Verifying Multi-threaded Software using SMT-based Context-Bounded Model Checking

Lucas Cordeiro
University of Southampton
lcc08r@ecs.soton.ac.uk

Bernd Fischer
University of Southampton
b.fischer@ecs.soton.ac.uk

ABSTRACT

We describe and evaluate three approaches to model check multi-threaded software with shared variables and locks using bounded model checking based on Satisfiability Modulo Theories (SMT) and our modelling of the synchronization primitives of the Pthread library. In the lazy approach, we generate all possible interleavings and call the SMT solver on each of them individually, until we either find a bug, or have systematically explored all interleavings. In the schedule recording approach, we encode all possible interleavings into one single formula and then exploit the high speed of the SMT solvers. In the underapproximation and widening approach, we reduce the state space by abstracting the number of interleavings from the proofs of unsatisfiability generated by the SMT solvers. In all three approaches, we bound the number of context switches allowed among threads in order to reduce the number of interleavings explored. We implemented these approaches in ESBMC, our SMT-based bounded model checker for ANSI-C programs. Our experiments show that ESBMC can analyze larger problems and substantially reduce the verification time compared to state-of-the-art techniques that use iterative context-bounding algorithms or counter-example guided abstraction refinement.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Model checking; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Mechanical verification

General Terms

Computer-Aided Verification

Keywords

Formal Software Verification, SAT Modulo Theories, Symbolic and Explicit Model Checking, Multi-threaded systems

1. INTRODUCTION

Bounded model checking (BMC) has already been successfully applied to verify software and to discover subtle errors in real systems [3]. In an attempt to cope with growing system complexity, Boolean Satisfiability (SAT) solvers are increasingly replaced by Satisfiability Modulo Theories (SMT) solvers to prove the generated verification conditions (VCs) [1, 8, 12]. There have also been attempts to extend BMC to the verification of multi-threaded software [13, 16, 17, 25]. The main challenge here is the state space explosion, as the number of possible interleavings grows exponentially with the number of threads and program statements. However, most concurrency bugs in real applications have been found to be shallow so that only a few context switches are required to expose them [23]. We can thus use a context-bounded analysis [18, 28] that limits the number of context switches it explores. Also, SAT and SMT solvers produce unsatisfiable cores that allow us to remove possible undesired models of the system in order to satisfy a given property [19]. Grumberg et al. [15] showed that the unsatisfiable cores can also be used to control the number of allowed interleavings of the given set of threads. They proposed a SAT-based BMC method to model check a multi-threaded system using a series of under-approximated models. However, their method does not use SMT solvers and does not combine context-bounded analysis with symbolic algorithms, which limits its usefulness for verifying multi-threaded software.

In our prior work [8], we extended the encodings from previous SMT-based BMC [1, 12] to provide more accurate support for variables of finite bit width, bit-vector operations, arrays, structures, unions and pointers, and implemented these in the ESBMC tool, which is built on top of the CBMC model checker [5]. Here, we continue this work and develop and evaluate three related approaches for model checking multi-threaded ANSI-C software. In contrast to previous fully symbolic approaches (e.g., [13, 16, 17, 25, 15]), we combine symbolic model checking with explicit state space exploration. In particular, we explicitly explore the possible interleavings (up to the given context bound) while we treat each interleaving itself symbolically. This approach is similar to the recent ESST approach by Cimatti et al. [4], but we handle ANSI-C instead of SystemC, use BMC instead of predicate abstraction, and place no restrictions on the scheduler. Our approaches all implicitly use the reachability tree (RT) derived from the system, but differ in the way they exploit it. In the *lazy* approach, we traverse the RT depth-first, and simply call the single-threaded BMC procedure on the interleaving whenever we reach an RT leaf node. We stop the

RT traversal either when we find a bug, or have systematically explored all interleavings. In the *scheduling recording* approach, we use the RT to encode all the possible execution paths into one single formula, which is then fed into the SMT solver. In a third approach, we extend the under-approximation and widening (UW) algorithm [15] with the purpose of addressing the verification of real-world C code using different background theories and SMT solvers.

We make two major novel contributions. First, we exploit SMT to improve BMC of multi-threaded software. We describe a comprehensive SMT-based BMC procedure to support the checking of multi-threaded C programs that use the synchronization primitives of the POSIX Pthread Library [20]. Second, we describe and evaluate three related approaches to SMT-based BMC. This work also marks the first application of the UW algorithm in combination with context-bounded model checking to verify non-trivial multi-threaded C software. Experiments obtained with the extended ESBMC show that our approaches can analyze larger problems and substantially reduce the verification time over state-of-the-art techniques that use iterative context-bounding algorithms and others that implement counter-example guided abstraction refinement (CEGAR) techniques.

2. PRELIMINARIES

In the widely adopted interleaving paradigm for multi-threaded programs, the notion of concurrency is represented by that of interleaving, i.e., the non-deterministic choice between activities of the simultaneously acting threads [2]. An interleaving thus represents a possible execution of the program where all of the concurrent events are arranged in a linear order. Any change of the active thread in an interleaving is a context switch.

The interleaving paradigm relies on a scheduler, which selects the concurrently executing threads according to a given strategy. This abstracts from the speed of the participating threads and thus models any possible realization by a single-core machine or by several cores with arbitrary speeds. However, in order to fully verify a multi-threaded program against a given specification, all possible interleavings must be considered. This results in a large state space that must be explored by a model checker.

2.1 Goto Programs

We consider multi-threaded ANSI-C programs in asynchronous mode and assume that all threads in the program only communicate through shared global variables. ESBMC handles full ANSI-C, but for presentation, we use a minimal language similar to the internal *goto*-language of the CBMC model checker [5]. It is expressive enough to model multi-threaded programs:

$$
\begin{aligned}
\mathit{Fml} &::= \mathit{Var} \mid \mathit{true} \mid \mathit{false} \mid \mathit{Fml} \wedge \mathit{Fml} \mid \ldots \mid \mathit{Exp} = \mathit{Exp} \mid \ldots \\
\mathit{Exp} &::= \mathit{Var} \mid \mathit{Const} \mid \mathit{Var}[\mathit{Exp}] \mid \mathit{Exp} + \mathit{Exp} \mid \ldots \\
\mathit{Cmd} &::= \mathit{skip} \mid \mathit{Var} = \mathit{Exp} \mid \mathit{Var} = * \mid \mathit{assume}\ \mathit{Fml} \\
&\mid \mathit{assert}\ \mathit{Fml} \mid \mathit{goto}\ l \mid \mathit{if}\ \mathit{Fml}\ \mathit{goto}\ l \mid \mathit{begin\_atomic} \\
&\mid \mathit{end\_atomic} \mid \mathit{begin\_thread}\ \mathit{Id} \mid \mathit{end\_thread} \\
&\mid \mathit{Var} = \mathit{start\_thread}\ \mathit{Id} \mid \mathit{join\_thread}\ \mathit{Var} \\
\mathit{Prog} &::= \mathit{Cmd}; \ldots; \mathit{Cmd}
\end{aligned}
$$

A goto-program is a (numbered) list of commands. Commands include assignments, nondeterministic assignments ($Var = *$), blocking statements (*assume*) to cut off subsequent executions paths, and assertion statements (*assert*) to indicate user-specified properties. All control structures are represented by explicit (conditional) jumps to a statement $l \in \{1, \ldots, n\}$. A thread t is a sublist of commands between *begin_thread* and *end_thread*. Threads are created via asynchronous procedure calls (*start_thread*), which return an integer that can be used as thread identifier for synchronization (*join_thread*); hence, dynamic thread creation is allowed. Atomic statements (*atomic_begin* and *atomic_end*) indicate that a code segment cannot be preempted by another thread. Figure 1 shows an example of a multi-threaded C program and its representation.

