Trace-Driven Verification of Multithreaded Programs*

Zijiang Yang[1] and Karem Sakallah[2]

[1] Western Michigan University, Kalamazoo, MI 49008, USA
[2] University of Michigan, Ann Arbor, MI 48109, USA

Abstract. We present a new method that combines the efficiency of testing with the reasoning power of satisfiability modulo theory (SMT) solvers for the verification of multithreaded programs under a user specified test vector. Our method performs dynamic executions to obtain both under- and over-approximations of the program, represented as quantifier-free first order logic formulas. The formulas are then analyzed by an SMT solver which implicitly considers all possible thread interleavings. The symbolic analysis may return the following results: (1) it reports a real bug, (2) it proves that the program has no bug under the given input, or (3) it remains inconclusive because the analysis is based on abstractions. In the last case, we present a refinement procedure that uses symbolic analysis to guide further executions.

1 Introduction

One of the main challenges in testing multithreaded programs is that the absence of bugs in a particular execution does not necessarily imply error-free operation under that input. To completely verify program behavior for a given test input, *all executions* permissible under that input must be examined. However, this is often an infeasible task considering the exponentially large number of possible interleavings of a typical multithreaded program. A program with n threads, each executing k statements, can have up to $(nk)!/(k!)^n \geq (n!)^k$ thread interleavings, a dependence that is exponential in both n and k.

In this paper we address this challenge by an approach called *Trace-Driven Verification* (TDV) that combines the efficiency of testing with the reasoning power of satisfiability modulo theory (SMT) solvers. TDV performs dynamic executions to obtain approximations, represented as quantifier-free first order logic (FOL) formulas, of the program under verification. The formulas are then analyzed by an SMT solver which implicitly considers all possible thread interleavings. The symbolic analysis may return one of the following results: (1) it reports a real bug, (2) it proves that the program has no bug under the given input, or (3) it remains inconclusive because the analysis is based on abstractions. In the last case, we present a refinement procedure that uses symbolic analysis to guide further executions. The features of TDV include:

* The work was supported by NSF Grant CCF-0811287.

– **Implicit consideration of thread interleavings.** As explicit enumeration of executions is intractable, the alternative we present is to capture thread interleavings implicitly as a set of constraints in a satisfiability formula. These constraints belong to the family of quantifier-free first order logic formulas for which efficient SMT solvers are available.
– **Integration of dynamic executions and symbolic analysis.** At any given time, TDV analyzes only the statements that appear in a particular execution under a user-specified test vector. It may report a real bug, or prove that the program behaves as expected under all thread interleavings stimulated by the given input. In either case, TDV avoids the analysis of statements that do not appear in an execution. However, it is also possible that the symbolic analysis, being an abstraction of program behavior, remains inconclusive. In such a case, TDV uses the symbolic analysis result to guide future concrete executions.
– **Abstraction with both under- and over-approximations.** Based on an execution, TDV infers both under- and over-approximations of the entire program. The under-approximation is complete so that any bug detected in the model is a real bug; while the over-approximation is sound so that it can be used to prove the absence of bugs.

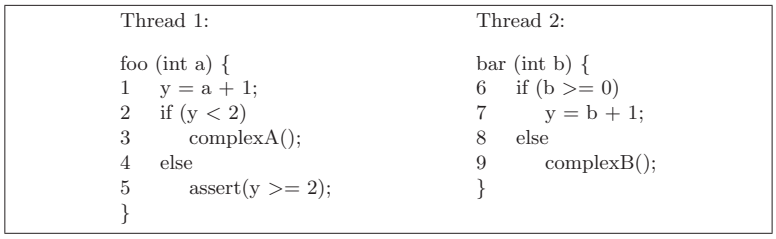
The rest of the paper is organized as follows. After giving the algorithm overview in Section 2, we present the symbolic encoding of program traces in Section 3. The refinement procedure is illustrated in Section 4. In Section 5 we outline several encoding and algorithmic optimizations to improve scalability. We discuss related work in Section 6. Finally we present experimental results in Section 7 and conclude the paper in Section 8.

2 Algorithm Overview

Consider a multithreaded program P where threads communicate via shared variables. Without loss of generality, we assume there is at most one shared variable access at a program statement[3]. Then each statement constitutes an *atomic computational step*, at which granularity the thread scheduler can switch control between threads during the execution.

Consider the program, shown in Fig. 1, that consists of two concurrently running threads. In a typical testing environment, even if we run the program multiple times under the test input $a = 1, b = 0$, we may obtain the same executed trace $\pi_1 = \langle 1, 2, 5, 6, 7\rangle$ where the integer values indicate the line numbers. In general, an executed trace is an ordered sequence of program statements executed by the different threads. Although π_1 does not cause an assertion failure

[3] If there are multiple shared variable accesses in one statement, we can introduce additional local variables and split the statement into multiple statements such that each statement has at most one shared variable. For example, consider a statement $a = x + y$ with shared variables x, y and local variable a. It can be split into two statements $t = y$ and $a = x + t$ with the help of a temporary local variable t.



```
Thread 1:                          Thread 2:

foo (int a) {                      bar (int b) {
1    y = a + 1;                    6    if (b >= 0)
2    if (y < 2)                    7        y = b + 1;
3        complexA();               8    else
4    else                          9        complexB();
5        assert(y >= 2);           }
}
```

Fig. 1. A program with the shared variable y and local variables a, b.

