Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Exploiting Traces in Static Program Analysis

Better Model Checking through `printfs`

Alex Groce, Rajeev Joshi

Laboratory for Reliable Software
Jet Propulsion Laboratory,
California Institute of Technology,
Pasadena, CA 91109, USA
e-mail: {Alex.D.Groce,Rajeev.Joshi}@jpl.nasa.gov
http://eis.jpl.nasa.gov/lars ⋆

The date of receipt and acceptance will be inserted by the editor

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 *what exactly happened* in order to avoid future failures arising from the same causes. Petroski has argued that failure analysis is the royal road to progress in engineering: understanding past failures is the key to future successes [34]. In the case of software engineering, 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 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 the most primitive form of *dynamic* analysis [4] and is the world's most widespread debugging technique [42]. At heart, `printf`-debugging is the production of failure traces, in the hope of exploiting these traces for (manual) program analysis.

The goal of our work is to exploit failure traces in order to increase the scalability of precise program analyses. Our approach is general enough to be applied as a reduction method for programs, and thus usable for any type of analysis, but our particular application and implementation are targeted for (software) model checking [7,36,9].

In particular, we show how restricting program behaviors given a trace can dramatically decrease the size of the SAT formulas in bounded model checking [6]. 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.

Bounded model checking is a natural choice as a verification method to combine with trace analysis: BMC works by converting a program into a SAT equation.

⋆ 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.

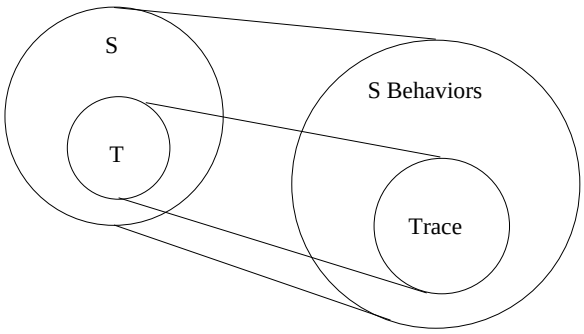


Fig. 1: Reducing a program S by a trace

The solutions to the SAT equation are counterexamples for the properties being model checked. This equational, execution-centered approach to verification integrates cleanly with the algorithm described below, which adds constraints and removes variables to make the SAT problem more tractable. The existence of a program trace may also help to mitigate one of the major limitations of bounded model checking, the need to "guess" loop bounds — a trace may provide important clues or even exact information about loop behavior in the counterexample.

Figure 1 shows the basic concept, which applies to any static analysis technique, not just to bounded model checking:

- We begin with a program S.
- We "restrict" S to a *new program* T, as described in Section 2.

The behaviors (executions) of T are a subset of the behaviors of S: in particular, the only behaviors of T are those which produce the trace of S in question. We expect that, as T has a smaller set of possible behaviors and is likely to be syntactically smaller than S, T will prove easier to analyze than the original program S. This reduction can be seen as an unusual kind of program slicing [40], an intermediate between static slicing and dynamic slicing (see Section 5 for a discussion of related work).

Because our 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 [1]. An `EVR` is a command which appends information to a running log. The log is eventually downlinked to ground control as part of spacecraft telemetry. For our purposes, 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. Event reports are used to diagnose and troubleshoot mission anomalies, including the famous "Spirit Flash anomaly" that jeopardized the Mars Exploration Rover mission [38][1].

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 extends "Exploiting Traces in Program Analysis," which appeared in TACAS 2006 [18]. We first describe 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. We then make use of the information gathered in the `assume` statement-generation to slice 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 `EVR`(a) can be seen as an operation that appends the string a to a history variable, `log`. Adding `assume`(`log` = σ) at the end of a 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`[2]. 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. The suffix variation is particularly important, both for handling cases where there is an absence of trace information about early behavior of the program and for potentially finding shorter traces leading to the same failure. A program's log may contain execution history stretching over days or weeks of execution time, much of which is irrelevant to the exhibited

[1] A lack of easy-to-access event telemetry for the relevant software aspects is noted as a contributing factor to the near-mission loss in the anomaly report.

[2] As noted below, the restriction to determinized programs with inputs is not a significant limitation for model checking

failure. Using a suffix of a trace allows us to experiment with reproducing the failure from only the evidence in the (more likely to be relevant) tail of the full trace.

We implemented our approach as an extension to CBMC [24], a bounded model checker for ANSI-C programs (see Section 3 for details of the implementation and for the analysis of a small program). Analyzing a trace with known length allows us to avoid considering loops and non-terminating execution, simplifying the implementation. CBMC determinizes C programs by making all external inputs explicit (a common approach in software model checking).

Section 4 presents our experimental results. We analyzed a model of a small file system (Section 4.1) and a resource arbitration algorithm based on that used in the Mars Exploration Rovers Spirit and Opportunity (Section 4.2). 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. The improvement over model checking without a trace to restrict the program is from over 17,000 seconds to 105 seconds[3].