2.2 Formal Model of Multi-threaded Software

The multi-threaded software to be analyzed is modelled as a tuple $M = \langle S, S_0, T, V \rangle$, where:

- S is a finite set of states, with initial states $S_0 \subset S$;
- $T = t_0, t_1, ..., t_n$ is the set of threads, where n represents the total number of threads;
- $V = V_{global} \cup \bigcup V_j$ where V_{global} is the set of global variables and V_j is the set of local variables of t_j.

We assume that each variable ranges over a finite domain. A state $s \in S$ consists of the values of the global and local variables, including a local program counter for each thread. Each thread j is a tuple $t_j = (R^j, l^j)$, where:

- $R^j \subseteq S \times S$ is the transition relation of thread t_j;
- $l^j = \langle l_i^j \rangle$ is the sequence of thread locations l_i^j at time step i.

The execution of the instructions of each thread t_j is modelled by means of transition relations and we use the notation $R_i^j(s, s')$ to denote that s' is a successor of s obtained by executing at time step i an instruction of thread t_j. Finally, a particular program location, denoted by l_0^j is designated as the entry point of thread t_j.

3. CONTEXT-BOUNDED MODEL CHECKING OF MULTI-THREADED SOFTWARE

3.1 Exploring the Reachability Tree

In order to describe reachable states of a multi-threaded goto program, we use a reachability tree (RT) that is obtained by unfolding the set of running threads. For a multi-threaded program with n active threads, each node in the RT is a tuple $\nu = (A_i, C_i, s_i, \langle l_i^j, G_i^j \rangle_{j=1}^n)_i$ for a given time step i, where:

- A_i represents the currently active thread;
- C_i represents the context switch number;
- s_i represents the current state;
- l_i^j represents the current location of thread j;
- G_i^j represents the control flow guards accumulated in thread j along the path from l_0^j to l_i^j.

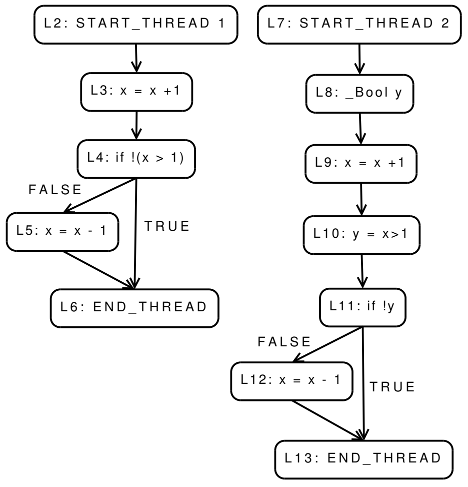
Since threads only communicate via global variables, we only need to consider context switches at visible instructions, i.e., synchronization points and statements containing global variables. As in [13], we do not model context switches inside individual visible statements. This is safe as long as the statements only read or write a single global variable, but in

```
#include <pthread.h>
int x=0;
void* t1(void* arg) {
  x++;
  if (x>1) x--;
  return NULL;
}
void* t2(void* arg) {
  _Bool y;
  x++;
  y = (x>1);
  if (y) x--;
  return NULL;
}
int main(void) {
  pthread_t id1, id2;
  pthread_create(&id1,NULL,t1,NULL);
  pthread_create(&id2,NULL,t2,NULL);
  pthread_join(&id1,NULL);
  pthread_join(&id2,NULL);
  assert(x==1)
  return 0;
}
```

(a)

```
int x = 0;
begin_thread t1;
  x = x + 1;
  if !(x > 1) then goto L6;
    x = x - 1;
L6: end_thread;
begin_thread t2;
  _Bool y;
  x = x + 1;
  y = x>1;
  if !(y) then goto L13;
    x = x - 1;
L13: end_thread;
id1 = start_thread t1;
id2 = start_thread t2;
join_thread id1;
join_thread id2;
assert(x==1);
return 0;
```

(b)



(c)

Figure 1: (a) A multi-threaded C program with an assertion violation. (b) The C program of (a) converted into goto form. (c) Control-flow graph of two threads of the goto program in (b).

general it is an under-approximation. However, we have not encountered any problems in the benchmarks we have used. Additionally, we do not model context switches between a visible control-flow test and the next visible statement, since the test cannot influence the state. We can simulate the effect of a context switch right after a visible test by hoisting the test out of the conditional, and assigning its result to a new auxiliary variable, as shown in thread t_2 in Figure 1(a). ESBMC can be configured to automatically insert such auxiliary variables. We also assume sequential consistency, as is common in model checking multi-threaded systems.

In order to expand the RT and explore all possible interleavings, we symbolically execute each instruction of the multi-threaded goto-program. This takes as input the program and the current RT node, and generates its children according to the set of rules described below. We assume that we expand an RT node ν at time step i and that the guard $G_i^{A_i}$ of the thread t_{A_i} is enabled in state s_i (i.e., that the corresponding formula is satisfiable), so that the thread can potentially execute the instruction I at location $l_i^{A_i}$. Note that our current implementation does not check the satisfiability of the accumulated guards, and simply assumes that all running threads are enabled, unless they have explicitly been blocked. Implementing this could further reduce the size of the RT to be explored.

R1: If I is an assignment $x = e$, then we symbolically execute I, which generates a new state s_{i+1}. We then add as child to ν a new node $\nu' = (A_i, C_i, s_{i+1}, \langle l_{i+1}^j, G_i^j\rangle)_{i+1}$, where the active thread remains unchanged. We increment the location of the active thread only (i.e., $l_{i+1}^{A_i} = l_i^{A_i} + 1$) and leave all other locations and all guards unchanged; however, note that the evaluation of the guards can change under the new state s_{i+1}, and hence threads may become enabled.

We have fully expanded ν if

- $l_i^{A_i}$ is within an atomic block; or
- I contains no global variable (since we allow context switches only at visible instructions); or
- we have reached the upper bound of context switches to be explored (i.e., $C_i = C$).

If ν is not yet fully expanded, we then also explore all context switches, up to the given context bound C. For each thread $j \neq A_i$ where G_i^j is enabled in s_{i+1}, we thus create a new child node $\nu_j' = (j, C_i + 1, s_{i+1}, \langle l_i^j, G_i^j\rangle)_{i+1}$. In ν_j' we then continue the RT exploration with thread j executing in the state produced by the current thread A_i.

R2: If I is a *skip*-statement or an unconditional *goto*-statement with target l, then we simply increment (resp. set) the location of the current thread and continue with it. We explore no context switches, i.e., we only add a single child node $\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, G_i^j\rangle)_{i+1}$, where $l_{i+1}^j = l_i^j + 1$ (resp. $l_{i+1}^j = l$) if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise.

R3: If I is a conditional *goto*-statement with test c and target l, then we create two child nodes ν' and ν'' for both possible outcomes of the test. For ν', we assume that c is true and proceed with the target instruction of the jump, similar to unconditional jumps. However, we also add c to the guards of all other threads, since it may contain global variables, and may thus enable or disable other transitions. Hence, we construct $\nu' = (A_i, C_i, s_i, \langle l_{i+1}^j, c \wedge G_i^j\rangle)_{i+1}$ (where $l_{i+1}^j = l$ if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise). For ν'', we add $\neg c$ to the guards and continue with the next instruction in the current thread, i.e., $\nu'' = (A_i, C_i, s_i, \langle l_{i+1}^j, \neg c \wedge G_i^j\rangle)_{i+1}$ (where $l_{i+1}^j = l_i^j + 1$ if $j = A_i$ and $l_{i+1}^j = l_i^j$ otherwise). We prune one of the nodes if the condition is determined in the current state (i.e., either evalutes to true or to false). Note that we are not exploring any possible context switches (even if I is visible), since the condition cannot change the global state.