on Line 5, we cannot conclude the absence of assertion failures in this program as this input admits other interleavings of these two threads. Table 1 shows the set $\Pi(\pi_1)$ of all 10 possible interleavings of π_1. For each trace in the table, the bottom row indicates whether the assertion on Line 5 holds (h) or fails (f). However, not all the interleavings in $\Pi(\pi_1)$ are valid executions. Closer examination of π_6 and π_9 shows that they are infeasible traces, due to the violation of program semantics. In particular, after y is updated by Thread 2 on Line 7, it is not possible for Thread 1 to follow the `Else` branch on Line 2. Let $\Pi_P(\pi_1)$ be the set of interleavings derived from π_1 that are consistent with the semantics of the program P. We have $\Pi_P(\pi_1) = \{\pi_1, \pi_2, \pi_3, \pi_4, \pi_5, \pi_7, \pi_8, \pi_{10}\}$. We call a trace $\pi_i \in \Pi_P(\pi_1) \backslash \{\pi_1\}$ an *induced trace* of π_1.

Table 1. $\Pi(\pi_1)$: all the thread interleavings of π_1. The two interleavings marked with an asterisk are invalid since they violate program semantics.

Step	π_1	π_2	π_3	π_4	π_5	π_6^*	π_7	π_8	π_9^*	π_{10}
1	1	1	1	1	1	1	6	6	6	6
2	2	2	2	6	6	6	1	1	1	7
3	5	6	6	2	2	7	2	2	7	1
4	6	5	7	5	7	2	5	7	2	2
5	7	7	5	7	5	5	7	5	5	5
assert	h	h	f	h	f	f	h	f	f	h

In order to check for assertion failures not only in π_1 but also in its induced traces, we construct an FOL formula $\varphi(\pi)$ that implicitly models all the traces in $\Pi_p(\pi)$ (see Section 3.1 for details). A satisfying assignment to $\varphi(\pi)$ indicates a true assertion failure and can be used to identify the particular thread interleaving that produces it. If $\varphi(\pi)$ is unsatisfiable, however, we cannot conclude correctness because $\varphi(\pi)$ is an under-approximation of program behavior. To understand the reason consider a statement `assert(`$C_{complexA}$`)` inside `complexA()` on Line 3 in Fig. 1. Given the executed trace $\pi_1 = \langle 1, 2, 5, 6, 7 \rangle$, $\varphi(\pi_1)$ itself cannot reveal any assertion failure inside `complexA()` since the `assert(`$C_{complexA}$`)` statement does not even appear in any traces of $\Pi_p(\pi_1)$. On the other hand, there exist valid executions that execute `complexA()` (e.g. $\pi' = \langle 1, 6, 7, 2, 3, \ldots \rangle$). Thus an assertion failure is still possible under the test input $a = 1, b = 0$.

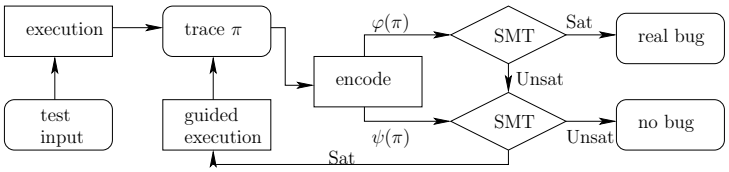


Fig. 2. Trace-driven verification flow.

To insure correctness (absence of assertion failures), all execution traces permissible under that input must be examined. We relax, or abstract $\varphi(\pi)$, by making changes to and dropping some of its constraints (see Section 3.2 for details). This leads to $\psi(\pi)$, an FOL formula that represents an over-approximation to the program behavior under the specified input. If $\psi(\pi)$ is unsatisfiable, we can provably conclude the absence of assertion failures for all thread interleavings under the specified input. Otherwise we need to check if the reported violation is true or spurious. In the latter case, TDV performs refinement by modifying the control flow in order to examine other executions of P under the same test input.

As illustrated in Fig. 2, TDV consists of the following steps:

1. Run the program under a given user input to obtain an initial execution trace π.
2. Using an encoding along the lines illustrated in Section 3.1, construct an FOL formula $\varphi(\pi)$.
3. Using an SMT solver, check the satisfiability of $\varphi(\pi)$.
   - If $\varphi(\pi)$ is found to be satisfiable, a real bug is found. Based on the solution to $\varphi(\pi)$ we can report to the user the specific scheduling that exposes the bug.
   - If $\varphi(\pi)$ is found to be unsatisfiable, we relax $\varphi(\pi)$ to obtain $\psi(\pi)$. This allows us to examine *sibling* traces, i.e., traces that conform to the same input but cover different statements.
     - If $\psi(\pi)$ is found to be unsatisfiable, we can conclude that the property holds under all possible thread interleavings under the given test input.
     - If $\psi(\pi)$ is found to be satisfiable, the SMT solver returns a counter-example, which is used to guide new executions that are guaranteed to touch new statements that have not appeared in previous executions.

3 Symbolic Encoding of Execution Traces

An executed trace is a sequence $\pi = \langle (t_1, l_1.o_1, Q_1), \ldots, (t_n, l_n.o_n, Q_n) \rangle$ that lists the statements executed by the various threads. Each tuple $(t, l.o, Q) \in \pi$ is considered to be an atomic computational step where t is the thread id, l is

the line number for the statement, o is an *occurrence index* that distinguishes the different executions of the same statement, and Q is the statement type that can be one of *assign, branch, jump, fork, join* or *assert*. In this paper we assume all the executions eventually terminate[4].