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[4]. 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 2. 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`.

```
<S> ::=  v := E
       | IF E THEN S [ ELSE S ] FI
       | WHILE E DO S END
       | S ; S
       | SKIP
       | assert(E)
       | assume(E)
       | EVR(a)
```

Fig. 2: Language syntax

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 most of the constructs of this language, and thus omit a full semantics. The semantics of an `assume` statement is given by the following weakest precondition equation: for any predicates P, Q

$$wp(\texttt{assume}(P), Q) \;=\; (P \Rightarrow Q) \qquad (1)$$

The statement $\texttt{assume}(P)$ always terminates, thus:

$$wp(\texttt{assume}(P), Q) = wlp(\texttt{assume}(P), Q)$$

Operationally, if we view program statements as relations on pre- and post-states, $\texttt{assume}(P)$ is a subset of the identity relation (i.e., `SKIP`), defined only on those pre-states satisfying the predicate P. Note — this means that $\texttt{assume}(P)$ is not total (and hence does not satisfy Dijkstra's Law of the Excluded Miracle [33]). In particular, $\texttt{assume}(\mathit{false})$ corresponds to the empty relation, which establishes *any* postcondition (and is therefore sometimes referred to as a "miracle" [31])

The semantics of the remaining construct, the `EVR` statement, is given as follows: for any symbol a in Σ, $\texttt{EVR}(a)$ is equivalent to "$\texttt{log} := \texttt{log} \bullet a$". That is, $\texttt{EVR}(a)$ appends the symbol a to the variable `log`.

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[5]. It is not hard to show that

[3] This is for an incomplete trace lacking the critical event; with a full trace, the model checking time is reduced to less than a second.

[4] We consider reduction of *statements* rather than *programs per se* as in our language, these are roughly equivalent – a program is a (compound) statement, a set of variables, and an event language.

[5] In practice, we might wish to construct a program $(\mathcal{V}', \Sigma, T)$, in which variables not appearing in T are not included.

the desired statement T satisfies the following statement equality:

$$T \;=\; \texttt{assume}\,(\texttt{log} = \langle\rangle)\;;\; S\;;\; \texttt{assume}\,(\texttt{log} = \sigma) \quad (2)$$

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[6]. 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 3 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 3a, 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 3b, 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.

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

$$\text{solve } P \;:\quad S\;;\; \texttt{assume}(Q) \;\subseteq\; \texttt{assume}(P)\;;\; S \quad (3)$$

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 $\neg wp(S, \neg Q)$. Recall that $wp(S, Q)$ denotes the set of states from which all executions of S terminate in states satisfying Q, whereas $wlp(S, Q)$ denotes states from which all *terminating* executions of S end in states satisfying Q. Therefore, the dual expression $\neg wp(S, \neg Q)$ 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 (3) 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:

$$\text{solve } P \;:\quad S\;;\; \texttt{assume}(Q) \;=\; \texttt{assume}(P)\;;\; S \quad (4)$$

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

`(x := x+1)` $\square$ `(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[7](in the sense that they can be executed from any state), equation (4) 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 (4) has a unique solution in P, given by $wlp(S, Q)$, the weakest liberal precondition of Q with respect to S.*

Proof of Lemma 1. We use the fact that a total, deterministic program S satisfies the following conditions [11], for all predicates Q, R:

$$wp(S, R) \;=\; \neg wlp(S, \neg R) \quad (5)$$

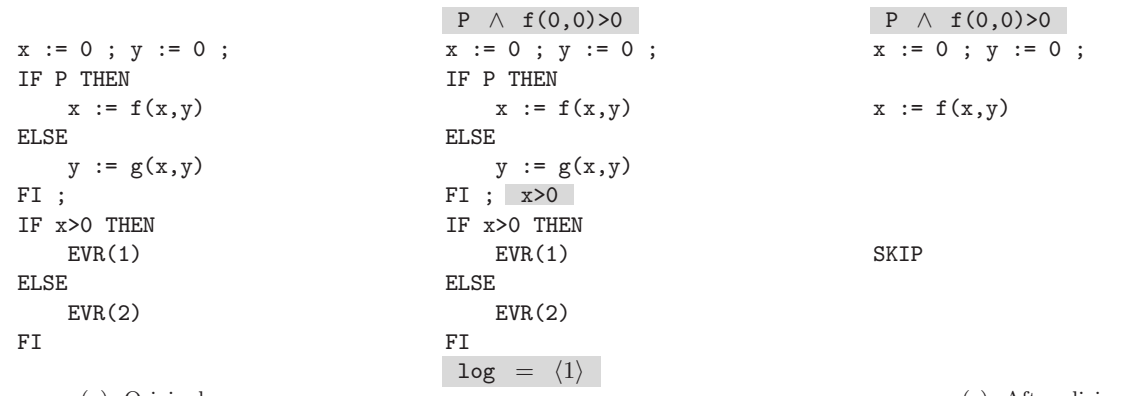
$$wp(S, Q \vee R) \;=\; wp(S, Q) \vee wp(S, R) \quad (6)$$

Recall that the statement of the lemma requires us to show:

$$\texttt{assume}(wlp(S, Q))\;;\; S \;=\; S\;;\; \texttt{assume}(Q)$$

[6] 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.