R4: If I is an *assume*- or *assert*-statement with argument c, then we proceed similar to the way described in R1. We continue with the unchanged state s_i but add c to all guards, as described in R3. If I is an *assume*, and $c \wedge G_i^j$ evaluates to *false*, we prune the execution paths and if I is an *assert*, we generate a VC to check the validity of c.

R5: If I is a *start_thread* instruction, we just add the indicated thread to the set of active threads, i.e., we add a node $\nu' = (A_i, C_i, s_i, \langle l^j_{i+1}, G^j_{i+1}\rangle^{n+1}_{j=1})_{i+1}$, where l^{n+1}_{i+1} is the initial location of the indicated thread, and $G^{n+1}_{i+1} = G^{A_i}_i$, i.e., the thread starts with the guards of the currently active thread.
R6: If I is a *join_thread* instruction with argument *Id*, then we add a child node $\nu' = (A_i, C_i, s_i, \langle l^j_{i+1}, G^j_i\rangle)_{i+1}$, (where $l^j_{i+1} = l^{A_i}_i + 1$) only if the joining thread *Id* has exited. We model this by an additional variable $exit_j$ that is set to *false* when *begin_thread Id* is called. When *end_thread* is reached, we set $exit_j$ to *true* to indicate that thread *Id* has exited.

The remaining instructions (*begin_atomic*, *end_atomic*, *begin_thread*, and *end_thread*) are just scoping constructs and do not contribute to the expansion of the RT. As example, we consider the C program with two threads and the corresponding goto-program, as shown in Figure 1(a) and (b). This example is modified slightly from [14], where it is used to check (by increasing the number of increments) the scalability of different context-bounded analysis algorithms. Both threads increment a global variable x, and then, depending on the value of x, decrement it again. t_2 uses a local variable y to store the value of x and uses this in the test (cf. lines 12–13). This simulates a possible context switch between the evaluation of the guard and the execution of the next statement. Figure 1(c) shows the CFG representation of the two threads t_1 and t_2. Note that this example contains an assertion violation in line 23, where the invariant $x = 1$ does not hold under specific thread interleavings.

Figure 2 shows a fragment of the reachability tree for threads t_1 and t_2. We build this by first executing the goto-program of Figure 1(b) sequentially, i.e., in the same order that the threads are created. In this case, we first execute the statements of t_1 (i.e., lines 3-5), followed by the statements of t_2 (i.e., lines 8-12). The initial node of the RT fragment is $\nu_0 = (t_0, 0, s_0, \langle (L_{16}, true), (L_2, true), (L_7, true)\rangle)$, i.e., the main thread t_0 is active at line 16, the program is before the first context switch, the state s_0 has $x = 0$ and y undefined, and both threads t_1 and t_2 have just been started, i.e., are at their initial location with guards true. To expand the RT, we check which threads are enabled from ν_0. Since t_1 and t_2 are both enabled and since our approach always expands the enabled thread with the smallest index, we expand the transitions of t_1. The transition relation $R^1_1(s_0, s_1)$ of t_1 that represents the assignment $x = x + 1$ is defined as follows:

$$\begin{aligned} R^1_1(s_0, s_1) \quad \Leftrightarrow \quad & l^1_1 = L3 \wedge x_1 = x_0 + 1 \\ & \wedge \forall v \in V \setminus \{x\} : v_1 = v_0 \end{aligned}$$

The first term corresponds to the unconditional edge from line 2 to 3 (see Figure 1(c)). The second term defines the new value of the shared variable x. The third term ensures that the values of V, but not x, do not change in the transition from s_0 to s_1. To create node ν_1, we apply rule R1, which gives us $\nu_1 = (t_1, 1, s_1, \langle (L_{16}, true), (L_3, true), (L_7, true)\rangle)$. We then check again which threads are enabled and expand t_1 as the enabled thread with the smallest index. The transition relation that represents the branch at program location L4 is defined by a case-split on the value of x in state s_1.

$$\begin{aligned} R^1_2(s_1, s_2) \quad \Leftrightarrow \quad & l^1_2 = \begin{cases} L6 : \neg(x_1 > 1), \\ L5 : otherwise \end{cases} \\ & \wedge \forall v \in V : v_2 = v_1 \end{aligned}$$

The transition does not affect the global state (as the condition $\neg(x_1 > 1)$ holds), so we only increment the program location but do not create a new node in the RT (described in rule R3). Therefore, to expand the next node from ν_1, we check again which threads are enabled and since t_1 has executed all its statements, we then expand the first instruction of thread t_2. The transition relation $R^2_3(s_2, s_3)$ of t_2 is similar to $R^1_1(s_0, s_1)$. We thus apply rules R1 and R2 to derive $\nu_2 = (t_2, 2, s_2, \langle (L_{16}, true), (L_6, true), (L_9, true)\rangle)$. ν_3 and ν_4 are derived in the same way. After creating ν_4, both t_1 and t_2 do not have enabled transitions and we backtrack to explore pending transitions from previous nodes; in this case, we have already explored ν_3 and ν_2 and continue the RT exploration at ν_1.

3.2 Lazy Approach

The idea of the lazy approach to verify multi-threaded software is to traverse the RT depth-first, and to call the single-threaded BMC procedure on each interleaving whenever we reach an RT leaf node. We stop the RT traversal either when we find a bug, or have systematically explored all interleavings. Figure 3 details how the lazy approach works. Formally, given an RT $\Upsilon = \{\nu_1, \ldots, \nu_N\}$ that represents the program unfolding for a context bound C and a bound k, and a property ϕ, we derive a VC ψ^π_k for a given interleaving (or computation path) $\pi = \{\nu_1, \ldots, \nu_k\}$ such that ψ^π_k is satisfiable if and only if ϕ has a counterexample of depth k that is exhibited by π. The VC ψ^π_k is a quantifier-free formula in a decidable subset of first-order logic, which is checked for satisfiability by an SMT solver. The model checking problem associated with SMT-based BMC of a given π is formulated by constructing the logical formula [1, 12]:

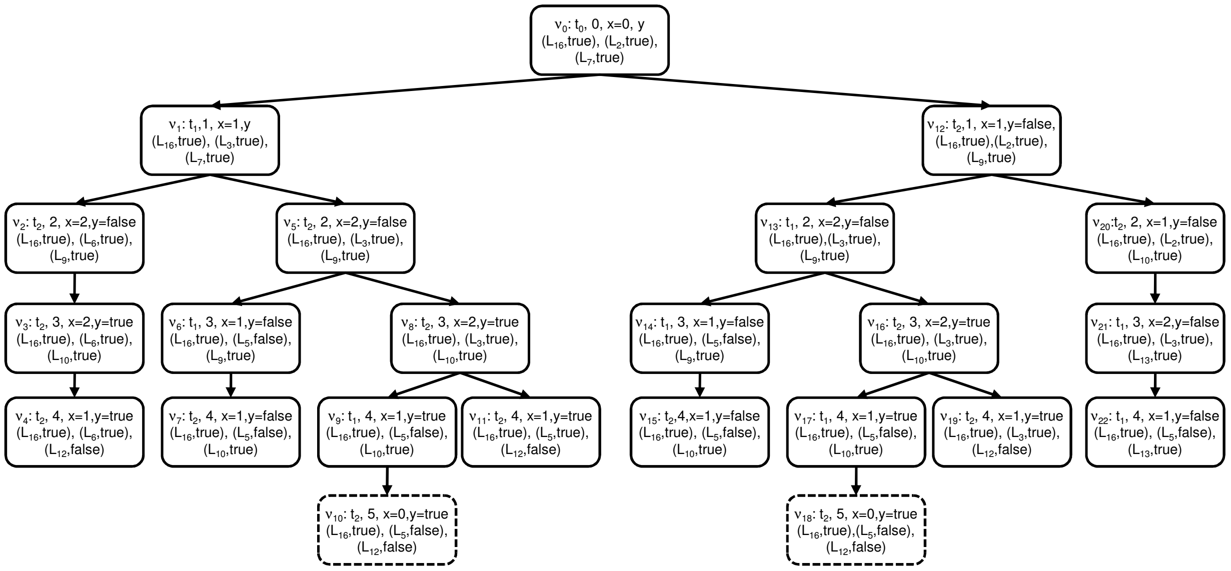
$$\psi^\pi_k = \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{constraints} \wedge \overbrace{\neg\phi_k}^{property} \qquad (1)$$