We consider three basic types of statements: assignment $v = E$ where E is an arithmetic expression, branch $C?l.o$ where C is a relational expression, and jump *goto* $l.o$. Note that $C?l.o$ only lists the destination if C holds because no two branches can be taken simultaneously in an executed trace[5]. Besides the basic types, we also allow `assert(C)` for checking assertions, `exit` for signaling the termination of a thread, and the synchronization primitives. `fork(t)` and `join(t)` allow a thread to dispatch and wait for the completion of another Thread t. Given a program written in a full-fledged programming languages like C, one can use pre-processing [21] to simplify its executed traces into the basic statements described above.

3.1 Under-Approximation FOL Formula $\varphi(\pi)$

The key to the TDV algorithm is the construction of appropriate FOL formulas that can be easily checked with SMT solvers.

Let V_G and $V_L(t)$ denote the set of global and local variables in Thread t, respectively. Let the set of variables visible to t be $V(t) = V_G \cup V_L(t)$. In addition to program variables, we introduce a statement location variable L_t for each thread, whose domain includes all the possible line numbers and occurrence indices. To model nondeterminism in the scheduler, we add a variable T whose domain is the set of thread indices. A transition in Thread t is executed only when $T = t$. At every transition step we add a fresh copy for each variable. That is, $v[i]$ denotes the copy of v at the i-th step. Given an executed trace π, $\varphi(\pi)$ consists of following constraints:

- **Program transition constraint** δ_π that expresses the effect of executing a particular statement of the program by a particular thread. For each tuple $(t, l.o, Q)$ except when Q is $exit$, we assume the next tuple to be executed by Thread t is $(t, l'.o', Q')$. Once the last tuple $(t, l.o, exit)$ of Thread t has been executed, we use $\triangle$ to indicate the end of Thread t. Let $\delta_{t,l.o}[i]$ denote the constraints of $(t, l.o, Q) \in \pi$ at step i. Fig. 3 shows the encoding for different types of tuples. For example, the one for $(t, l.o, v = E)$ states that if Thread t executes the statement at step i, the following updates occur at step $i + 1$:
  1. the next statement for Thread t to execute is $l'.o'$;
  2. the value of v at step $i + 1$ is $E|_{V \to V[i]}$ with all variables in E replaced by their corresponding versions at step i; and
  3. other visible variables remain unchanged.

[4] For nonterminating programs, our procedure can be used as a bounded analysis tool to search for bugs up to a bounded number of execution steps.

[5] A conditional branch such as `if C then` $l_1 : \ldots$ `else` $l_2 : \ldots$ results in the executed trace $C?l_1$ if the `then` branch is executed, and $\neg C?l_2$ otherwise.

The program transition constraint δ_π is defined as

$$\delta_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l.o)} \delta_{t,l.o}[i] \tag{1}$$

assignment: (t, l.o, v = E)
$T[i] = t \wedge L_t[i] = l.o \rightarrow$
$L_t[i+1] = l'.o' \wedge v[i+1] = E|_{V \rightarrow V[i]} \wedge V(t) \backslash v[i+1] = V(t) \backslash v[i]$
conditional branch: (t, l.o, C?l'.o')
$T[i] = t \wedge L_t[i] = l.o \wedge C|_{V \rightarrow V[i]} \rightarrow L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i]$
thread termination: (t, l.o, exit)
$T[i] = t \wedge L_t[i] = l.o \rightarrow L_t[i+1] = \Delta \wedge V(t)[i+1] = V(t)[i]$
unconditional jump: (t, l.o, goto l'.o')
$T[i] = t \wedge L_t[i] = l.o \rightarrow L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i]$
thread fork: (t, l.o, fork(t'))
$T[i] = t \wedge L_t[i] = l.o \rightarrow$
$L_t[i+1] = l'.o' \wedge L_{t'}[i+1] = s_{t'} \wedge V(t)[i+1] = V(t)[i]$
thread join: (t, l.o, join(t'))
$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge L_{t'}[i] = \Delta \rightarrow \\ L_t[i+1] = l'.o' \wedge V(t)[i+1] = V(t)[i] \end{pmatrix} \wedge$
$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge L_{t'}[i] \neq \Delta \rightarrow \\ L_t[i+1] = l.o \wedge V(t)[i+1] = V(t)[i] \end{pmatrix}$
lock: (t, l.o, lock(lk))
$\begin{pmatrix} T[i] = t \wedge L_t[i] = l.o \wedge \neg lk[i] \rightarrow \\ L_t[i+1] = l'.o' \wedge lk[i+1] = true \wedge V(t) \backslash lk[i+1] = V(t) \backslash lk[i] \end{pmatrix} \wedge$
$(T[i] = t \wedge L_t[i] = l.o \wedge lk[i] \rightarrow L_t[i+1] = l.o \wedge V(t)[i+1] = V(t)[i])$
unlock: (t, l.o, unlock(lk))
$T[i] = t \wedge L_t[i] = l.o \rightarrow$
$L_t[i+1] = l'.o' \wedge lk[i+1] = false \wedge V(t) \backslash lk[i+1] = V(t) \backslash lk[i]$

Fig. 3. Program transition constraints. $T[i]$ is the active thread at step i; $L_t[i]$ ($L_t[i+1]$) is the statement location at step i ($i+1$); $E|_{V \rightarrow V[i]}(C|_{V \rightarrow V[i]})$ substitute all variables in $E(C)$ by the by their corresponding versions at step i; $V(t) \backslash v[i+1] = V(t) \backslash v[i]$ denotes all visible variables in t keep their value except variable v.