[7] Such programs are sometimes called "non-miraculous" since they satisfy Dijkstra's Law of the Excluded Miracle [10]



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

Fig. 3: Example program for trace reduction. Shaded expressions are assumptions.

Now this is an equality between programs, so it suffices to show that the formulas for weakest preconditions of each side with respect to an arbitrary predicate R are identical. This follows from the following calculation, starting with the weakest precondition of the LHS above:

$$
\begin{array}{ll}
 & wp(\texttt{assume}(wlp(S,Q))\ ;\ S, R) \\
\equiv & \quad \{\ \text{Since } wp(\texttt{assume}(Q), R) \ = \ (\neg Q \ \vee \ R)\ \} \\
 & \neg wlp(S,Q) \ \vee \ wp(S,R) \\
\equiv & \quad \{\ \text{From condition (5) above}\ \} \\
 & wp(S, \neg Q) \ \vee \ wp(S,R) \\
\equiv & \quad \{\ \text{From condition (6) above}\ \} \\
 & wp(S, \neg Q \ \vee \ R) \\
\equiv & \quad \{\ \text{Weakest precondition of } \texttt{assume}\ \} \\
 & wp(S\ ;\ \texttt{assume}(Q), R)
\end{array}
$$

(End of Proof.)

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

$$\text{solve } Q \ : \qquad \texttt{assume}(P)\ ;\ S \ \subseteq \ S\ ;\ \texttt{assume}(Q) \quad (7)$$

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 (4) has a solution in Q for arbitrary P only if S is invertible[8]. 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 (7) 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 [16,27]. Because CBMC generates passive programs (based on a modified version of Static Single Assignment (SSA) [3] form), we use forward propagation in our implementation[9].

2.3.1 Slicing the Program

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. In a passive program, *any* assumption may be considered to appear before a given `IF`, as the temporal direction of assignments, conveniently, no longer applies. 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.

2.4 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

[8] To see this, replace S with its relational converse $\sim S$, and solve for Q instead of P in equation (4). The equation is then identical to (4) 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.

[9] We have also implemented a backwards propagation version of the algorithm; the results do not substantially differ from forward propagation for the examples we have considered; we report on the more mature forward propagation implementation below.

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 up to i") denote the first[10] 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

$$(\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; log = \sigma \uparrow i) \tag{8}$$

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 (8). As an example, consider the program in Figure 3. The desired `log` is $\langle 1 \rangle$. The vector t representing the conditions under which some portion of σ has been produced at this point in execution is therefore $[true, false]$ at the beginning of the program, and remains so at all program points until the `EVR` statements. At the end of the program, the vector t may be given as $[false, \texttt{x} > \texttt{0}]$ — under no conditions can execution reach the end of the program without producing any `EVR`s, and the trace $\langle 1 \rangle$ will be produced at this point iff `x > 0`. After SSA-transformation, using substitution, we may slice away the assignment to `y`, since `x > 0` implies that `P` must hold in the initial state (and that `f(x,y) > 0`, though this fact is not useful in slicing). Section 3.1 shows in more detail how this construction is used in analyzing a program.

The usefulness of this notion is due to the following:

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 2. 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

$$\begin{array}{ll} & wp(\texttt{EVR}(a), \widehat{t}) \\ \equiv & \quad \{\ \text{definition of } \widehat{t}\ \} \\ & wp(\texttt{EVR}(a), (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; \texttt{log} = \sigma \uparrow i)) \\ \equiv & \quad \{\ \text{semantics of } \texttt{EVR}(a);\ t_i \text{ are well-formed}\ \} \\ & (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; wp(\texttt{EVR}(a), \texttt{log} = \sigma \uparrow i)) \\ \equiv & \quad \{\ \text{meaning of } \texttt{EVR}(a) \text{ as appending to } \texttt{log}\ \} \\ & (\exists\, i \,:\, 0 \leq i \leq n \;\wedge\; t_i \;\wedge\; \texttt{log} \bullet a = \sigma \uparrow i) \\ \equiv & \quad \{\ \text{properties of } \bullet;\ \sigma[i-1] = i^{th} \text{ char. in } \sigma\ \} \\ & (\exists\, i \,:\, 0 < i \leq n \;\wedge\; t_i \;\wedge\; \sigma[i-1] = a \\ & \;\wedge\; \texttt{log} = \sigma \uparrow (i-1)) \\ \equiv & \quad \{\ \text{introducing } u;\ \text{replace } i \text{ with } j+1\ \} \\ & (\exists\, j \,:\, 0 \leq j \leq n \;\wedge\; u_j \;\wedge\; \texttt{log} = \sigma \uparrow j) \\ \equiv & \quad \{\ \text{definition of } \widehat{u}\ \} \\ & \widehat{u} \end{array}$$

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

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

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 (2). 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 $[true, false, \ldots, false]$ (. As shown in this section, we can push these predicates through a 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,

$$T \;=\; \texttt{assume}(\texttt{log} = \langle\rangle)\ ;\ S\ ;\ \texttt{assume}(\texttt{log} \downarrow n = \sigma) \tag{9}$$

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.