Here, ϕ_k represents a safety property ϕ in step k, I is the set of initial states and $R(s_i, s_{i+1})$ is the transition relation at time steps i and $i + 1$, as described by the states in the nodes of π. In order to check if the logical formula (1) is satisfiable or unsatisfiable, the SMT solver constrains some symbols by a given background theory (e.g., the theory of arithmetics restricts the interpretation of symbols such as $+$, $\leq$, 0, and 1) [10]. If (1) is satisfiable, then ϕ is violated and the SMT solver provides a satisfying assignment, from which we can extract the values of the program variables to construct a counterexample, i.e., a sequence of states $s_0, s_1, \ldots, s_k$ with $s_0 \in S_0$, and $R(s_i, s_{i+1})$ for $0 \leq i < k$. If (1) is unsatisfiable, we can conclude that no error state is reachable in length k along π.

Step 1: Initialize the stack with the initial node ν_0 and the initial path $\pi_0 = \langle \nu_0 \rangle$.
Step 2: If the stack is empty, terminate with "no error".
Step 3: Pop the current node ν and current path π off the stack and compute the set ν' of successors of ν using rules R1-R6.
Step 4: If ν' is empty, derive the VC ψ^π_k for π using formula (1), and call the SMT solver on it. If ψ^π_k is satisfiable, terminate with "error"; otherwise, goto step 2.
Step 5: If ν' is not empty, then for each node $\nu \in \nu'$, add ν to π, and push node and extended path on the stack. Goto step 3.

Figure 3: Algorithm of the lazy approach.

On the face of it, the lazy approach seems to be naive:



ν0: t0, 0, x=0, y (L16,true), (L2,true), (L7,true)
ν1: t1,1, x=1,y (L16,true), (L3,true), (L7,true)
ν12: t2,1, x=1,y=false, (L16,true),(L2,true), (L9,true)
ν2: t2, 2, x=2,y=false (L16,true), (L6,true), (L9,true)
ν5: t2, 2, x=2,y=false (L16,true), (L3,true), (L9,true)
ν13: t1, 2, x=2,y=false (L16,true),(L3,true), (L9,true)
ν20:t2, 2, x=1,y=false (L16,true), (L2,true), (L10,true)
ν3: t2, 3, x=2,y=true (L16,true), (L6,true), (L10,true)
ν6: t1, 3, x=1,y=false (L16,true), (L5,false), (L9,true)
ν8: t2, 3, x=2,y=true (L16,true), (L3,true), (L10,true)
ν14: t1, 3, x=1,y=false (L16,true), (L5,false), (L9,true)
ν16: t2, 3, x=2,y=true (L16,true), (L3,true), (L10,true)
ν21: t1, 3, x=2,y=false (L16,true), (L3,true), (L13,true)
ν4: t2, 4, x=1,y=true (L16,true), (L6,true), (L12,false)
ν7: t2, 4, x=1,y=false (L16,true), (L5,false), (L10,true)
ν9: t1, 4, x=1,y=true (L16,true), (L5,false), (L10,true)
ν11: t2, 4, x=1,y=true (L16,true), (L5,true), (L12,false)
ν15: t2,4,x=1,y=false (L16,true), (L5,false), (L10,true)
ν17: t1, 4, x=1,y=true (L16,true), (L5,false), (L10,true)
ν19: t2, 4, x=1,y=true (L16,true), (L3,true), (L12,false)
ν22: t1, 4, x=1,y=false (L16,true), (L5,false), (L13,true)
ν10: t2, 5, x=0,y=true (L16,true), (L5,false), (L12,false)
ν18: t2, 5, x=0,y=true (L16,true),(L5,false), (L12,false)

Figure 2: Fragment of the reachability tree of the multi-threaded goto-program of Figure 1(b). Nodes with dashed line represent program locations that violate the assertion statement in line 18 of Figure 1(b).

despite the context-bounding, the RT and thus the number of interleavings can grow very quickly, and we need to invoke the SMT solver several times to check the satisfiability of formula (1), which might slow down the verification process. However, there are several observations that make this approach worthwhile. First, if the program contains any errors at all, they will often be exhibited in a substantial fraction of the interleavings (cf. [23] for experience on real applications), so that in practice we only need to explore a small part of the search space until we find the first error. In our running example, the invariant $x = 1$ does not hold for the two nodes ν_{10} and ν_{18} and if we traverse the RT depth-first and left-to-right, the error already shows up in the third interleaving. Second, we do not need to actually build the entire RT; instead, we only keep in memory nodes on computation paths that are still unexplored and expand them one path at a time. We then construct the VC for the chosen computation path and feed it into the SMT solver to check for satisfiability. Third, and most important, we can leverage the optimizations from the CBMC front-end (e.g., constant propagation and folding) to exploit which transitions are enabled in a given state to drive the exploration of the interleavings and to reduce both the number of interleavings to be explored and the size of the formulas sent to the SMT solver. For example, if we continue to explore thread t_1 from node ν_1, the front-end exploits the fact that $x = 1$ to infer that the guard in line 4 holds. t_1 thus continues in line 6, and terminates, so that the exploration continues with a context switch to thread t_2, as shown in node ν_2.

In summary, the lazy approach guides the symbolic execution between the threads and systematically explore all the possible interleavings in a lazy way. This approach can find bugs fast, but as the front-end invokes the SMT solver, once for each possible computation path, it can suffer performance degradation, in particular for correct programs where we explore all possible interleavings.

3.3 Schedule Recording Approach

State-of-the-art SMT solvers are built on top of efficient SAT solvers to speed up the performance on large problems by exploiting the support for conflict clauses and non-chronological backtracking. In the schedule recording approach we leverage this and avoid invoking the SMT solver repeatedly. We build the RT as before to systematically explore the interleavings, but we now add schedule guards [17] to record in which order the scheduler has executed the program. Figure 4 shows how schedule guards are added to the program during the exploration of the left-hand side of the RT in Figure 2. We then encode all interleavings into a *single* formula, which is finally passed to the SMT solver.

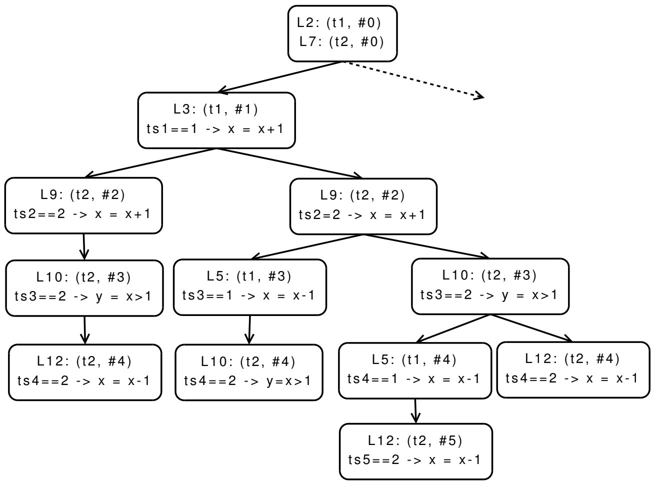


Figure 4: Schedule recording applied to the left-hand side of the RT in Figure 2.