– **Initial condition constraint** ι_π that specifies the starting locations for each thread as well the initial values of program variables, including the values set by the input vector.

– **Trace enforcement constraint** ε_π that restricts the encoded behavior to include only the statements appearing in an executed trace π. For each $(t, l.o, C?l'.o') \in \pi$ we assume condition C holds on line l at o-th occurrence in π. Then we have

$$\varepsilon_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l.o)} (T[i] = t \wedge L[i] = l.o \rightarrow C|_{V \rightarrow V[i]}) \tag{2}$$

– **Thread control constraint** τ_π that (1) insures that the local state of a thread (the values of its local variables) remains unchanged when the thread is not executing, and (2) insures that the thread cannot be selected for execution after it has terminated. These two constraints are specified in Equation 3.

$$\begin{array}{l} \tau_{t,idle}[i] \;\equiv\; T[i] \neq t \rightarrow L_t[i+1] = L_t[i] \wedge V_L(t)[i+1] = V_L(t)[i] \\ \tau_{t,done}[i] \;\equiv\; L_t[i] = \Delta \rightarrow T[i] \neq t \end{array} \tag{3}$$

The thread control constraint is defined as follows:

$$\tau_\pi \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{t=1}^{N} (\tau_{t,idle}[i] \wedge \tau_{t,done}[i] \wedge \tau_{other}) \tag{4}$$

In τ_{other}, additional optional constraints can be included to model particular scheduling policy.

– **Property constraint** ρ_P that indicates the correctness conditions, specified as assertions within the program in this paper, that we would like to check for validity under all possible executions. Note that many common programming errors can be modeled as assertions [21]. Let $(t, l, assert(C))$ be an assertion on line l in Thread t. The property constraint can be specified as follows:

$$\rho_P \equiv \bigwedge_{i=1}^{|\pi|} \bigwedge_{(t,l)} (T[i] = t \wedge L[i] = l \rightarrow C|_{V_C \rightarrow V_C[i]}) \tag{5}$$

Note that properties encoded by ρ_P are not necessarily the assertions appearing in π only; the assertions may appear anywhere in the program P. This is a crucial requirement for our trace-based method to find real failures anywhere in the program, or to prove the absence of assertion failures of the program.

Whether the property ρ_P holds for all possible thread interleavings in $\Pi_P(\pi)$ is determined by checking the validity of the formula: $\iota_\pi \wedge \delta_\pi \wedge \tau_\pi \wedge \varepsilon_\pi \rightarrow \rho_P$, which is equivalent to checking the *satisfiability* of the formula

$$\varphi(\pi) \equiv \iota_\pi \wedge \delta_\pi \wedge \tau_\pi \wedge \varepsilon_\pi \wedge \neg\rho_P \tag{6}$$

Equation 6, which implicitly represents all thread interleavings of $\Pi_P(\pi)$, is still an under-approximation of the behavior of program P under the given test input. Therefore, a solution to $\varphi(\pi)$ reveals real errors in the program, but the unsatisfiability of $\varphi(\pi)$ does not prove the absence of errors.

3.2 Over-Approximation FOL Formula $\psi(\pi)$

Let $\Pi_P(\overrightarrow{v})$ be the set of all possible execution traces of program P under the test input $\overrightarrow{v}$. The set of interleavings considered by $\varphi(\pi)$ is $\Pi_P(\pi) \subseteq \Pi_P(\overrightarrow{v})$.

To catch assertion violations in branches not yet executed in π, or to establish the absence of such violations in all traces, we need an over-approximation of $\Pi_P(\overrightarrow{v})$. The over-approximated encoding can be obtained from $\varphi(\pi)$ with the following changes:

- Remove the trace enforcement constraint ε_π that prohibits any trace $\pi' \notin \Pi_P(\pi)$ from being considered in $\varphi(\pi)$. In Fig. 1, for example, a trace starting from $\langle 1, 6, 7, 2, 3, \ldots \rangle$ can be a valid execution according to the program. However, the ε_π constraint $T[i] = 1 \wedge L[i] = 2 \rightarrow y[i] \geq 2$ prohibits the trace from being considered.
- Collapse multiple occurrences. For statements that occur more than once, we consider only one instance in the transition constraint. Thus the occurrence index o is no longer needed. This leads to a modified transition constraint δ_π^o.
- Add control flow constraints λ_π for un-executed statements. λ_π keeps the control flow logic but ignores the data logic in those statements that do not occur in π. The purpose of λ_π is to force the over-approximated behavior to at least follow the control flow logic of program P. Here we consider assignments and conditional branches. Given a conditional branch $(t, l, C?l_1 : l_2) \notin \pi$ that executes l_1 next if C is true and l_2 next otherwise, we add a constraint to $\lambda_\pi[i]$:

$$T[i] = t \wedge L_t[i] = l \rightarrow L_t[i+1] = l_1 \vee L_t[i+1] = l_2. \tag{7}$$