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

```
exact
RESET                    EVR(``RESET'');
FORMAT                   EVR(``FORMAT'');
MOUNT_SUCC               EVR(``MOUNT_SUCC'');
PICK 0                   EVR_value(``PICK'',fd);
CREAT_SUCC               EVR(``CREAT_SUCC'');
PICK 0                   EVR_value(``PICK'',fd);
WRITE_SUCC               EVR(``WRITE_SUCC'');
PICK 0                   EVR_value(``PICK'',fd);
CLOSE_SUCC               EVR(``CLOSE_SUCC'');
PICK 1                   EVR_value(``PICK'',fd);
RESET
MOUNT_SUCC               EVR(``MOUNT_SUCC'');
PICK 0                   EVR_value(``PICK'',fd);
OPEN_SUCC                EVR(``OPEN_SUCC'');
PICK 0                   EVR_value(``PICK'',fd);
READ_FAIL                EVR(``READ_FAIL'');
PICK 0                   EVR_value(``PICK'',fd);
```

Fig. 4: A file system trace

3 Implementation

The analysis described above is implemented as an extension to CBMC [24], a bounded model checker [6] 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 [32], Limmat [5], or MiniSAT [13]. 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, including pointer arithmetic, and checks safety properties such as assertion violations, null pointer dereferences, memory safety, arithmetic overflow, 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.

As an example, Figure 4 shows a complete trace that might be produced by our simple file system, in the format in which CBMC stores trace files, with the calls that would produce the trace on the right. The first line, `exact`, indicates that this trace is a complete trace, not a suffix of a longer log. This semantic choice can be overridden by changing options when calling CBMC or by altering the file (we might wish to treat a complete trace as a suffix or vice versa, in some cases). `PICK` events denote the selection of a file descriptor variable to operate on, and the other events indicate system reset or success or failure on basic file system operation calls. The example trace is a failure in which a file is created, written to, and closed, but not available for reading after a well-placed system reset. We will revisit this counterexample in the experimental results.

3.1 Analyzing a Simple Program

Consider the program in Figure 5. 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.

As discussed in Section 2.3, our implementation uses a forward analysis to compute assumptions and slices as CBMC generates the equational (SSA-like) form of the program. This avoids a second pass over the transformed source code. The right side of Figure 5 shows the passive equational form of `example.c` (the effects of calls to `foo` and `bar` are inlined). In the remainder, we will omit the renamings of `p` and `q`, as these inputs are never reassigned.

CBMC produces predicate vectors (as described in Section 2.4) as it converts the program equations into SAT equations. 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. (8), the interpretation of $[t_0, t_1, t_2]$ is $(t_0 \wedge log = \langle\rangle) \vee (t_1 \wedge log = \langle$`foo 1`$\rangle) \vee (t_2 \wedge log = \langle$`foo 2, foo 1`$\rangle)$.

Table 1 shows the elements of the vectors at 8 program locations (labeled as 1-8 in Figure 5. 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` call match the value in σ. Restriction by variable values is not discussed in earlier sections, but can be considered as a simple case of alphabet matching: the expression producing the trace value is constrained to match the observed value as part of the condition for the event.

The use of references to expressions appearing earlier in the table makes clear the pattern by which the constraints "march across" the program with each event, including propagation into branches of conditional statements and a "merge" at the end of each conditional structure. The implementation also makes use of ref-

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

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

```
void bar() {
  x++;
  EVR("bar");
}
```

```
x#1 == input#0
x#2 == x#1 - 1;
x#3 == (p#0 ? x#2 : x#1)
x#4 == x#3 - 1;
x#5 == (q#0 ? x#4 : x#3)

x#6 == x#5 + 1;
x#7 == x#5 - 1;
x#8 == (p#0 ∧ q#0 ? x#6 : x#7)
assert ((x#8 + 1) == input#0)
```

Fig. 5: `example.c`

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`).

erences to previously generated expressions in order to conserve memory (and speed the translation to CNF).

Consider the derivation of the values in row 5. The first column (**5A**) gives conditions under which no input symbols have been consumed: when both `p` and `q` are false (and thus neither location 2 nor location 4 has been reached — note that **2A** and **4A** are both false). This condition is derived from the disjunction (`q` ∧ **4A**) ∨ (`¬q` ∧ the implied `¬p` for the implicit else-branch). The left side of the disjunction simplifies to false, leaving us with `¬q ∧ ¬p`. We could write this as `¬q ∧` **3A** to show that the condition on `p` originates in the earlier branch over `p`. The condition at **5B** makes this a bit clearer: at the end of the conditional on `q`, the condition for consumption of one input symbol is a disjunction of (1) the *positive* valuation of the guard conjoined with the condition for consumption of one input symbol *inside* the branch (**4B**) and (2) the *negative* valuation of the guard conjoined with the condition for consumption of one input symbol if the guard is not taken (**3B** since the else-branch is empty).