Since control-flow tests cannot influence the state, we only need to add guards to *effective statements*, i.e., assignments and assertions. Similarly, we only need to record *effective context switches* (ECS), i.e., context switches to an effective statement. Each effective program statement is then prefixed by a schedule guard $ts_i = j$ where ts_i is the thread selection variable for the i-th ECS and j is the thread identifier. Its intuitive interpretation is that the statement can only be executed if thread j is scheduled to run after the i-th ECS. For example, the schedule guard $ts_1 = 1$ at L_3 encodes that $x = x + 1$ can only be executed if t_1 runs after the first ECS.

The schedule guards are added when program statements are executed symbolically and become part of the produced VCs. They can be derived from the RT nodes, i.e., for node ν_i we construct the guard $ts_{C_i} = A_i$. The thread selection variables are free variables that the SMT solver will instantiate with concrete values. The instantiation of all thread selection variables corresponds to the choice of a specific interleaving. In our example, if the SMT solver chooses $ts_1 = 1$, $ts_2 = 2$, $ts_3 = 2$, and $ts_4 = 2$, then the model checker simulates the effect of executing the program statements at L_3, L_9, L_{10}, and L_{12} (in that order). Note that the ordering of statements within a thread is of course still ensured by the program order semantics, so that program statement at L_{10} will not be executed before program statement at L_9. We further define a schedule SCH to determine which interleavings are considered and encode the guards in (2) as:

$$\psi_k \;=\; \overbrace{I(s_0) \wedge R(s_0, s_1) \wedge \ldots \wedge R(s_{k-1}, s_k)}^{constraints} \wedge \overbrace{\neg\phi_k}^{property}$$
$$\wedge \overbrace{SCH(s_0) \wedge \ldots \wedge SCH(s_k)}^{scheduler} \qquad (2)$$

Here $SCH(s_i)$ represents a constraint on the schedule guard of state s_i. If we do not add any constraints, then we formulate $\bigwedge_{i=0}^{k} SCH(s_i) = true$ and all possible interleavings are considered. However, if we want to apply aggressive reductions (for example by exploiting the proofs of unsatisfiability as described in the next subsection), we can add constraints to SCH to force the removal of interleavings that do not contribute to checking a given property. Although, we can bound the number of preemptions and exploit which transitions are enabled in a given state when we build formula (2), the number of threads and context switches can still grow very large quickly, and easily "blow-up" the solver.

3.4 UW Approach

The core idea of the under-approximation and widening (UW) approach is to check models with an increasing set of allowed interleavings [15]. We start from an underapproximation describing a single interleaving and widen the model by adding more interleavings incrementally based on the proof objects generated from an SMT solver [10]. We define ψ' as an underapproximated model of ψ, i.e., $\psi' = \psi \wedge SCH(s_0) \wedge \ldots \wedge SCH(s_k)$, where we introduce constraints on the schedule guards. We can see that if ψ is unsatisfiable, then ψ' is also unsatisfiable; however, it is possible that ψ is satisfiable while ψ' is not, due to the constraints on the schedule. Thus, ψ' can be thought of as an underapproximation of ψ and each satisfying assignment of ψ' is also a satisfying assignment to ψ. The main steps of the UW algorithm are shown in Figure 5.

The additional literals cl_{ij} introduce constraints on the schedule guards (e.g., $cl_{ij} \rightarrow ts_i = j$), which allow us to guide the widening process according to the variables that participate in the proof of unsatisfiability produced by the SMT solver. This means that the schedule is now updated based on the information extracted from the proof, which aims to remove interleavings that are not relevant for checking a given property [15, 19]. Note that the way that we encode the underapproximation differs from [15]. The authors in [15] encode an underapproximation using $m \times n$ control literals, where m is the number of control points that guard each program statement and n is the number of processes. In our encoding, we use $e \times n$ control literals, where e is the number of ECS (with $e \leq m$) and n is the number of threads. If we were to include a control literal for each statement as in [15], then our solution might not scale in practice to large multi-threaded software systems.

Step 1: Add control literals cl_{ij} (where i is the ECS number and j is the thread identifier) to the VC ψ_k.
Step 2: Add negated control literals $\neg cl_{ij}$ to the schedule SCH, except those enabling the first interleaving.
Step 3: Check satisfiability of ψ_k; if ψ_k is satisfiable, then terminate with "error".
Step 4: Check whether the proof objects generated by the SMT solver contains any control literals; if not terminate with "no error".
Step 5: Remove literals that are contained in the proof objects from the schedule SCH and go to step 3.

Figure 5: Algorithm of the UW approach.

4. MODELLING SYNCHRONIZATION PRIMITIVES IN PTHREAD

This section presents our modelling of the synchronization primitives of the Pthread library [20]. We assume that the library function implementations are correct and focus our effort on verifying only the client programs that use them. We thus provide an instrumented model of the Pthread functions and use this to model check the client code.

4.1 Modelling Mutex Locking Operations

The Pthread library supports two functions to implement mutual exclusion between threads, *pthread_mutex_lock* and *pthread_mutex_unlock*. Both functions take as argument a data structure called *mutex* that has two states, "locked" and "unlocked". *pthread_mutex_lock* locks the mutex if it is unlocked; otherwise it blocks the current thread until the mutex is unlocked and can successfully be locked again. *pthread_mutex_unlock* simply unlocks a locked mutex. Computation paths are blocked on a mutex when a thread tries to lock a mutex that has already been locked by another thread. As an example, consider the threads t_A and t_B, which both lock and unlock the same mutex m, as shown in Figure 6. The paths $A_0; A_1; B_0; B_1$ and $B_0; B_1; A_0; A_1$ are non-blocking or *wait-free* while the other two paths are blocked.

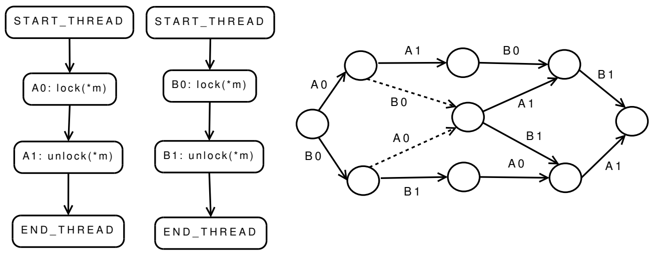


Figure 6: Computation paths blocking on a mutex.

A strategy to model mutex operations based on the notion of wait-free paths was proposed in [24, 25]. Instead of blocking the computation paths starting with $A_0; B_0$ and $B_0; A_0$, they are simply ignored by modelling *pthread_mutex_lock* as *atomic* $\{assume(*m == 0); *m = 1\}$, where the statement $assume(*m == 0)$ cuts off subsequent paths if the mutex is already locked. *pthread_mutex_unlock* is then modelled as *atomic* $\{assert(*m == 1); *m = 0\}$, which simply checks if the mutex is already locked. If so, the lock is released; otherwise, a thread tries to unlock a mutex that has not been locked previously, and we have detected an error.