  Similarly, for an assignment statement $(t, l, v = E) \notin \pi$ that executes l_1 next, the constraint added to $\lambda_\pi[i]$ is

$$T[i] = t \wedge L_t[i] = l \rightarrow L_t[i+1] = l_1 \tag{8}$$

After the modifications above we obtain the following over-approximation:

$$\psi(\pi) \equiv \iota_\pi \wedge \delta_\pi^o \wedge \tau_\pi \wedge \lambda_\pi \wedge \neg\rho_P \tag{9}$$

Let $\Omega(\pi)$ be the set of interleavings considered by ψ_π; then $\Omega(\pi) \supseteq \Pi_P(\overrightarrow{v})$ is an over-approximation of the program behavior under the test vector $\overrightarrow{v}$. In general, the unsatisfiability of $\psi(\pi)$ proves P has no assertion failures under the test vector $\overrightarrow{v}$. The downside of using $\psi(\pi)$ is the inevitability of invalid executions which need to be filtered out afterwards. In the running example in Fig. 1, the SMT solver may report π_6 in Table 1 as a satisfiable solution of $\psi(\pi)$. However, it is not a feasible trace since the behavior of the step in line 2 is unspecified in $\psi(\pi)$ when $y < 2$.

4 Refinement

4.1 Analysis-Guided Execution

Let CEX_π be a satisfiable assignment to all variables in $\psi(\pi)$; it is called a potential counterexample. In the counterexample guided abstraction refinement

(CEGAR) framework, a decision procedure (theorem prover, SAT solver, or BDDs) [8, 2, 1, 27] has been used to check whether CEX_π is feasible in P, and if not, to refine the over-approximation. Such an approach may not be scalable for handling multithreaded software due to the program complexity and the length of the counterexamples.

Instead, we use guided concrete execution rather than a theorem prover or a SAT solver. Let $T = \cup_{i=1}^{|\pi|}\{T[i]\}$ be the set of thread selection variables at all time steps, and let $L = \cup_{i=1}^{|\pi|} \cup_{t=1}^{N} \{L_t[i]\}$ be the set of line number variables. Given CEX_π, we first extract a thread schedule $SCH_\pi = \exists_{v \in \{T \cup L\}}.CEX_\pi$, and organize it as a sequence

$$\pi_{SCH} = \langle (t_1, l_1), (t_2, l_2), \ldots, (t_{|\pi|}, l_{|\pi|}) \rangle \ .$$

Note that the occurrence index is not needed as the sequence uniquely identifies a trace (although it may be infeasible). The program is then re-executed by trying to follow π_{SCH}; this is implemented by using check-point and restart techniques as in [30]. If the re-execution can follow π_{SCH} to completion, then π_{SCH} represents a real bug. Otherwise, we obtain a new executed trace

$$\pi' = \langle (t_1, l_1.o_1), \ldots, (t_{k-1}, l_{k-1}.o_{k-1}), (t'_k, l'_k.o'_k), \ldots, (t'_{|\pi'|}, l'_{|\pi'|}.o'_{|\pi'|}) \rangle \ .$$

π and π' have the same thread ids and line numbers for the first $k-1$ steps. But starting from the k-th step π' can no longer follow π and completes the execution on its own.

To sum up, by performing a guided execution after analyzing the over-approximation $\psi(\pi)$, we are able to either validate the potential counterexample CEX_π, or obtain a new execution π' for a further analysis.

4.2 Avoid Redundant Checks

To avoid performing symbolic analysis on executed traces that have been analyzed before, we maintain a set χ of already inspected traces. Let $\{\pi_1, \ldots, \pi_m\}$ be the set of executed traces in the first m iterations that have been analyzed. If $\psi(\pi_m)$ is satisfiable, we are only interested in a solution $\overrightarrow{S}$ such that the trace $\pi_{\overrightarrow{S}}$ corresponding to $\overrightarrow{S}$ satisfies $\pi_{\overrightarrow{S}} \notin \Pi_P(\pi_i)$ for all $1 \leq i \leq m$. Such requirement is not only for performance, but also for the termination of the algorithm: without χ our algorithm may analyze the same executed trace infinitely.

Let π_t be a subsequence of π that is executed by Thread t. For two such subsequences π_t^1 and π_t^2 from two different executed traces, if they visit the same set of branch statements in t and have the same truth value of the conditionals at each branch, then $\pi_t^1 \equiv \pi_t^2$ (same statements are visited in the same order). Therefore, the trace enforcement constraint ε_{π_t} uniquely identifies a trace π_t in Thread t. As $\Pi_P(\pi)$ is the interleavings among the traces $\pi_{t_1}, \ldots, \pi_{t_N}$, they are identified by $\varepsilon_\pi = \varepsilon_{\pi_{t_1}} \wedge \ldots \wedge \varepsilon_{\pi_{t_N}}$. In the other words, in order to find a trace not in $\Pi_P(\pi)$, we must add the constraint $\neg\varepsilon_\pi$. Assume $\{\pi_1, \ldots, \pi_m\}$ are the

traces that have been executed so far, we have

$$\chi_m \equiv \bigwedge_{k=1}^{m} \neg\varepsilon_{\pi_k}. \tag{10}$$

The over-approximation formula at the $(m+1)$-th iteration becomes

$$\psi(\pi) \equiv \iota_\pi \wedge \delta_\pi^o \wedge \tau_\pi \wedge \lambda_\pi \wedge \chi_m \wedge \neg\rho_P. \tag{11}$$