Observe that 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 away branches in other conditions as well, as discussed below). 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 (a kind of cone-of-influence reduction). 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.

```
exact
RESET,FORMAT,MOUNT_SUCC,MOUNT_FAIL,CREAT_SUCC
CREAT_FAIL,OPEN_SUCC,OPEN_FAIL,WRITE_SUCC
WRITE_FAIL,READ_SUCC,READ_FAIL
end
RESET                   EVR(``RESET'');
FORMAT                  EVR(``FORMAT'');
MOUNT_SUCC              EVR(``MOUNT_SUCC'');
CREAT_SUCC              EVR(``CREAT_SUCC'');
WRITE_SUCC              EVR(``WRITE_SUCC'');
CLOSE_SUCC              EVR(``CLOSE_SUCC'');
-RESET                  Any EVR other than a RESET
-RESET
-RESET
MOUNT_SUCC              EVR(``MOUNT_SUCC'');
OPEN_SUCC,OPEN_FAIL     EVR(``OPEN_*'');
READ_FAIL               EVR(``READ_FAIL'');
```

Fig. 6: A "specification" trace with alphabet restriction and negated events

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 $\neg$(`p` $\wedge$ `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. Determining the "best" slice is as hard as the model checking problem, although it is possible that a more aggressive and computationally expensive approach than our syntactic analysis (using a SAT solver, for instance) might be valuable for certain programs and traces.

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. Figure 6 shows a file system trace of this type: the elements after the semantic indicator `exact` and before the `end` are the alphabet to be used. Other events are simply ignored. In the absence of an alphabet, the tool defaults to observing all events. Here, ignoring the `PICK` events provides a more general (and false) specification than the original property that written files should be readable, as the read can apply to a file descriptor that has not been opened. The `-RESET` action indicates that any action except a `RESET` is allowable at this point in the trace, and the `OPEN_SUCC,OPEN_FAIL` allows either a failed or successful open operation.

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 as specification, 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

In principle, it is possible that a reduction in the size of a program, or even in its state-space, may not result in better scalability for verification. In order to empirically justify the usefulness of our approach, we applied the technique to real programs and observed a dramatic improvement in verification times. Given the small number of examples, it is impossible to draw definite conclusions, other than that further study, with larger examples, is warranted. We expect that a re-implementation in the latest version of CBMC would serve as a good baseline for such a confirmation, in that some problems we observed with CBMC stability when encoding large C programs with considerable library usage appear to have been mitigated in recent versions of the tool[11].

All experiments were performed on a dual-core Xeon (3.2 GHz) with 8 GB of RAM, under Red Hat Enterprise Linux 4.0.2-8. We used Limmat version 1.3 as our SAT solver in all cases, in order to provide some uniformity in results, as it worked best on the largest and most difficult file system instances.

4.1 Simple File System Model

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

[11] The conversion to GOTO programs introduced in recent versions of CBMC unfortunately prevents a naive adaptation of our original extension.