This is sufficient to find bugs related to data races and lock acquisition ordering, but not to detect local and global deadlocks [24, 25]. We thus model *pthread_mutex_lock* in such a way that we can detect global and local deadlock caused by the wrong use of the mutexes; *pthread_mutex_unlock* remains unchanged. For this, we first need to look in more detail at the different possible states that our model allows for a thread: *(i) Join state:* the thread is waiting for another thread to terminate; *(ii) Lock state:* the thread is waiting for a mutex to be unlocked; *(iii) Wait state:* the thread is waiting for a signal or broadcast to wake up; *(iv) Exit state:* the thread has already exited; *(v) Free state:* the thread is not in any of the above four states and is free to execute its instructions. A thread is blocked if it is in one of the *join*, *lock* or *wait* states, and is supposed to be running if it is not in *exit* state. Global deadlock occurs when all threads wait for a mutex and a local deadlock occurs when some of the threads form a waiting cycle. In both cases, we can detect the deadlock if there is no running thread in the free state, i.e., the number of blocked threads is equal to the number of running threads. Figure 7 presents our modelling to detect global and local deadlock with mutexes, which maintains counts on both blocked and running threads with global variables.

```
int pthread_mutex_lock(pthread_mutex_t *m) {
extern uint trds_in_run, c_lock=0;
  atomic {
    unlocked = (mutex_lock_field(*m)==0);
    if (unlocked) mutex_lock_field(*m) = 1;
    else c_lock = c_lock + 1;
  }
  atomic {
    if (mutex_lock_field(*m)==0)
      c_lock = c_lock - 1;
    if (!unlocked) {
      deadlock_mutex = (c_lock<trds_in_run);
      assert(deadlock_mutex);
      assume(!deadlock_mutex);
    }
  }
  return 0;
}
```

Figure 7: Modelling mutex lock operation.

mutex_lock_field retrieves the state of the mutex. We also use the variable *c_lock* to count the number of threads that are in the lock state due to mutex m, and *trds_in_run* to count the number of threads that are currently running. Initially, the mutex is unlocked and we only lock it after the first call to *pthread_mutex_lock*. In subsequent calls, we increase the value of the variable *c_lock*, allow context switches, check if the mutex m was unlocked, and then assert $c\_lock < trds\_in\_run$. If the assertion fails, a deadlock was detected: a thread is blocked by a lock operation on a mutex and the required mutex never gets unlocked by the thread that owns it, either because the locking thread has exited or because it has been blocked by another operation. If the assertion holds, we then eliminate this execution as described above.

4.2 Modelling Conditional Waiting

We model functions *pthread_cond_wait*, *pthread_cond_signal*, and *pthread_cond_broadcast* from the Pthread library that implement conditional waiting. All functions take as argument a condition variable c that has also two states, "locked" and "unlocked"; *pthread_cond_wait* also takes a mutex argument. Our modelling of the conditional waiting operation again employs the notion of wait-free execution paths. *pthread_cond_wait* is used to block the thread on a condition variable; the blocked thread is woken up only if another thread calls signal or broadcast. If several threads are blocked on a condition variable, then *pthread_cond_signal* non-deterministically unblocks at least one of them while *pthread_cond_broadcast* unblocks all threads blocked on the specified condition variable.

Figure 8 shows our modelling for the wait-operation. We use the variable *c_wait* to count the number of threads that are in the wait state due to condition c. Whenever a thread calls *pthread_cond_wait*, we atomically lock the condition variable c, assert that the mutex m is currently locked, release the mutex (so that other threads that access it can make progress), and then increment the number of threads in wait state (i.e., threads that are waiting for a signal or broadcast to wake up). We then allow context switches before we check whether the number of threads in wait state is less than the total number of the threads that are currently running with the assertion $c\_wait < trds\_in\_run$. If the assertion holds or the variable c is locked, we simply eliminate this execution as described above.

```
int pthread_cond_wait(pthread_cond_t *c,
                      pthread_mutex_t *m) {
extern uint trds_in_run, c_wait=0;
  atomic {
    cond_lock_field(*c) = 1
    assert(mutex_lock_field(*m))
    mutex_lock_field(*m) = 0
    c_wait = c_wait + 1
  }
  atomic {
    deadlock_wait = (c_wait<trds_in_run)
    assert(deadlock_wait);
    assume(!deadlock_wait
          || cond_lock_field(*c)==0);
    c_wait = c_wait - 1
  }
  mutex_lock_field(*m) = 1
  return 0;
}
```

Figure 8: Modelling conditional waiting operation.

To model signal-operations, we simply release the condition variable, i.e., $c = 0$. To model broadcast-operations, we create a global variable called *broadcast_id*, which records the number of broadcast operations that have executed and which gets incremented inside *pthread_cond_broadcast*. In the wait-operation, the thread records the current value of *broadcast_id* before it is forced to make context switches to other threads. When the context is switched back to the current thread, an assertion checks if a broadcast operation has occurred by checking whether the current value of *broadcast_id* is greater than the recorded value. The deadlock is detected if there is no path with broadcast operations.

5. EXPERIMENTAL EVALUATION

We have implemented the lazy, schedule recording, and UW approaches described in Section 3 in our ESBMC tool that supports the SMT logics QF_AUFBV and QF_AUFLIRA as specified in the SMT-LIB [27]. In our experiments, we have used ESBMC v1.15.1 together with Z3 v2.11 [10], which was the most efficient SMT solver in our previous experiments [8]. ESBMC and the benchmarks are available at `http://users.ecs.soton.ac.uk/lcc08r/esbmc/`. All experiments were conducted on an otherwise idle Intel Pentium Dual CPU with 4 GB of RAM running Windows and Linux OS respectively. For all benchmarks, the time limit has been set to 3600 seconds to check all properties at once. All times given are wall clock time in seconds as measured through a single execution. In our experiments, we chose CHESS v0.1.30626.0 [21] and SATABS v2.5 [6] as two of the most widely used verification tools.

5.1 Comparison to CHESS

CHESS is a concurrency testing tool for C# programs. It implements iterative context-bounding and explores the various thread schedules deterministically [21]. CHESS requires idempotent unit tests that it repeatedly executes in a loop, exploring a different interleaving on each iteration. In this respect, it is similar to our lazy approach; however, CHESS is a purely dynamic, test-based tool and originally employed a stateless search technique, but its latest version (v0.1.30626.0) performs state hashing based on a happens-before graph to avoid exploring the same state repeatedly.

Table 1 shows the detailed results of the comparison between ESBMC and CHESS on a 2GHz machine. *reorder*, *twostage*, and *wrong lock* are different versions of a reader/writer program [26]. The numbers (x, y) indicate that we have x instance(s) of thread t_{set} and y instance(s) of thread t_{reader}. According to [26], increasing the number of instances of a given thread while keeping constant the number of instances of the other thread, substantially increases the "semantic hardness" of the error discovery. Note that all these benchmarks only check for a single, violated property. *micro* is a synthetic micro-benchmark [14] which checks a single valid property. It is used to check the scalability of multi-threaded software verification tools. The number in brackets indicates the total number of visible statements on each thread. In the table, L is the size of the code (in lines), and T the total number of threads. B is the number of BMC unrolling steps for each loop, while C is the context switch bound. Except for *reorder_6_bad*, C is set to the minimum number of context switches required to expose the error. *Time* is the time in seconds until the error is found; timeouts are denoted by TO. For ESBMC, I is the total number of generated interleavings, while *FI* is the total number of failed interleavings. The column *iter* gives the number of iterations required to prove or disprove the property in the UW approach. For CHESS, *Tests* reports the approximate number of tests executed, which is not related to the number of interleavings. Both tools identify the property violation (resp. confirm that it holds) in all cases where they do not run out of time or memory.