4.3 An Illustrative Example

Fig. 4 shows a program with two methods `foo` and `bar`. At Line 0 `foo` creates a new thread and invoke `bar`. There is a recursive call on Line 3 in `foo`, therefore, multiple threads may be created depending on the input value of a. In the program, x and y are global variables with initial value 1, while a and b are thread local variables. We would like to check whether there can be an assertion failure on Line 11 under the test value $a = 1$.

```
foo(int a) {
0    create a new thread t to invoke bar(1);
1    x = a;
2    if (x>0)
3        foo(a-1);
4    else
5        if (y<=1 && y! =0)
6            x = y-x;
7        else if (y > 10)
8            complexA();
9        else
10           x = x-y;
11           assert(0);
12   wait for t to complete;
}
```

```
bar(int b) {
13   y = b;
14   if (y>0)
15       x = x-y;
16       y = y-1;
17   else
18       complexB();
}
```

Fig. 4. A program with recursion and dynamically created threads.

Assume the first executed trace is $\pi_1 = \langle(1,0.1),(1,1.1),(1,2.1),(1,3),(1,0.2),(1,1.2),(1,2.2),(1,5),(1,6),(2,13),(2,14),(2,15),(2,16),(3,13),(3,14),(3,15),(3,16),(1,12.1),(1,12.2)\rangle$, in which Thread 1 creates Thread 2 and 3 that execute $bar(1)$. Note that in π_1 we drop the occurrence index if a statement of a thread occurs only once. An under-approximated symbolic analysis on π_1 does not yield an assertion violation, but the over-approximated symbolic analysis produces a counter-example $CEX_1 = \langle(1,0),(1,1),(2,13),(2,14),(2,15),(1,2),(1,5),(1,7),(1,10),(1,11)\rangle$, which leads to an assertion failure on Line 11. An execution following CEX_1 shows that the counterexample is spurious as it can only follow

up to $(1,5)$, because the else branch on Line 5 cannot be taken. The complete executed trace is $\pi_2 = \langle(1,0),(1,1),(2,13),(2,14),(2,15),(1,2),(1,5),(1,6),(2,16),(1,12)\rangle$. There is no assertion failure in π_2, but the counterexample obtained from the over-approximated analysis is $CEX_2 = \langle(1,0),(1,1),(2,13),(2,14),(2,15),(2,16),(1,2),(1,5),(1,7),(1,10),(1,11)\rangle$. A further execution is able to follow the complete trace of CEX_2 and therefore reveals a real assertion failure on line 11.

5 Optimizations

We apply *peephole partial order reduction* (PPOR) [29] to exploit the equivalence of interleavings due to independent transitions. Unlike classical partial order reduction [17, 15], PPOR is able to reduce the search space symbolically in an SMT solver.

Given an executed trace $\pi = \langle(t_1, l_1.o_1, Q_1), \ldots (t_n, l_n.o_n, Q_n)\rangle$, we add a special scheduling constraint for every pair of tuples $(t_p, l_p.o_p, Q_p)$ and $(t_q, l_q.o_q, Q_q)$ such that $t_p \neq t_q$ and Q_p and Q_q are not dependent. Two statements are dependent if they access the same shared variable and at least one access is a write. For example, consider two statements $Q_p : a[k1] = e_1$ and $Q_q : a[k2] = e_2$ that are independent if the array index expressions do not have the same value. We add the following constraint to $\varphi(\pi)$:

$$L_p[i] = l_p.o_p \wedge L_q[i] = l_q.o_q \wedge k1|_{V \to V[i]} \neq k2|_{V \to V[i]} \rightarrow \neg(T[i] = q \wedge T[i+1] = p), \tag{12}$$

which prohibits Q_p being executed immediately after Q_q. Similar constraints can be added to over-approximated satisfiability formula $\psi(\pi)$.

Another optimization is a new thread-local static single assignment (TL-SSA) form to efficiently encode the thread-local statements. TL-SSA can significantly reduce the number of variables and the number of constraints needed in $\varphi(\pi)$ and $\psi(\pi)$, which are crucial since they often directly affect the performance of an SMT solver. Our observation is that the encoding in Section 3 may produce many redundant variables and constraints, due to the fact that it has to assign a fresh copy to every variable at every step. However, statements involving only local variables do not need a fresh copy of the local variables and constraints at every step. Furthermore, in a typical program execution, each statement writes to one variable at a time; a vast number of constraints, in the form of $v[i+1] = v[i]$, are used to keep the current values of the uninvolved variables.

In a purely sequential program, one can use Static Single Assignment (SSA) form [9] to simplify the encoding of a SAT formula [7]. However, SSA is not meant to be used in multithreaded programs (it remains an open problem as to what a SSA-style IR should be for concurrent programs), since a *use-define* chain for any shared variable cannot be established at compile time. Our observation here is that, while shared global variables cannot take advantages of the SSA form, local variables can still utilize the reduction power of SSA. The proposed TL-SSA form exploits the fact that, in any particular execution trace, the *use-define* chain of every *local* variable can be determined. Consider an executed trace

snippet $\langle \ldots$ (y=a+1), ..., (a=y), ..., (y=y+a)$\rangle$, where y is a shared variable and a is a local variables. In addition, no other statements in the trace access a. The trace with corresponding sequence of TL-SSA statements are $\langle \ldots$ (y=a_0+1),..., (a_1=y), ..., (y=y+a_1)$\rangle$. Instead of creating fresh copies for local variables at every step,the TL-SSA form creates only two copies of a. In addition, there is no need for the constraints $a[i+1] = a[i]$ to keep the value of a at each step where a is not assigned.