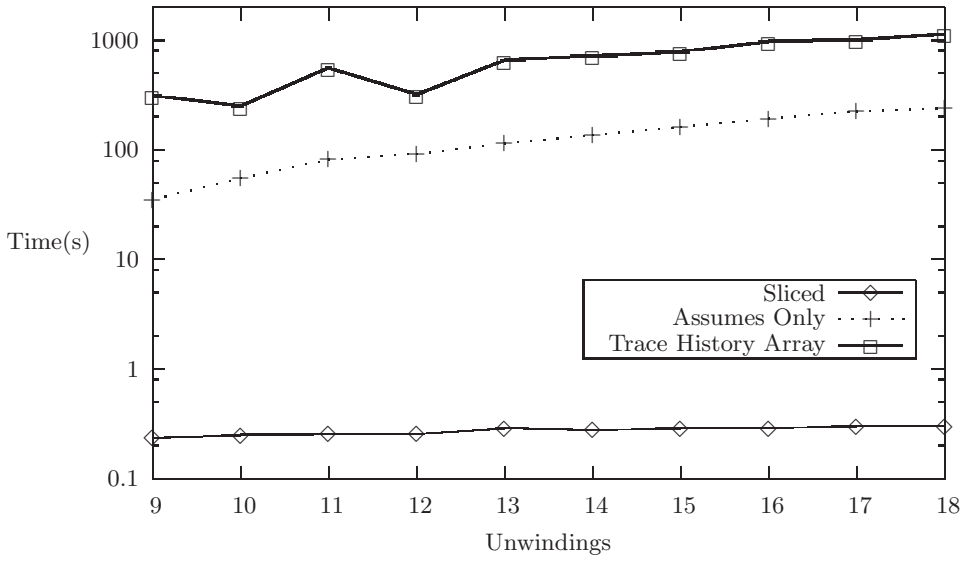


Fig. 7: Results for 8 maximum files, without blind search

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`[12], 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 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 is a slightly simplified version of the actual failure trace used). Figure 7 provides a logscale graph of SAT run-times, given a complete trace for the file system in the smallest configuration we examined. For comparison, although the results are not strictly comparable (the replacement of wildcards with resets provides a more constrained problem, as noted above), we also show a graph comparing running times for a smaller set of instances with the blind monitor-based search mentioned above (Figure 8) and more detailed SAT instance information for the same set of experiments (Table 4.1). In this table, **TO** indicates that the SAT solver did not complete its search within 8 hours. We omit blind/monitor results in subsequent tables. The results were always timeouts, the SAT variables and clauses for the monitor-based search followed a predictable linear pattern as in Table 4.1, and the searches themselves are not really comparable to the EVR-constrained model checking problems.

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 3 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, using a monitor — was consistently roughly an order of magnitude (or more) slower than search using a trace array,

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

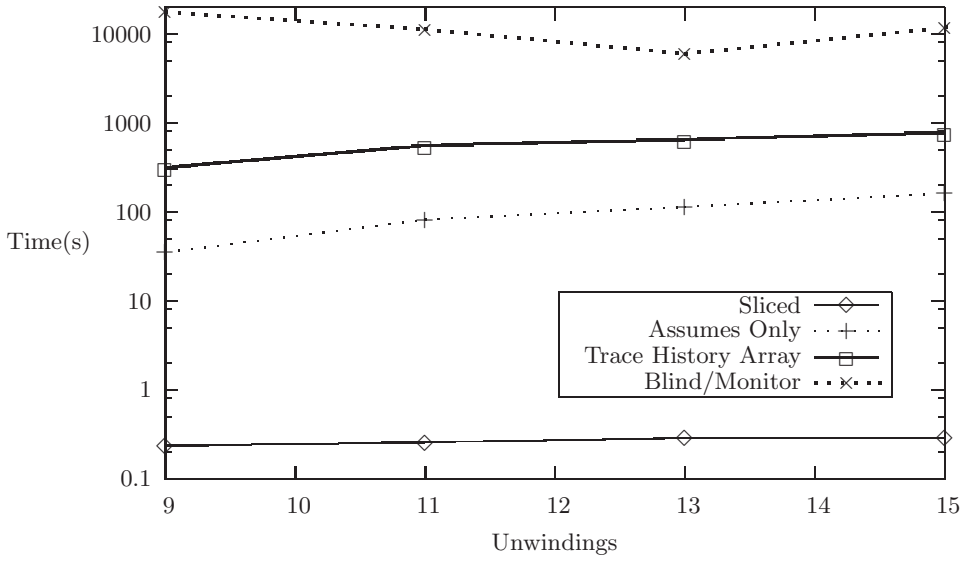


Fig. 8: Results for 8 maximum files, including blind search

	Sliced			Assumes Only			Trace Array			Blind/Monitor		
U	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time
9	14850	54774	0.24	588253	2014890	34	631134	2323868	312	613857	2108934	17609
11	15626	57154	0.26	752229	2576685	82	805164	2999913	560	784071	2694364	11247
13	16402	59534	0.29	916205	3139500	114	989491	3739004	653	954239	3281105	5943
15	17178	61914	0.29	1080181	3703335	161	1165265	4456961	781	1124408	3869253	11787
17	17954	64294	0.30	1244208	4268275	225	1341090	5191959	1003	1294576	4458802	**TO**

Table 2: Some more detailed results for 8 maximum files.

and did not complete within a timeout period for larger system configurations such as those shown in Table 3.

4.2 Resource Arbiter Model

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 trace (providing only requests and responses, but not the actual resources — a serious under-constraint on the actual error path), 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 requiring that certain critical resources be acquired within a time period after request, unless the request was rescinded), 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 (indistinguishable from the "noise" of CBMC's parsing and pre-processing steps). We omit this essentially constant factor from the results.

4.3 Context-bounded Model Checking via Trace Analysis

We speculated that trace-based analysis might be useful for context-bounded model checking [35]: if context-switches invoke an EVR call, a trace of exactly n switches can be used to restrict a concurrent program to a context-bounded set of executions. CBMC lacks native support for concurrency, forcing us to explicitly model thread program counters and context-switching. Our test case was a model of a flawed mutual exclusion protocol based on atomic store and load operations on a processor (derived from methods used to implement reference counting pointers on the processor).