As we can see in Table 1, CHESS is effective for programs where there are a small number of threads, but it does not scale that well and consistently runs out of time when we increase the number of threads. In general, CHESS times out when the number of threads increases beyond six. The relatively poor scalability of CHESS has already been observed by [26]. In contrast, our lazy algorithm is able to find bugs quickly even when we increase the number of threads and the context bound, and consistently outperforms CHESS as well as the schedule recording and UW approaches. However, note that it runs out of memory for test cases 16 and 18 when we increase the number of context switches to 18 and 13 respectively.

5.2 Comparison to SATABS

SATABS is an ANSI-C model checker which supports the verification of multi-threaded software with shared variables using the CEGAR technique. We compare our approaches against SATABS v2.5 [6] based on Cadence SMV using a number of multi-threaded programs taken from standard benchmark suites. Table 2 shows the results achieved on a 3Ghz machine. Programs that end on "bad" contain an error (i.e., at least one of the properties is satisfiable) while those that end on "ok" are correct. Here, #P gives the number of properties to be verified for each program, which includes array bounds, pointer safety, division by zero, deadlock and order violations checks. A context bound of ∞ means that we did not specify a bound. A "-" result indicates that the tool failed with an error such as internal (†) and refinement (RF) failure, memory overflow (MO), time-out (TO), or failed to detect errors in the program. A "+" indicates that the tool detected the error or proved all VCs.

Programs 1-6 are concurrent implementations of stack, queue, and circular buffer data structures; programs 1 and 2 are extracted from an embedded application [7]. Programs 7-14 are from the INSPECT benchmark [29] and use mutex and condition synchronization primitives from the Pthread library. Programs 15-17 are from the VV-lab benchmarks [26] and contain common concurrency bugs such as data races, atomicity and order violations. Programs 18-20 are embedded applications that run on a dual core processor; they are implemented in a commercial set-top box product from NXP semiconductors [22]. Program 21 is the same synthetic micro-benchmark described in Section 5.1, but here we increase further the number of context switches to check the scalability of our approaches.

As we can see in Table 2, SATABS produces refinement failures (RF) for most programs. These programs contain linear arithmetic operations with arrays and the predicate abstraction technique implemented in SATABS seems to suffer from a lack of precision when dealing with arrays. However, the ability of a verification tool to check such programs is particularly important as many real-world multi-threaded programs belong to this class. SATABS also times out for large programs or for programs with many threads (cf. programs 7, 8, 9, 13, and 21). Additionally, SATABS gives false positives on programs 14-16, which contain known bugs related to data races, atomicity and order violations. Note that SATABS uses predicate abstraction and refinement, and in some sense tries to solve a harder problem than BMC. However, the results in Table 2 indicate that this problem may still be too hard for multi-threaded applications, as SATABS is unable to prove the required properties.

We can also see in Table 2 that if the program contains errors at all, these errors indeed generally occur in most interleavings explored; consequently, the lazy approach is very fast for these cases. The notable exception is *wronglock_bad*, where less than 0.1% of the interleavings expose the error

						CHESS		Lazy		Schedule	UW	
	Test Program	#L	#T	B	C	Time	Tests	Time	#FI / #I	Time	Time	Iter
1	reorder_3_bad (2,1)	84	3	3	4	1	200	**<1**	1/29	**<1**	**<1**	4
2	reorder_4_bad (3,1)	84	4	4	5	98	13000	**<1**	1/82	1	4	5
3	reorder_5_bad (4,1)	84	5	5	6	TO	429000	**<1**	1/277	4	18	6
4	reorder_6_bad (5,1)	84	6	6	7	TO	396000	**<1**	1/853	36	72	7
5	reorder_6_bad (5,1)	84	6	6	8	TO	371000	**<1**	1/2810	225	592	7
6	reorder_6_bad (5,1)	84	6	6	9	TO	367000	**<1**	1/8124	MO	MO	1
7	twostage_3_bad (2,1)	128	3	3	4	4	500	**1**	1/35	**1**	3	5
8	twostage_4_bad (3,1)	128	4	4	4	215	27000	2	1/42	**1**	4	5
9	twostage_5_bad (4,1)	128	5	5	4	TO	384000	2	1/44	**1**	5	5
10	twostage_6_bad (5,1)	128	6	6	4	TO	366000	**2**	1/45	**2**	5	5
11	wronglock_4_bad (1,3)	110	4	4	8	21	3000	**5**	2/489	10	89	9
12	wronglock_5_bad (1,4)	110	5	5	8	724	93000	**10**	3/2869	50	408	9
13	wronglock_6_bad (1,5)	110	6	6	8	TO	356000	**18**	4/12106	225	2060	9
14	wronglock_7_bad (1,6)	110	7	7	8	TO	330000	**34**	5/39100	MO	MO	1
15	micro_2_ok (100)	247	2	1	2	316	35855	**<1**	0/4	**<1**	**<1**	1
16	micro_2_ok (100)	247	2	1	17	TO	400000	**1095**	0/131072	MO	MO	1
17	micro_3_ok (100)	365	3	1	2	TO	272000	**<1**	0/9	**<1**	**<1**	1
18	micro_3_ok (100)	365	3	1	12	TO	290000	**1021**	0/121393	MO	MO	1

Table 1: Results of the comparison between ESBMC (v1.15.1) and CHESS (v0.1.30626.0).

							SATABS		Lazy			Schedule		UW		
	Test Program	#L	#T	#P	B	C	Time	Result	Time	Result	#FI / #I	Time	Result	Time	Result	Iter
1	circular_buffer_ok [7]	111	2	9	8	∞	†	−	**477**	+	0/12870	MO	−	MO	−	1
2	circular_buffer_bad [7]	109	2	8	8	5	†	−	**<1**	+	3/32	2	+	11	+	6
3	queue_ok [9]	147	2	12	41	∞	RF	−	**3**	+	0/6	**3**	+	**3**	+	1
4	queue_bad [9]	153	2	15	41	8	†	−	**3**	+	91/256	50	+	373	+	7
5	stack_ok [9]	105	2	5	11	12	†	−	**225**	+	0/4094	1026	+	1097	+	1
6	stack_bad [9]	106	2	6	11	4	RF	−	**<1**	+	4/16	**2**	+	6	+	4
7	fsbench_ok [29]	81	26	47	26	2	†	−	**252**	+	0/676	304	+	301	+	1
8	fsbench_bad [29]	80	27	48	27	2	†	−	**<1**	+	729/729	360	+	786	+	2
9	indexer_ok [29]	77	13	21	129	4	TO	−	595	+	0/17160	220	+	**218**	+	1
10	stateful20_ok [29]	60	2	3	20	10	†	−	**95**	+	0/1024	487	+	518	+	1
11	sync02_ok [29]	74	2	6	21	21	RF	−	**44**	+	0/121	60	+	60	+	1
12	sync02_bad [29]	74	2	6	21	21	RF	−	**8**	+	5/186	132	+	383	+	3
13	aget-0.4_bad [29]	1233	3	279	200	2	3346	+	137	+	1/1	127	+	**125**	+	1
14	bzip2smp_ok [29]	6366	3	8568	1	9	TO	−	**1800**	+	0/1294	MO	−	MO	−	1
15	reorder_10_bad (9,1) [26]	84	10	7	10	11	**1**	−	<1	+	1/154574	MO	−	MO	−	1
16	twostage_100_bad (99,1) [26]	128	100	13	100	4	**2**	−	88	+	1/139	93	+	195	+	5
17	wronglock_8_bad (1,7) [26]	110	8	8	8	8	**2**	−	90	+	6/104015	MO	−	MO	−	1
18	exStbHDMI_ok [22]	1060	2	24	16	20	TO	+	229	+	0/1	226	+	**213**	+	1
19	exStbLED_ok [22]	425	2	45	10	10	RF	−	**73**	+	0/11	73	+	787	+	1
20	exStbThumbs_bad [22]	1109	2	249	2	1	317	+	95	+	3/3	14	+	**12**	+	1
21	micro_10_ok (100) [14]	1171	10	10	1	17	TO	−	**254**	+	0/29260	MO	−	MO	−	1

Table 2: Results of the comparison between SATABS (v2.5) and ESBMC (v1.15.1).

and SATABS is substantially faster than ESBMC (but fails to find the error); however, even here the lazy approach outperforms both the schedule recording and UW approaches. Similarly, the lazy approach is capable of handling safe programs in which the number of threads and context switches grows quickly, which makes the formula harder and often "blows up" the SMT solver. The UW approach is typically slower than schedule recording. We suspect that the proof generation of the SMT solver (which is required to produce the unsatisfiable cores) causes memory overhead and corresponding slowdowns; this was also reported previously [11].

