and writing files; it also supports reset events, which re-initialize all data structures except the disk contents (which is modeled as an array).

As written, the system is not robust across resets: a file can be opened, written to, and closed; if a reset happens at this point, the data in the file can be lost (the sync to disk in the close operation is faulty). We first consider the use of a partial trace as a specification. Using a trace with an `open`, `write`, `close`, a sequence of wildcard actions (not allowing a `delete`), and an `open` followed by a failed `read`[5], we can specify that data should not be lost across any file system event sequence (of a bounded length), even if `reset`s are present. Finding a counterexample (an execution matching this bad trace) requires 105 seconds, when using our technique and this trace as a specification. The utility of guiding the search with a trace is evident: CBMC requires 17,608 seconds to find a counterexample when checking the same property using a hand-coded monitor automaton ("blind" search) as a specification but without even a partial trace of execution. Because the wildcard actions limit the amount of slicing possible, the reduction in the size of the SAT problem is less impressive than the

[5] In the log, success or failure is recorded in addition to which operation is performed.

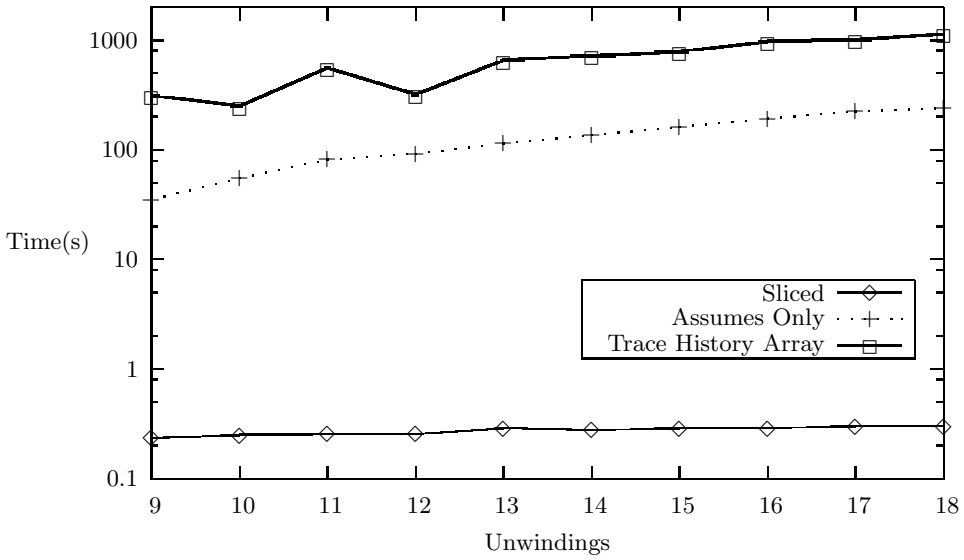


Fig. 4: Results for 8 maximum files

decrease in running time: the monitor-based approach produces a SAT instance with 613,857 variables and 2,108,934 clauses; our approach brings this down to 328,142 variables and 1,128,272 clauses.

A more significant reduction in the size of the SAT problem is seen when examining the same trace with reset in place of wildcards. Figure 4 provides a logscale graph of SAT run-times, given a complete trace for the file system in the smallest configuration we examined. Across a range of unwinding depths, full application of our approach results in a reduction of running time by several orders of magnitude. Applying our analysis to produce an assumption but using no slicing produces a smaller, but still quite significant, reduction over using a trace array semantics. Table 2 shows timing and SAT instance sizes for other configurations of the file system. Checking the property on the *largest* configuration and unwinding depth requires only 26,916 SAT variables when slicing is used; the *smallest* configuration uses 899,989 variables if slicing is not applied, and uses 3,266,123 variables in the largest configuration; running times for the sliced version are uniformly less than one second; over a thousand seconds are needed without slicing. Blind search — without a trace array — was consistently at least an order of magnitude slower than search using a trace array, and did not complete within a timeout period for larger system configurations such as those shown in Table 2.

Applying trace-based analysis to a small model of the core of the resource arbitration algorithm for the Mars Exploration Rovers also improved SAT problem sizes and running times significantly. Adding assumptions to match a failure

	Sliced			Assumes Only			Trace Array		
U	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time
File System Results (System Size = 10)									
11	17,884	65,816	0.29	899,989	3,085,814	91.83	952,924	3,509,042	334.04
12	18,280	67,031	0.30	998,527	3,423,893	119.28	1,067,554	3,960,332	412.91
13	18,676	68,246	0.32	1,097,065	3,762,227	146.00	1,172,149	4,370,541	550.51
14	19,072	69,461	0.32	1,195,603	4,100,816	181.05	1,276,744	4,784,989	1,152.70
15	19,468	70,676	0.32	1,294,141	4,439,660	206.25	1,381,339	5,203,676	624.28
16	19,864	71,891	0.33	1,392,727	4,778,839	248.86	1,485,982	5,626,682	806.59
17	20,260	73,106	0.34	1,491,268	5,118,198	269.77	1,590,580	6,053,852	1,495.01
18	20,656	74,321	0.34	1,589,809	5,457,812	331.40	1,695,178	6,485,261	2,115.49
File System Results (System Size = 12)									
30	26,916	94,931	0.57	3,266,123	11,291,540	1,216.78	3,451,137	13,761,421	2,889.41
Resource Arbiter Results (Safety)									
40	10,497	34,118	0.12	39,273	142,399	1.19	38,936	141,388	1.77
Resource Arbiter Results (Liveness)									
40	21,311	72,142	0.29	73,244	259,308	1.30	72,099	255,639	32.96

Table 2: Results for file system and arbiter. **U** indicates the unwinding depth for loops.

trace the SAT instance grew slightly, but the search time decreased. Applying slicing to remove unreachable portions of the source code reduced the running time to 0.12 seconds. Scaling up to a more complex version of the same model with more properties (including some bounded liveness properties), blind search required 33 seconds, unsliced assumptions needed a little over a second, and with slicing the search time was only 0.29 seconds.