6 Related Work

Since we are not the first in modeling high-level source code semantics using a constraint language, it is helpful to briefly mention some of the successful approaches that have been reported. Noting the large gap between high-level programming languages and those of the formal logics, existing symbolic model checking tools, including [2, 7, 21], often restrict their representations to the pure Boolean domain; that is, they extract a Boolean-level model from the given program and then apply Binary Decision Diagrams (BDDs) [4] or SAT solvers (e.g., [11]) to perform verification. Although modeling all variables as bit-vectors is accurate, such high-precision approaches are often not needed and may generate models that are too large. In addition, bit vectors cannot model floating point arithmetic. In [32], sequential C programs are modeled at the word, as opposed to the bit, level using polyhedral analysis. This approach was shown to be very competitive for handling sequential C programs of non-trivial sizes. Unlike [32] that uses polyhedra library Omega [26] to perform reachability computation, we leverage the recently-demonstrated performance advances of SMT solvers to perform satisfiability checking.

Approaches based on similar ideas that augment testing with formal analysis include [20, 18, 6, 24, 28]. While Synergy [20] considers only sequential programs, we concentrate on multithreaded programs. Concolic testing [18, 6, 24] runs symbolic executions simultaneously with concrete executions, but the purpose is to generate new test inputs for better path coverage. In our approach, the purpose of symbolic analysis is to consider all related feasible thread interleavings implicitly, and in the event of inconclusive results, to guide the next concrete execution to follow a different thread schedule (that obeys program control flow semantics) under the same test vector. Predictive analysis [28] encodes a single execution symbolically without further refinement. The approach that augments testing with formal analysis has also been applied in other domains such as MCAPI [13, 12] and service computing [14].

Although integrated under- and over-approximations have been used in a decision procedure [3] for bit-vector arithmetic, most previous works on hardware and software model checking follow the paradigm of CEGAR [22, 8], which is based solely on over-approximations and uses spurious counterexamples to refine the over-approximations. In [19], Grumberg *et al.* presented a software model checking procedure based on a series of under-approximations.

Table 2. Bounded model checking (BMC) v.s trace-driven verification (TDV) for the multithreaded program in Fig. 4

	BMC		TDV		
#threads	mem(Mb)	time(s)	mem(Mb)	time(s)	speedup
5	21.15	1.16	20.14	1.07	1.08
10	54.29	3.18	51.92	3.15	1.01
15	129.11	66.01	100.72	7.64	8.64
20	219.34	169.84	166.81	18.69	9.09
25	317.40	215.87	250.13	44.33	4.87
30	420.54	222.85	348.18	45.83	4.86
35	538.75	140.21	461.85	42.11	3.33
40	692.62	150.66	597.55	73.69	2.04
45	-	-	745.86	77.79	-
50	-	-	906.9	143.27	-
55	-	-	1106.1	122.93	-
60	-	-	1330.5	182.53	-
65	-	-	1510.4	222.87	-
70	-	-	1737.5	289.86	-
75	-	-	2003.6	438.82	-
80	-	-	2270.1	407.07	-

There are several research projects that target concurrent program verification directly. Inspect [30] and CHESS [25] can check multithreaded C/C++ programs by explicitly executing different interleavings using dynamic partial order reduction [16]. However, explicitly exploring the thread interleavings does not scale well in the presence of a large number of (equivalence classes of) interleavings. The recent development in CHESS [25] also allows the tool to perform context bounded model checking. However, it is not intuitive to ask from user for a preset value on the number of context switches. CheckFence [5] checks all concurrent executions of a given C program on a relaxed memory model and verifies that they are observationally equivalent to a sequential execution, which targets a different application than ours.

7 Experiments

We have implemented a prototype of TDV using the Yices SMT solver [10], which is capable of deciding formulas with a combination of theories including propositional logic, linear arithmetic, and arrays. We performed two case studies. The first case study is on the example shown in Fig. 4 with multiple threads and recursions, and the second case study is on a file system implementation, which was previously used in [16]. Our experiments were conducted on a workstation with Pentium D 2.8 GHz CPU and 4GB memory running Red Hat Linux 7.2.

Table 2 shows the results of the first case study. By changing the value of the test variable a, we can increase the number of threads and the level of recursion.

Table 3. Bounded model checking (BMC), trace-driven verification (TDV), and trace-driven verification with optimizations (TDVO)

filesystem example			BMC		TDV			TDVO		
#threads	depth	prop	mem	time	mem	time	speedup	mem	time	speedup
2	10	sat	13.51	34.8	8.86	15.7	2.22	7.58	1.7	20.47
2	16	sat	42.72	665.0	18.14	126.0	5.28	12.82	9.4	70.74
2	22	sat	40.56	2324.6	23.44	212.5	10.94	25.37	15.9	91.63
3	21	sat	194.21	49642.3	42.77	1823.1	27.23	30.95	381.8	130.02
2	10	unsat	7.50	7.2	5.36	1.03	6.99	5.27	0.26	27.69
2	16	unsat	15.85	824.9	13.37	82.7	9.97	7.76	1.24	665.24
3	15	unsat	73.07	9488.7	11.83	122.7	77.33	9.16	8.1	1171.44

Column 1 lists the number of threads. Columns 2 and 3 show the peak memory and total time usage for Bounded Model Checking (BMC) without dynamic execution and abstraction. Columns 4 and 5 show the peak memory and total time usage for TDV. Note that optimizations has been applied to both methods. The last Column shows the speedup of the new method. A one-hour timeout limit is used in all the experiments. BMC ran out of time for test cases with more than 50 threads, while our method took only 407 seconds to complete 80 threads.