Unfortunately, the slicing provided by the EVR representation of context-switches is negligible, when compared to a version using `assume` statements to enforce context-bounding: the sliced SAT instance needs 234,617

	Sliced			Assumes Only			Trace Array		
U	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time
File System Results (System Size = 10)									
11	17884	65816	0.29	899989	3085814	91.83	952924	3509042	334.04
12	18280	67031	0.30	998527	3423893	119.28	1067554	3960332	412.91
13	18676	68246	0.32	1097065	3762227	146.00	1172149	4370541	550.51
14	19072	69461	0.32	1195603	4100816	181.05	1276744	4784989	1152.70
15	19468	70676	0.32	1294141	4439660	206.25	1381339	5203676	624.28
16	19864	71891	0.33	1392727	4778839	248.86	1485982	5626682	806.59
17	20260	73106	0.34	1491268	5118198	269.77	1590580	6053852	1495.01
18	20656	74321	0.34	1589809	5457812	331.40	1695178	6485261	2115.49
File System Results (System Size = 12)									
30	26916	94931	0.57	3266123	11291540	1216.78	3451137	13761421	2889.41

Resource Arbiter Results (Safety)									
	Sliced			Assumes Only			Blind		
U	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time
40	10497	34118	0.12	39273	142399	1.19	38936	141388	1.77

Resource Arbiter Results (Liveness)									
	Sliced			Assumes Only			Blind		
U	Vars	Clauses	Time	Vars	Clauses	Time	Vars	Clauses	Time
40	21311	72142	0.29	73244	259308	1.30	72099	255639	32.96

Table 3: Results for file system and arbiter. **U** indicates the unwinding depth for loops. Note: the final set of vars/clauses/time for the Resource Arbiter properties is for a blind search, not for a trace array search.

variables and 753,541 clauses, while the version using assumptions only (and no trace analysis) requires 241,819 variables and 804,298 clauses. More importantly, Limmat solves the larger instance in only 39 seconds, but requires 83 seconds for the smaller instance. The relatively small reduction in SAT problem size indicates that (at least in this example) context-switching does not constrain the control flow of the system sufficiently to obtain significant benefits. It is also possible that these results are influenced by an inefficient representation of concurrency. More conclusive results must await integration with a native representation of concurrency (e.g., that of Rabinovitz and Grumberg [37] or the in-progress extension of CBMC).

5 Related Work

This paper extends a 2006 TACAS paper [18] on the 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 [21,41,15] 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 input choices that cannot match a given trace. Such methods are considerably 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, and doing so will often require the introduction of a history variable, increasing the size of the state space.

Removing code irrelevant to a given program trace is an extension of the idea of *program slicing* [40] — in particular dynamic slicing [2]. 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 kind of reduction, for a known set of inputs. Parametric program slicing [14] makes use of a more general constraint, allowing for only partial knowledge of inputs. Static slicing's utility is limited by aliasing and error handling paths (it is often the case that under *some* set of conditions, almost every line of a program is relevant to a given property). Unfortunately, dynamic slicing is of little utility when many program traces must be considered — for verification or bug hunting. A failure trace is very unlikely to allow full reconstruction of program inputs.

The *path slicing* [23] of BLAST [20] 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 slic-

ing and true dynamic slicing. The approaches differ in purpose: we apply slicing before model checking in order to limit system behaviors, while path slicing is a step in a counterexample-refinement loop. Our approach addresses both concrete execution paths and event traces while BLAST's slicing is based on a fixed control flow.

More generally, the cone-of-influence reduction used in model checking is a kind of static slicing[13]. Millett and Teitelbaum applied more traditional program slicing to Promela models [30], Clarke et al. have proposed to use slicing for hardware description languages [8], and the Bandera project has devoted considerable effort to static slicing as an aid to software model checking [19, 12]. Ours is, to our knowledge, the first work to provide slicing based on a given *event trace*, in a model checking (or general program analysis) context.

Howard et al. [22] use model checking to analyze traces produced by software, Roger and Goubault-Larrecq propose similar techniques for use in log auditing for intrusion detection [39], and Gannod and Murthy [17] describe the use of model checking to reverse engineer software architectures from a set of log files, in a largely non-automated approach. These works are all either relatively limited in scope or lacking in automation. Our approach is an automated general-purpose program slicing method, and does not limit itself to any particular domain of application, despite the motivation in spacecraft event reporting.

More closely related to our efforts is Postmortem Symbolic Evaluation (PSE) [29], which makes use of static analysis to *produce* possible program traces given only a failure's location and type. This approach uses (and produces) less information than our slicing method, but is intended to scale to very large programs and in-the-field bug reports which may not even include a stack trace.

PSE builds on the work of Liblit and Aiken on the use of backtraces in debugging [28]. 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 to ours about the advantages of backwards over forwards analysis, for largely independent reasons. More recent work addresses optimizing path lengths in a similar context (though application to event logs is not the primary goal, the authors note that a small modification allows for use similar to that in Liblit and Aiken's earlier work) [26].

[13] It is unclear who first devised or published the cone-of-influence reduction, though Kurshan [25] is a plausible candidate.

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.