For both the resource arbiter and the file system, the additional overhead for trace-based analysis (performed while computing the passive form of the programs and unrolling loops) prior to calling the SAT solver was negligible.

5 Related Work

This paper presents a use of traces in program analysis — as slicing criteria and specification method — that differs in both motivation and technique from most previous work on related topics.

Assumptions and never-claims are used in many program verifiers [10, 6] to restrict explored system behavior; this kind of restriction is more general than what is described here, but does not provide any *a-priori* state-space reduction — the model checker may explore fewer states in an on-the-fly manner, but these techniques do not preclude exploration of branches that cannot match a given trace. Such methods are also less convenient than our approach for expressing the constraint that system behavior must be able (or not able) to produce a given sequence of events.

Removing code irrelevant to a given program trace is an extension of the idea of *program slicing* [20] — in particular dynamic slicing [1]. Static slicing removes

the portions of a program that are not relevant to the analysis of a particular program point, under *any* set of inputs. Dynamic slicing performs the same task, for a known set of inputs. Parametric program slicing [5] makes use of a more general constraint, allowing for partial knowledge of inputs. Static slicing's utility is limited by aliasing and error handling paths, while dynamic slicing is of little utility when many program traces must be considered — for verification or bug hunting. The *path slicing* [12] of BLAST [9] removes portions of an abstract counterexample that are irrelevant to the feasibility of the path. Path slicing resembles our approach in that both are hybrids of purely static slicing and true dynamic slicing; the approaches differ in purpose (we apply slicing before model checking in order to limit system behaviors; path slicing is a step in a counterexample-refinement loop) and representation of multiple paths (a sequence of trace events vs. a fixed control flow). Millett and Teitelbaum applied more traditional program slicing to Promela models [17]. Only our approach addresses the notion of slicing based on a given event trace.

Howard et al. [11] use model checking to analyze traces produced by software, Roger and Goubault-Larrecq propose similar techniques for use in log auditing for intrusion detection [19], and Gannod and Murthy [8] describe the use of model checking to reverse engineer software architectures from a set of log files, in a largely non-automated approach.

Postmortem Symbolic Evaluation (PSE) [16] uses static analysis to *produce* possible program traces given only a failure's location and type. PSE builds on the work of Liblit and Aiken on the use of backtraces in debugging [15]. The work of Liblit and Aiken is closely related to our approach, in that they consider event traces derived from "`printf` debugging," including the suffix and multiple trace variations. Their work focuses on producing all CFL-reachable paths to a failure, rather than producing only feasible complete concrete executions. It is interesting to note that Liblit and Aiken come to similar conclusions about the advantages of backwards over forwards analysis, for largely independent reasons.

6 Summary and Future Work

We have addressed the problem of analyzing a given program given one of its traces, and demonstrated the utility of our approach for small examples such as the file system and the resource arbiter. A larger concern is how to optimize placement of `EVR`s in order to allow maximal slicing. The placement of `EVR`s is at present largely an ad-hoc process: developing a methodology for placing `EVR`s is critical if we are to analyze larger programs. We are pursuing these problems while applying our method to a larger, in-development, production-quality file system with over 2,000 lines of C source.

References

1. H. Agrawal and J. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, pages 246–256, 1990.

2. A. Biere. The evolution from Limmat to Nanosat. Technical Report 444, Dept. of Computer Science, ETH Zŭrich, 2004.
3. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.
4. E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
5. J. Field, G Ramalingam, and F. Tip. Parametric program slicing. In *Principles of Programming Languages*, pages 379–392, 1995.
6. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 234–245, May 2002.
7. C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles of Programming Languages*, pages 193–205, 2002.
8. G. Gannod and S. Murthy. Using log files to reconstruct state-based software architectures. In *WCRE'02 Workshop on Software Architecture Reconstruction*, 2002.
9. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
10. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
11. Y. Howard, S. Gruner, A. Gravell, C. Ferreira, and J. Augusto. Model-based trace-checking. In *SoftTest: UK Software Testing Research Workshop II*, 2003.
12. R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.
13. D. Kroening, E. Clarke, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.
14. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6), 2005.
15. B. Liblit and A. Aiken. Building a better backtrace: Techniques for postmortem program analysis. Technical Report UCB CSD-02-1203, Computer Science Division, University of California at Berkeley, 2002.
16. R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 63–72, 2004.
17. L. Millett and T. Teitelbaum. Slicing Promela and its applications to model checking, simulation, and protocol understanding. In *SPIN Workshop on Model Checking of Software*, pages 75–83, 1998.
18. M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535, 2001.
19. M. Roger and J. Goubault-Larrecq. Log auditing through model-checking. In *IEEE Workshop on Computer Security Foundations*, page 220, 2001.
20. F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.