6. RELATED WORK

SMT-based BMC is gaining popularity in the formal verification community [10]. However, most work focuses on sequential software, uses only restricted sets of theories (e.g., integer and real arithmetic) that do not precisely reflect the ANSI-C semantics [12] or does not address important language constructs [1].

Cimatti et al. [4] describe an approach to verify SystemC that similarly combines explicit state space exploration (i.e., the explicit exploration of the different possible interleavings) with symbolic model checking (i.e., the symbolic representation and updates of the state). However, we use BMC instead of predicate abstraction, and we implement a realistic scheduler, i.e., our scheduler may preempt a thread at any visible instruction in its execution, whereas [4] encodes the semantics of the non-preempting SystemC scheduler. We also exploit the SMT techniques on large problems by encoding all possible interleavings into a single formula.

Qadeer and Rehof present a pragmatic method to discover bugs in concurrent software in which the program analysis is restricted to executions with a bounded number of context switches [23]. However, they do not apply it to realistic concurrent software benchmarks and the integration of the context-bounded algorithm into the explicit state model checker ZING is left for future work. Rabinovitz and Grumberg describe an extension of CBMC to concurrent C programs [25], which translates each C thread individually into SSA form and adds constraints for a bounded number of context-switches, as in [23]. This approach, however, is limited to two threads, and requires the user to run CBMC twice in order to detect different types of bugs ("regular" and concurrency bugs). It is also only evaluated on a concurrent bubblesort, but not on a set of realistic applications.

Ganai and Gupta describe a lazy method for modelling multi-threaded concurrent systems using shared variables [13], but this again is restricted to two threads. Gupta et al. [17] extend [13, 16] by supporting more than two threads and by combining dynamic partial order reduction with symbolic state space exploration. The benchmarks that have been reported are a parameterized version of the dining philosophers model, which are untypical multi-threaded C programs. Grumberg et al. propose an algorithmic method based on SAT and BMC to model check a multi-process system based on a series of under-approximated models [15]. This approach, however, does not integrate context-bounded analysis and it does not address the problem of model checking multi-threaded C software.

7. CONCLUSIONS AND FUTURE WORK

We have presented the lazy, schedule recording, and UW algorithms to model check multi-threaded ANSI-C software with shared variable communication between the threads. We have also presented our modelling of the synchronization primitives of the Pthread library that allows us to detect not only atomicity and order violations, but also local and global deadlock, that previous attempts are unable to find [13, 16, 17, 25]. Surprisingly, our approach to check constraints lazily is extremely fast for programs that contain errors and to a lesser extent even for safe programs in which the number of threads and context switches grows quickly. The experimental results also show that the lazy approach generally outperforms not only the schedule recording and UW approaches, but also CHESS [21] and SATABS [6] tools on several non-trivial benchmarks. As far as we are aware, there is no other work that considers a comprehensive SMT-based BMC procedure to verify multi-threaded ANSI-C software by combining symbolic model checking with explicit state space exploration. In future, we plan to explore the use of Craig interpolants to prove non-interference of context switches, and to develop an efficient method on top of ESBMC to localize faults in multi-threaded programs.

8. REFERENCES

[1] A. Armando, J. Mantovani, and L. Platania. Bounded model checking of software using SMT solvers instead of SAT solvers. *STTT*, vol. 11 (1), pp. 69–83, 2009.

[2] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.

[3] A. Biere. Bounded model checking. In *Handbook of Satisfiability*, pp. 457–481. 2009.

[4] A. Cimatti et al. Verifying SystemC: a software model checking approach. *FMCAD*, 2010.

[5] E. Clarke, D. Kroening, and F. Lerda A tool for checking ANSI-C programs. *TACAS*, *LNCS* 2988, pp. 168–176, 2004.

[6] E. Clarke, D. Kroening, N. Sharygina, and K. Yorav SATABS: SAT-based predicate abstraction for ANSI-C. *TACAS*, *LNCS* 3440, pp. 570–574, 2005.

[7] L. Cordeiro et al. Agile development methodology for embedded systems: A platform-based design approach. *ECBS*, pp. 195–202, 2007.

[8] L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based bounded model checking for embedded ANSI-C software. *ASE*, pp. 137–148, 2009.

[9] T. H. Cormen et al. *Introduction to algorithms*. Second edition, 2001.

[10] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS*, *LNCS* 4963, pp. 337–340, 2008.

[11] L. M. de Moura and N. Bjørner. Proofs and refutations, and Z3. In *LPAR* Workshops, 2008.

[12] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. *ICCAD*, pp. 794–801, 2006.

[13] M. K. Ganai and A. Gupta. Efficient modeling of concurrent systems in BMC. *SPIN*, *LNCS* 5156, pp. 114–133, 2008.

[14] N. Ghafari, A. Hu, and Z. Rakamaric. Context-bounded translations for concurrent software: An empirical evaluation. *SPIN*, *LNCS* 6349, pp. 227–244, 2010.

[15] O. Grumberg et al. Proof-guided underapproximation-widening for multi-process systems. *POPL*, pp. 122–131, 2005.

[16] V. Kahlon et al. Semantic reduction of thread interleavings in concurrent programs. *TACAS*, *LNCS* 5505, pp. 124–138, 2009.

[17] V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. *CAV*, *LNCS* 5643, pp. 398–413, 2009.

[18] A. Lal and T. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Form. Methods Syst. Des.*, 35(1):73–97, 2009.

[19] K. L. McMillan and N. Amla. Automatic abstraction without counterexamples. *TACAS*, *LNCS* 2619, pp. 2–17, 2003.

[20] F. Mueller. A library implementation of posix threads under unix. *USENIX*, pp. 29–41, 1993.

[21] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *PLDI*, pp. 446–455, 2007.

[22] http://www.nxp.com/, 2009.

[23] S. Qadeer and J. Rehof. Context-bounded model checking of concurrent software. *TACAS*, *LNCS* 3440, pp. 93–107, 2005.

[24] S. Qadeer and D. Wu. Kiss: keep it simple and sequential. In *PLDI*, pp. 14–24, 2004.

[25] I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. *CAV*, *LNCS* 3576, pp. 82–97, 2005.

[26] N. Rungta and E. G. Mercer. Clash of the titans: tools and techniques for hunting bugs in concurrent programs. *PADTAD*, pp. 1–10, 2009.

[27] SMT-LIB. *The Satisfiability Modulo Theories Library.* http://combination.cs.uiowa.edu/smtlib, 2009.

[28] S. L. Torre, P. Madhusudan, and G. Parlato. Reducing context-bounded concurrent reachability to sequential reachability. *CAV*, LNCS 5643, pages 477–492, 2009.

[29] http://www.cs.utah.edu/, 2010.