Avenues of future research include determining if the failure of context-bounded model checking to combine well with our approach is fundamental, or only a limitation of the hand-coding of concurrency, and investigation of the tradeoffs between aggressive slicing and expensive pre-SAT computation. We are also interested in applying a variation of our approach to explicit-state model checking in SPIN [21].

A larger concern is how to optimize placement of `EVR`s in order to allow maximal slicing while retaining (or improving) EVR utility to humans in diagnosing problems. 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, and there are concerns that the current approach may be less than ideal for human analysis [38]. We are pursuing these problems while applying our method to a larger, in-development, flight-quality Flash file system with over 4,000 lines of C source.

Acknowledgements: The authors would like to thank Daniel Kroening for assistance with CBMC and Thomas Reps for related work suggestions.

References

1. `http://stardust.jpl.nasa.gov/acronyms.html`.
2. Hiralal Agrawal and Jospeh R. Horgan. Dynamic program slicing. In *Programming Language Design and Implementation*, pages 246–256, 1990.
3. Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Principles of Programming Languages*, pages 1–11, 1988.
4. Thomas Ball. The concept of dynamic analysis. In *European Software Engineering Conference/Foundations of Software Engineering*, pages 216–234, 1999.
5. Armin Biere. The evolution from Limmat to Nanosat. Technical Report 444, Dept. of Computer Science, ETH Zŭrich, 2004.
6. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, 1999.
7. Edmund M. Clarke and E. Emerson. The design and synthesis of synchronization skeletons using temporal logic. In *Workshop on Logics of Programs*, pages 52–71, 1981.
8. Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 298–312, 1999.
9. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 2000.

10. Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, New Jersey, 1976.
11. Edsger W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.
12. Matthew B. Dwyer, John Hatcliff, Matthew Hoosier, Venkatesh Prasad Ranganath, Robby, and Todd Wallentine. Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 73–89, 2006.
13. Niklas Een and Niklas Sorensson. An extensible SAT-solver. In *Symposium on the Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518, 2003.
14. John Field, Ganesan Ramalingam, and Frank Tip. Parametric program slicing. In *Principles of Programming Languages*, pages 379–392, 1995.
15. Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie 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.
16. Cormac Flanagan and James B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In *Principles of Programming Languages*, pages 193–205, 2002.
17. Gerald Gannod and Shilpa Murthy. Using log files to reconstruct state-based software architectures. In *WCRE'02 Workshop on Software Architecture Reconstruction*, 2002.
18. Alex Groce and Rajeev Joshi. Exploiting traces in program analysis. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 379–393, 2006.
19. John Hatcliff, Matthew B. Dwyer, and Hongjun Zheng. Slicing software for model construction. *Higher-Order and Symbolic Computation*, 13(4):315–353, 2000.
20. Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Lazy abstraction. In *Principles of Programming Languages*, pages 58–70, 2002.
21. Gerard J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, 2003.
22. Yvonne Howard, Stefan Gruner, Andrew M. Gravell, Carla Ferreira, and Juan Carlos Augusto. Model-based trace-checking. In *SoftTest: UK Software Testing Research Workshop II*, 2003.
23. Ranjit Jhala and Rupak Majumdar. Path slicing. In *Programming Language Design and Implementation*, pages 38–47, 2005.
24. Daniel Kroening, Edmund M. Clarke, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176, 2004.
25. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes: The Automata- Theoretic Approach*. Princeton University Press, 1995.
26. Akash Lal, Junghee Lim, Marina Polishchuk, and Ben Liblit. Path optimization in programs and its application to debugging. In *European Symposium on Programming*, pages 246–263, 2006.
27. K. Rustan M. Leino. Efficient weakest preconditions. *Information Processing Letters*, 93(6), 2005.
28. Ben Liblit and Alex 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.
29. Roman Manevich, Manu Sridharan, Stephen Adams, Manuvir Das, and Zhe Yang. PSE: explaining program failures via postmortem static analysis. In *Foundations of Software Engineering*, pages 63–72, 2004.
30. Lynette I. Millett and Tim 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.
31. Carroll Morgan. The specification statement. *ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.
32. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Linao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Design Automation Conference*, pages 530–535, 2001.
33. Greg Nelson. A Generalization of Dijkstra's Calculus. *TOPLAS*, 11(4):517–561, Oct. 1989.
34. Henry Petroski. *To Engineer is Human: The Role of Failure in Successful Design*. St. Martin's Press, 1985.
35. Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, April 2005.
36. Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in CESAR. In *International Symposium on Programming*, pages 337–351, 1982.
37. Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In *Computer-Aided Verification*, pages 82–97, 2005.
38. Glenn Reeves and Tracy Neilson. The Mars Rover Spirit Flash anomaly. In *IEEE Aerospace Conference*, 2005.
39. Muriel Roger and Jean Goubault-Larrecq. Log auditing through model-checking. In *IEEE Workshop on Computer Security Foundations*, page 220, 2001.
40. Frank Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.
41. Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.
42. Andreas Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.