We also performed the experiments on the file system example, which is derived from a synchronization idiom found in the Frangipani file system. Table 3 shows the results we obtained by comparing BMC and TDV, both without and with optimizations. The results show that TDV gains a speedup from 1.46 to 77.33 over BMC, and the TDV with optimizations gains a speedup from 5.87 to 1171.44 over BMC, with an average speedup of 299.

8 Conclusion and Future Work

We have presented a new method to combine the efficiency of dynamic executions with the reasoning power of an SMT solver for the verification of safety properties of multithreaded programs. The main contributions are (1) a new symbolic encoding of executions of a multithreaded program, (2) using both under- and over-approximations in the same trace-driven abstraction framework, where refinement involving the mutual guidance between concrete program execution and symbolic analysis. For future work, we plan to investigate performance enhancement techniques, such as minimal unsatisfiable core analysis [23] and dynamic path reduction [31], to allow TDV to scale to larger programs.

References

1. Z. Andraus and K. Sakallah. Automatic abstraction and verification of verilog models. In *Design Automation Conference (DAC)*, pages 218–223, San Diego, California, 2004.

2. T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Automatic predicate abstraction of c programs. In *Programming Language Design and Implementation*, pages 203–213, 2001.
3. R. Bryant, D. Kroening, J. Ouaknine, S. A. Seshia, O. Strichman, and B. Brady. Deciding bit-vector arithmetic with abstraction. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, March 2007.
4. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8), 1986.
5. S. Burckhardt, R. Alur, and M. M. K. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *Programming language design and implementation*, pages 12–21, New York, NY, USA, 2007. ACM Press.
6. C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, and D. Engler. EXE: automatically generating inputs of death. In *ACM conference on Computer and communications security*. ACM, 2006.
7. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004. LNCS 2988.
8. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification*, pages 154–169, 2000.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
10. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *Computer-Aided Verification conference*, pages 81–94. Springer, 2006. LNCS 4144.
11. N. Een and N. Sorensson. An extensible sat-solver. In *Satisfiability Workshop*, 2003.
12. M. Elwakil and Z. Yang. Debugging Support Tool for MCAPI Applications. In *Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, 2010.
13. M. Elwakil, Z. Yang, and L. Wang. CRI: Symbolic Debugger for MCAPI Applications. In *The 8th International Symposium on Automated Technology for Verification and Analysis*. Springer-Verlag, 2010.
14. M. Elwakil, Z. Yang, L. Wang, and Q. Chen. Message race detection for web services by an smt-based analysis. In *The 7th International Conference on Autonomic and Trusted Computing*. Springer-Verlag, 2010.
15. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of programming languages*, pages 110–121, 2005.
16. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. *SIGPLAN Not.*, 40(1):110–121, 2005.
17. P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*. Springer, 1996. LNCS 1032.
18. P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI*, pages 213–223, 2005.
19. O. Grumberg, F. Lerda, O. Strichman, and M. Theobald. Proof-guided underapproximation-widening for multi-process systems. *SIGPLAN Notices*, 40(1):122–131, 2005.
20. B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. SYNERGY: a new algorithm for property checking. In *Foundations of software engineering*, pages 117–127, 2006.

21. F. Ivančić, I. Shlyakhter, A. Gupta, M. Ganai, V. Kahlon, C. Wang, and Z. Yang. Model checking C program using F-Soft. In *IEEE International Conference on Computer Design*, San Jose, CA, Oct. 2005.
22. R. P. Kurshan. *Computer-aided verification of coordinating processes*. Princeton University Press, Princeton, NJ, USA, 1994.
23. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008.
24. R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*. IEEE, 2007.
25. M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *SIGPLAN Not.*, 42(6):446–455, 2007.
26. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *ACM/IEEE conference on Supercomputing*, pages 4–13. ACM, 1991.
27. C. Wang, H. Kim, and A. Gupta. Hybrid CEGAR: combining variable hiding and predicate abstraction. In *International Conference on Computer Aided Design*, pages 310–317, 2007.
28. C. Wang, S. Kundu, M. Ganai, and A. Gupta. Symbolic predictive analysis for concurrent programs. In *the 2nd World Congress on Formal Methods*, pages 256–272, Berlin, Heidelberg, 2009. Springer-Verlag.
29. C. Wang, Z. Yang, V. Kahlon, and A. Gupta. Peephole partial order reduction. In *Tools and algorithms for the construction and analysis of systems (TACAS'08)*, 2008.
30. Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Runtime model checking of multithreaded C/C++ programs. Technical Report UUCS-07-008, School of Computing, University of Utah, 2007.
31. Z. Yang, B. Al-Rawi, K. Sakallah, X. Huang, S. Smolka, and R. Grosu. Dynamic path reduction for software model checking. In *The 7th International Conference on Integrated Formal Methods*, 2009.
32. Z. Yang, C. Wang, A. Gupta, and F. Ivanvčić. Model checking sequential software programs via mixed symbolic analysis. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–26, 2009.